An Evolutionary Programming Approach to Probabilistic Model-based Fault Diagnosis of Chemical Processes

by
Carlos Rojas-Guzmán

M.S.C.E.P., Massachusetts Institute of Technology
February 1991

Ingeniero Químico, Universidad Nacional Autónoma de México
August 1989

Submitted to the Department of Chemical Engineering
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Author

Department of Chemical Engineering
December 9, 1994

Certified by

Mark A. Kramer
Thesis Supervisor

Accepted by

Robert E. Cohen
Chairman, Committee for Graduate Students

An Evolutionary Programming Approach to Probabilistic Model-based Fault Diagnosis of Chemical Processes

by
Carlos Rojas-Guzmán

Submitted to the Department of Chemical Engineering
on December 9, 1994,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Keywords: Process systems engineering, knowledge engineering abductive inference, chemical industry, fault diagnosis, probability theory, system failure nd recovery, expert systems, artificial intelligence, Bayesian belief networks, genetic algorithms, evolutionary programming.

Abstract

Uncertainty is intrinsic to fault diagnosis in chemical processes. The need to model uncertainty derives primarily from engineering limitations both in process modeling and measurement. Deterministic models for the effects of faults cannot always be constructed, and measurements to diagnose every fault uniquely may be missing, noisy or inaccurate. This project develops and exploits the advantages of a probabilistic approach to construct an improved theoretical framework capable of modeling and handling uncertain, incomplete and conflicting information to perform fault diagnosis in complex dynamic chemical processes.

The type of reasoning involved in diagnosis is called abductive inference and consists of deriving the *best* (most probable) globally consistent explanation for a given set of evidence (measured or observed variables). Bayesian networks are graphs used to model and reason about uncertain systems by qualitatively (through the network topology) and quantitatively (through the probabilistic parameters) encoding conditional dependence and independence among the system variables. Bayesian networks have a sound theoretical basis, are consistent with probability theory, use results from graph theory, and constitute a powerful tool in probabilistic reasoning. The Bayesian network framework was selected as the basis for a unifying representation for a probabilistic framework in which knowledge from different sources can be integrated.

Recently developed methods to propagate probability information in the belief network structure use distributed parallel computations in which probabilistic values are locally propagated between neighboring nodes. However, abductive inference in belief networks belongs to the class of NP-hard problems. Complexity increases drastically as a function of the number of undirected cycles, discrete states per variable, and variables in the network.

Approximate near-optimal methods constitute an alternative. A graph-based genetic algorithm is developed and implemented as part of this project by posing inference as search in a large discrete multi-dimensional space where the metric (phenotype) is the probability of each diagnostic hypothesis. The space is initially sampled randomly creating a population of diagnostic hypotheses. The search effort is allocated in parallel to $O(N^3)$ hyperplanes. Through several iterations (generations), solutions (individuals) are selected and combined (reproduction and crossover) to improve (evolve) the quality of the solution set (population) towards the optimal solution. Convergence to a local optimum is avoided by introducing low frequency random changes (mutations). Efficiency results from the ease of evaluating any given solution, and from the genotype representation (solution specification) as a graph structure. By performing crossover in graphs, as opposed to strings, *semantic closeness* is preserved (meaningful sets of directly related variables are kept together) and the *compact building block hypothesis* is satisfied. This hypothesis states that highly fit, short-defining-length schemata (or similarity templates) are propagated through generations by giving exponentially increasing samples to high phenotype individuals.

The Bayesian belief network theoretical paradigm is extended to handle temporal reasoning and dynamics through the incorporation of time-indexed nodes. The resulting extended framework, the Multi-Stage Bayesian Network (MSBN) can take advantage of the same inference algorithms developed for Bayesian networks.

Experiments conducted on a well-known model are used to quantify the performance of the algorithm, and to optimize the algorithm parameters. A real-time on-line industrial implementation in an unmanned, remotely controlled chemical plant is used to illustrate the capabilities of the methodology.

Thesis Supervisor: Prof. Mark A. Kramer

Acknowledgments

The evolution of this project was greatly enriched by the continuous advice of Prof. Mark Kramer and from the ideas, criticism, and curiosity of the thesis committee members, Prof. Charles Cooney, Prof. George Stephanopoulos and Prof. Peter Szolovits. Also invaluable were the frequent discussions within our research group, Mike Thompson, Jon Tan, Lloyd Johnston, and during the early stages, Jim Leonard and Bertrand Sliwa.

Contents

1 Introduction 15

1.1 Motivation 15

1.2 Problem Definition 18

1.3 Summary of Research Objectives 19

1.3.1 Knowledge representation 20

1.3.2 Knowledge integration 20

1.3.3 Knowledge acquisition and translation 21

1.3.4 Dynamic behavior representation 21

1.3.5 Diagnosis based on probabilistic reasoning 22

1.3.6 Transparency of diagnostic conclusions 22

2 Previous Approaches to Fault Diagnosis in Chemical Engineering 24

2.1 Quantitative Model-Based Approaches 25

2.1.1 Direct estimation approaches 25

2.1.2 Comparative models 26

2.1.2.1 Generalized likelihood ratio method 26

2.1.2.2 Serial elimination 27

2.2 Qualitative Model-Based Approaches 28

2.2.1 Causality-based methods 28

2.2.1.1 Signed directed graph methods 28

2.2.1.2 Signed directed graph extensions 29

2.2.1.3 The Model-Integrated Diagnostic Analysis System 29
2.2.1.4 Model-based monitoring of dynamic systems 29
2.2.2 Residual incidence structure 30
2.2.2.1 The Governing Equations Method 30
2.2.2.2 The Diagnostic Model Processor 30
2.2.2.3 Parity-space methods 31
2.2.3 Causal models 31
2.3 Heuristic Approaches 31
2.4 Methods Based on Solved Cases 32
2.4.1 Pattern recognition approaches 32
2.4.2 Case-based reasoning 33

3 Probabilistic Reasoning Under Uncertainty 35
3.1 Previous Approaches 35
3.1.1 Probability Theory 36
3.1.2 Logical approaches 36
3.1.2.1 Default reasoning and truth maintenance systems 36
3.1.2.2 Autoepistemic or nonmonotonic logic 37
3.1.2.3 Circumscription 37
3.1.3 Quantitative approaches to uncertainty 38
3.1.3.1 Probabilistic logic 38
3.1.3.2 Paris's maximum entropy 38
3.1.3.3 Bundy's incidence calculus 38
3.1.3.4 The Dempster-Shafer theory of evidence 39
3.1.3.5 Bayesian belief networks 39
3.1.4 Non-normative approaches 39
3.1.4.1 Endorsements 40

3.1.4.2 Certainty factors in rule-based systems 40
3.1.4.3 Fuzzy logic 41
3.1.4.4 Possibilistic logic 41
3.2 Bayesian Belief Networks 41
3.2.1 Graph representation 42
3.2.2 Probabilistic parameters and their sources 45
3.2.3 Inference features of belief networks 50

4 A Probabilistic Framework for Model-based Diagnosis 52
4.1 Elements of the Methodology 52
4.1.1 Knowledge representation 54
4.1.2 Knowledge integration 55
4.1.3 Knowledge acquisition 56
4.1.4 Dynamic behavior representation 57
4.1.5 Diagnosis based on probabilistic reasoning 57
4.2 Initial Evaluation Criteria 59
4.3 Probabilistic Modeling of Chemical Processes 59
4.3.1 A fluid catalytic cracker rule-based expert system 61
4.3.2 Examination of the rule-based expert system approach 65
4.3.2.1 Selection of rule structure 66
4.3.2.2 Expression of fault-symptom relations 67
4.3.2.3 Representation of uncertainty 67
4.3.2.4 Handling incomplete evidence 68
4.3.2.5 Combination of evidence 68
4.3.2.6 Validation and modification of compiled knowledge 69
4.3.3 Advantages of the Bayesian network modeling approach 69
4.3.4 Bayesian network representation of a FCC unit 70

4.3.5 Illustration of inference features 73

4.4 Multi-Stage Bayesian Networks 77

4.4.1 Signed directed graph and residual-based methods 77

4.4.2 Capabilities of SDG and residual-based approaches 78

4.4.3 Multi-Stage Bayesian Networks 81

4.4.4 Model construction and comparison 84

4.4.5 Summary 90

5 A Graph-based Genetic Algorithm for Abductive Inference 91

5.1 The Mathematical Problem 91

5.2 Motivation 92

5.3 Genetic Algorithms 94

5.4 A Graph-based Genetic Algorithm 97

5.4.1 Problem representation 99

5.4.2 Space metric 101

5.4.3 Algorithm parameters 101

5.4.4 Convergence criteria 102

5.4.5 Selection 103

5.4.6 Mutation 104

5.4.7 Crossover in graph-based genetic algorithms 104

5.4.7.1 Characterization of link distributions 108

5.4.7.2 Comparison of crossover strategies 111

5.4.8 Performance metrics 112

5.5 Theoretical Analysis: Admissibility 113

5.6 Experimental Comparison of Parent Selection Criteria 115

5.7 Performance on the ALARM Network 121

5.7.1 Model description 122

5.7.2 Experiments 122

5.7.3 Regression model 124

5.7.4 Parameter optimization 128

5.8 Comparison of MPEs and Posterior Distributions 128

5.8.1 Introduction 128

5.8.2 Description of experiments 130

5.8.3 Results 131

5.8.4 Conclusions 131

5.9 Discussion 132

5.10 Summary 134

6 Industrial Implementation 135

6.1 Introduction 135

6.2 On-line Diagnosis 137

6.3 Fault Suppression Strategy 137

6.4 Implementation 138

6.5 Plant Description 139

6.6 Model Construction 142

6.7 Model Testing, Validation, and Maintenance 145

7 Conclusions 148

8 Future work 151

8.1 A Genetic Algorithm with a Marginalization Algorithm 152

8.1.1 Limitations of nested dissection 153

8.1.2 Basic concept 153

8.1.3 Time complexity 154

8.1.4 Hybrid algorithms 154

8.1.5 Complement for abductive inference algorithms 154

8.2 User Interaction 155

8.2.1 Generation of explanations 155

8.2.2 Inclusion of additional observables 156

8.2.3 Suggestion of corrective action 156

8.2.4 Automated response on simple cases 156

8.2.5 Optimal test generation 156

8.3 A Hybrid Search Algorithm 157

8.3.1 Previous work on hybrid algorithms 157

8.3.2 Experiments 157

8.3.3 Preliminary results 158

8.3.4 Discussion 158

8.4 Residual-based Probabilistic Diagnosis 159

8.4.1 Limitations of previous approaches 161

8.4.2 Multi-Stage Bayesian Networks 161

8.4.3 Desired model properties 162

8.4.4 Network transformations 163

8.4.5 Problem representation 163

8.4.6 Summary 165

9 References 166

Appendix A. Prior and Conditional Distributions 180

Appendix B. Exhaustive Systematic Enumeration of Space States 181

Appendix C. Probability Distributions for Network BN1 183

Appendix D. Comparison of Crossover Strategies 184

Appendix E. Node Removal 187

Appendix F. Alternative Approaches: Mixed Integer Non-linear Programming 190

Appendix G. Inverse Response Modeling with MSBNs 192

Appendix H. Directed Cycles in Bayesian Networks 194

Appendix I. Diagnostic System, User's Guide 197

Appendix J. GALGO© Source Code 226

Appendix K. Monte Carlo Simulation using GALGO 375

Appendix L. Comparison of MPEs and Posterior Distributions 388

List of Figures

3-1	Mathematically modeled sensor to illustrate Monte Carlo simulation	49
4-1	Elements of the diagnostic methodology	54
4-2	FCC process hierarchy	62
4-3	Hypothetical fault-symptom relations	66
4-4	Bayesian network topology for a FCC section	71
4-5	Effect of context on competing hypotheses or *explaining away*	75
4-6	Digraph semantics	80
4-7	Semantics of graph elements in MSBN	81
4-8	Graphical representation of temporal causal relations in the MSBN	82
4-9	Modeling of causal relations with uncertain time delays in the MSBN	83
4-10	Two tank system	85
4-11	Digraph representation for the two tank system	86
4-12	Bipartite graph associating faults with equation residuals	87
4-13	Multi-Stage Bayesian Network for the two tank system	89
5-1	Bayesian networks and genetic algorithms, a powerful combination	95
5-2	Basic steps of the genetic algorithm	99
5-3	The topology for a small Bayesian network model	104
5-4	Traditional crossover in the string representation	106
5-5	Crossover with the graph representation	109

5-6 Ten node Bayesian network (BN0) 110

5-7 Graph-based crossover with an expansion level of 1 116

5-8 Topology for Bayesian network model BN3 117

5-9 Average selection frequency for random centers and expansion levels 118

5-10 Topology for the ALARM Bayesian network from Beinlich, *et al* 123

6-1 On-line use of the probabilistic diagnostic system 140

6-2 Flow diagram for a membrane separation plant 141

6-3 Knowledge acquisition form to rank competing root causes 144

6-4 Elicitation of prior probabilities 144

6-5 Form to elicit probabilistic distributions given a scenario 145

8-1 Residual-based diagnosis 160

List of Tables

3-1	Discrete states and probability distributions for a sensor model	49
3-2	Conditional probabilities from Monte Carlo simulation	50
4-1	Conditional probability distribution for the *Header Pressure*	72
4-2	Conditonal probability distribution for the *Feed Dump*	72
4-3	Conditional probability table for the *Load on Wet Gas Compressors*	73
5-1	Parameter sets for crossover partition strategies	111
5-2a	Parent selection with uniform distribution	119
5-2b	Parent selection with proportional distribution	119
5-2c	Parent selection with a logarithmic transformation distribution	120
5-3	Summary of performance on different cases and parameter sets	124
6-1	Discrete states for key variables	144

Chapter 1

Introduction

Safety, environmental and economical concerns motivate the development of computer tools to aid and improve the performance of plant operators. Various diagnostic tools for detection of malfunctions and operator advising have been developed to help operators deal with unexpected events encountered during process operation. The methods proposed are diverse and range from numerical estimation to qualitative cause and effect reasoning and expert systems. Each approach has its own strengths and weaknesses, but something which appears to be missing is a methodology able to adequately handle the uncertain information unavoidably associated to dynamic chemical processes. Even though some recognize the existence of process knowledge in different forms at different sources, a proper integration of this knowledge is still required.

1.1 Motivation

Uncertainty is intrinsic to fault diagnosis of chemical processes. The need to model uncertainty derives primarily from engineering limitations both in process modeling and measurement. Deterministic models for the effects of faults cannot always be constructed due to the variability in fault-symptom relationships. Faults exist which exhibit the same or similar symptoms and, as a result, distinguishing among suspected faults is difficult.

Other sources of uncertainty are the need to discretize continuous variables and the fact that prior failure probabilities are not equal. In addition, measurements to diagnose every fault uniquely may be missing, noisy or inaccurate. The uncertainty in the measurements is a result both of sensor and process noise. Consequently, chemical process diagnosis requires a framework that can represent and process uncertain knowledge. This project deals with the use of probability theory to represent the uncertain elements of the diagnosis problem, the examination of the Bayesian network framework, the extension to model temporal relations and the development of an algorithm to perform abductive inference to determine the most probable diagnostic hypotheses using the resulting models.

Although several methods for dealing with uncertainty in diagnostic reasoning have been studied (*e.g.* certainty factors, fuzzy logic, Dempster-Shafer theory), probability theory appears to provide the most complete and consistent framework for dealing with uncertain knowledge (Neapolitan 1990; Pearl, 1988). While limitations of non-probabilistic approaches have been known for several years, recently methods have emerged for calculating probabilities in probabilistic knowledge bases with an ease approaching that of certainty factor updating in rule-based expert systems.

These advances are based on the representation of knowledge by Bayesian belief networks, also called Bayesian networks, causal networks and, when augmented with decision vertices, probabilistic influence diagrams. Influence diagrams were formalized by Howard and Matheson (1981) as a tool for decision analysis. Related work in decision analysis includes (Miller and co-workers, 1976) and (Howard and Matheson, 1984). In the context of the Dempster-Shafer theory, belief networks are called galleries (Lowrance, Garvey and Strat, 1986), qualitative Markov networks (Shafer, Shenoy and Melouilli, 1987) or constraint networks (Montanari, 1974).

A Bayesian belief network is a directed acyclic graph whose nodes represent variables defined in the same probability space and whose arcs represent probabilistic links between variables. It efficiently summarizes the joint probability distribution of this set of variables by associating a prior probability distribution to each root node and a probability distribution to each non-root node conditioned only on the value of its parent variables, that is, only those on which it directly depends.

Inference algorithms on belief networks have been developed by Pearl (1988), Lauritzen and Spiegelhalter, (1988), Cooper (1990), Rojas-Guzmán and Kramer (1993b) and others. Methods handling continuous Gaussian probability distributions have been proposed for influence diagrams by Shachter and Kenley (1989) and Kenley (1986).

A significant number of applications of Bayesian networks have been to medical diagnosis. Different methodologies and forms of representing knowledge have been used in the medical domain to perform diagnosis. Some similarities and differences exist between medical diagnosis and chemical engineering diagnosis. In chemical engineering, often a known design is used when building a process plant and consequently mathematical models exist which describe the original structure and behavior of the system. In contrast, in medical diagnosis, 'blue prints' for the human body do not exist yet, although qualitative causal knowledge is available for some physiological processes. An important difference is that the functions and structure of the human body are fundamentally the same for each new patient, whereas each chemical process differs widely from others. Consequently, large databases with examples exist which can be used for medical diagnosis. Very often chemical processes have features never observed before as a result of a new design or a process modification.

Nonetheless, Bayesian networks are attractive for chemical process diagnosis. Belief networks can clearly represent the causal relations known for chemical processes. In addition, the uncertainty associated with model information and sensor data can be adequately captured by the probabilistic parameters of the Bayesian networks. Within this framework, abductive inference can be performed to obtain diagnostic conclusions based on measured or observed symptoms of the process.

1.2 Problem Definition

This research project focuses on the development of a diagnostic methodology capable of representing complex non-linear uncertain systems and performing inference about their behavior under normal and abnormal operation. Several elements of the methodology require careful consideration. It is necessary to select a framework to represent the available uncertain knowledge, the framework must be capable of integrating and exploiting knowledge from different sources. Knowledge acquisition and translation require attention to ensure acceptance of the methodology among the intended users: researchers, process engineers and plant operators. An important component of the methodology is an efficient algorithm to perform inference on the resulting model. Time constraints make an efficient implementation necessary. In addition, the implementation of the methodology in a computer program, the construction, and testing of an industrial diagnostic system are necessary to realistically evaluate the capabilities of the proposed methodology.

This thesis is organized as follows. This chapter briefly describes the requirements of a diagnostic methodology for chemical processes and summarizes the research objectives of this project. Chapter 2 surveys previous approaches to fault diagnosis in chemical

engineering. Several techniques for probabilistic reasoning under uncertainty are reviewed in Chapter 3. Chapter 4 describes the proposed probabilistic diagnostic methodology and introduces the Multi-Stage Bayesian Network framework. Chapter 5 focuses on a non-conventional graph-based genetic algorithm developed to perform approximate inference on Bayesian networks. An industrial implementation is described in Chapter 6 and conclusions are presented in Chapter 7. Chapter 8 outlines promising areas of research and Chapter 9 lists references used in this work.

1.3 Summary of Research Objectives

The main goal of this project is to develop an improved theoretical framework capable of modeling and handling uncertain, incomplete, and conflicting information to perform fault diagnosis in complex dynamic chemical processes. The project aims to develop and exploit the advantages of a probabilistic approach to deal with uncertainty in diagnostic reasoning and to combine quantitative and qualitative knowledge from different sources.

Diagnosis in chemical plants involves the detection, location and identification of one or more faults as well as the determination of their cause or causes. Moore and Kramer (1986) have described the useful characteristics of a diagnosis system. The system should be able to encode and combine uncertain, incomplete, and conflicting information describing chemical processes and their behavior. Since knowledge is frequently obtained from different sources, this framework must be able to integrate knowledge expressed in different forms. Temporal reasoning is also necessary. The diagnostic conclusion should be obtained in real-time from an on-line system. The most probable diagnosis should be produced, and a likelihood ranking of the diagnostic conclusions should be given. The

system should include suggestion of corrective actions, and selection of useful additional tests, if appropriate.

From a theoretical standpoint, the main objective of this project is to develop an improved probabilistic framework for dynamic systems.

Specific objectives and an outline of the issues to be addressed in order to develop the probabilistic diagnostic methodology follows.

1.3.1 Knowledge representation

In order to reason about any system, means to represent it as an abstraction are necessary. The appropriateness of the model depends both on the original entity to be represented and on its intended use. A model should be clear and should reflect and emphasize those aspects of reality of interest for the problem to be solved, while omitting irrelevant information. In addition, the language chosen to think about the system must lend itself to performing the computations necessary for diagnostic reasoning.

The first objective of this project is to evaluate Bayesian belief networks as the unifying representation for a probabilistic framework and compare and contrast this approach to rule-based expert systems.

1.3.2 Knowledge integration

The knowledge available about chemical processes exists in different forms. Sources of process behavior descriptions include mathematical models, plant history data, direct or indirect measurements (obtained from sensors or from direct observation or measurement by a human operator), and empirical knowledge (obtained by a human operator based on previous experience and observation of the process). Often, only one of these sources is

used in a given framework, even when it is recognized that some valuable information is being neglected. The combination and integration of information expressed in different formats is a difficult problem.

The second objective addressed in this research project is to demonstrate how to efficiently encode and integrate qualitative and quantitative knowledge from different sources within the same framework using Bayesian belief networks as the unifying representation.

1.3.3 Knowledge acquisition and translation

In addition to the diagnostic framework, methodologies are required to acquire, encode and process knowledge. The construction of a knowledge base should be a more formalized procedure than the one which has currently been used in expert systems. Parameters for a probabilistic model can be obtained from plant history data, through subjective interpretation of observations or, in some cases, by performing Monte Carlo simulations.

The third objective is to provide guidelines to formalize the acquisition and translation of information describing a chemical process and its behavior into the chosen representation, in this case the probabilistic framework. Both the topology and the parameters should be determined by a formalized methodology.

1.3.4 Dynamic behavior representation

Chemical processes are often dynamic, and even those operating at steady-state will undergo transients as a result of the occurrence of a fault. This motivates the incorporation of dynamic modeling in the diagnostic framework of Bayesian belief

networks. Additionally, temporal capabilities of probabilistic reasoning should be explored to assist in early fault detection.

The fourth objective is to develop a methodology to handle both deterministic and probabilistic dynamic behavior. Through the extension of the Bayesian network framework to incorporate time-indexed variables, dynamic relations can be handled.

1.3.5 Diagnosis based on probabilistic reasoning

The inference methodology is an important aspect of a diagnostic system. An efficient algorithm is necessary to perform inference on Bayesian networks models of chemical processes. When operating under time constraints, a fast advice is of value.

The fifth objective is to find or develop adequate methods to perform abductive inference for the specific class of problems addressed. This objective will be achieved by considering exact and approximate methods, and combinations of them and by characterizing the topology arising from the representation of chemical processes. This involves studying the connectivity and sparseness of the network. Inference methodologies must also be capable of working on the dynamic extensions of Bayesian belief networks.

1.3.6 Transparency of diagnostic conclusions

Diagnostic conclusions are most useful when presented in a simple and concise form that enables the plant operator to respond quickly to the malfunction. The selection and format of the information to be presented to the operator is crucial. The generation of explanations of the process leading to a diagnostic conclusion is by itself a research area. The generation of explanations constitutes a very useful feature in any automated reasoning system.

The sixth objective is to select and summarize relevant information to plant operators to maximize the usefulness of the computer-aided diagnostic tool.

Chapter 2

Previous approaches to Fault Diagnosis in Chemical Engineering

Fault diagnosis methodologies can be broadly classified in four groups according to the type of knowledge they use. The first group is based on quantitative models, typically sets of algebraic or differential equations. The second group uses qualitative models, for example, the signed digraph. A third group is based on heuristic and empirical knowledge. Some of these methods try to capture human expertise by encoding functional behavior as compiled if-then rules. Finally, the fourth group is based on previously solved cases. Here the idea is to *learn* associations between symptoms and failure modes from previously classified cases. Within this group two approaches exist, pattern recognition and case-based reasoning. A brief discussion follows for each group of methods.

2.1 Quantitative Model-Based Approaches

Quantitative models are used by two main groups of methods, direct estimation approaches and comparative approaches. In addition, quantitative models can be used indirectly for feature production which can then be processed qualitatively, heuristically or by pattern recognition.

2.1.1 Direct estimation approaches

Estimation is the process of extracting information from data which is often noisy. An optimal estimator is a computational algorithm that processes measurements to deduce a minimum error estimate of the state of a system. The minimum error estimate is based on a stated criterion of optimality. The technique uses knowledge of system and measurement dynamics, assumed statistics of system noises and measurement errors, and initial condition information (Gelb, 1974). Because criteria of optimality are stated as a function of the quantitative estimated states, estimation methodologies require quantitative process models.

Among the advantages of this class of techniques are that the estimation error is minimized in a well-defined statistical sense and all measurement data plus prior knowledge about the system are utilized. One disadvantage is its sensitivity to model error including assumed statistics. Another disadvantage is the limitation on the number of parameters that can be estimated simultaneously with reasonable certainty in the result. For example, sensor biases cannot generally be estimated simultaneously with extents of process faults.

Filtering refers to estimating the state vector at the current time, based upon all past measurements. A recursive filter is one in which there is no need to store all past measurements for the purpose of computing present estimates. Early work in linear state estimation techniques was done by Luenberg (1964, 1971).

One of the most common and powerful optimal filtering techniques was developed by Kalman (1960) for estimating the state of a linear system. Several optimal recursive filter techniques based on state-space, time domain formulations are described in (Kalman,

1960, 1961; Blum, 1961). Optimal estimation techniques perform best where multiple, noisy measurements are strongly related in the mathematical model. According to Gelb (1974) the Kalman filter is, in essence, a recursive solution to Gauss' original least-squares problem that enables sequential processing of measurement data, as opposed to batch processing. Both discrete and continuous formulations of the Kalman filter can be written.

Watanabe and Himmelblau (1983, 1984), among many other authors, have applied the extended Kalman filter to identify process parameters that would indicate process faults caused by deterioration of components.

2.1.2 Comparative models

2.1.2.1 Generalized likelihood ratio method

Because of the difficulty of simultaneously estimating many process parameters by direct estimation approaches, the problem of fault diagnosis is often reduced to a problem of selection among rival models. Each fault model will contain only a small number of parameters to be estimated. Willsky and Jones (1974) and Narasimhan and Mah (1987) proposed methods based on statistical tests to detect, identify and estimate gross errors in steady state processes. The Generalized Likelihood Ratio (GLR) method (Narasimhan and Mah, 1987) has been proposed to discriminate rival models. The GLR method is based on the likelihood ratio statistical test. The GLR approach provides a general framework for the identification of any type of gross error that can be mathematically modeled. Advantages of this method are its generality: both steady state and dynamic systems can be modeled. Additionally, an estimate of the magnitude of the gross error is obtained, which can be used to judge the impact of the gross error in the process. The methodology requires extensive process modeling, since a model of the process in each possible fault state is required.

2.1.2.2 Serial elimination

A specialized technique for comparison of multiple models is available when sensor failures and leaks comprise the set of possible faults. Through the use of serial compensation (Romagnoli and Stephanopoulos, 1981) multiple gross errors can be identified. Gross errors are caused by non-random events (i.e. sensor biases, leaks) as opposed to random noise errors. This procedure exploits the association of a gross error with a measurement. Consequently it is not applicable to errors that are not directly associated with measurements or individual balance equations. The removal of a constraint from the initial model enables the system to identify an alternate model consistent with the observations. By identifying the normal parts of the model, the faults are located. If a leak is suspected, the removal of a balance equation around a certain equipment may render the rest of the model consistent with the measurements, and will help to determine the location of the fault.

In general, methods for gross error detection use statistical tests in combination with an identification strategy. These methods are able to detect whether or not a gross error is present but do not perform diagnosis, understood as the determination of the cause, or the classification of the gross error. Several statistical tests have been developed such as the measurement test (Mah and Tamhane, 1982) and the nodal test (Mah and co-workers, 1976). Tamhane and Mah (1985) reviewed statistical tests for gross error detection in steady state chemical processes. These gross errors reflect sensor biases or failure, or leaks. Also, a Bayesian approach to detect gross errors was presented in (Tamhane and co-workers, 1988).

2.2 Qualitative Model-Based Approaches

Several methods which use qualitative models are discussed in this section. These methods can be grouped in causality-based methods and residual incidence methods.

2.2.1 Causality-based methods

2.2.1.1 Signed directed graph methods

Iri and co-workers (1979) proposed an algorithm to diagnose failures applying a signed digraph, a model based on graph theory which represents influences between the elements of the process. Motivated by the need to eliminate complicated and inefficient quantitative simulations, the propagation of failures is simplified in a qualitative fashion. The representation is limited since each variable is represented only as high, normal, or low, and the single fault assumption is used to make the method feasible. A more general framework that subsumes several existing methodologies including those based on the signed directed graphs is described in section 4.4.

2.2.1.2 Signed directed graph extensions

Yu and Lee (1991) proposed a model-based diagnostic system using signed directed graphs combining quantitative knowledge (steady state gains) and qualitative process knowledge. They use fuzzy logic to calculate truth values of hypotheses. Shiozaki and co-workers (1984) used the signed directed graph to identify causes and the optimal location for sensors to monitor the process conditions.

Order-of-magnitude concepts have been used in diagnostic systems for digital circuits in (Davis, 1984; Hamscher, 1984) which use a hierarchic representation of time to reason about events that occur at different time scales. Mavrovouniotis and Stephanopoulos (1988) proposed a formalization for reasoning with approximate relations for process

engineering activities. Inference strategies use assumption-based truth-maintenance and are based on propagation of order-of-magnitude relations, solved or unsolved algebraic constraints and rules. The system enables qualitative or semi-quantitative reasoning by capturing engineering commonsense about parameter sizes, and by offering a vocabulary to formalize concepts that deal with rough parameter magnitudes.

2.2.1.3 The Model-Integrated Diagnostic Analysis System

MIDAS (Finch and Kramer, 1990) is a system that performs diagnosis of abnormal process conditions. The method determines the root causes of disturbances using a formalism called *event modeling* that integrates knowledge from qualitative causal models and quantitative constraint equation models. The event model feature is the use of transitions between process states, rather than the states themselves.

2.2.1.4 Model-based monitoring of dynamic systems

MIMIC (Dvorak and Kuipers, 1989) involves diagnosing a dynamic system continuously by *tracking* the system with the most appropriate qualitative model. The main techniques used are (1) modeling the physical system with dynamic qualitative and quantitative models, (2) inducing diagnostic knowledge from qualitative simulations, (3) continuously comparing or tracking observations against fault-model predictions, and (4) incrementally creating and testing multiple-fault hypotheses.

Tracking is the process of using the observations to follow a path through the behavior graph of a model. By tracking, the system maintains a set of candidate models. When the diagnostic task identifies a fault, it incorporates the fault in the model. MIMIC performs both tasks in a hypothesize and match cycle that combines associative and model-based reasoning. Observations generate hypotheses which are incorporated into the model.

Using the updated fault models, the behavior is predicted through qualitative simulation. Finally, the predictions and observations are matched keeping track of the system state.

2.2.2 Residual incidence structure

This approach uses quantitative models to produce features which are then qualitatively processed. Two methodologies using model equation residuals are discussed below.

2.2.2.1 The Governing Equations Method

The Governing Equations Method (Kramer, 1987) is based on the association of each quantitative constraint on a process with a set of faults sufficient to cause violation of the constraint. The relations between assumptions and constraints are represented by a two level network. The method uses non-Boolean reasoning to avoid noise effects which become dominant when constraints are near threshold values. The residual assigned to each equation is assumed to be a normally distributed random variable with zero mean. The variance can be determined experimentally, or can be estimated from the variances of the measured quantities in the constraint. In order to smooth the classification function and thus diminish instability, a sigmoidal function of the equations residuals is used and a factor to quantify uncertainty is used.

2.2.2.2 The Diagnostic Model Processor

The Diagnostic Model Processor (Petti and Dhurjati, 1990) also uses model equation residuals. The violation of an equation indicates that at least one of its associated assumptions is invalid. An assumption that is common to many violated equations is strongly suspect, whereas satisfaction of equations provides evidence that associated assumptions are valid. Multiple faults are considered and the sensitivity value is used to weigh model equations as evidence.

2.2.2.3 Parity-space methods

Analytical redundancy can be used as the basis of diagnostic systems if a mathematical model of the chemical process is available. A parity equation is an input-output equation of the plant, rearranged so that ideally it returns a zero value, or residual. However, in the presence of noise, modeling errors or failures, the residual from the parity equation is non-zero. The determination of the presence of a failure is usually based on statistical tests applied to the residuals. One technique to generate residuals relies on parity equations (Chow and Willsky, 1984; Gertler, 1988; Gertler , Bennani, and Phillips, 1989). When residuals are independently tested, special model structures may enhance failure isolation, that is, the ability to distinguish among different failures. Residuals should be orthogonal to failures, and this places a constraint on the structure of the set of parity equations. A detailed procedure to generate a robust parity equation model is described in (Gertler and Luo, 1989).

2.2.3 Causal models

Peng and Reggia (1987) used connectionist methods for diagnosis as part of the Parsimonious Covering Theory. The goal of their work was to derive a formal model to capture causal knowledge to perform inference. They formalize causal and probabilistic associative knowledge in a two level network where associations between disorders and manifestations are represented. One difficulty that arises is the computational complexity involved, especially if multiple disorders may occur simultaneously.

2.3 Heuristic Approaches

Rule-based systems are the most common heuristic approach. If-then rules, also called production rules are used to organize knowledge as empirical associations. Each rule

should represent only one inferential empirical association. One of the earliest expert systems, MYCIN, (Buchanan, and Shortliffe, 1984) performed infectious disease diagnosis and therapy selection. Related methods have been developed which extend the basic representation and methodology. Some rule-based expert systems use certainty factors (not probabilities) to quantify the uncertainty on each diagnosis rule. In the chemical engineering domain CATCRACKER (Ramesh, Davis, and Schwenzer, 1989) uses a hierarchical classification rule-based approach to diagnose faults in a fluid catalytic cracking unit.

2.4 Methods Based on Solved Cases

Classification is a key concept in this approach. The problem of classification consists basically of partitioning the feature space into regions. Each region corresponds to one category. Since it is in general impossible to avoid incorrect decisions, methods attempt to minimize the probability of error, or the average cost of errors. Classification can be viewed as a problem in statistical decision theory.

2.4.1 Pattern recognition approaches

Several pattern classification methods are described in (Duda and Hart, 1973). Some methods use maximum likelihood or Bayesian procedures for estimating statistical parameters from samples. Gaussian classifiers are Bayesian classifiers which assume that the underlying distributions of all inputs are jointly Gaussian.

If distribution of classes is severely non-Gaussian, nonparametric techniques can be used. In this case, a large number of samples is required. The k-nearest neighbor estimator is an example of this approach. Training data is stored and an Euclidean distance metric is used

to determine the k stored training cases that are closest to each new entry. The output of the classifier is the class which occurs most frequently in those k examples.

In supervised learning methods, the correct mapping from symptoms to faults is learned from previously solved (already classified) examples. In contrast, unsupervised learning (clustering) utilizes unsolved cases (Carpenter and Grossberg, 1991).

Lee and Lippmann (1989) review several classification techniques parametric Gaussian, k-nearest neighbor, standard backpropagation, adaptive-stepsize backpropagation, feature-maps, learning vector quantizers, and binary decision trees. Venkatasubramanian and co-workers (1990) use neural networks for steady state processes and include multiple causal origins of the symptoms. A method using radial basis functions (Leonard and Kramer, 1991) uses previous examples to determine the parameters and the topology of the neural network. The method also provides reliability measures for results. A review of several neural net models is presented in (Lippmann, 1987).

2.4.2 Case-based reasoning

Case-based reasoning is a general paradigm whose objective is to reason based on experience. Case-based reasoning can be viewed as a psychological theory of human cognition, and is based on the claim that conceptual memory is episodic in nature. The paradigm requires a memory model for representing, indexing, and organizing past cases, and a process model for retrieving and modifying previous cases and assimilating new ones (Slade, 1991).

Previously classified cases are stored and assigned indices based on key predictive features in the domain of interest. When confronted with a new case, indices are used to retrieve similar past cases from storage. Since more than one case may be retrieved, a similarity

metric is required to decide which case is closer to the current case. When old cases do not perfectly match, they may be modified to fit. After being classified, the new case is incorporated to the knowledge base. An example in diagnosis is PROTOS (Bareiss 1988; Bareiss, Porter, and Wier 1988), a system in the domain of clinical audiology. Other diagnostic systems in the medical domain are CASEY (Koton, 1988), IVY (Hunter, 1989), and MEDIC (Turner, 1988). Diagnosis of machines was done by CBD (Hammond & Hurwitz, 1988).

Chapter 3

Probabilistic Reasoning Under Uncertainty

The type of reasoning involved in diagnosis is called abductive inference. The goal of this kind of reasoning mechanism is to find the causes of a given set of symptoms or observables in the system. In the case of chemical process diagnosis, the measured symptoms are usually noisy, missing or inaccurate. This unavoidable uncertainty is the result of measurement and process noise, and imperfect modeling. Therefore, representing and handling uncertainty is an essential step to diagnose chemical processes.

3.1 Previous Approaches

Approaches to reasoning under uncertainty can be classified in four groups. The first approach is classical probability theory. A second group encompasses several logic related methodologies. A third group is based on quantitative approaches. And a fourth group is made up of non-normative approaches. Several proposed approaches for inference under uncertainty are surveyed by Sheridan (1991) and briefly discussed by Spiegelhalter (1985).

3.1.1 Probability Theory

Probability theory, by itself, provides means to express uncertainty. In addition, it constitutes the basis of some of the present methods to reason under uncertainty.

The earliest developments of the Probability Theory were done in the 17th century. The first general methods for solving probability problems were discussed in a famous correspondence between Pascal and Fermat, which began in 1654, and in a book by Huygens in 1657. Probability calculus was pioneered and developed by gamblers as Cardano (1501-1576) and De Moivre (1667-1754). Rigorous probability theory began with the work of Bernoulli in 1713 and De Moivre in 1730. Further developments of the theory were provided by Laplace, Poisson, and Gauss. Sums of random variables were originally studied by Chebyshev and Markov. Probability theory was finally axiomatized by Kolmogorov in 1933 (Cormen, Leiserson and Rivest, 1990; Pearl, 1988).

3.1.2 Logical approaches

It is a fact that classical logic cannot, in general, express uncertainty or handle conflicting evidence, it assumes that every proposition is either true or false and the combination of propositional variables is restricted to conjunction and disjunction. These limitations have motivated several extensions to logic which attempt to incorporate uncertainty. The simplest extension corresponds to multivalued logic where a new truth value is added. The truth values *uncertain*, *true*, and *false* provide a simple but limited tool to reason under uncertainty. Other extensions are default reasoning, autoepistemic logic, and modal logic.

3.1.2.1 Default reasoning and truth maintenance systems

A reasoning system trying to infer conclusions often needs information which is not available. One way to deal with incompleteness is to assume that propositions with an

unknown truth value take a default value. These assumptions may be withdrawn later if new information reveals they were false. This type of nonmonotonic inference is called default reasoning.

Original work was restricted to only one plausible world or set of statements describing the state of the system under study. This limitation was overcome by De Kleer and Williams (1986). They proposed an assumption-based truth maintenance system which can keep track of multiple conflicting compound hypotheses. The output of this system is usually a partial possible world, it is not a complete description of a world but just a consistent set of statements.

3.1.2.2 Autoepistemic or nonmonotonic logic

Autoepistemic logic (Moore, 1988) is a nonmonotonic logic for modeling the set of beliefs that an ideally rational agent with unlimited resources would hold when reflecting on what it knows and what it does not know. With this methodology it is possible to determine whether, given some data, a rational agent would conclude a given formula. One disadvantage it has is that often the algorithm requires exponential time. A discussion on autoepistemic logic is presented in (Measor, 1991; Obeid and Turner 1991).

3.1.2.3 Circumscription

Given a collection of propositions, there are many things the propositions could be true about. This logic avoids the explicit statement of the rules necessary in default reasoning but obtains similar effects. Circumscription is risky since it is possible that what it infers will contradict the information stored in the knowledge base.

3.1.3 Quantitative approaches to uncertainty

Several approaches which are strongly based on Probability Theory are included in this group. These approaches either generalize the theory or make simplifying assumptions. Another approach which formalizes reasoning with approximate relations for qualitative or semi-quantitative reasoning is included.

3.1.3.1 Probabilistic logic

Probabilistic logic (Nilsson, 1986) is standard probability theory which uses standard mathematics to deduce some statements given a set of known statements. Both the given and the deduced statements are probabilities. The method requires a partition into disjoint and exhaustive possible worlds and a decision procedure to determine whether each of these worlds is consistent. The probability of a statement is the sum of the probability of the worlds in which it is true.

3.1.3.2 Paris's maximum entropy

One of the disadvantages of probability theory is that probabilities are often unknown. The maximum entropy technique (Paris and Vencovská, 1990) ensures that in guessing these probabilities, only the necessary information is assumed, not more. A disadvantage is that computations are, in general, NP-complete.

3.1.3.3 Bundy's incidence calculus

The objective of this method is to obtain a good approximation to the probability of the conjunction or disjunction of two statements. This is accomplished by expressing information about the correlation of statements among themselves. Each statement is associated with a set of points called the statement's incidence, each point corresponds roughly to a possible world. If two statements are independent, their incidences will be

fairly random with respect to each other. An advantage is that the accuracy of the approximation can be improved by increasing the number of points.

3.1.3.4 The Dempster-Shafer theory of evidence

Shafer (1976) proposed an extension to probability theory. Basic probability assignments (bpa) are given to the identified elements of the set of all mutually exclusive and exhaustive possibilities. The difference with probabilities is that these values for the complete set do not have to sum to 1. Another difference is that some of the probability (bpa) may be left unassigned. In addition, belief can be assigned to a collection of statements as a whole. Related work has been published in (Shafer and Pearl, 1990).

3.1.3.5 Bayesian belief networks.

Pearl (1986) proposed a methodology based based on and consistent with probability theory. Belief networks are directed graphs used to represent causal relations and probabilistic dependencies. Two types of reasoning processes can be performed. One represents causal or anticipatory support, it is directed from causes to effects. The other process reflects diagnostic support and is directed from effects to explanations. The belief network paradigm is described in Section 3.2. A case study contrasting belief networks with a hierarchical classification rule-based representation is discussed in section 4.3.

3.1.4 Non-normative approaches

These approaches do not extend an existing theory. Instead they consist of *ad hoc* strategies with pragmatical objectives.

3.1.4.1 Endorsements

This is a general problem solving strategy which uses domain dependent heuristics. Cohen (1986) implemented a framework to use arbitrary heuristic techniques. Most of the work of an implementation lies in the domain-specific heuristics. The obvious disadvantage is that work done in one domain gives little help to work in another.

3.1.4.2 Certainty factors in rule-based systems

Expert systems attempt to capture problem solving abilities as a set of heuristic rules. Rule-based expert systems have been successfully used in many different domains, however, the construction of a diagnostic expert system remains a rather unstructured exercise. One common approach to quantify uncertainty in rule-based expert systems is the incorporation of certainty factors. Certainty factors have been used since the earliest rule-based expert systems, such as MYCIN (Buchanan, and Shortliffe, 1984), a medical diagnosis program. Certainty factors, which are not probabilities, are numbers between -1 and 1 used to quantify the uncertainty associated to evidence and to empirical associations. Certainty factors require a strong assumption which is often false, about the independence of pieces of evidence. Certainty factors represent changes in the level of belief, not a level of belief itself, and can be thought of as ratios of odds. Several ways to combine certainty factors have been used (Micro Data Base Systems, 1985).

In the discussion presented in section 4.3, the CATCRACKER expert system developed by Ramesh, Davis and Schwenzer (1989) is used as an example to show how several limitations of expert system approaches can be overcome by the Bayesian belief network representation. The approach of CATCRACKER and related systems constitutes an improvement over unstructured rule based systems, but nonetheless some of the inherent limitations of expert systems remain. This case study attempts to show how the probabilistic approach contrasts with the expert system approach to fault diagnosis. The

points compared in the methodologies are (1) the selection of rule structure, (2) the expression of fault-symptom relations, (3) the representation of uncertainty, (4) the handling of incomplete evidence, (5) the combination of evidence, and (6) the validation and modification of compiled knowledge. Many questions which arise from the examination of conventional expert systems are addressed by belief networks.

3.1.4.3 Fuzzy logic

The imprecise language that often characterizes expert knowledge motivated the work on fuzzy reasoning (Zadeh, 1983). Within this framework, the extent to which a proposition is true can be quantified. The basic notion of fuzzy logic is the degree of truth of a proposition, which can take any value between 0 and 1. Propositions can be composed, and the degree of truth of a disjunction is the maximum of the degrees of the disjuncts. Conjunctions are analogously interpreted as the minimum in the set.

3.1.4.4 Possibilistic logic

Possibilistic reasoning is precise reasoning on an incompletely described situation. Possibilistic calculus can be seen as a quasi-qualitative calculus in which numbers are only compared, not added or multiplied. The answer to a query in this logic can be *surely true*, *surely false*, *possibly true* or *possibly false*. (Dubois and Prade, 1988).

3.2 Bayesian Belief Networks

Belief networks are introduced in this section, which discusses the graph representation and its inference capabilities. The focus of the representation discussion is on the meaning of the graph structure and the probabilistic parameters.

3.2.1 Graph representation

Bayesian networks consist of a set of propositional variables represented by nodes in a directed acyclic graph. Each variable can assume an arbitrary number of mutually exclusive and exhaustive values. Directed arcs represent the probabilistic relationships between nodes. The absence of a link between two variables indicates independence between them given that the values of their parents are known. The prior probability of each state of a root node is required. For a non-root node, the conditional probability of each possible value, given the states of the parent nodes or direct causes, is needed. Deterministic relations are a particular case which can be handled by having each conditional probability be either a 0 or a 1.

Although several methods for dealing with uncertainty in diagnostic reasoning have been studied (certainty factors, fuzzy logic, Dempster-Shafer theory), probability theory appears to provide the most complete and consistent framework for dealing with uncertain knowledge. (Pearl, 1988; Neapolitan, 1990). While limitations of non-probabilistic approaches have been known for several years, recently methods have emerged for calculating probabilities in probabilistic knowledge bases with an ease approaching that of certainty factor updating in rule-based expert systems.

These advances are based on the representation of knowledge by *Bayesian belief networks,* also called Bayesian networks, causal networks and, when augmented with decision vertices, probabilistic influence diagrams. Influence diagrams were formalized by Howard and Matheson as a tool for decision analysis (1981). In the context of the Dempster-Shafer theory, belief networks are called galleries, qualitative Markov networks or constraint networks (Neapolitan, 1990).

An important distinction must be made between singly and multiply connected networks. A directed acyclic graph is singly connected if there is at most one chain (or undirected path) between each pair of variables. Formally, a directed acyclic graph DAG=(V,E) is singly connected if for every node u in V and v in V, there is at most one chain between u and v (otherwise it is multiply connected), where V is a finite set of vertices or nodes and E is an irreflexive binary relation E on V (called an adjacency relation). And a chain is defined as follows: Let G=(V,E) be a graph (directed or undirected). A sequence of vertices $(v_0,v_1,v_2,...,v_m)$ is a chain of length m in G between v_0 and v_m if either (v_{i-1}, v_i) or (v_i, v_{i-1}) are elements of E for i=1,2,...,m. Often in chemical processes effects could be the result of several causes, each of which might cause more than one effect, rendering networks multiply connected (with undirected cycles). Some algorithms work only on singly connected networks, consequently additional work is necessary to render them singly connected. Two well-known techniques are cutset conditioning (Pearl, 1986) and clustering (Lauritzen, 1988). Conditioning includes identifying the loops and selecting the minimal set of nodes whose instantiation eliminates cycles, a problem not unlike finding tearing streams in chemical process flowsheets. Clustering involves the aggregation of several nodes into a single node whose possible states are combinations of the states of the individual nodes. In addition, methods for Gaussian continuous variables have been proposed (Shachter, 1988). The global behavior of a feedback loop can be described by representing it as a node in the network. Calculation algorithms for directed cycles do not yet exist but are being developed (Wen, 1989).

Abductive inference is any reasoning process which derives the best explanations for a given set of evidence (problem features, symptoms, or observed facts). The task can be conceptually divided in three steps: 1) generating plausible hypotheses 2) ranking them and 3) obtaining their probability. A set of exact algorithms for abductive inference is based on parallel distributed calculations or local message passing that update the

probability of the state of each node in the best explanation. When the value of a node is known or when receiving a message, other messages are sent from the node to all its neighbors. This type of evidence can be incorporated at the beginning of the analysis or later during the inference process. Messages are sent in two directions, the causal or predictive direction and the diagnostic or evidential direction. An explanation (hypothesis or system description) is a set containing all unknown variables of interest and an associated value for each. When the explanation set contains more than one variable, the values in the set must be consistent among themselves; independently choosing the *most probable state for each variable* might include some contradictory conclusions and is not equivalent to the *most probable set* of values. Due to the presence of uncertainty, explanations are not unique, and it is necessary to know which one is most likely to be true. One explanation is quantitatively better than another if its probability (conditional on evidence) is greater. Since the number of possible explanations grows exponentially with the number of variables, determining the n-th most probable explanation using the brute force approach of enumerating all possible cases, computing their probabilities, and then ranking them, is usually not feasible. Some inference processes capable of doing it more efficiently use distributed network (parallel) computations on belief networks, often simultaneously performing the first two conceptual steps mentioned. Other algorithms rely on a parallel search in the space of possible solutions and find a set of high probability diagnostic hypotheses (Rojas-Guzmán and Kramer, 1993b). It is possible, in general, to determine the two most probable explanations (Pearl, 1988), and for simpler problems, to find the n best (Cooper, 1984). Once an explanation is found, regardless of the method used to find it, it is of interest to determine its probability. The Nested Dissection Method by Cooper (1990) is an efficient way of performing this calculation.

3.2.2 Probabilistic parameters and their sources

Uncertainty in the causal relations is represented by conditional probabilities if the node has parents, or prior probabilities if it is a root node. Sometimes the probability parameters of the Bayesian network model can be determined by calculating the relative frequency of its variables states, but more often they are subjective estimates. Alternatively, frequencies can be determined from process historical data or by performing Monte Carlo simulations if a mathematical model is available. Probabilities need not be drawn from the same source.

To completely quantify the probabilistic model it is necessary to encode all possible system states and the probability of each state. Each system state includes a value for each observed and for each unobserved variable. The set of all possible states and their probabilities is called the joint probability distribution of the system. The prior and conditional probabilities in the belief network constitute an equivalent and more efficient representation than the joint probability distribution of all propositional variables. Far fewer numbers have to be given by the expert, in preparing the Bayesian network model.

Clearly one of the "drawbacks" of belief networks is the requirement for probability values to parameterize the network. Failure frequencies are one source of objective values for probabilities. However, large databases are usually not available, mainly due to the infrequent occurrence of failures. Combining failure frequencies from different processes to obtain probability estimates is not always valid since probabilities are sensitive to the context and conditions under which the failures occurred. Probability estimates constitute a valuable source of information. In the following sections, we discuss the use of subjective probability estimates in belief networks. More information is given by Neapolitan (1990).

Among the approaches to probability theory, three main positions have been clearly established. First, there is the classical approach which views probability as a ratio of equipossible events such as the roll of a die. In this approach, probability is defined as a ratio. For each event in a subset of a sample space there corresponds a real number P(E), called the probability of E. This number is obtained by dividing the number of equipossible alternatives favorable to E by the total number of equipossible alternatives. Second, there is the limiting frequency approach which considers probability an objective representation of the frequency of an event. This approach requires repeatable experiments to obtain a value whose uncertainty depends on the amount of data available. The frequentist concept of probability is applicable when there is sufficient reason to believe that the relative frequency of the observed attribute would tend to a fixed limit if the observations were indefinitely continued. Third is the subjective approach which is based on the belief in the occurrence of a particular event. The accuracy of the latter approach depends on the quality of a judgment. The subjective approach is the only one that applies to unique events which can not be repeated and whose outcome has a significant impact. Given that failures are infrequent events, the subjective approach is required in building belief networks for process diagnosis.

The subjectivist approach has been developed largely, from the work of de Finetti (1964), Savage (1954), and Lindley (1979) and holds that "...probability statements may be made regarding any potentially verifiable proposition; whether a chance mechanism can be imagined or not. The only constraints " ... "are that the probability statements should cohere, in the sense that if, say, statements p(a), p(b), p(b|a) and p(a|b) are made, then the equation p(a)p(b|a)=p(a|b)p(b) should hold." (Spiegelhalter, 1985) The numerical value attached to the uncertainty is only relative to each particular situation and must be obtained from a human's careful analysis of its likelihood. From this perspective, uncertainty only represents a belief relative to the particular situation at hand as described

by Pearl (1988). From a pragmatic point of view probability theory can be used as the basis for decisions and actions when uncertainty exists. De Finetti defines the probability P(E) as the fraction of a whole unit value which one would feel is the fair amount to exchange for the promise that one would receive a whole unit value if E turns out to be true, and zero units if E turns out to be false (De Finetti, 1972).

In the worst case, neither frequencies based on past data nor estimates based on knowledge exist. Nevertheless, decisions must be made. Within the framework of the classical approach to probability, probabilities are assigned by using the Principle of Indifference (Keynes, 1948). The idea is that "alternatives are always to be judged equiprobable if we have no reason to expect or prefer one over the other." (Weatherford, 1982). The principle of indifference was extended by Jaynes (1979) who calls this rule for assigning probabilities the *principle of maximum entropy.* It says that "the least presumptuous assignment of probabilities is the one which comes as close as possible to distributing the probabilities equitably while satisfying the information."(Neapolitan, 1990). If entropy is defined as $H=\Sigma p_i \ln(p_i)$, the sum is minimized when some $p_i=1$ and maximized when $p_i=1/n$ for all i. The Maximum Entropy Technique is a way to overcome one of probability theory's practical disadvantages: there are often probabilities we do not know (Paris, 1989). The Maximum Entropy Principle ensures that in guessing these, we are not assuming any more information than we must (Sheridan, 1991). According to Jaynes, this principle achieves *objectivity* by basing predictions only on the information that is in fact available (Jaynes, 1979).

While some researchers are mainly concerned about the understanding of human reasoning processes, others focus their efforts on developing practically useful techniques and methodologies, which are evaluated based on their performance. In the words of Spiegelhalter, a subjectivist Bayesian view of uncertainty, if flexibly applied, can provide

many of the features demanded by expert systems (Spiegelhalter, 1985). Even without precise probabilities, reasoning conclusions are qualitatively correct, since the network structure by itself encodes a significant part of the inference model. Ben-Bassat has said: "Bayesian methods tolerate large deviations in prior and conditional probabilities. Even rough estimates for which qualitative expressions such as *rare*, *frequent* and *probable* are used may be accurate enough to result in the recommendation of the correct decision" (Ben-Bassat, 1980). It has been shown that probability calculus gives correct qualitative behaviors even with uncertain probabilities (Pearl, 1988). The main advantage of Bayesian networks is that *once probability values are chosen, their combination in the belief network is rigorous and no additional uncertainties are introduced by inadequate structuring of rules, certainty factor calculi, or ad hoc handling of missing evidence.* Variables and their states are often chosen to reflect those found in behavioral descriptions. The number of states for each variable depends on the level of detail available. Both deterministic (probabilities of 1 or 0) and probabilistic relations can be represented.

Another potential source of probabilities is Monte Carlo simulations using conventional mathematical models. Given mathematical models including the effects of failures and failure probabilities, this method can determine the probabilities of arbitrarily defined states. For example, simulation can determine the conditional probability of a variable X whose reading, Y, directly depends on the true value of X, a sensor bias (miscalibration) and random measurement noise. The part of the Bayesian network corresponding to these four variables is shown in Fig. 3-1.

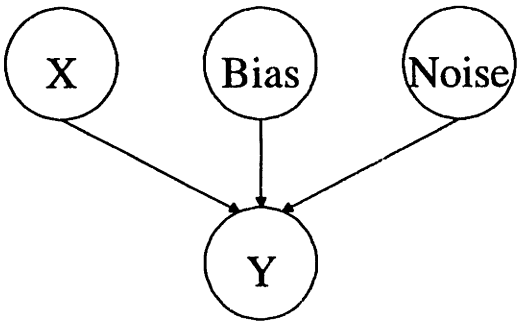


Figure 3-1. Mathematically modeled sensor to illustrate Monte Carlo simulation

The sensor would typically be part of a larger system, however, thanks to the locality of the relations, the simulation requires only the probability distributions of the parent variables and the mathematical model which relates them. In this case the model is the algebraic relation $Y = X + Bias + Noise$. Under the assumption that both X and Bias are uniformly distributed and Noise is normally distributed with a zero mean, the discrete states each variable can assume and their probability distributions are shown in Table 3-1, and a sample of the obtained conditional probabilities is shown in Table 3-2.

Table 3-1. Discrete states and probability distributions for a sensor model

Y	X	Bias	Noise
Sat. low (<1.0)	Low (-0.8,-0.2)	Neglibible (<0.05)	Normal N(0,0.02)
Low (-1.0, -0.5)	Med (-0.2, 0.2)	Small (0.05, 0.1)	Excessive N(0,0.1)
Med (-0.5, 0.5)	High (0.2, 0.8)	Large (>0.1)	
High (0.5, 1.0)			
Sat. high (>1.0)			

Table 3-2. Conditional probabilities obtained from Monte Carlo simulation

X	Bias	Noise	Y				
			Sat low	Low	Med	High	Sat high
Low	Negligible	Normal	0.0	0.489	0.511	0.0	0.0
Low	Negligible	Excessive	0.0	0.495	0.505	0.0	0.0
Low	Small	Normal	0.0	0.412	0.588	0.0	0.0
Low	Small	Excessive	0.006	0.407	0.587	0.0	0.0
Low	Large	Normal	0.293	0.179	0.052	0.262	0.214

3.2.3 Inference features of belief networks

The belief network is very well suited to describe the variability between faults and symptoms. All causal relations are included and their strength quantified using conditional probabilities. When diagnosis is performed, all expected symptoms of a fault need not be present to diagnose the fault. Each symptom contributes confirming or disconfirming evidence according to the laws of probability. The fact that a variable may or may not be observed is handled transparently in the belief network. When a variable is unknown, the inference process will provide the most probable value that is globally consistent with the available evidence.

Details of the inference features on belief networks are given in Pearl (1988) and Neapolitan (1990). In Pearl's algorithm the nodes in the network perform calculations asynchronously and independently in response to messages received from other nodes. The use of distributed algorithms with asynchronous processing enables incremental addition of new evidence as it becomes available. New information updates the conclusions by locally propagating through the network without the need for repeating all

the calculations. As a result, hypotheses explaining the same symptoms automatically compete in the belief network.

In some cases it is reasonable to assume that at most one fault can occur at a time (or that when several faults occur there is no interaction between them). When the single fault assumption is valid, it is sufficient to independently calculate the most probable value for each variable. However, a more realistic analysis should consider the possibility of multiple interacting simultaneous faults. To diagnose such a system it is necessary to find the most probable consistent set of values which explains the observed symptoms and the complexity of the calculations increases. The most probable set of values may be different from the set containing the most probable value for each individual variable. Just assembling individual analyses may yield a suboptimal or incorrect diagnosis.

When a symptom is explained by more than one cause, evidence indicating the presence one cause should decrease the belief that the other cause is present, an effect called "explaining away" by Pearl (1988). Explaining away is captured by probability theory. A related problem involves handling the context in which evidence is obtained.

Chapter 4

A Probabilistic Framework for Model-based Diagnosis

Two major difficulties in developing probabilistic knowledge representations of chemical processes exist. First, the representational problem, how to structure and encode knowledge which exists in the form of mathematical models, data, and human beliefs into a coherent, integrated probabilistic form, and second, the inferential problem, the issue of formally generating, evaluating and selecting hypotheses about the state of the system.

4.1 Elements of the Methodology

The probabilistic methodology for discrete-state estimation in dynamic systems proposed in this project is based on the Bayesian belief network as the unifying representation. The proposed framework uses a discrete multi-variable space to represent chemical processes. Let S be a Probability space = (Ω, F, P) as defined by Kolmogorov (1950), where,

(1) Ω is a set of sample points in the space. The sample space is a mutually exclusive, collectively exhaustive listing of all possible outcomes of a model of a non-deterministic process. A discrete-state description of a chemical process has a finite

but extremely large amount of possible outcomes. The size of the sample space grows exponentially with the number of variables and the number of discrete states per variable.

(2) F is a set of events relative to Ω,

(3) $\{E_1,E_2,...,E_n\}$ is a set of mutually exclusive and exhaustive events $P(E_i,E_j)=0 \ \forall \ i,j$

(4) P is a function which assigns a unique real number to each E in F. P satisfies the following three conditions:

$$P(E) \geq 0 \forall E \in F \text{ where } P(E) \in R$$

$$P(\Omega) = 1$$

if E_1 and E_2 are disjoint subsets of F then $P(E_1 \vee E_2) = P(E_1) + P(E_2)$

A finite propositional variable A on the probability space is a function from Ω to a subset A of F containing mutually exclusive and exhaustive events. $A = \{a_1,a_2,a_3,...,a_n\}$ where A can have n possible values. Propositional variables can represent observable or non-observable events. In this framework the belief network representation is used as a starting point and extended to handle temporal reasoning.

Unmeasured variables can be included in the belief network. Even though some variables are not measured, their most probable values, consistent with the rest of the evidence can be estimated from the calculations in the belief network.

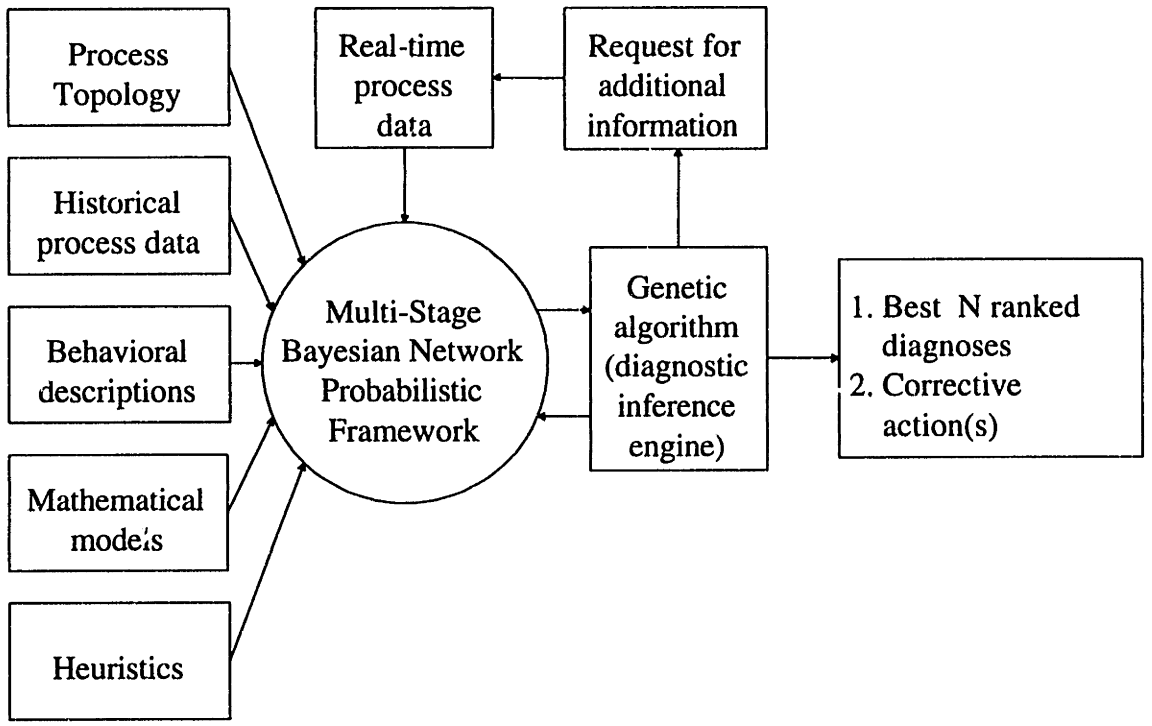


Figure 4-1. Elements of the diagnostic methodology

4.1.1 Knowledge representation

When trying to use a probabilistic framework to represent chemical processes and perform diagnosis, several difficulties arise. One problem which is addressed in this project is that not all the variables in a chemical process are measurable and even those which usually are may not always be available. In addition, part of the sensor data is often biased or incorrect. The intended use of the Bayesian belief network representation is to capture this uncertainty and the probabilistic relations that quantify uncertainty in a well defined framework in which abductive inference can be performed in a form consistent with probability theory. The first step in the project involved the evaluation of Bayesian belief networks as the unifying representation for a probabilistic framework.

A second problem is that some of the variables which describe a chemical process are discrete, some are continuous and both types must be handled simultaneously. One option is to discretize continuous variables and to find criteria, that do not yet exist, for optimal selection of states and thresholds. Alternatively, inference algorithms can be used to handle continuous variables. If the process can be represented by a linear quadratic Gaussian model then influence diagrams (Shachter and Kenley, 1989) can be used with continuous Gaussian probability distributions.

A third problem is that belief networks must be extended to handle temporal reasoning, an issue which deserves individual attention and which will be discussed below.

The Bayesian belief network representation keeps the domain specific knowledge (variables, their interrelations and the probabilistic parameters) independent from the inference mechanisms (parallel message passing algorithms, stochastic simulation algorithms, or evolutionary programming algorithms). The use of Bayesian networks makes possible the incremental construction of models and simplifies model modifications by keeping all interactions local in the network. The Bayesian network constitutes a natural, transparent and efficient representation of the problem to be solved.

Once the probabilistic model is built, it is expected to perform adequately in situations which have not yet been observed. The fact that the model encodes an understanding of the process and contains local causal relations constitutes an advantage over purely statistical methodologies.

4.1.2 Knowledge integration

The second objective is to demonstrate how to efficiently encode and integrate qualitative and quantitative knowledge from different sources within the same framework.

The knowledge related to a chemical process can be obtained from different sources including present sensor data, past plant history data, human estimates of expected behavior, functional process descriptions, and formal models such as systems of equations. Implementing and testing the proposed methodology in a real plant environment would help to evaluate its advantages and limitations. The case study will also serve to illustrate the acquisition and integration processes.

The identification of the relevant variables and their causal links can be obtained from the understanding of the process operation that operators, engineers, and designers have and from the basic concepts of chemical engineering. Objective frequencies of failures and records of the probability distribution of the values an identified variable can take can be used to determine and update the prior and conditional probabilities required in the model. The use of recursive filtering techniques facilitates the use of previous measurement information.

4.1.3 Knowledge acquisition

The third objective is to develop a reliable formal methodology to acquire and translate information describing a chemical process and its behavior into the chosen representation. Both the topology and the parameters should be determined by a formalized methodology. Formalizing this tasks has several advantages which include reproducibility, consistency, and the potential of being partially automated.

This objective will be achieved by recording and analyzing the activities performed while implementing the proposed diagnostic system in an industrial setting. One difficult part in doing this is the human subjective interaction. Not only additional evidence should be easily incorporated while diagnosing the process operation, but also changes in network

topology. This task is simplified thanks to the modular and local properties of belief networks.

Work has been done on building networks from databases (Cooper, 1991). Possible applications of this methodology might be useful and their benefits should be considered.

4.1.4 Dynamic behavior representation

The fourth objective is to develop a methodology to handle both deterministic and probabilistic dynamic behavior in time. A fault in a chemical process introduces transient events which often invalidate the steady state assumption. This will be accomplished by extending the Bayesian network framework to represent and reason about temporal relations. One difficulty is the existence of transient events which occur at different time scales.

4.1.5 Diagnosis based on probabilistic reasoning

The fifth objective is to find adequate methods to perform abductive inference for the specific class of problems to be addressed. Exact and approximate methods, and combinations of them and the topology arising from the representation of chemical processes is studied.

One major challenge is to solve what in general is an NP hard problem in real-time. It is of interest to study and quantify the relation between sparseness in an incidence matrix and degree of connectivity and the consequent inference complexity.

Some approaches to diagnosis are based on two distinct steps, hypotheses generation and the subsequent testing to determine which one is the best. Abductive inference in Bayesian networks can be thought of as a simultaneous generation and testing of

hypotheses. It is not necessary to systematically search among all possibilities. Abductive inference algorithms can be extended to generate suboptimal hypotheses if necessary. Not only unknown variables can be instantiated (new evidence) at any time but also known variables can be re-instantiated (revised evidence) if they are found to be in a different state at a later time.

The hierarchical organization of network structures will be explored since it may help to diagnose multiple faults. This idea is based on the assumption that sections of a system will be connected within them but sparsely connected with other sections. When this is the case, a hierarchical structure based on topological considerations can be used. Another possible use of hierarchical structuring is the diagnosis of faults at different levels of detail.

Conditioning is not an easy problem, since the optimal selection of a loop cutset is NP hard. One option is to use heuristic algorithms which work in polynomial time (Suermondt and Cooper, 1990;1988). When conditioning, loop cutset nodes can be sequentially instantiated to improve efficiency. Once an explanation is found, its probability can be calculated efficiently using the Nested Dissection method by Cooper (1990). Monte Carlo simulation methods to generate conditional probabilities from deterministic models will be implemented and evaluated.

Implementation of abductive inference algorithms to explore and test proposed ideas is being done in an object-oriented language. C++ is very well suited to enhance modularity. Portability is desirable for the testing stage of the project.

The selection of information and interaction between the computer system and the human user are part of the practical industrial application of the proposed methodology and are discussed in Chapter 6.

4.2 Initial Evaluation Criteria

The industrial case study will help to determine if the following objectives were accomplished. Specifically, this project will be considered successful if the following goals are achieved.

(1) The probabilistic framework system should be able to represent both discrete and continuous variables and to combine qualitative and quantitative knowledge from different sources.

(2) The diagnostic conclusions should be accurate.

(3) The construction (knowledge acquisition) and maintenance (knowledge base modification) should be easier than with conventional expert system approaches.

(4) The methodology should overcome the limitations of rule-based expert systems and hierarchical classification systems.

(5) The diagnostic system should be able to perform temporal reasoning in dynamic chemical processes, to diagnose multiple faults in real time as an on-line system.

4.3 Probabilistic Modeling of Chemical Processes

Chemical process diagnosis requires a framework that can represent and process uncertain knowledge. Several factors give rise to this requirement. Deterministic models for the effects of faults cannot always be constructed, and measurements to diagnose every fault

uniquely may be missing, noisy or inaccurate. A single *fault* may represent a class of closely related events or processes whose observable effects are similar, but not identical. Thus the associations between symptom patterns and malfunctions may not be one to one. Faults may occur with unequal prior probabilities. Furthermore there may be uncertainties in diagnostic heuristics provided by experts. This section deals with the use of probability theory to represent the uncertain elements of the diagnosis problem, and the use of abductive inference computations to determine the most probable diagnostic hypotheses given such representations.

This example attempts to show how the probabilistic approach contrasts with the expert system approach to fault diagnosis. An expert system attempts to capture the problem solving abilities of experts as a set of heuristic rules. Rule-based expert systems have been successfully used in many domains, however, the construction of a diagnostic expert system remains a rather unstructured exercise. The probabilistic approach not only contributes to improved uncertainty handling, but also imposes a logical order on the construction of knowledge bases.

In this section, we use the example of the CATCRACKER expert system developed by Ramesh, Davis and Schwenzer (1989). CATCRACKER treats diagnosis as a hierarchical classification problem, which provides additional structure to the knowledge base. Several large systems have been built using this approach. The approach of CATCRACKER and related systems constitutes an improvement over unstructured rule based systems, but nonetheless some of the inherent limitations of expert systems remain.

4.3.1 A fluid catalytic cracker rule-based expert system

CATCRACKER is a rule-based expert system that diagnoses a fluid catalytic cracker (FCC) unit, made up of the reactor-regenerator and three ancillary systems, feed, catalyst and separation systems.

The FCC unit performs cracking through a catalytic mechanism to produce olefins. Olefins are hydrocarbon molecules with double bonds between carbon atoms which are used for motor fuels and as intermediates in chemical industrial synthesis. Due to their reactivity, olefins are present only in small amounts in the crude oil and natural gas. After extraction of oil and natural gas, oil is distilled at atmospheric pressure to separate fractions of different molecular weight based on the differences in their boiling temperatures. The main fractions are refinery gas, liquid gas, crude naphtha, kerosene, gas oil, heavy fuel oil, and a heavy mixture. The gas oil fraction undergoes cracking to produce ethylene and propene (via catalytic, hydrocatalytic or thermal cracking).

Catalytic cracking converts higher boiling distillation fractions into saturated branched paraffins, cyclohexane, and aromatics. Common acid cracking catalysts are zeolites (crystalline aluminum silicates). In the process, the catalyst must be regenerated continuously by burning off the coke layer which results from coke deposition. If coke were not removed catalyst activity would be hindered. Usually the process operates at about 450-500 degrees Celsius and a slight excess pressure of a few bar (Weissermel, 1978). In the regenerator, carbon and tars burn in the presence of oxygen in an exothermic reaction (releasing energy). In the reactor the heavy petroleum fraction undergoes endothermic cracking (absorbing energy). The high rate of solid catalyst circulation between the regenerator and the reactor maintains a heat balance and keeps the catalyst in a high-activity state.

Hydrocarbon cracking is sensitive to temperature, residence time and partial pressure of hydrocarbons. The addition of steam decreases the partial pressure of the hydrocarbon and consequently improves the olefin yield. Residence times must be short since hydrocarbons are thermodynamically unstable at cracking temperature. Operating conditions must be carefully monitored and any deviation from normal conditions must be promptly and accurately diagnosed to avoid uncontrolled reactions due to the presence of oxygen (in the regenerator), hydrogen (a reaction product), hydrocarbons (from the feed), high temperature and large heat exchange within the unit.

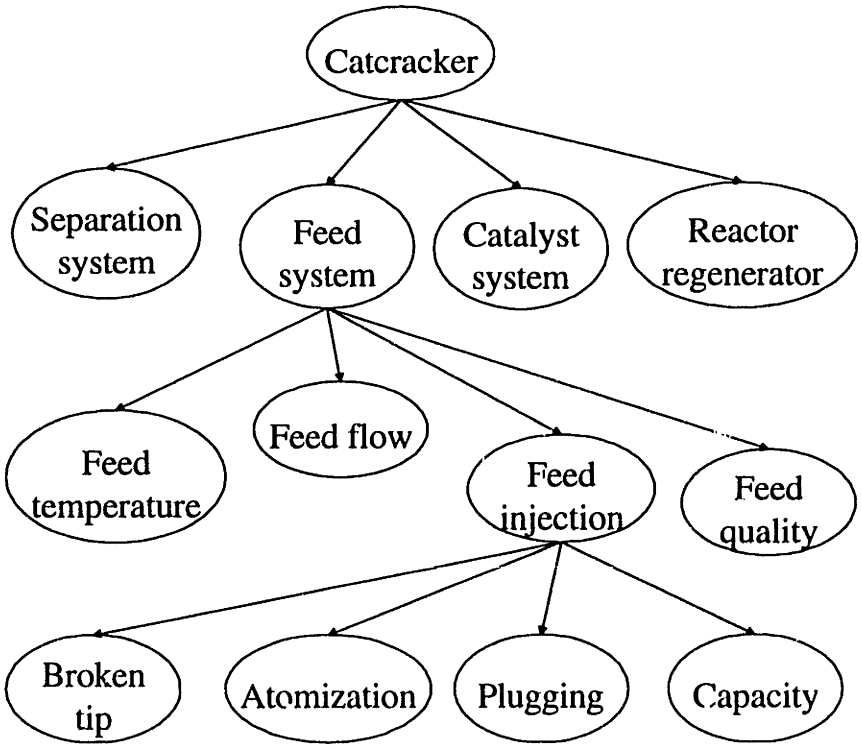


Figure 4-2. FCC process hierarchy.

The process hierarchy, with details on the section corresponding to the feed system, is shown in Figure 4-2. These classification hierarchies are used in the CATCRACKER system to organize knowledge as malfunction hypotheses. Each node of the hierarchy

contains rules to confirm or reject the hypothesis that the system represented by the node is functioning normally. Diagnosis in the CATCRACKER system proceeds from the top of the hierarchy (the most general level) downward to the level of most detail.

Rules at each node are developed in several steps from functional descriptions of the FCC unit. Two examples of a *causal description* of a section of the feed system of the FCC unit follow:

1. If the injector is plugged, the atomization of the feed/steam mixture deteriorates. With poorer atomization, the mass transfer area reduces, affecting the kinetics of the cracking reaction, and favoring undesirable side reactions that produce more coke and hydrogen.
2. The increased coke make affects the heat balance in the reactor. The cracking reaction is endothermic, hence greater coke production means that more heat is needed for the reaction. The reactor temperature control tries to compensate by increasing the catalyst circulation rate (since regeneration is exothermic, the regenerated catalyst provides the heat for the cracking reaction). An increased catalyst circulation rate increases the rate of heat removal from the regenerator, thus causing the temperature to drop.

Also included in the basic knowledge is a *behavioral description* of the effects of faults in terms of process observables. The following is the behavioral description for plugged injectors:

1. If the plug is severe then header pressure may increase enough to activate a trip which causes the feed to be dumped.

2. For less severe plugs, the header pressure will increase but not necessarily enough to cause a feed dump. The header pressure is monitored on a strip chart but is not alarmed. If the increase is not enough to cause a trip it may or may not be observed by the operator.
3. In the event of a plug, although the header pressure may not increase enough to reach the feed switch limits, there will likely be a strong effect on the FCC operation and product quality.
4. The increased hydrogen production would be observed as an increase in load on wet gas compressors.
5. The increased coke production would be seen as a drop in regenerator temperature if operating at maximum air capacity or an increase in air rate (blower speed) to maintain regenerator temperature.

Diagnostic rules for each node in the hierarchy are generated from the causal and behavioral descriptions. Given a feed system problem, Ramesh, Davis and Schwenzer (1989) suggest the following rules to determine the cause (temperature, flow, injection or feed quality):

1. If the feed temperature alarm is not on and the temperature measurement is normal then there is no evidence of problem in the feed temperature control.
2. If the feed flow alarm is not on, the flow measurement is normal, and no gross deviation exists in the material balance closure, then there is no evidence of a feed flow problem.
3. If the injector pressure is high, then there is strong evidence of either a feed injection problem or the presence of light hydrocarbons and water contaminating the feed.

4. If the material balance closure is within 1%, the header pressure has increased steadily, the yield distribution has not changed and the feed flow and feed preheat pressure have remained steady, then there is evidence of an injection problem and not a contamination problem.

When there is evidence of feed injection system malfunction, this subsystem is considered in more detail, by evaluating the following rules:

5. If there is an increase in the header pressure, then there cannot be a broken injector tip.
6. If the atomization flowrate or steam pressure are out of range then there is strong evidence of an atomization steam problem.
7. If the atomization flowrate is normal and the steam pressure is normal then there is inconclusive evidence that there is no problem with the atomization flowrate.
8. If conversion is not steady, the sour water make has increased, the CO make has increased and the vapor traffic in the column has increased, then there is evidence of atomization flowrate problems.
9. If there is no evidence of other problems and switching to standby injectors corrects the observed behavior, then conclude plugging in injectors.

4.3.2 Examination of the rule-based expert system approach

In this section we raise a number of questions regarding the methodology for developing rules within the CATCRACKER hierarchy. These questions refer to the construction of rules on one level of the hierarchy, not to the hierarchy itself. The rules at each level of the hierarchy are small, self-contained expert systems that use patterns of observable symptoms to identify which subsystems are likely sources of the observed abnormalities. Examining rule construction in a single hierarchy level reveals certain problems that are

also present in engineering more complex expert systems. The questions raised here could also apply to other expert systems besides CATCRACKER.

4.3.2.1 Selection of rule structure

A formal description of the process of producing rules from the behavioral description is desirable to ensure consistency and reproducibility. But exactly how is this translation achieved? For example, consider the selection of observables in rule antecedents. Could *atomization problem* occur without the presence of the four symptoms in Rule 8? What motivates the process of elimination used in Rule 9? Should other symptoms of broken injector tip be included in Rule 5? If increased CO acts as confirming evidence for atomization flow problems in Rule 8, should it also appear as negative evidence for other faults? Should rules act competitively (with positive evidence for one fault acting as negative evidence for another) as in Rule 4, or non-competitively, as in Rule 6?

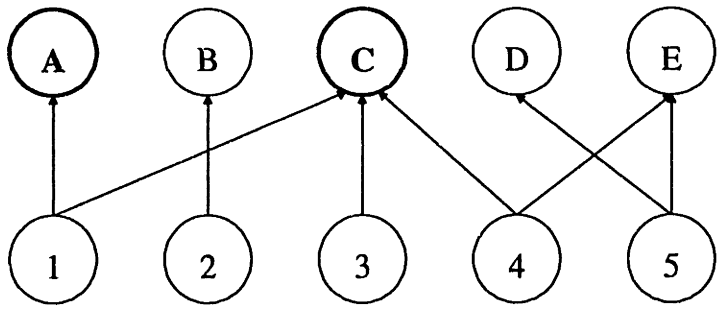


Figure 4-3. Hypothetical fault-symptom relations.

The basic problem in selecting rule structure can be illustrated by the hypothetical scenario shown in Figure 4-3 which abstracts relations between symptoms and faults. Given that symptoms A and C are observed, should only abnormal symptoms be included as in "If A and C then Fault 1", or both normal and abnormal symptoms as in "If A and C and not E then Fault 1"? Should rules in general refer to other hypotheses, as in "If not Fault 4 and

A then Fault 1"? When competitive hypotheses exist, should evidence in favor of one provide evidence against others, as in "If A and C then Fault 1 and not Fault 3"? In the rules of the CATCRACKER system, there is no consistent approach to selecting the observables that constitute the rule antecedents, and a mixture of different structures is used.

4.3.2.2 Expression of fault-symptom relations

A given fault does not always cause the same symptoms due to the operational context at the time of the fault, or due to variations in the intensity of the fault. One way to account for symptomatic variability is to include rules covering all possible outcomes of each possible fault. This requires many patterns to be enumerated in rules, and yet the system is unable to respond to patterns that are not explicitly enumerated in the rules. For example, a partial match of 3 out of 4 of the symptoms in Rule 8 would result in no contribution to evidence for atomization problems. Is this the desired effect, or should a partial match contribute partial evidence? A rule base without adequate coverage of partial symptom patterns may exhibit "brittleness", failing to diagnose a problem whose symptoms are a close but imperfect match to the symptom patterns in specific rules.

4.3.2.3 Representation of uncertainty

Rules in CATCRACKER use certain qualifiers such as *no evidence*, *strong evidence*, or *absolute certainty*, expressing the certainty of the hypothesis given observed symptoms. In CATCRACKER, these modifiers come from process experts. But given the use of uncertainty qualifiers in the behavioral and causal descriptions (e.g., for severe plugs "the header pressure *may increase enough* to activate a trip...") could the qualifiers in the rules be determined systematically from the behavioral or causal descriptions? In the rule-based system, there is no systematic way to derive qualifiers in diagnostic rules from qualifiers in behavioral descriptions.

4.3.2.4 Handling incomplete evidence

Symptomatic variables available for diagnosis are not always the same because sensors may be out of service, and tests may or may not be performed. For example, the second behavioral description states that "if the increase in header pressure is not enough to cause a trip it *may or may not be observed* by the operator". How should header pressure be treated in the rules if it may or may not have a value? Once rules are created, there is a commitment to which should be the measured or observed variables. Although most expert systems can accept *unknown* as a legitimate value, on what basis can we feel confident in the conclusions of the system if certain variables are not observed? Postulating new rules to deal explicitly with missing evidence develops into a combinatorial problem when multiple missing symptoms are considered.

In CATCRACKER, when missing evidence becomes available, previous results are cleared from the system and the analysis begins anew. Ideally, when a value changes from *unknown* to a known value, inference should not have to begin again from scratch. Is there a more efficient updating scheme that can add new information incrementally when it becomes available?

4.3.2.5 Combination of evidence

In CATCRACKER, a hypothesis is concluded by at most one rule. But in general, the likelihood of a hypothesis is based on several pieces of evidence and partial conclusions which must be combined. When an event is contained in the consequent of more than one rule, intuition tends to increase the strength of evidence. The certainty factor calculus typically used in expert systems combines partial results in an arbitrary way. In addition, if there is a partial match of symptoms to a rule antecedent, what type of matching and certainty factor accounting should be used to contribute evidence to the rule consequent?

4.3.2.6 Validation and modification of compiled knowledge

When the process changes or new behaviors are observed, the knowledge base must be updated and validated. The modularity introduced by the hierarchical approach simplifies this task, however, how this can be done in general is not clear. Modifications would be more easily incorporated at the level of the behavioral description than at the level of diagnostic rules. The behavioral effects of a modification should be more readily apparent than the effect of a change on the rules.

4.3.3 Advantages of the Bayesian network modeling approach

Many of the foregoing questions are addressed by Bayesian networks. The Bayesian network framework is described in Section 3.2. The description includes the semantics of the graph representation, a discussion on model construction, sources of the probabilistic parameters, and inference features of the representation.

The Bayesian network framework is capable of modeling uncertain systems by encoding probabilistic relations between variables that often represent causal mechanisms, enabling the framework to diagnose scenarios which have not been observed before. This approach is flexible, it has the capability to perform inference based on any subset of measured variables. When the hypothesis contains more than one variable, the values in the set must be consistent among themselves; independently choosing the *most probable state for each variable* might include some contradictory conclusions and is not equivalent to the *most probable set* of values. An additional advantage of using Bayesian network is that both exact and approximate algorithms exist to perform abductive inference to find global diagnostic hypotheses.

Sometimes the probability parameters of the belief network model can be determined by calculating the relative frequency of its variables states, but more often they are subjective estimates. Alternatively, frequencies can be determined from process historical data or by performing Monte Carlo simulations if a mathematical model is available. Probabilities need not be drawn from the same source. The main advantage of belief networks is that *once probability values are chosen, their combination in the belief network is rigorous and no additional uncertainties are introduced by inadequate structuring of rules, certainty factor calculi, or* ad hoc *handling of missing evidence.*

4.3.4 Bayesian network representation of a FCC unit

In the following sections, we will illustrate the process of constructing a belief network for the fluid catalytic cracker (FCC) unit. The first step is to identify the variables of interest, which become the nodes in the belief network. The variables chosen are the ones mentioned in the behavioral and causal descriptions. The nodes are not limited to measured or observed variables. For each node, a small number of discrete states is used to construct the set of mutually exclusive and exhaustive values that a variable can assume. For example, the node *plugging* can assume the states: *none*, *moderate* and *severe*. Only those states mentioned in the behavioral description for each variable are included, reflecting the degree of precision of the original description.

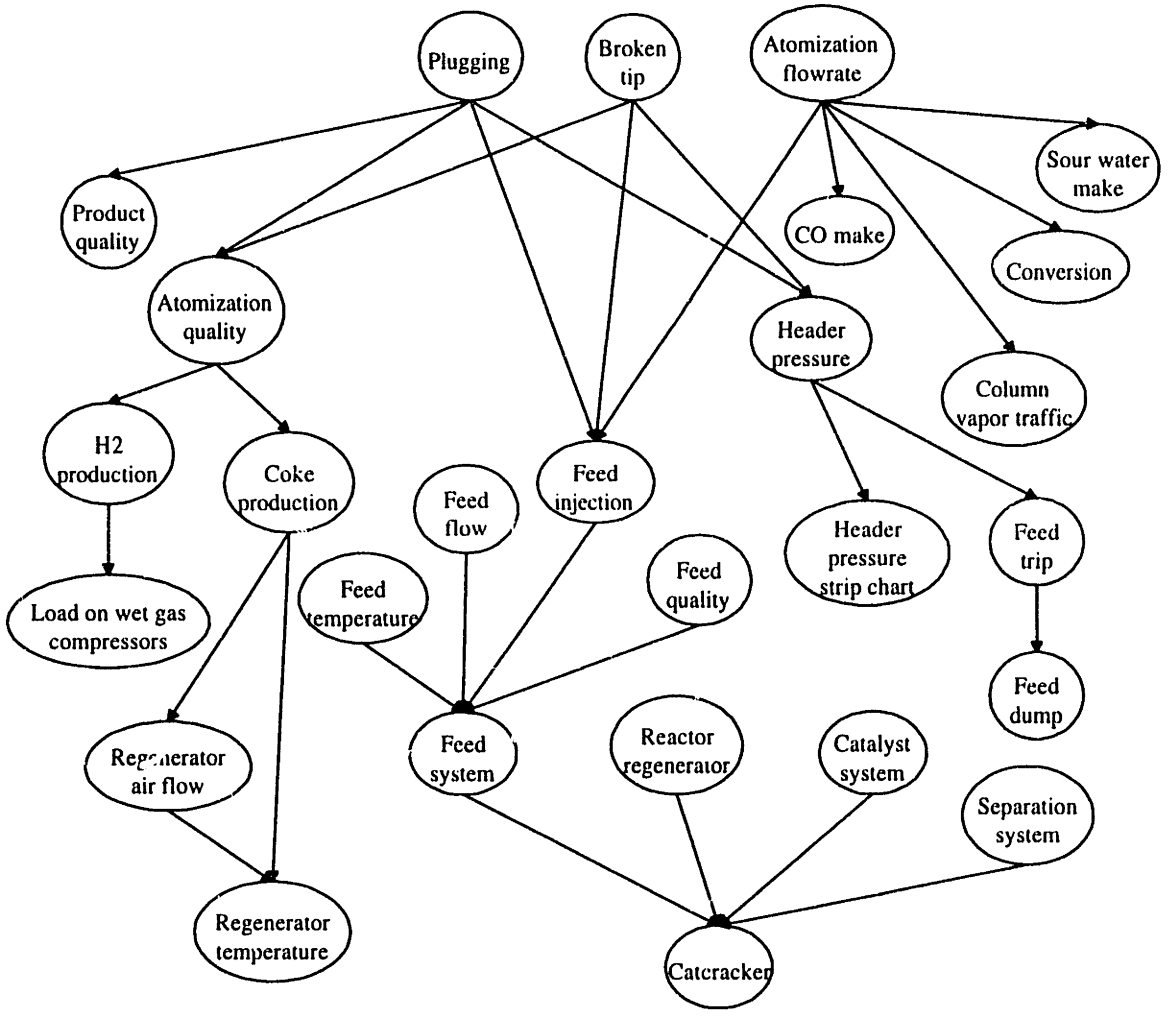


Figure 4.4. Bayesian network topology for a FCC section.

Next the links between nodes are identified. For example, *atomization quality* directly affects *hydrogen production*, so there is a link, represented by a directed arrow from cause to effect. Variables which are not directly related probabilistically are not linked. Finally, the prior and conditional probabilities are assigned.

The Bayesian network in this example has been extended to include the process hierarchy. The objective of this extension is to summarize in a simple way the location of the fault within the process. The fault location can be specified at different degrees of detail. A

deeper level in the tree hierarchy corresponds to an increasing degree in detail. The variables in the hierarchy are obtained by decomposing the system and subsystems of the process. The links between nodes in the hierarchy are deterministic, indicating that a section is in an abnormal state if and only if any of its subsections is working abnormally. For example, when *feed injection* is abnormal, then *feed system* and *catcracker* are set to abnormal. Figure 4-4 shows the graph for the FCC unit section focusing on feed injection problems and Table 4-1 illustrates probabilities obtained from behavioral descriptions.

Table 4.1 Conditional probability distribution for the *Header Pressure*

NODE: Header Pressure (Given BROKEN TIP = NO)			
	Plugging		
Header Pressure	None	Moderate	Severe
Low	0.0	0.0	0.0
Normal	1.0	0.0	0.0
Moderately High	0.0	**0.5**	**0.25**
Very High	0.0	**0.5**	**0.75**
Note: *Very High* is considered enough to activate trip			

Table 4-2 Conditional probability distribution for the *Feed Dump*

NODE: Feed Dump		
	Feed Trip	
Feed Dump	Activated	Not Activated
Yes	1.0	**0.0**
No	0.0	**1.0**

Table 4-3 Conditional probability table for the Load on Wet Gas Compressor

NODE: Load on Wet Gas Compressors		
Load on Wet Gas Compressors	H2 Production	
	Normal	High
Yes	1.0	**0.0**
No	0.0	**1.0**

Tables 4-2, 4-2, and 4-3 encode the conditional probability values obtained from behavioral descriptions: (1) If **plug is severe** then header pressure **may increase enough** to activate trip which causes the feed to be dumped. (2) For **less severe plugs**, the header pressure will increase but not necessarily enough to cause a feed dump. (3) The increased H2 production **would be observed as** an increase in load on wet gas compressors.

4.3.5 Illustration of inference features

Belief networks are flexible probabilistic models which can be used for prediction and diagnosis. The following discussion includes qualitative reasoning features and numerical results for a section of the FCC unit belief network shown in Figure 4-4. The probability distributions for each node are shown in Appendix A, and the joint probability distribution is listed in Appendix B with a total of 48 possible system states, and their probabilities. Each state description includes a value for each variable in the system. The exhaustive system states enumeration is possible only for very small systems and constitutes a brute force approach used here to compare results with the Bayesian network conclusions.

Predictive capabilities find the likely effects of given causes, that is, predictions are probabilistic. If the *injector tip* is found to be *broken*, then instantiation of the *injector tip* variable and the resulting messages propagation yield State 1 shown in Appendix B as the

best system description with *Plugging = none*, *Atomization Quality = poor*, *Injector Tip = broken*, *Product Quality = normal* and *Hydrogen Production = high*. The broken tip causes poor atomization quality which in turn causes a high production of H_2.

Diagnostic reasoning involves a more complex process than forward or predictive reasoning. The objective is to find the most probable causes given some observations. When *product quality* is determined to be *low*, the result of propagating this evidence generates State 39 with *Plugging = severe*, *Atomization Quality = poor*, *Injector Tip = not broken*, *Product Quality = low* and *Hydrogen Production = high*. *Low product quality* indicates it is likely to have *severe plugging*. Consistently with this scenario, atomization quality is poor and consequently *H_2 production* is *high*. The *injector tip* is diagnosed as being in a normal state. This case illustrates the global consistency of the conclusions which is one of the main features of belief networks. Global consistency, as explained before, is a key requirement when interactions are known to exist among faults and when competing faults share some symptoms.

The Bayesian network model can provide a consistent system description even when there is no available evidence. In this case the best system description is based on the prior probabilities. The network structure captures the qualitative relations among variables, and that structure by itself provides correct qualitative responses even with imperfect probabilistic parameters. As expected, the most probable state found *a priori* was that which contained all variables in a normal state, State 12 (shown in Appendix B) which has a probability of 0.455. The probabilities in the example are used only to illustrate some qualitative features of Bayesian networks but were not obtained from a real system. The flexibility gained by being able to perform inference with incomplete information is crucial when diagnosing real systems since partial measurements are common. Additional

flexibility results from not having a commitment to measure some specific subset of variables.

Often both predictive and diagnostic capabilities are used simultaneously. Hypotheses explaining the same symptoms automatically compete in the belief network. When a symptom is explained by more than one cause, evidence indicating the presence of one cause should decrease the belief that the other cause is present, an effect called *explaining away* by Pearl (1988). This is exemplified in Figure 4-5, where *poor atomization quality* (A) can be caused by a *broken injector tip* (T) or *injector plugging* (P). When *atomization quality* is found to be *poor*, then the probability of both causes increase. When further evidence is gathered supporting *plugging* as the failure, the probability of *broken tip* should be reduced.

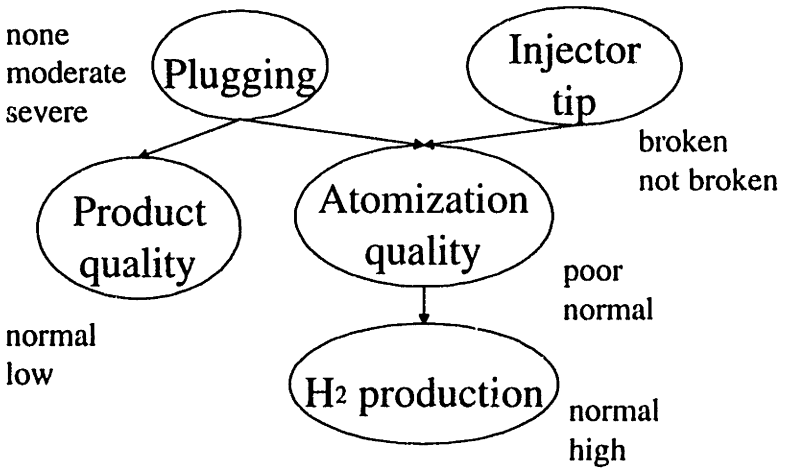


Figure 4-5. Effect of context on competing hypotheses or *explaining away*

Explaining away is captured by probability theory. When a feed injection problem is detected, the probability of both failures increases from their prior probabilities: $P(P|A)=P(A|P)P(P)/P(A)$ and $P(T|A)=P(A|T)P(T)/P(A)$. When further evidence tends to

confirm *plugging*, the probability of *broken tip* is reduced to $P(T|P,A)=P(A|T,P)P(T)/P(A|P)\gg P(T)$, since $P(A|P,T)$ is of comparable magnitude to $P(A|P)$.

When *poor atomization quality* was incorporated into the network, the most probable diagnostic conclusion was State 39 which includes *severe plugging* and consequently *low product quality*. Later, when it is found that the *injector tip* is *broken*, the evidence is incorporated and propagated through the network. The *broken tip* "explains away" the *poor atomization quality* and the most probable state consistent with the new observations is State 1. *Broken injector tip* and *poor atomization quality* were given as evidence. The inference conclusion additionally indicates *high H_2 production* and, more importantly, takes back part of the previous conclusion, now indicating no plugging, and normal product quality which best describe the system in the light of new evidence. This non-monotonic reasoning is a valuable feature of Bayesian networks.

A related problem involves handling the context in which evidence is obtained. In Figure 4-5, an increase in the probability that the *atomization* is *poor* can cause either an increase or a decrease in the probability that the *injector tip* is *broken*, depending on the context in which it occurs. If the *poor atomization* probability increases as a result of the onset of *high H_2 production*, then the probability of *broken injector tip* increases. On the other hand, if the probability of *poor atomization quality* increases as a consequence of finding a *low product quality*, then the probability of *broken injector tip* decreases, since the atomization problem is most likely due to *plugging*. The source of change in belief of atomization quality must be considered to move the probability of *broken injector tip* in the correct direction.

4.4 Multi-Stage Bayesian Networks

The main limitation of Bayesian networks is the lack of a formal representation of dynamics. This section describes an extension of the framework to represent and reason about temporal probabilistic relations. The multi-stage Bayesian network (MSBN) is a general framework for representing the dynamic behavior of systems of discrete variables. It offers a formal, rigorous approach to fault diagnosis using probability theory. The MSBN subsumes several existing diagnostic methodologies including those based on the signed directed graph (SDG) and its relatives, and techniques based on patterns of satisfied or violated constraint residuals.

4.4.1 Signed directed graph and residual-based methods

Fault diagnosis can be performed using *diagnostic-direction knowledge,* connecting symptoms to root causes, or with *model-direction knowledge,* connecting root causes with symptoms (Kramer and Mah, 1993). Diagnostic-direction knowledge is leveraged through deductive or pattern-driven techniques such as expert systems, neural network classifiers and decision trees. Model-direction knowledge is leveraged for diagnosis through a process of inversion: a model capable of predicting symptoms from given causes is used to infer causes from observed symptoms.

In this section, we focus on model-based diagnosis using models where the system and its malfunctions are represented by discrete variables. Several existing model-based diagnostic methods fit this general framework. Iri and co-workers (1979) proposed the use of the signed directed graph (SDG), where each variable is represented as high, low, or normal and influences between variables are represented by directed arcs. Each arc is parameterized by a sign which represents the direction of influence (sign of the gain) from one variable to another. Diagnosis is carried out by search for an unobserved node which roots a consistent subgraph involving all abnormal observed nodes. Many workers

have extended the basic SDG technique by introducing more than three discrete states (Shiozaki, *et al,* 1985), probability, gain and delay information on arcs (Kokawa, *et al,* 1983; Tsuge, *et al,* 1985; Yu and Lee, 1991), robust inference techniques for multiple faults (Finch, *et al,* 1990), continuous fuzzy membership functions for discrete variables (Yu and Lee, 1991), a multi-stage representation of time (Hashimoto, *et al,* 1991), and distributed reasoning (Mohindra and Clark, 1993).

A closely related set of model-based techniques utilizes patterns of process-model residuals (constraint violations) to locate root causes. These methods include the Governing Equations Method (Kramer, 1987), and the Diagnostic Model Processor (Petti, *et al,* 1990). Causal relations in these methods can be represented by two-level (bipartite) graphs, with nodes representing root causes connected to nodes representing equation residuals. Nodes can be represented by Boolean or fuzzy variables. These residual methods are special cases of SDG techniques where effect nodes are calculated residuals rather than directly observed quantities.

In this section, we show how both digraph and residual-violation methods are subsumed by a new representation called the multi-stage Bayesian network (MSBN), based on the Bayesian belief network representation originally proposed by Pearl (1988). The MSBN allows formal representation and rigorous solution of a wide range of diagnostic problems modeled by causal networks of discrete variables.

4.4.2 Capabilities of SDG and residual-based approaches

Figure 4-6 summarizes modeling features found in SDG and residual-based methods. The representation involves cause nodes (with one or more outputs), effect nodes (with one or more inputs) and directed causal arcs. Nodes are discrete variables with a finite number of mutually-exclusive states (usually 3 or 5 states, representing levels high/normal/low or

very high/high/normal/low/very low). Root cause nodes (with no inputs) represent malfunctions or unmeasured parameters whose values are to be determined. Effect nodes can represent either observable symptoms or unmeasured intermediate variables that influence subsequent effect nodes through one or more output arcs. In some representations, multiple binary-valued (T/F) nodes are used to express the state of a single variable (Wilcox and Himmelblau, 1994); alternately, nodes can represent state transitions instead of states (Finch, *et al*, 1990).

Arcs represent causality between variables, and are often parameterized by a sign (+/-) representing a direction of influence of the cause on the effect. Multiple arcs terminating at a node are interpreted according to "or" logic. If an effect node is acted upon by simultaneous positive and negative inputs, the resultant state is indeterminate. Several authors have added quantitative gain information or time delay information to arcs to reduce ambiguity caused by influences of opposite sign.

Many algorithms have been proposed to determine active root causes in SDGs. Most techniques restrict the analysis to single causes. Under the single-cause assumption, a viable root cause must "explain" all active symptoms through consistent causal pathways originating from the root cause node. If multiple root causes are considered, the goal is to identify a minimal set of root causes that account for observed symptoms. Most (if not all) current algorithms lack a formal basis such as that provided by probability theory.

The SDG, even with extensions, does not constitute a general discrete-state model due to the following limitations:

- The SDG is an "or" graph, with each cause acting independently upon an effect node. Potential interactions between causes at effect nodes are not modeled.

- Signed arcs (and arcs with gains) are only compatible with ordered states such as high/normal/low. Nodes with unordered states cannot be handled.
- The SDG provides no differentiation of *may-cause* and *must-cause* arcs.
- The semantics for handling time are weak and there is no consensus about the proper handling of time in SDG techniques.

These limitations also apply to diagnostics based on residual patterns. Some techniques (Kramer, 1987), associate residual patterns and faults with if-then rules and certainty factors, an approach which presents difficulties when combining partial diagnostic conclusions (Rojas-Guzmán, and Kramer, 1993a). The assumption that faults with competing effects do not simultaneously affect the same constraint limits the scope of these methodologies.

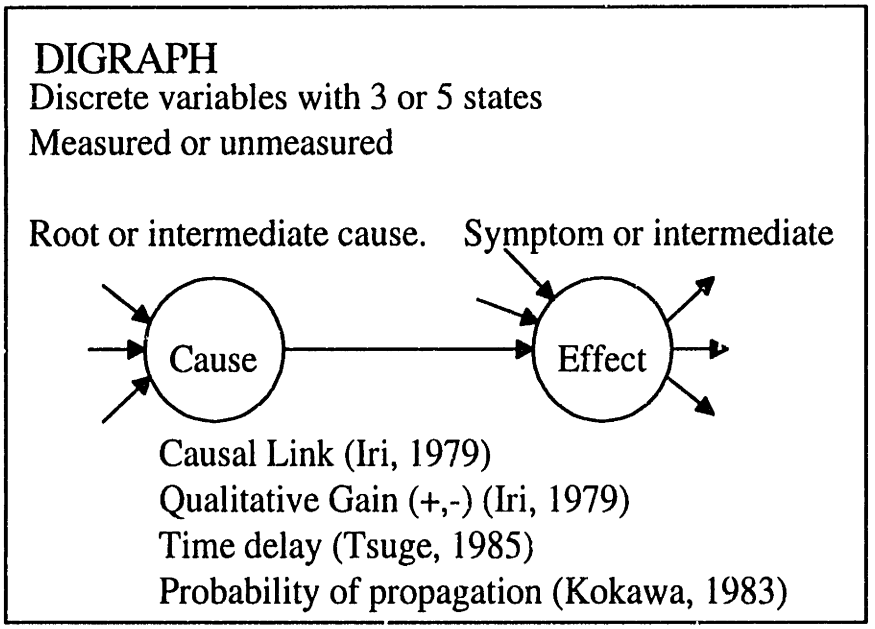


Figure 4-6. Digraph semantics.

4.4.3 Multi-Stage Bayesian Networks

Bayesian networks (also called belief networks) have been suggested as a method of representing and solving chemical engineering diagnosis problems (Rojas-Guzmán and Kramer, 1992; Rojas-Guzmán and Kramer, 1993a). The Bayesian network is a formal graphical representation capable of quantitatively modeling systems of discrete variables. As in the SDG, each node represents either a root cause, an intermediate variable, a symptom or an equation residual. Algorithms exist to perform diagnostic inference on these graphs rigorously, using the laws of probability, resulting in the most probable assignment of all unobserved variables (Pearl, 1988; Shimony and Charniak, 1990; Rojas-Guzmán and Kramer, 1993b). In a Bayesian network, any variables can be observed or unobserved, giving the method great flexibility in dealing with partially-measured systems. The Bayesian network method is not limited to single malfunctions. Approximate solution techniques which reduce calculation time and generate rankings of possible hypotheses have also been introduced (Rojas-Guzmán and Kramer, 1993b).

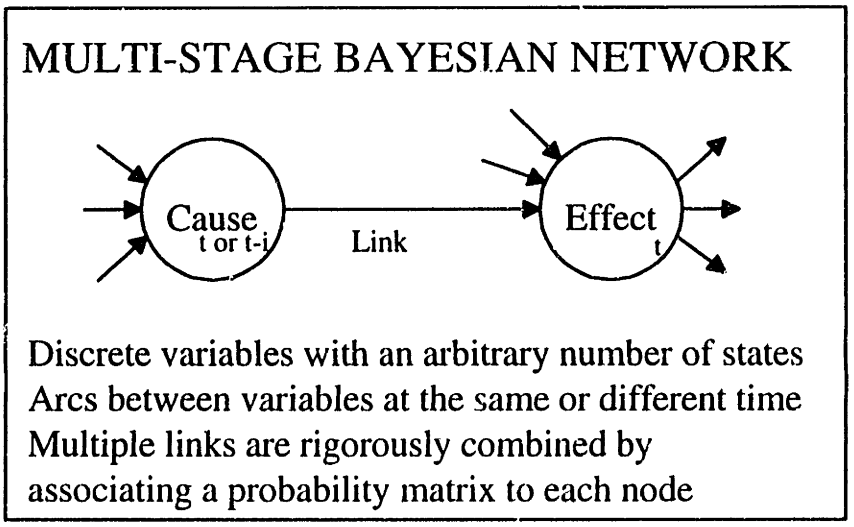


Figure 4-7. Semantics of graph elements in MSBN.

In this section we show the extension of the Bayesian network to the multi-stage Bayesian network (MSBN) to handle systems with time delays and other dynamics. In the MSBN, nodes represent variables at different times, and inference is performed with time as an implicit variable over the aggregate network spanning multiple time slices. Because time is implicit in the MSBN, all the theoretical properties of the Bayesian network extend directly to the MSBN. The MSBN encompasses methods based on SDGs with time delays and multi-stage SDGs. The MSBN provides in a single framework the union of the features and capabilities of all digraph methods.

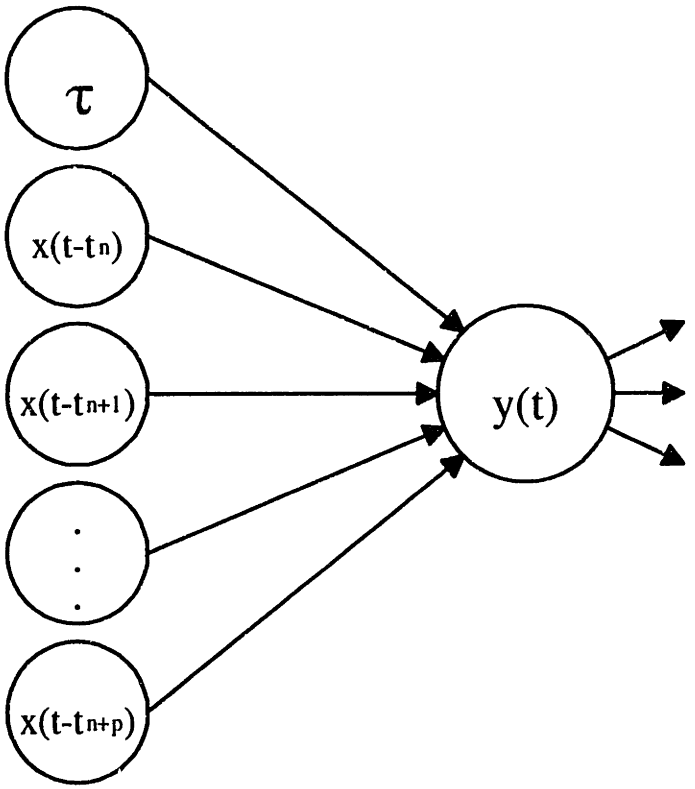


Figure 4-8. Graphical representation of temporal causal relations in the MSBN.

Figure 4-7 shows the semantics of the MSBN. Variables are represented by time-indexed nodes that take on mutually-exclusive, exhaustive states. The absence of an arc between

two variables indicates independence between them given that the values of their parents are known. Arcs between the same variable at different times are allowed, accounting for the effect of a variable on its own future state. No parameters are directly associated with arcs; instead, delays are accounted for via time indexing of nodes. Effect nodes are parameterized by a link matrix which gives the distribution of probabilities of the states of the effect node for each different combination of inputs.

The MSBN can be arranged into time "slices", where each slice involves variables at a single time. Slices can be linked not only by delayed effects between variables, but also by variables influencing their own future states. Delayed feedback loops become open spirals in the MSBN. Uncertain time delays can be modeled by introducing nodes representing time delay parameters.

τ	$x(t-t_n)$	$x(t-t_{n+1})$	P_0	P_1
0	0	any	a_1	a_2
0	1	any	b_1	b_2
1	any	0	c_1	c_2
1	any	1	d	d_2

$\tau = 0$ indicates a time delay of $(t-t_n)$
$\tau = 1$ indicates a time delay of $(t-t_{n+1})$

$P_0 = p(y=0 \mid \tau, x(t-t_n), x(t-t_{n+1}))$
$P_1 = p(y=1 \mid \tau, x(t-t_n), x(t-t_{n+1}))$

Figure 4-9. Modeling of causal relations with uncertain time delays in the MSBN

Figure 4-8 and Figure 4-9 depict the MSBN modeling of the causal relation x(t - τ) -> y(t), where τ is an uncertain delay described by a prior probability distribution P(τ) with $t_n \leq \tau \leq t_{n+p}$. The link matrix states that if $\tau = t_i$, the state of y(t) is dependent only on the state of x(t - t_i). Inference on the MSBN results in an improved estimate (posterior probability distribution) for the uncertain delay parameter.

Compared to the SDG, the MSBN has several advantages:

- The link matrix allows for arbitrary distribution of states in an effect node as a function of all input effects. It is not limited to "or" logic.
- The number of states per variable is not limited to 3 or 5, and arbitrary, unordered states are permitted.
- Delayed propagation of effects and uncertain delays can be represented without special extensions or new inference rules.
- Both rigorous reasoning methods and time-saving approximate methods are available for determining the states of unobserved nodes, both based on the laws of probability.
- The MSBN may be extended arbitrarily backwards and forwards in time, and can be used to predict future states as well as estimate the present and past state of the system.

4.4.4 Model construction and comparison

The following example illustrates the construction of a Multi-Stage Bayesian Network and compares it with a digraph and a residual-pattern bipartite graph. The system, shown in Figure 4-10, consists of two interconnected tanks in series with a non-compressible fluid

flowing through pipes F_0 , F_1 and F_2. Possible faults in this system include partial blockage at pipes F_1 and F_2 and leaks from the tanks, represented by streams Q_1 and Q_2.

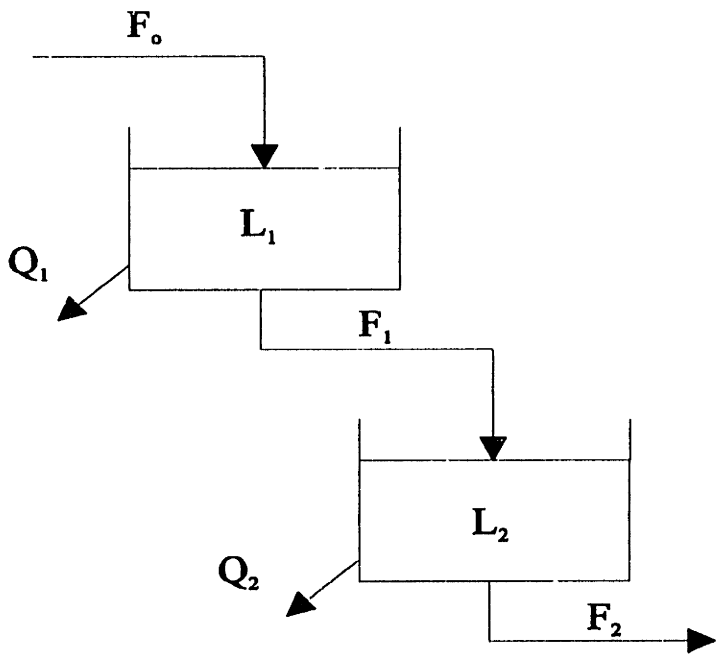


Figure 4-10. Two tank system

The behavior of the two tank system can be described by the following equations:

$$A_1 \frac{dL_1}{dt} = F_o - F_1 - Q_1 \qquad [4\text{-}1]$$

$$F_1 = b_2 a_2 \sqrt{L_1} \qquad [4\text{-}2]$$

$$A_2 \frac{dL_2}{dt} = F_1 - F_2 - Q_2 \qquad [4\text{-}3]$$

$$F_2 = b_3 a_3 \sqrt{L_2} \qquad [4\text{-}4]$$

L_1 and L_2 represent the level of liquid in the tanks whose base areas are A_1 and A_2. Output flow from each tank is proportional to the square root of its level, with a_2 and a_3 being constants. Pipe plugging is quantified with b_2 and b_3. Under normal operating

conditions $Q_1 = 0$, $Q_2 = 0$, $b_2 = 1$, and $b_3 = 1$. Ideally a diagnostic technique should be able to detect leaks ($Q_i > 0$), partial pipe blockages ($0 < b_i < 1$) or any combination of them.

The traditional digraph representation of the two tank system is shown in Figure 4-11. Directed links and their associated qualitative gains represent causal interactions, with high/low thresholds, and constitute the complete digraph model.

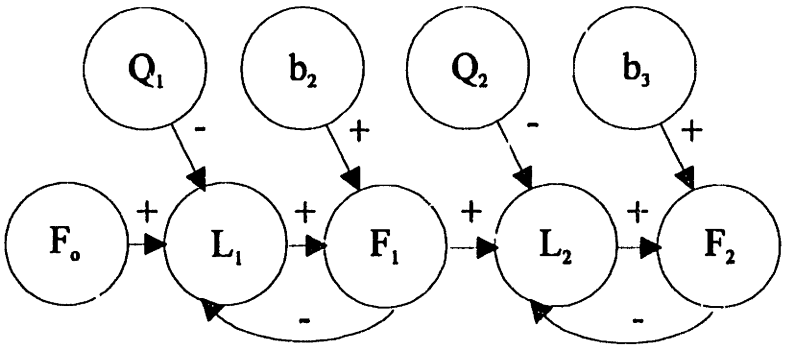


Figure 4-11. Digraph representation for the two tank system

A bipartite graph that relates faults (nodes in the first level) with equation residuals (nodes in the second level) is shown in Figure 4-12. Nodes e_i represent equation residuals and are calculated from equations describing normal operation. Equation residuals are typically assumed to be normally distributed with zero mean. When a fault is present, the values of some residuals have a larger magnitude. Diagnosis is performed by discriminating the distinct pattern of residuals which result from each fault. All the quantities involved in the equations used in the bipartite graph are known or measured.

Note that redundant equations can be used and contribute to the construction of residual patterns.

If F_0, F_2, L_1 and L_2 are measured and A_1, A_2, a_2 and a_3 are known constants, equations [4-5], [4-6], [4-7] and [4-8] can be used to calculate residuals. The derivative of level with respect to time can be approximated from level values measured at different times.

$$\varepsilon_1 = A_1 \frac{dL_1}{dt} - F_o + a_2\sqrt{L_1} \qquad [4\text{-}5]$$

$$\varepsilon_2 = A_2 \frac{dL_2}{dt} + F_2 - a_2\sqrt{L_1} \qquad [4\text{-}6]$$

$$\varepsilon_3 = A_1 \frac{dL_1}{dt} + A_2 \frac{dL_2}{dt} - F_0 + F_2 \qquad [4\text{-}7]$$

$$\varepsilon_4 = F_2 - a_3\sqrt{L_2} \qquad [4\text{-}8]$$

Residual values calculated from normal operation equations reflect the presence of faults and their magnitudes are related to the unmeasured quantities b_2, b_3, Q_1, Q_2, and random noise from sensors by the following equations.

$$\varepsilon_1 = (1-b_2)a_2\sqrt{L_1} - Q_1 + N(0,\sigma_1) \qquad [4\text{-}9]$$

$$\varepsilon_2 = (b_2-1)a_2\sqrt{L_1} - Q_2 + N(0,\sigma_2) \qquad [4\text{-}10]$$

$$\varepsilon_3 = -Q_1 - Q_2 + N(0,\sigma_3) \qquad [4\text{-}11]$$

$$\varepsilon_4 = (b_3-1)a_3\sqrt{L_2} + N(0,\sigma_4) \qquad [4\text{-}12]$$

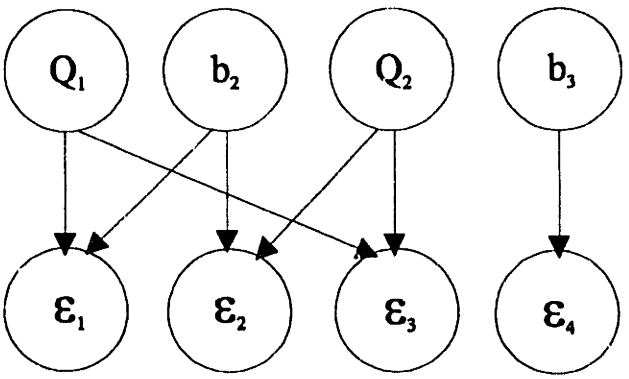


Figure 4-12. Bipartite graph associating faults with equation residuals

The same relations captured by a SDG can be modeled by a MSBN with time indexing and probabilities of 0 or 1. One possible MSBN corresponding to the two tank system is shown in Figure 4-13. By using probabilities between 0 and 1 instead of signs, the MSBN provides a more general and powerful model. Immediate effects are represented by links between variables at the same time slice and temporal relations are modeled by links between variables at different time slices. The persistence of faults is indicated by a link from the fault variable to the same variable at the next time slice. The value of exogenous variables such as F_0 can be estimated from their value at a previous time.

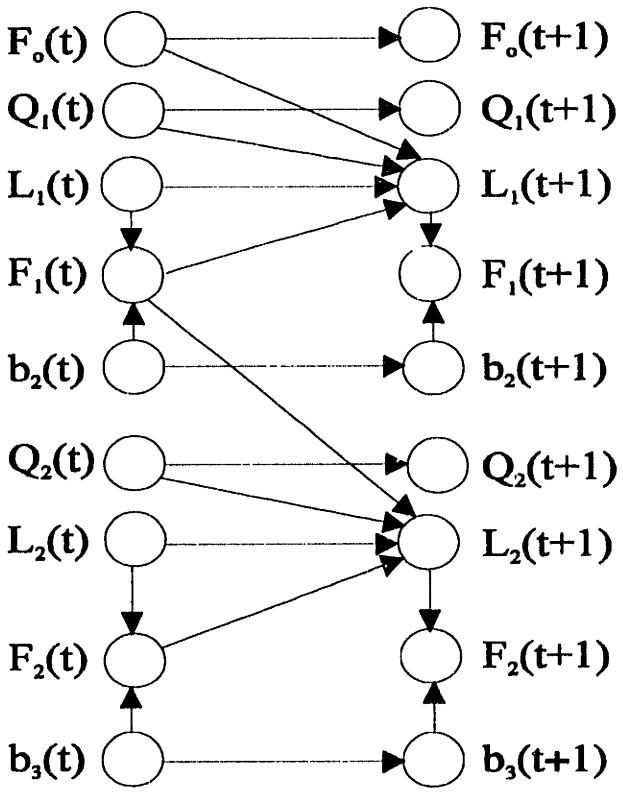


Figure 4-13. Multi-stage Bayesian Network for the two tank system.

The MSBN framework can also use residuals by including one node in the network for each residual. The MSBN corresponding to the residuals defined in equations [4-5], [4-6], [4-7], and [4-8] would have the same structure as the bipartite graph shown in Figure 4-12 and would have probabilities associated to each link.

In the general case, Multi-Stage Bayesian Network combines time indexing, residual nodes and probabilities offering a powerful and flexible representation which subsumes SDGs and residual-based methods and extends their modeling capabilities.

Probabilistic parameters for the MSBN can be obtained from statistics based on recorded process measurements, subjective estimates from experts or from deterministic mathematical models. Discrete probability distributions can be approximated with an

arbitrary degree of precision from the process model using Monte Carlo simulations. First, a value is assigned to each root node according to its prior probability distribution and the process is dynamically simulated. Next, the state of each node and its immediate parents is recorded. The number of times the simulation is repeated and the number of time steps used in each simulation depend on the required degree of precision. Finally, the states of each node and its parents are counted to estimate the discrete probability distribution of the states of each node given the states of its parent nodes.

4.4.5 Summary

We have introduced the MSBN as a general model of discrete dynamic systems and compared its properties to previous SDG and residual-pattern methods. Simple SDG methods constitute a particular case of Bayesian networks where the discrete states are high, low and normal, and probabilistic parameters are either 0 or 1. The relations between equation residuals and constraints in residual-based diagnostic methods are represented by a two-level network, another particular case of Bayesian networks. The main benefits of the MSBN are that (1) arbitrarily complex causal relations between discrete variables can be expressed, and (2) dynamic behavior with uncertain time delays can be modeled.

Chapter 5

A Graph-based Genetic Algorithm for Abductive Inference

Diagnostic reasoning, also called abductive inference, involves determining the global most probable system description given the values of any subset of variables. In some cases abductive inference can be performed with exact algorithms using distributed network computations, but the problem is NP-hard and complexity increases significantly with the presence of undirected cycles, the number of discrete states per variable, and the number of variables in the network. This paper describes an approximate method based on a graph-based genetic algorithm that uses non-binary alphabets, graphs instead of strings, and graph operators to perform abductive inference in multiply connected networks for which systematic search methods are not feasible. The motivation, theoretical basis and adequacy of the method are discussed and experimental results are presented.

5.1 The Mathematical Problem

Abductive inference in the Bayesian belief network framework can be viewed as an optimization problem in a large discrete multi-dimensional search space. The objective is

to find H, the *best* set of discrete values for the unknown variables (given the values of a subset of known variables), in other words, to maximize the probability of the inferential hypothesis given the evidence:

$$H = \{x_m \ni p(x_m \mid y) \geq p(x_n \mid y) \forall x_n, y \in \Omega\} \qquad [5\text{-}1]$$

where x is a set of unknown variable-value pairs (a full instantiation of the unknown variables subset) and

y is a set of measured, observed, or otherwise known variable-value pairs,

$x \cup y = \Omega$ and $x \cap y = \varnothing$,

and $p(x \mid y) \in [0,1]$.

The size of the resulting multi-dimensional discrete space in the Bayesian network renders the search combinatorically complex. The size of the search space, S, can be calculated as follows:

$$S = \prod_{i=1}^{n} (d_i) \qquad [5\text{-}2]$$

where n is the number of nodes in the network and d_i is the number of discrete states that node i can assume. A near-optimal search algorithm is described in the following sections to efficiently perform this task.

5.2 Motivation

Recently developed methods for propagating probability information in the Bayesian network structure have improved the ease with which probability data can be manipulated. These methods use distributed parallel computations in which probabilistic values are

locally propagated between neighboring nodes in the network (Pearl 1988). However, for large multiply connected networks, exact inference may not be feasible, rendering approximate algorithms an attractive alternative. Abductive inference with Bayesian belief networks is an NP-hard problem (Cooper 1990) and similarly all the exact methods are highly sensitive to the connectedness of the networks (Horvitz 1990). Complexity increases with the number of variables in the system, the number of states per variable, and the number of undirected cycles in the network.

Different methods exist to find the most probable globally consistent explanation given the evidence, also called most probable explanation (MPE). Pearl (1988) proposed an exact algorithm that can find the two best explanations for singly connected networks, but its growth is exponential for multiply connected networks. Shimony and Charniak (1990) obtain the maximum a-posteriori (MAP) assignment of values by using a best-first search on a modified belief network. The algorithm naturally extends to find next-best assignments, is linear in the size of polytrees but exponential in the general case. Peng and Reggia (1987) formalize causal and probabilistic associative knowledge in a two level network which associates disorders and manifestations. The structure is a special case of a belief network and calculations are computationally complex if multiple simultaneous disorders may occur. Li and D'Ambrosio (1993) developed an algorithm based on finding an ordering of distributions in the network and efficiently combining them. An optimal factoring can be found for singly connected networks and an efficient algorithm is proposed for multiply connected networks. The next MPE can be found in linear time. Poole (1993) developed an anytime algorithm that estimates probabilities and provides error bounds. The algorithm exploits only the probability distributions, not the structure, and is efficient only for extreme distributions with values close to 0 or 1.

This chapter explores the properties and performance of Genetic Algorithms (GAs) to find approximate near-optimal solutions in large and multiply connected belief networks. Section 5.3 describes the fundamentals of genetic algorithms. Section 5.4 discusses the adequacy of applying GAs to abductive inference in belief networks and describes in detail one genetic algorithm used. Section 5.5 examines the proposed genetic algorithm under the set of requirements for *admissibility*, shows that it satisfies the requirements, is admissible, and therefore exhibits *implicit parallelism*. Section 5.6 compares different parent selection criteria. Section 5.7 presents experimental results on a larger, multiply connected well-known network using an object-oriented implementation. A discussion of the results is presented in Section 5.8. Conclusions are summarized in Section 5.9.

5.3 Genetic Algorithms

Approximate algorithms constitute an alternative when the size and topology of a problem render it intractable. The trade-off involves accepting a near-optimal solution (often the exact optimal solution) in a feasible time. The method proposed in this paper to perform inference takes advantage of the BN framework to represent an uncertain system. The use of GAs to solve Bayesian belief networks (Figure 5-1) seems to be a simple yet powerful combination of a knowledge representation paradigm and an efficient inference engine (Rojas-Guzmán and Kramer, 1993b). The development of the algorithm was motivated by the need to perform automated reasoning in complex systems with non-linear interactions for real-time applications. To understand why GAs are particularly suited to perform inference in BNs, a review of the underlying concepts of this approach is necessary.

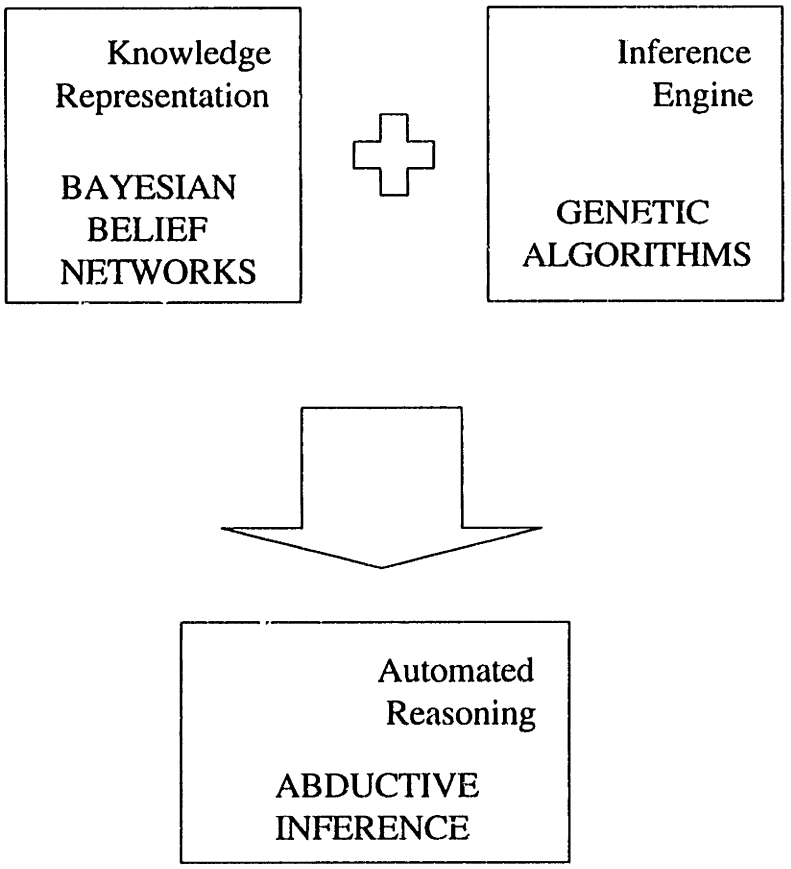


Figure 5-1. Abductive inference through Bayesian belief networks and genetic algorithms

Genetic algorithms are search procedures based on the mechanics of natural selection and natural genetics (Goldberg 1989). The methodology, architecture, and theoretical analysis were developed by Holland (1992) for studying existing natural adaptive systems and designing artificial adaptive systems. The idea is that ..."adaptation can be usefully modeled as a form of search through a space of structural changes ... to improve its behavioral characteristics" (De Jong 1985). The structural modification space is conventionally represented by strings of symbols chosen from some finite alphabet.

The notion of *implicit parallelism* is a key strength of GAs and is responsible for the allocation of the search effort simultaneously to many hyperplanes (Grefenstette 1989). Hyperplanes can be thought of as subsets of points in the space which are consistent with a partial point specification. The Schema Theorem proposed by Holland in 1975 (Holland 1992) has been called the Fundamental Theorem of Genetic Algorithms (Goldberg, 1989). "The power of a genetic algorithm derives largely from its *implicit parallelism*, i.e., the simultaneous allocation of search effort to many regions of the search space" (Grefenstette, 1991). It means that trials are allocated in parallel to a large number of hyperplanes. A hyperplane with persistently above-average fitness grows at an exponential rate, while trials to those with below-average fitness rapidly decline. GAs process $O(N^3)$ hyperplanes in a population of size N. A theoretical discussion of the underlying principles of GAs is presented in (Holland 1992; Goldberg 1989; Grefenstette 1989). The concept of a *schema* is used in the theoretical analysis of GAs by Holland (1992). A *schema* is a similarity template (a specification of alleles for a subset of the genes) describing a set of strings with similarities at certain positions. *Schemata* can be thought of as pattern matching devices. An important result (Holland 1992) is that highly fit, short-defining-length schemata, also called *compact building blocks*, are propagated through generations by giving exponentially increasing samples to the observed best individual.

"Genetic algorithms are theoretically and empirically proven to provide robust search in complex spaces" (Goldberg, 1989). GAs have several advantages over other optimization methods to solve problems in discrete and in continuous spaces. GAs improve over the local scope of traditional search methods (calculus-based, enumerative or random) by searching in parallel many subspaces in multidimensional spaces with complex topologies. Enumerative approaches are often not feasible or too slow for systems under time constraints whereas random search algorithms simply lack the efficiency of GAs.

According to Goldberg (1989) GAs differ from other methods in the following ways: (1) GAs work with a coding of the parameter set, *i.e.* the natural parameter set of the problem is typically coded as finite-length string over some finite alphabet, (2) GAs search from a population of points, not from a single point, (3) GAs use an objective function without any auxiliary knowledge, and (4) GAs use probabilistic transition rules, not deterministic rules.

Commonly used optimization techniques such as integer and mixed integer non-linear programming (MINLP) techniques have been successfully used in optimization problems involving discrete variables. However, MINLP techniques are not adequate for inference with Bayesian networks for the following reasons. Due to the interactions that exist among variables within the network, a decomposition in subproblems to be solved in sequence is, in general, not possible. This requirement of dynamic programming is not satisfied in Bayesian networks. Furthermore, since variables in a Bayesian network are not restricted to being binary, it is necessary to preprocess the model to an equivalent formulation containing only binary variables with the consequent problem growth. The relaxation would work for continuous variables represented as discrete variables but it would not be meaningful for variables whose discrete states are not intrinsically ordered.

5.4 A Graph-based Genetic Algorithm

The algorithm described in this paper is part of a larger probabilistic reasoning system. The algorithm can perform diagnostic reasoning and predictive inference. For large multiply connected networks, exact inference may not be feasible (the problem is NP-hard), rendering approximate algorithms an attractive alternative. Systematic enumeration

is often not feasible due to combinatorial explosion. Approximate algorithms constitute a viable trade-off.

The use of an evolutionary programming approach for inference in Bayesian networks seems to be a simple yet powerful combination of a knowledge representation paradigm and an efficient inference engine. The proposed GA uses non-binary alphabets since each variable or node in the network has its own small set of discrete states. *Genotypes* are sets containing variable-value assignments for all nodes in the network. A key feature of the algorithm is the use of graphs instead of strings to represent the genotypes. Crossover is based on a graph operator which partitions the graph in clusters to be exchanged. *Compact blocks* are built from *semantically close* elements (adjacent nodes in the graph). The *phenotype* or metric in the solution space is the absolute probability of each set of assignments. This specific case of the probabilistic calculation in which all nodes have been *instantiated* (assigned a value) is fortunately straightforward in the Bayesian network framework.

Conventional GAs use three genetic operators (1) selection, (2) crossover, and (3) mutation, which are introduced in this section. The iterative nature of GAs is illustrated in Figure 5-2. The proposed algorithm extends the conventional notion of GAs and can be described more accurately as *evolution programming* (Michalewicz, 1992) since it uses non-binary alphabets, graphs instead of strings, and graph operators.

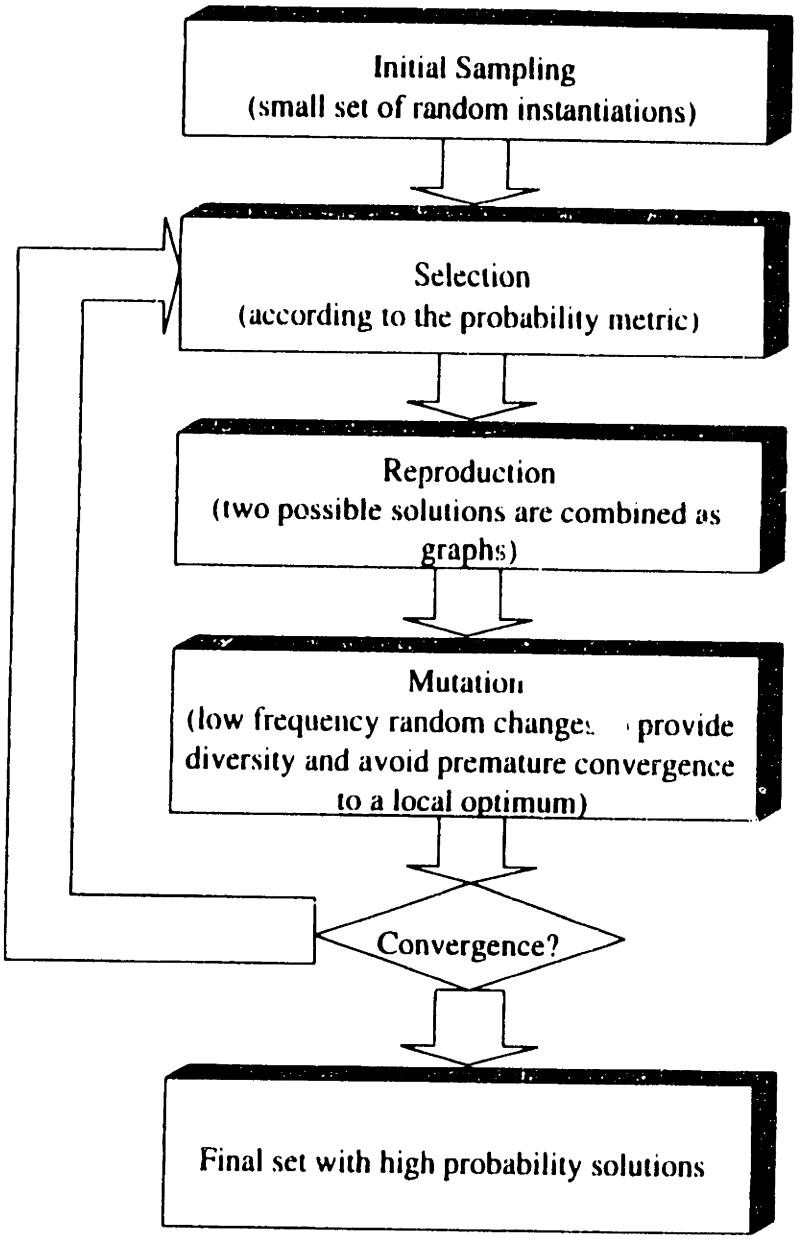


Figure 5-2. Basic steps of the graph-based genetic algorithm

5.4.1 Problem representation

A solution or individual is conventionally represented by a string of integers or chromosome which encodes the individual genotype. Each position or gene in the string corresponds to one variable in the Bayesian belief network. Each gene can take a number of values or alleles from a finite discrete alphabet which may be different for each gene and corresponds to the number of discrete values that the variable can assume. This feature

naturally maps into Bayesian networks where each variable has a small number of possible states.

A non-conventional representation is used in this paper to represent the genotype. Since the evolution of individual solutions is based on the notion of successful compact blocks inherited through generations, it seems reasonable to construct these blocks in a form such that their elements are adjacent on the genotype. Adjacency in the graph corresponds to the notion of *semantic closeness*. It is desirable to have neighboring nodes in the graph be neighbors in the genotype string. However, in mapping arbitrary graphs to strings, some nodes are necessarily separated. The structure which naturally satisfies *semantic closeness* is the graph itself. Therefore, in our algorithm, individuals in the population are graphs whose structure is that of the Bayesian network. During crossover, at least one (and very often both) of the graph sections exchanged during crossover is made up of neighboring nodes.

Experimental comparison of implementations of both the string and the graph representation confirm the expected benefits of *semantic closeness* achieved by representing individual genotypes as graphs. Both C++ implementations were run with the same model (the ALARM model described in section 7), data and algorithm parameters (obtained in Section 7.4). The string algorithm (30 runs) converged to sub-optimal solutions 16.7% of the time, required an average of 39.3 generations for convergence and had an average running time of 62.8 seconds. The graph algorithm (35 runs) converged to sub-optimal solutions in only 8.6% of the cases, with an average of 33 generations for convergence and an average running time of 57.2 seconds. Results show that the graph representation provides higher diagnostic accuracy and requires less generations to converge. The time per generation is about 6% shorter for the string

representation, but due to the larger number of generations required the graph algorithm has an overall faster performance.

5.4.2 Space metric

GAs require the existence of a metric in the space of possible solutions. In this case there is a clearly defined metric, namely the absolute probability of each possible solution. Within the Bayesian network framework, evaluating the probability of a solution is straightforward when all the nodes have been instantiated (assigned a value). Fortunately each member of the population is precisely a fully-instantiated solution. Each genotype (set of variable-value assignments) corresponds to a phenotype, P, (fitness metric or probability), calculated as follows:

$$P = \prod_{i=1}^{n} p(v_i | v_{i1}, v_{i2}, \ldots, v_{im}) \qquad [5\text{-}3]$$

Each factor is either a prior probability (for root nodes) or a conditional probability (for internal and leaf nodes). The phenotype, P, is a product with n factors, one for each node, v_i, conditioned on its parents, namely v_{i1} through v_{im}, where m is the number of parents of node v_i (for root nodes, m=0) and n the number of nodes in the network. These probabilities are efficiently stored and retrieved using multidimensional arrays.

5.4.3 Algorithm parameters

The GA algorithm requires the specification of several parameters. The main parameters quantify the population size, crossover rate and mutation frequency. The *total population* is the number of possible solutions and is usually a very large number that can be calculated as the product of the number of states of all unknown nodes as given by equation [5-2]. The *evolving population* is the number of individuals from which the final solution will evolve and is a small fraction, called *evolving fraction*, of the total

population. The *breeding selectivity* is the fraction of the *evolving population* which constitutes the pool from which parents are chosen for breeding. The *breeding population* is the number of individuals which make up the parents pool. The *average lifetime* of individuals can be specified and represents the average number of generations an individual stays in the evolving population set, before being displaced by better individuals. The inverse of average lifetime specifies the fraction of the population to be replaced at each generation. The *mutation frequency* is the fraction of the *evolving population* which suffers a mutation each generation. Mutations conventionally occur with a low frequency and consist of random changes introduced into the population genotypes to guarantee diversity and to prevent convergence to a local maximum. Finally the number of *generations* refers to the number of iterations in the algorithm. As suggested by (Grefenstette 1986) these parameters can be meta-optimized with another GA.

5.4.4 Convergence criteria

As the population of candidate genotypes evolves, the average phenotype increases, first as a result of new and better genotypes being generated, and at a later stage as a result of the convergence of the population to the same best genotype found. Evolution can be stopped when the time available for inference ends or when convergence is reached. Convergence is tested every N generations (with N=10) and the evolution is stopped when the phenotype of the best member of the population has not changed in the last N generations and when the sum of the phenotypes of the population has reached M% (with M=95%) of the value it would have if all its members were equal to the best. Perfect uniformity (M=100%) is unlikely when the mutation frequency is larger than 0.

5.4.5 Selection

The initial population of individuals is created randomly and constitutes a very small fraction of the total population. The initial population is expected to randomly and uniformly sample all the search space. Note that the search space is reduced when a subset of the variables is known and therefore not changed throughout the evolution. In GAs, common ways of handling invalid genotypes involve either penalizing invalid genotypes by significantly reducing their phenotype in the metric function, by fixing invalid genotypes whenever they are generated, or ensuring that all genotypes are valid. The latter approach is the most natural and trivial for this problem. All individuals are guaranteed to have legal genotypes by assigning to each uninstantiated gene in the initial population, an allele (value) from the specific alphabet of each gene (variable). When a mutation occurs, the new value is randomly selected from the possible discrete values of the variable involved. An arbitrarily specified fraction (1/average lifetime) of the evolving population is replaced during each generation by new individuals. Individuals with the lowest fitness are replaced. New individuals are created by combining parents, which are selected among the best found in the previous generation.

Selection of the best individuals can proceed according to different criteria. Three methods have been implemented. In the first, the probability of being selected as a parent (for each individual contained in the breeding population) is proportional to the phenotype of the individual. In the second method the probability of being selected is proportional to a monotonic function of the phenotype of the individual as shown in Equation 4. This function can be used to control the sensitivity of the algorithm to fitness values. In the third method the probability for each individual is the same for all elements of the breeding population. This criterion reduces sensitivity to phenotype values. Sensitivity reduction can also be accomplished by using a function of the rank of each individual. These criteria were evaluated and the results are described in the experimental section.

Immediately after each new individual is created, its phenotype or fitness is assessed and stored. In an iterative procedure, elements of each generation are sorted to choose those which will be replaced and those which will be used as parents for the next generation.

5.4.6 Mutation

A mutation is a random change in one allele of the genotype of one individual. The mutation frequency is typically low and its goal is to maintain diversity in the population in an attempt to avoid premature convergence. The BN can be used for predictive reasoning or diagnostic abductive inference in which case any arbitrary subset of the variables may be initially instantiated (assigned a known value) during the inference process. Instantiated values are not changed by the mutation to guarantee that all individuals retain legal and meaningful genotypes.

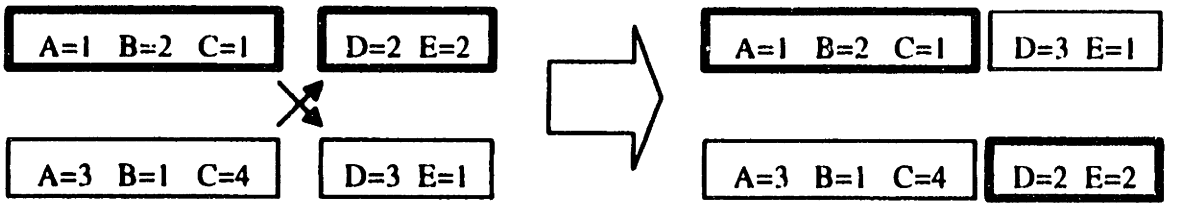


Figure 5-3. Traditional crossover in the string representation

5.4.7 Crossover in graph-based genetic algorithms

In genetic algorithms new individuals are obtained by crossover of selected individuals of the previous generation Two parents can create one or two children, the latter being preferred to avoid loosing potentially useful new individuals. The genotype of each new individual results from the combination of the genotypes of the parents. In traditional GAs

two parents are copied into two children, two positions are randomly chosen in the new strings and the genes located between the two positions are interchanged as shown in Figure 5-3.

When using graphs to represent genotypes, breaking a single link does not necessarily separate the graph into disjoint pieces. Instead, dividing the network into disjoint clusters requires breaking multiple links. Our strategy for identifying which links to break involves a focal node *C* and an expansion level *L*. Starting at the focal node, all nodes within *L* links distance of *C* are selected. The rest of the nodes makes up the second set specified by the partition. These nodes are then exchanged with the same nodes of another genotype. When $L=1$, node C and all its immediate neighbors (parents *and* children) are selected. The strategy satisfies the *compact block hypothesis*, *i.e.* at least one of the clusters should be made up of contiguous nodes in the network. Crossover in graphs is illustrated in Figure 5-4 where node E is the focal node and the expansion level is 1. More elaborate cluster construction algorithms might optimally identify loosely connected components but the computational cost would increase and the resulting overall performance would reflect the balance of both effects.

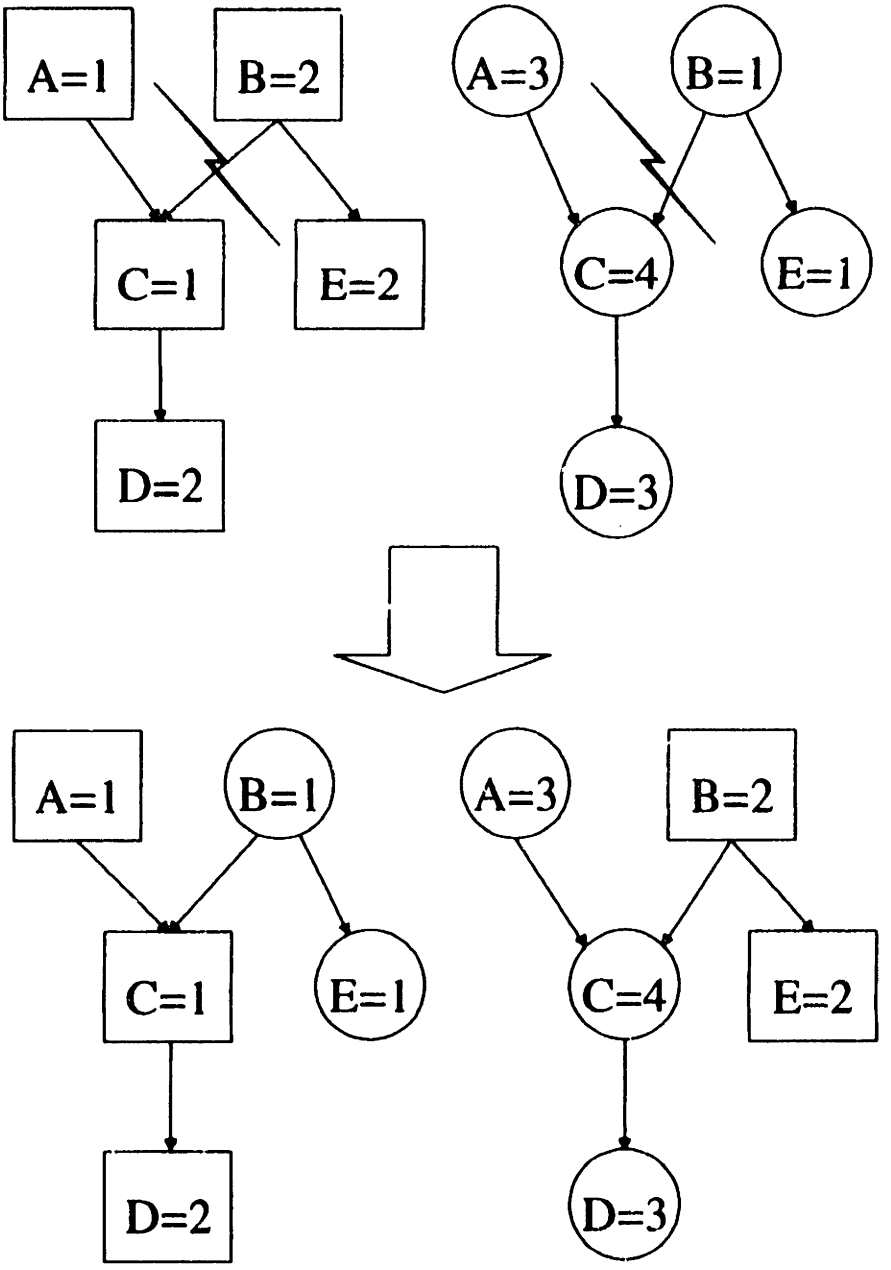


Figure 5-4. Crossover in the graph representation

The creation of new genotypes with a better phenotype is based on the combination of good network components. Therefore, by using a variety of boundaries at each crossover operation, the probability of providing a diverse pool of network components is higher and consequently more components should have an opportunity to be evaluated. Logically, it would be expected that convergence would be favored when the probability of breaking

any given link by some partition boundary is larger than zero. To achieve this, the focal node C is selected at random. There are many possible strategies to determine the expansion level. Experiments to compare the following 3 strategies are discussed below:

(1) The expansion level is a constant equal to L_{max}=*floor*(R/2) where R is the *longest directed path* in the network. Intuitively this strategy tries to create a partition with a diameter of about half the network size (diameter in a star shaped network, or longest directed path in an arbitrary network). A *path* in this context refers to an ordered, sequentially connected set of nodes which contains each node only once. The *longest undirected path* should not have an alternative shorter path between the two nodes which define its extremes. Note that the *longest directed path* is always equal to or smaller than the *longest undirected path* and that the set of *directed paths* is a subset of the set of *undirected paths* due to the constraint added by directionality.

(2) The expansion level is stochastically generated from a uniform probability distribution as an integer between 0 (only the center node is selected) and L_{max}=*floor*(R/2).

(3) The expansion level is stochastically generated from a specified non-uniform probability distribution.

For strategy (2) it is simple to prove that the probability of breaking any link is larger than zero: The probability of selecting any node as C, the center of the cluster is 1/S where S is the network size. The number of expansion levels is constrained to be an integer uniformly distributed in the interval $[0,L_{max}]$, therefore, the probability of selecting any integer in $[0, L_{max}]$ is $1/(L_{max}+1)$. For example, the probability of selecting C=(node i), with L=0 is $1/(S*(L_{max}+1))$. In this case, the boundary will only surround node i and all the links connected to node i will be broken. Since a link is connected to exactly 2 nodes, a lower bound for the probability of breaking any given link when L=0 is $2/(S*(L_{max}+1)) > 0$. It is also possible to break a link with a boundary resulting from an expansion level L

> 0. A link may be broken when C is r links away and L=r (a link is 0 links away from a node to which it is directly connected). However, a link will not be broken when both nodes to which the link is connected are included in the same partition set. The exact probability can be easily calculated for a given network topology by enumerating the cases corresponding to all L values.

The following examples compare several features of the outlined crossover strategies. The example described in Section 5.4.7.1 shows the effects of L on the distribution of broken links and identifies a special topology with links with a zero probability of being cut for any non-zero L. The example in Section 5.4.7.2 experimentally evaluates performance for three different strategies on a larger network.

5.4.7.1 Characterization of link distributions in crossover boundaries

This example shows how the choice of L affects the probability of cutting links during the crossover operation. We use a randomly generated network (BN3), shown in Figure 5-5, containing 20 nodes and 20 links. The longest directed path is 7, R/2=3.5, (L_{max}=3) and using strategy (1) with a constant L=3 or L=4 all the links can be cut at least once. With L=5 at least one link (H) is not cut. With L=8 only 4 links are cut (A and B by C=12 or C=19, and links L and T by C=1). As L grows from 0 to 7, those links with a previously high frequency receive a low frequency and *vice versa*. With small values of L, links closer to the center tend to have a higher probability of being elements of a boundary (links G, H, J, and N have the highest frequency for L=2), since they can be approached from both directions. Links on the network periphery are not favored (links A, B, L, and R have the lowest frequency for L=2). For L>9, the longest undirected path (n1-n12 or n1-n19), every boundary is drawn around the whole network and no links are cut (independently of the focal node selected) resulting in one single cluster containing all the nodes. If the selection of center nodes and expansion levels is distributed randomly, then

the expected frequency of breaking a link is the same for every link as indicated by the cumulative plot shown in Figure 5-6. This shows that an intermediate value of L has the best chance of cutting each link in the graph during crossover operations.

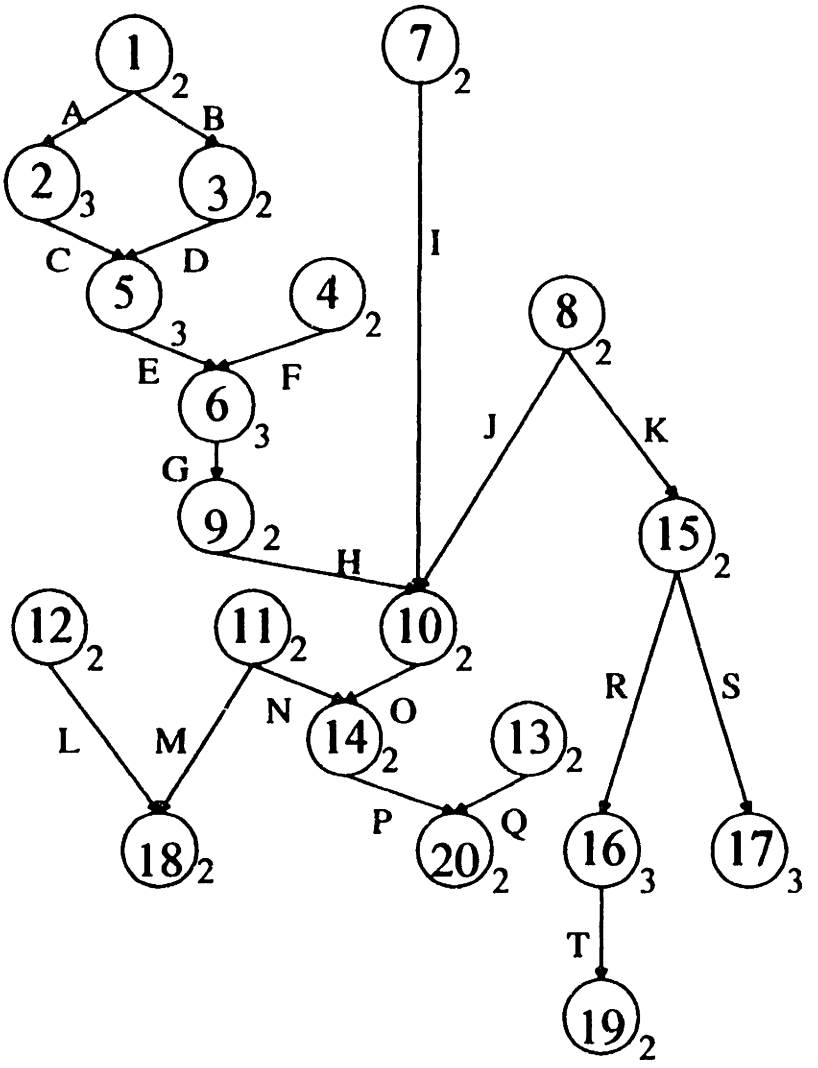


Figure 5-5. Randomly generated network topology (BN3)

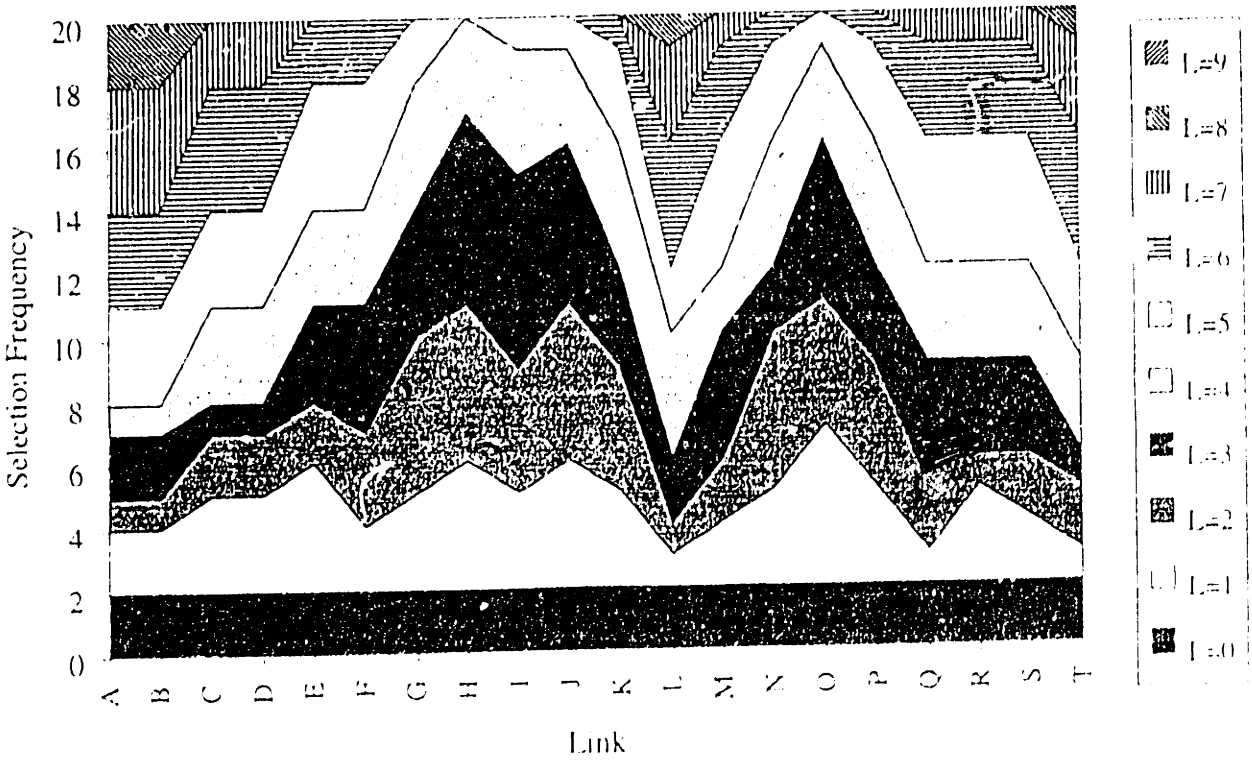


Figure 5-6. When both center nodes (C) and expansion levels (L) are randomly selected with a uniform distribution, the average selection frequency of each link is the same. The width of each band represents the number of times each link can be cut for the given value of L. For example, link H can be cut 6 ways given L=2, whereas link H cannot be cut using $L \geq 5$. Note that with L=9 no link can be cut.

Some links cannot be cut (except with L=0) if the distance between any focal node is the same to each of the two nodes connected by the link. As a result either none or both are contained in the cluster. It is therefore not possible to guarantee that all the links will be cut except when $P(L=0) \neq 0$. However, since all links can be cut by mutation we do not take special steps to deal with this special case of non-severable links.

5.4.7.2 Comparison of crossover strategies

Conventional genetic algorithms use strings to represent genotypes. However the graph representation is more natural when analyzing Bayesian networks. A discussion on the benefits of the graph representation, including experimental results, was presented in Section 4.1.

The overall efficiency of the genetic algorithm may be a function of the strategy used for the construction of the crossover boundaries. To compare the three proposed strategies to select L, experiments were conducted on the ALARM network described in Section 7, comparing performance for: (1) the expansion level was heuristically set to a constant value of 4, which corresponds to one half of the longest directed path in the network, (2) the expansion level was randomly selected at each crossover operation from a uniform distribution between 1 and 7, and (3) the expansion level was stochastically selected from a non-uniform probability distribution with a maximum at 4.

Table 5-1. Parameter sets selected for testing crossover partition strategies. G_{max} is the maximum number of generations, E is the size of the evolving population, M the mutation frequency, B the breeding selectivity, and A the average lifetime.

Parameter Set	G_{max}	E	M	B	A
PS22	200	50	0.5	0.6	5
PS10	200	300	0.01	0.2	3
PS6	200	75	0.25	0.7	5
PS21	200	250	0.25	0.1	7
PS13	200	50	0.1	0.5	6

Each of the crossover strategies was implemented using five different parameter sets with different choices for mutation frequency, average lifetime, etc. (shown in Table 1), resulting in 15 extended parameter sets each of which was run on 5 diagnostic cases

corresponding to the ALARM network. Each run was repeated 10 times recording (1) run time average (2) average of the best phenotype found in each run, and (3) number of times the run converged to the optimum. Results indicate that there is no significant difference between the three partitioning strategies (the fraction of times the optimum was found is 67/250 for the constant L, 65/250 for the uniform distribution and 68/250 for the peaked distribution. The standard deviations were similar for the three cases). Experiments indicate that the 3 proposed distributions of the expansion level (a parameter of the crossover operator) are similarly effective and therefore *any* of the three crossover strategies discussed can be selected. Note that all the strategies randomly select a focal node, an expansion level, and exchange a connected cluster and differ only in the strategies in the distribution of L, the expansion level. In the light of the above observations, we recommend the simplest, namely strategy (1), with a constant value of $L=floor(R/2)$, where R is the longest directed path in the network. The only disadvantage is that certain pathological links might not be breakable by the crossover, but all links are breakable via mutation.

5.4.8 Performance metrics

The performance of the genetic algorithm must be judged both on speed and accuracy. The latter can be quantified in terms of one of the following accuracy measures: (1) the presence or absence of the optimal solution (and the presence of the 2nd, 3rd, ... , nth) in the evolving population, (2) the distance between the best obtained individual and the optimal one, (3) the accumulated probability mass in the best n solutions, (4) the fraction of offspring which improves over its ancestors, (5) the average fitness in the population, (6) the fitness of the best individual. Criteria (1) and (2) can be used for algorithm development purposes by comparing with the result of a systematic enumeration (only on small problems) of all possible solutions. We use (1), (2), and (6) to measure performance of the algorithm.

5.5 Theoretical Analysis: Admissibility

This section shows that the graph-based GA satisfies the criteria proposed by Grefenstette (1991) to determine whether a genetic algorithm exhibits *implicit parallelism*. Grefenstette focuses on two algorithm design decisions: the way the user-defined objective function is mapped to a fitness measure, and the way the fitness measure is used to assign offspring to parents. A GA is then called *admissible* if it meets "what seem to be the weakest reasonable requirements" for the two algorithm decisions, and he shows that "any admissible genetic algorithm exhibits a form of *implicit parallelism*."

The growth rate gr(x) is a function that calculates the expected number of offspring for an individual x and can be written as a composition of three functions: $gr(x) \equiv select(u(f(x)))$ where f is the objective function, u is the fitness function, and *select* is the selection algorithm. In our algorithm, the objective function is the probability of a hypothesis (a full instantiation of the Bayesian network model), x: $f(x) = p(x)$. The fitness function u is usually defined to map the range of the objective function into a non-negative interval and to normalize it into the [0,1] interval, for example. In our algorithm the objective function f(x) corresponds to the phenotype and, since it is a probability, it is already guaranteed to be an element of [0,1]. Therefore it may appear that the identity function u_1 can be used as the fitness function, but this choice leads to premature convergence due to the large differences (typically several orders of magnitude) existing among the phenotypes (absolute probabilities) of hypotheses in the initial population. A reduction in the sensitivity to phenotype values, and better convergence properties can be achieved by using a transformed phenotype. An example of such a function follows:

$$u_2(f(x)) = \frac{\varepsilon_2}{(\log(p(x)+\varepsilon_1))^2 + \varepsilon_2} \qquad [5\text{-}4]$$

where ε_1=1e-12 and ε_2=0.01 are possible values which satisfy the requirements. The function is monotonic and defined for p(x)$\in$[0,1] and $u_2(0)$=0.01/144.01$\cong$0.00007, and u(1)$\cong$1.

The growth rate of an arbitrary subset of the space is defined as follows:

$$gr(A,t) \equiv \frac{1}{n}\sum_{i=1}^{n} gr(x_i) \qquad [5\text{-}5]$$

where { x_1, x_2, ... , x_n } are the representatives of A, an arbitrary subset of the search space in the population at time t. Functions of the form of u_2 have been empirically successful due to the reduced sensitivity to phenotype values. A fitness function is monotonic if and only if, $u(x_i) \le u(x_j)$ *iff* $f(x_i) \le f(x_j)$. A stronger condition, strict monotonicity, is satisfied if and only if the function is monotonic and if $f(x_i) < f(x_j)$ then $u(x_i) < u(x_j)$. A selection algorithm is monotonic if and only if $gr(x_i) \le gr(x_j)$ ift $u(x_i) \le u(x_j)$ and a selection algorithm is strictly monotonic iff it is monotonic and *if* $u(x_i) < u(x_j)$ then $gr(x_i) < gr(x_j)$.

The fitness function u_2 is strictly monotonic for f(x) in [0,1] and the selection algorithm in our approach is strictly monotonic since it assigns to each individual a probability of being chosen as a parent, proportional to its fitness. The algorithm is consequently *strict* (its fitness function and selection algorithm are both strictly monotonic). Grefenstette proves that in any *strict* GA algorithm, for any pair of subsets <A,B> represented in P(t), if $A <_p B$ then $gr(A,t) < gr(B,t)$ where $<_p$ indicates partial dominance among sets. B then grows exponentially faster than A. In other words, the algorithm used in our methodology is an *admissible algorithm* and therefore exhibits *implicit parallelism*. These results relate

f(x), u(z) and select(y) to the exponential allocation of trials heuristic advocated in 1975 by (Holland 1992).

5.6 Comparison of Parent Selection Criteria

Four networks are used to illustrate the algorithm and to explore its performance. These examples have different sizes and degrees of connectivity. The first network, BN1, is an abstraction of a singly connected belief network model for a section of a chemical plant (Rojas-Guzmán and Kramer 1992) whose topology is shown in Figure 5-7. This network has 13 nodes and its discrete probability distributions were obtained from behavioral descriptions. The number inside each circle is the node identifier and the small number outside the circle indicates the number of discrete states of the node. The discrete states of the variables are numbered from 1 to m, the number of discrete states the node can assume. From a systematic enumeration of the 12,288 points which comprise the search space, the best (most probable) system description was found to be D=(1221111211111) with a probability of 0.098. The ordering of the genes (variables) in D corresponds to the numbering of the nodes (i.e. the value of node i is in position i). The best 100 genotypes (0.8% of the total 12,288 possible genotypes) contain 62% of the probability mass.

The second network, BN2 has the same nodes as BN3, shown in Figure 5-3, but with 5-4 additional links (3-6, 7-15, 17-19, and 18-19). BN2 has 20 nodes and represents a significantly larger search space than BN1, with 7,962,624 possible states. BN2 is a multiply connected network with 5 undirected cycles, 15 binary variables and 5 ternary variables, and its most probable state is D=(21212212211222122211) with p=5.98e-5. BN3 has only one cycle and its optimal solution is D=(21121212212222122121) with p=4.42e-5. The fourth network, BN4, has no links among variables but has the same

search space size as BN2 and BN3, and its optimal solution is D=(22211312121222223212) with p=3.15e-5.

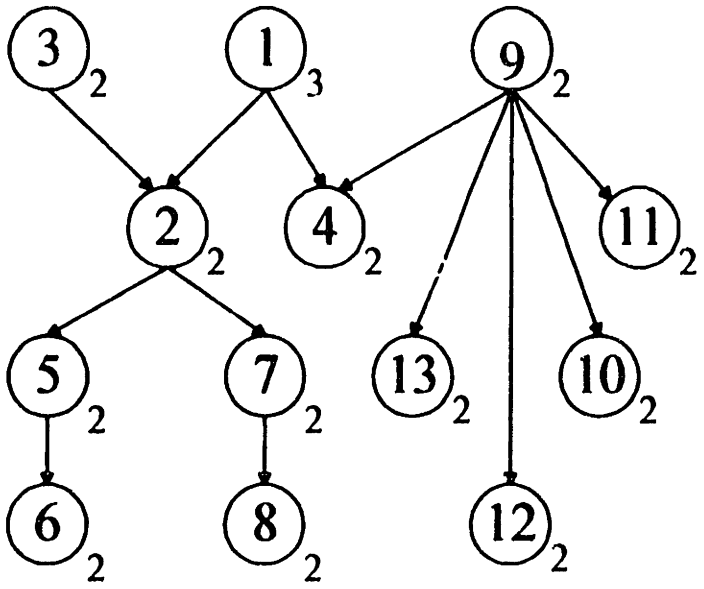


Figure 5-7. Topology for model BN1

Results from the proposed algorithm were compared with the solutions obtained by systematic exhaustive enumeration of all possible system states for BN1, BN2 and BN3. The best 50 solutions were stored in order in each run. For systematic enumeration on BN2 and BN3, each run required approximately 70 hours on a 486 33MHz PC running a C++ implementation. The best solution for BN4 is simply the product of the largest prior probability of each node. Results from 135 runs are summarized in Table 2a, 2b, and 2c. In all the runs, the average lifetime was set to 5 generations, which means that 20% of the individuals were replaced in each generation. Three parent selector criteria were used: (1) a uniform probability distribution, (2) a distribution proportional to the individual phenotype, and (3) a distribution proportional to a transformed value of the phenotype, where the transformation function is shown in equation [5-4]. The mutation frequency

was 0.025 for runs on BN1, and 0.075 for runs on BN2, BN3, and BN4. Each run of the GA required less than one minute.

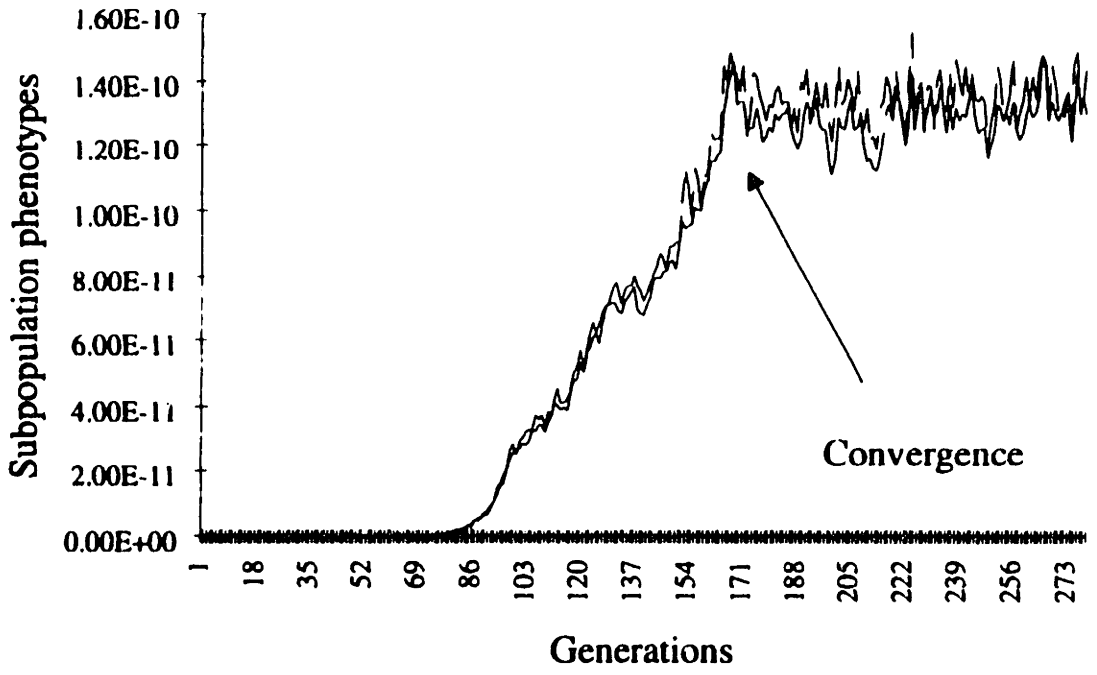


Figure 5-8. Typical phenotype evolution

In Table 2a, 2b, and 2c, TOP N = X% means that in X% of the runs, a solution among the top N was obtained. Note that the set containing the top 50 solutions includes only 0.00063% of all the possible solutions for BN2. *Rank* refers to the average rank of the solutions to which the algorithm converged (rank=1 corresponds to the optimum). G indicates the average generation number at which the converged solution was first created. G_c corresponds to the average generation number at which convergence was reached. The increase in the sum of the phenotypes of the population is shown in Figure 5-8 for a typical run. Figure 5-9 shows the evolution of the best phenotype for BN2. After a good solution is found, the population will take a few generations to converge. The point at

which the probability mass as a function of generations in becomes flat corresponds to convergence, the population is uniform and high frequency variations are due to mutations. EvalG indicates the number of individuals evaluated before G, and similarly, EvalGc is based on G_c. Note that Number of evaluations = initial population + Generations * (births per generation + mutations per generation). The size of the evolving population and the number of runs used are also indicated. Calculations to perform inference on networks with instantiated nodes are similar, except that mutations are not allowed on instantiated nodes. Note that complexity is a function of the number of non-instantiated nodes only.

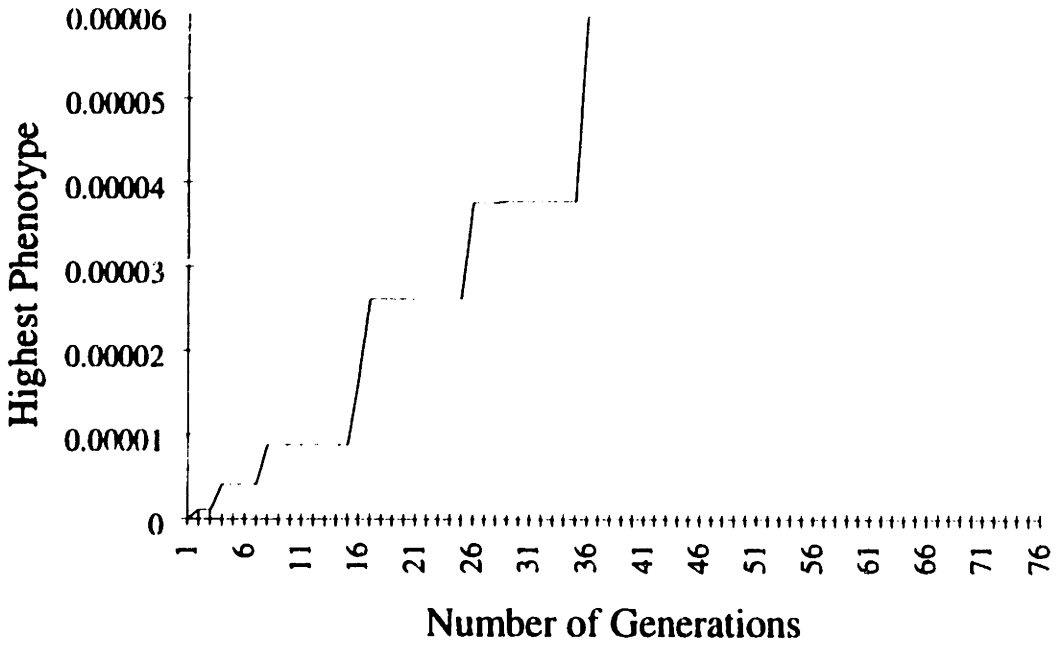


Figure 5-9. Phenotype of the best individual as a function of generations for BN2.

Table 5-2a. Results when parents are uniformly selected. The probability of selecting a genotype from the *breeding population* is the same for all genotypes.

Model name	BN1 (0 cycles)	BN2 (5 cycles)
Space size	12,288	7,962,624
TOP 1	95%	30%
TOP 10	100%	60%
TOP 50	100%	100%
Rank, std. dev.		12.8, 16.3
G, std. dev.	< 50	37.4 , 9.4
G_r	< 50	49 (estimated)
EvalG	< 1310	785
$EvalG_r$	< 1310	1006
% Evaluated	0.0	0.013
Evolving population	110	75
Total Runs	20	10

Table 5-2b. Results when the probability of selecting a node from the *breeding population* is proportional to its phenotype.

Model name	BN1 (0 cycles)	BN2 (5 cycles)
Space size	12,288	7,962,624
TOP 1	30%	0%
TOP 10	100%	45%
TOP 50	100%	75%
Rank	1.05	
G, std. dev.	< 50	28.6, 12.5
G_r	< 50	41 (estimated)
EvalG	< 1310	618
$EvalG_r$	< 1310	854
% Evaluated	< 10.7	0.011
Evolving population	110	75
Total Runs	10	20

Table 5-2c. Results when the probability of selecting a node from the *breeding population* is proportional to a logarithmic transformation of its phenotype.

Model name	BN2 (5 cycles)	BN3 (1 cycle)	BN4 (0 arcs)
Space size	7,962,624	7,962,624	7,962,624
TOP 1	20%	8%	100%
TOP 10	88%	44%	100%
TOP 50	100%	56%	100%
Rank, std. dev.	7.2, 10.8		1.0, 0.0
G, std. dev.	38.4, 15.0	45.6, 17.9	55.6, 16.9
G_c, std. dev.	49.9, 11.6	58.2, 15.8	75.1, 17.9
EvalG	805	941	1131
EvalG_t	1023	1180	1501
% Evaluated	0.013	0.015	0.019
Evolving population	75	75	75
Total Runs	25	25	25

A random search to obtain the optimum with a probability of 0.20 (as in BN2 with the transformed phenotype selector) would have required evaluating 1.6 million points at each attempt, a significantly larger amount than the approximately 1000 evaluations required by the GA. The complexity growth of the algorithm deserves careful attention and a larger amount of experiments is required to obtain significant statistics. However, preliminary results comparing BN1 and BN2 are encouraging. BN1 is singly connected and represents a space of 12,288 states, whereas BN2 has 5 cycles, and has a significantly larger search space (7.96 million states). Nevertheless, convergence to solutions among the top N (for small N) required a similar number of point evaluations (around 1000). In the ALARM network example discussed in Section 7 an average of 12,000 points were evaluated from a total space size of 1.73×10^{16}, corresponding to a significantly smaller percentage than in networks BN1 and BN2.

Intuitively the sensitivity to the number of cycles in the GA approach would be small because calculations do not directly depend on the number of cycles. Nevertheless

performance is expected to be affected by the degree of connectivity in the network, but not particularly by the number of cycles. By comparing results from BN2 and BN4 using the transformed parent selector it is clear that in the extreme case of 0 arcs the problem is simpler and a greedy algorithm (in which node values are assigned, one node at a time, trying to maximize the phenotype at each assignment) would be more efficient, as expected due to the absence of variable interactions and the resulting null connectivity. However, in connected networks a greedy algorithm may tend to converge to local optima.

Results indicate adequate convergence when parents are selected with a uniform probability and show premature convergence when the parent selection uses the proportional criteria due to the large differences in probabilities of solutions (several orders of magnitude), especially at early stages in the evolution. A better parent selection which reduces sensitivity to phenotype values but still gives preference to individuals with higher phenotypes is based on the use of a transformed phenotype. Experimental results from the three parent selection criteria on BN2, shown in Tables 5-2a, 5-2b, and 5-2c, indicate that the inverse square logarithmic transformation of the phenotype reduces premature convergence, especially at the early stages of evolution, when differences among individuals are typically of several orders of magnitude.

5.7 Performance on the ALARM Network

Algorithm performance is a function of many parameters. However, since the theoretical basis of genetic algorithms is still under development, the relation between parameters and performance is not yet completely understood. Experiments were conducted to model the effects of the evolution parameters on running time and diagnostic accuracy. These

models can be used to predict performance and to guide the search for an optimal set of parameters.

5.7.1 Model description

The Bayesian network used for this set of experiments is the ALARM monitoring system (Beinlich, *et al.* 1989) used in a medical diagnostic application. The network has 37 nodes, of which 8 are diagnostic nodes, 16 represent findings (or measured variables), and 13 are intermediate nodes. The topology of the network is shown in Figure 5-10. One hundred scenarios were used from a database containing 20,000 scenarios stochastically generated by (Herskovitz, 1991; Cooper, 1992) using the logic sampling method described in (Henrion, 1988). Each scenario consists of a completely instantiated network, in which each variable has been assigned a value.

5.7.2 Experiments

In the first set of experiments, 100 cases from the database were used. Given only the 16 values corresponding to the finding (measured) nodes, the algorithm was run trying to find the most probable set of variable-value assignments for the other 21 nodes. The diagnostic hypotheses obtained by the GA had in all the cases a higher probability than the stochastically generated scenario. This means that the *actual* explanation (as represented by the scenario in the database) is not the most likely explanation of the findings. This supports the strategy of storing the top N hypotheses generated throughout the evolution process to be considered by the user either for further refinement or as a basis in the decision making process.

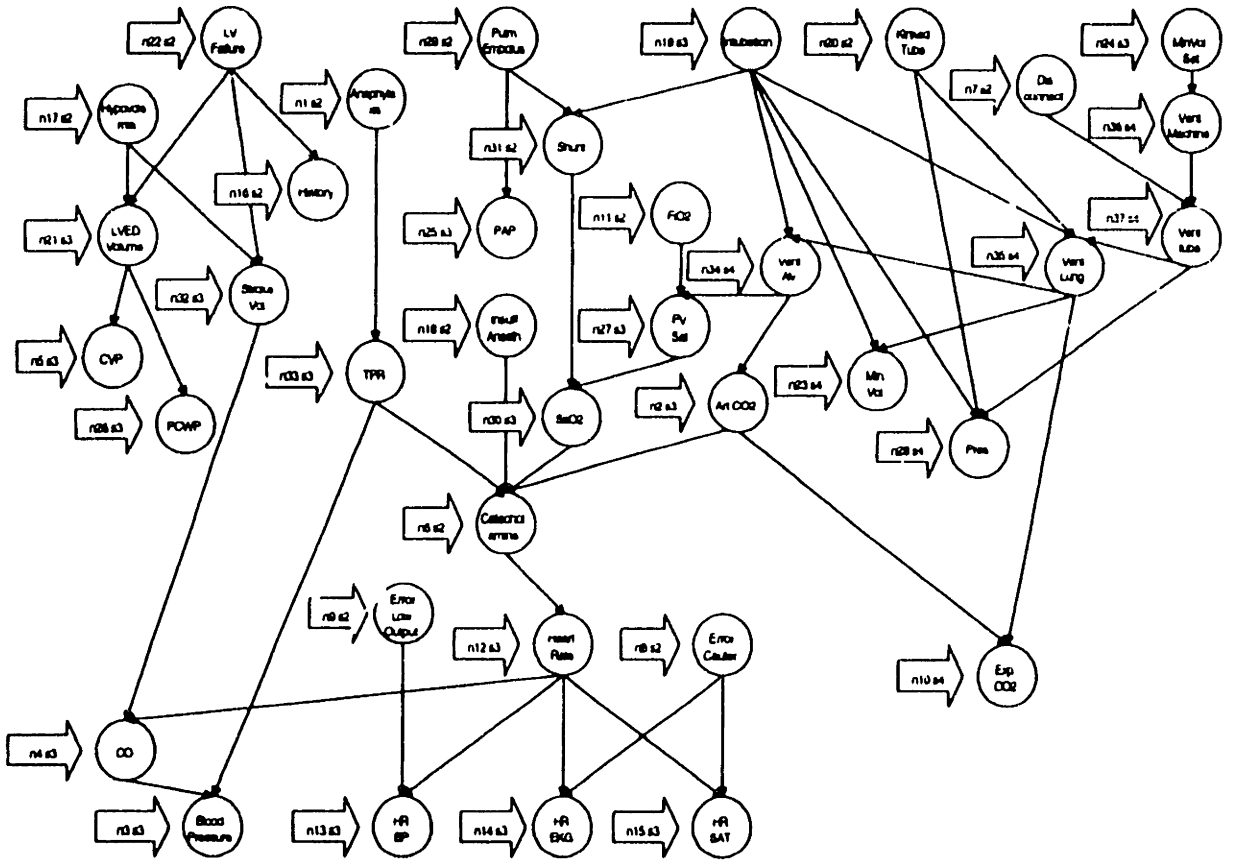


Figure 5-10. Topology from the ALARM Bayesian network from Beinlich, *et al.*

In the second set of experiments five parameters were independently varied: the evolving population size (E), the breeding selectivity (B) which specifies the fraction of the population to be used as a pool for parent selection, the mutation frequency (M), the fraction of the population which suffers a mutation at each generation, the average lifetime (A) whose inverse specifies the fraction of the population replaced at each generation, and the expansion level (L) that specifies the size of the cluster used for partitioning the network during the crossover operation.

For each set of parameters, two performance metrics were recorded: total inference time (T), the best phenotype found (PH), and time per generation (TPG). Twenty four sets of

parameters were run on 5 diagnostic cases each. Each of these 240 experimental points was repeated 10 times, yielding a total of 1200 points.

5.7.3 Regression model

A second degree polynomial was used to model the effect of the five GA parameters on total inference time and best phenotype. The coefficients in the following model were adjusted by linear regression based on the 1200 runs: $(T,PH) = f(E, E^2, B, B^2, M, M^2, A, A^2, L, L^2, E*M, E*B, E*A, E*L, M*B, M*A, M*L, B*A, B*L, A*L)$.

Parameters, their products, and time were normalized to have a zero mean and unit standard deviation. Since the probability of the optimal hypothesis can be orders of magnitude different for different scenarios, phenotypes were normalized first by dividing the logarithm of the probability of each hypothesis by the logarithm of the probability of the optimal hypothesis. The optimum value for each case was estimated to be the best obtained hypothesis from 250 runs using different parameter sets and in each a different starting population. Next, the average was subtracted and the result divided by the standard deviation resulting in a set with mean zero and standard deviation one. Parameters were varied in the following intervals: $E \in [50,500]$, $B \in [0.1,0.9]$, $M \in [0.0,0.5]$, $A \in [2,10]$, $L \in [2,5]$.

As shown in Table 5-3, there is a difference in the difficulty of each of the diagnostic cases. For case 300, the optimum was found in 47.5% of the cases, while in case 310 only in 20.4% of the runs. As will be shown later, a better set of parameters was found that obtains the optimum in about 80% of the runs for all the cases.

Table 5-3. Summary of performance on different cases and parameter sets. PH_{max} is the highest phenotype generated in all the runs for a given case

and is an estimate of the optimum for that case. PH_{max} fraction shows the number of times in which PH_{max} was generated by the GA, and $\%PH_{max}$ the percentage. The sum of ratios serves to quantify how close to the optimal were the best generated solutions. %Sum compares the obtained ratio with the ideal ratio. Ideally the Sum of ratios would be 240 and %Sum would be 100. Note that these experiments do not correspond to the best set of parameters found presented in Section 7.4.

Case	PH_{max}	PH_{max} fraction	$\% PH_{max}$	Sum of ratios $\sum_{i=1}^{i=240} PH_i / PH_{max}$	%Sum
300	5.59E-11	114/240	47.5	157	65.4
310	7.03E-13	49/240	20.4	64.1	26.7
320	2.333E-08	60/240	25	101	42
330	5.15E-11	57/240	23.8	87.3	36.4
340	2.604E-11	70/240	29.2	109	45.2

Time per generation (*TPG*) can be approximated by the following function with a regression coefficient of 0.96:

$$TPG = 6.10\text{x}10^{-3}\, E - 9.6\text{x}10^{-4}\, E^*A + 1.89\text{x}10^{-3}\, E^*B + 1.67\text{x}10^{-4}\, E^*L + 2.58\text{x}10^{-3}\, E^*M \quad [5\text{-}6]$$

The five terms retained in the time per generation function (neglecting the constant) are related to specific tasks within the algorithm. The size of the population increases the size of the initial space sample. The term E^*A has a negative contribution: with larger average lifetimes, a smaller number of new individuals is required at each generation, thus saving time. The term E^*B corresponds to the subpopulation from which parents are selected. The selection process involves sorting this subpopulation, applying a transformation to

their phenotypes, generating a probability distribution to favor the selection of better phenotypes, and finally using the distribution to select the parents to be used to generate new hypotheses for the next generation. A higher B implies a larger subpopulation and consequently a larger number of calculations. A very small B would save time but might result in premature convergence and overall lower performance. The product of E^*L reflects the overhead of performing crossover with higher expansion levels. The time spent defining a network partition is directly proportional to the number of crossovers (which in turn is proportional to E) and to the expansion level used at each crossover. The E^*M term represents the number of mutations per generation, each of which takes a constant amount of time. The constant term is, as expected, close to zero.

The total running time, T, (in seconds) can be approximated by the following function with a regression coefficient of 0.948:

$$T = 0.293E + 0.163E^*B - 0.021E^*A - 0.04E^*L + 0.482A^2 + \\ + 10.74L - 60.75M^2 - 4.129A - 0.992L^2 + 56.76M - 22.332 \qquad [5\text{-}7]$$

The overall inference time is also a function of evolving population. In contrast with time per generation, total time has quadratic terms in M, L, and A, suggesting the existence of competing effects. A larger M tends to increase exploration of the space versus exploitation of already found blocks increasing the probability of finding a certain block but also of destroying good blocks. Small expansion levels have the lowest cost in time, but reduced information exchange between networks (in the extreme case of L=0 would be equivalent to a mutation). Large expansion levels tend to create unbalanced partitions and, in the extreme case (not modeled by this function), no information exchange, thereby decreasing the efficiency of the algorithm. Large average lifetimes reduce time per

generation but tend to slow down the overall evolutionary process by causing a small birth and death rate.

Finally, the best phenotype obtained can be approximated by the following function with a regression coefficient of 0.48:

$$PH = -0.0012E + 1.74 \times 10^{-6} E^2 + 0.026L^2 - 0.102B + 0.00559L - 0.174M + 1.557 \quad [5\text{-}8]$$

PH in this function refers to the logarithm of the phenotype normalized with the logarithm of the optimal hypothesis. Thus $PH=1$ when the optimum is found, and $PH>1$ are sub-optimal solutions. The quadratic effects on *E* and *L* suggest the existence of competing effects. Larger populations tend to yield higher phenotypes partly because the pool of compact blocks is larger in larger populations. The increased search exploration due to *M* is shown (for the range of values used in the experiments 0 to 0.5). With higher values of *M* a decrease in performance is expected due to the disruptive effect of excessive mutation. High *B* reduces the risk of premature convergence. An intermediate value of *L* will achieve balanced sections (closer to 50%/50%) when performing crossover, with either extreme being less useful. This function suggests to use high values for *B* and *M*, and low values for *L*. Larger values of *E* tend to increase *PH*, since the E^2 term dominates.

These experiments show that performance is not directly correlated with the number of generations, but with the size of the evolving population and other parameters. The total running time is not a function of the diagnostic case. The models for individual cases are very similar to the models obtained from using all the normalized data simultaneously, which supports the use of these models for other new cases. The fact that the phenotype exhibits higher variability than time is reflected in a lower regression coefficient for

phenotype models. This variability is significantly decreased as the parameter set approaches the optimum set.

5.7.4 Parameter optimization

Performance is measured by the dual objectives of total inference time and diagnostic accuracy. Lower time and more accurate diagnoses are clearly preferred, but when selecting an optimum for the parameter set, it is necessary to specify a utility function that encodes the tradeoff between an improvement in time versus and improvement in accuracy. In general, however, the optimum should be obtained in a high fraction of the runs with the minimum inference time.

The best parameter set found (E=150, M=0.33 , or M_g=0.009, B=0.9, A=2, L=4) locates the optimum with an average and median of 80% of the runs, with a maximum of 90% and a minimum of 70% for different diagnostic cases.

5.8 Comparison of MPEs and Posterior Distributions

Diagnostic reasoning (or abductive inference) under uncertainty can be probabilistically modeled and approached as an optimization problem. This paper experimentally compares two definitions of the problem, and quantifies differences in the resulting diagnostic conclusions on a real-life model.

5.8.1 Introduction

Diagnosis can be mathematically defined in different forms. In addition, given a mathematical definition, diagnostic reasoning can be formally modeled within a probabilistic or a logical framework. In this work we have focused on probabilistic

models, specifically Bayesian networks (Pearl, 1988). This section experimentally compares diagnostic conclusions from two common definitions of diagnosis given a set of evidence: finding the *most probable explanations* or MPEs, (Pearl, 1988; Shimony, 1994, Rojas-Guzmán and Kramer, 1993) and finding posterior distributions, one type of calculations that can be done using a popular exact algorithm (Lauritzen and Spiegelhalter, 1988).

The first approach requires finding $p(v_i|y)$ a posterior distribution for each variable v_i given y, a set of evidence. In the second approach, determining a MPE (also called a maximum *a-posteriori* probability, MAP, instantiation of all the variables) given the evidence, the objective is to find H, the *best* set of discrete values for the unknown variables (given the values of a subset of known variables), in other words, to maximize the probability of the inferential hypothesis given the evidence, as shown in equation [5-1].

Diagnostic conclusions obtained using different optimality criteria may be different. Poole (1990) argues that the definition of an optimal diagnosis should take into account the utility of outcomes and what the diagnosis is used for. Poole concludes that there may be no *a priori* ontological definition of optimal diagnosis and that it is epistemological and situation-dependent. Obtaining a MPE is *not necessarily the same* as finding a system description based on the posterior distributions of its variables, as discussed by Pearl (1988) and Poole (1990). How different they are, and how often they are different, are questions of interest when dealing with larger problems and when selecting an inference algorithm. The following experimental comparison of MPEs and posterior distributions quantifies the differences between posterior distributions and MPEs using a well known model (Beinlich, 1989).

5.8.2 Description of experiments

Posterior distributions were determined using an exact algorithm by Lauritzen and Spiegelhalter (1988) implemented in ERGO, and MPEs were searched using an approximate graph-based genetic algorithm for Bayesian networks developed by Rojas-Guzmán and Kramer (1993) and implemented in GALGO.

One hundred cases were selected from a database of stochastically generated scenarios on the ALARM model, (Herskovitz, 1991; Cooper, 1992). The set of measured variables contained in each scenario were given as input evidence to both ERGO and GALGO. The same Bayesian network model (topology and probabilistic parameters) was used by both implementations. The output from ERGO contains one probability distribution for each variable, and a system description was assembled by selecting, for each variable, the state with the highest probability. The probability, P, of the full system description was calculated by GALGO using equation [5-3]. Note that the probability of a scenario represents $p(x, y)$, the probability of the unknown variables *and* the measured variables. $p(x,y)$ is sufficient to compare and rank diagnostic hypotheses based on the same evidence, and $p(x|y)$, the probability of x given y, can be calculated if $p(y)$ is known, using Bayes theorem.

Using the same set of evidence, GALGO was used to estimate the optimum MPE given the evidence. Since it is an approximate algorithm, each case was run 10 times and the highest probability description was used for the comparison. The full system description was evaluated using the same metric function used to evaluate system descriptions obtained with ERGO. Appendix A summarizes the results.

5.8.3 Results

As expected, the diagnoses from both methods are not always the same, even when given the same evidence. The probability of the scenario obtained by GALGO was higher than the one obtained from ERGO, in 42% of the cases. Both algorithms yielded the same results in 57% of the cases, and ERGO found a higher probability than GALGO in 1% of the cases.

The fact that the probabilities are the same for two scenarios does not guarantee that the system descriptions are identical, since more than one system description can have the same probability as illustrated by case 301. In some cases the probabilities from both implementations were very close, as in case 304 where their ratio is 1.11, or large, as in case 317 with a ratio of 303,860. Often, GALGO obtained several system descriptions with higher probability than the ERGO distribution for the same case.

5.8.4 Conclusions

The theoretical argument that definitions of diagnosis *are not necessarily the same*, does not say how often, if ever, diagnoses would differ in practical applications. Toy examples can be easily constructed to illustrate the difference, but it is not clear from them whether they constitute isolated or typical cases. Experimental results show that on real life models, different definitions of the diagnostic problem produce significantly different results, and therefore the problem should be carefully analyzed to select the best definition for a given application.

Using posterior distributions for systems where multiple faults may be present would often yield suboptimal diagnoses, even with exact methods. When the overall system description is required, a global system description should be determined.

Determining the location within the network of variables with different values in a suboptimal solution can help assess the benefit of a hybrid algorithm that performs a systematic search of a bounded depth on the approximate solution obtained by GALGO. If the nodes are disconnected, a greedy search of depth of one level would find the optimum. The degree of clustering of the nodes can help estimate the required depth to find the optimum, and experimental results are necessary to determine the maximum level of depth that would be practical.

5.9 Discussion

For large multiply connected networks, exact inference may not be feasible, rendering approximate algorithms an attractive alternative. Results have shown that graph-based genetic algorithms constitute a feasible approach to perform inference in multiply connected Bayesian networks for real systems. The proposed method yields sometimes sub-optimal (and most often optimal) solutions in tractable times and reduces the strong sensitivity to the number of undirected loops in the network which makes exact methods infeasible for large, complex models.

The proposed method of graph-based GA was compared to conventional string-based GA. The graph structure preserves compact building blocks which persist from generation to generation. In general it is not possible to map a graph to a string and preserve compact blocks during crossover. Experiments have confirmed the benefits of having *semantically close* compact blocks (resulting from representing genotypes as graphs instead of strings).

Convergence to solutions among the top N (for small N) required a similar number of point evaluations for networks of similar size (BN2, BN3) but with a different degree of

connectivity and number of loops. The ALARM network required a smaller percentage of genotype evaluations than smaller networks (BN1 and BN2). By comparing results from BN2 and BN4 it is clear that in the extreme case of 0 arcs the problem is simpler and a greedy algorithm would be more efficient. Results indicate adequate convergence when parents are selected with a uniform probability (the probability of being selected is 1/(size of breeding population) and is not a function of the phenotype) and show premature convergence when the parent selection uses the proportional criteria due to the large differences in probabilities of solutions (several orders of magnitude), especially at early stages in the evolution.

There is a class of problems which is hard for GAs in general. From a practical and theoretical standpoint it is of interest to study the BN and GA combination proposed in this paper to determine under which conditions hard problems arise. A problem is *deceptive* if certain hyperplanes guide the search toward some solution or genetic building block that is not globally competitive (Goldberg 1989). Whitley (1991) showed that the only problems which pose challenging optimization tasks are those that involve some degree of deception. According to Davidor (1991) three elements contribute to GA-hardness: (1) the structure of the solution space, (2) the representation of the solution space, and (3) the sampling error which results from finite and often small population sizes. By changing representations, GA-hardness may be diminished or avoided.

Davidor (1991) proposed the use of a statistic called *epistasis variance* to quantify the non-linearity in a representation. Epistasis (Klug 1986) refers to gene interactions. Some degree of interaction is necessary to guide the search in the space, but when interactions are too strong the problem is hard. Zero epistasis would occur in a network without links such as BN4. In that case the best genotype can be found by a simple greedy algorithm following an approach similar to (Koutsoupias 1992) starting from a random position and

changing genes, one at a time, to the allele which causes the largest improvement to the individual fitness. High epistasis would occur in a network with each node directly connected with all other nodes. A meaningful improvement in the fitness is expected to occur only when all the nodes are simultaneously moved to the optimal. Fortunately, the structure which results in BNs has usually enough links to guide the search, and is very seldom fully connected. It is this local modularity (gene interactions are limited to immediate neighbors) which supports the notion of small compact blocks making a GA approach attractive over a greedy algorithm.

5.10 Summary

This study was motivated by the requirements of real-time diagnostic reasoning tools for large, complex, and dynamic systems with strong non-linear interactions. The algorithm has an *anytime* performance, meaning that it can continue the search while time is available, or be stopped at anytime to yield the best solution found so far. Experimental and theoretical results contribute to a better understanding of the processes underlying the simulated evolution algorithm and confirm the speculation based on intuitive arguments that the graph representation is appropriate for genetic algorithms operating on Bayesian networks. The theoretical analysis was conducted to place a newly developed algorithm in the context of previously developed research. From the results by Grefenstette (1991), since the proposed algorithm is *admissible*, it exhibits *implicit parallelism*. Experiments complemented theoretical arguments to determine good parameter sets, strategies for parent selection, crossover between networks and to define a phenotype transformation for use during the selection step. Tests were performed on models of real interest and size such as the ALARM network indicating the feasibility of the approach, quantifying its performance, and comparing results when using different definitions for optimal diagnosis.

Chapter 6

Industrial Implementation

The methodology described in this paper has been implemented in a real-time on-line system and is being tested on an industrial chemical process at an industrial site to remotely monitor and diagnose air separation plants.

6.1 Introduction

Unmanned industrial nitrogen plants are currently being remotely monitored and diagnosed using an innovative technique involving genetic algorithms applied to Bayesian belief networks. The GALGO system has been implemented at a remote monitoring and operations center and is being used for proactive detection of incipient faults and for reactive diagnostics. When data is received from a remote facility, the genetic algorithm determines the best (most probable) globally consistent description(s) of the state of the process. This output is then converted to an English syntax natural-language explanation of the problem for the operator.

The diagnostic system has two main components, a semi-quantitative model of both normal and abnormal operation and an inference algorithm. The model is a Bayesian network and the inference algorithm is a non-conventional graph-based genetic algorithm.

The GALGO approach is suitable for integrating knowledge from different sources, including the process flow sheet, historical data, behavioral descriptions and troubleshooting expertise from plant designers and operators. The system diagnoses single or multiple faults, does not require a fixed set of evidence, and provides a probabilistically ranked set of diagnostic hypotheses. In addition, the modularity and local interactions in the network representation simplify knowledge acquisition, model construction and maintenance.

Industrial processes can improve overall operation efficiency with the support of on-line diagnostic tools by detecting incipient faults, accurately diagnosing root causes of malfunctions, and reducing downtime with faster diagnoses.

A methodology based on on-line data should have the flexibility to also take advantage of off-line data and on-site observations when available. The availability of on-line data is a requirement for remote real-time on-line diagnosis. A high degree of redundancy in measurements simplifies the diagnostic task but is expensive and often only justified in systems with potentially catastrophic malfunction scenarios such as nuclear power plants. Hydroelectric and fuel-based power plants, chemical and petrochemical plants and manufacturing facilities tend to have a lower degree of redundancy in their measurement systems.

The probabilistic diagnostic methodology presented in this paper uses a modeling framework based on Bayesian networks. Diagnosis is performed on the model using a non-conventional graph-based genetic algorithm on the Bayesian network model.

6.2 On-line Diagnosis

The GALGO system is an automated reasoning tool based on a genetic algorithm approach to monitor and diagnose faults, disturbances and equipment failures based on remotely available local-sensor data acquired by a computer. Diagnosis can be refined with on-site observations or requested additional measurements.

GALGO has been implemented at a remote monitoring and operations center and is being used for detection of incipient faults and for diagnosis of shutdown situations. When data is received from one of the unmanned remote facilities, the genetic algorithm at the operations center determines the best (most probable) globally consistent description(s) of the state of the system, a reasoning task called abductive inference. The best N system descriptions are then translated to a natural-language explanation of the problem for the operator. The explanation includes the root cause of the malfunction for corrective action and selected variables contained in the causal path relating the root cause to the observed symptoms. Explanations simplify the verification of the diagnostic hypotheses and increase their credibility. Ranked hypotheses are produced and focus operators attention on the most likely fault(s) of the system.

6.3 Fault Suppression Strategy

The top N hypotheses obtained in one run from the genetic algorithm may be similar variations of one faulty scenario, which may reduce the usefulness of a ranked set of hypotheses. The top N different hypotheses (containing clearly distinct patterns of faults) are especially useful in an industrial setting when interactive usage includes the possibility of conducting on-site tests to verify suspected faults.

One strategy that is capable of providing different hypotheses involves first, generating the most probable hypothesis, second suppressing a fault from the diagnosed scenario, and then generating another most probable hypothesis (that does not contain the initially suspected fault). Each additional iteration provides an additional hypothesis. When used in an interactive mode, the operator can generate and test hypotheses sequentially. The iterative process concludes when the correct diagnosis is found or when the most likely scenario does not contain faults.

6.4 Implementation

The diagnostic methodology described in this paper has been implemented and tested in an air separation plant. A nitrogen production plant was selected to demonstrate the feasibility of the methodology and to provide guidelines for usage in similar plants. The role of the GALGO system as part of the plant operation is shown in Figure 6-1.

From the academic perspective this case study has shown the feasibility of the methodology, including the Bayesian network knowledge representation, the knowledge acquisition technique, and the near-optimal graph-based genetic algorithm. From the industrial standpoint this work constitutes another step in the automatization of industrial plants to achieve better operation and illustrates the tangible benefits of computer-aided systems. The industrial tests illustrate the technology transfer process and the mutual benefits of interaction between industry and academia.

This project illustrates a research cycle that started with the motivation of improving process operation through better diagnosis, involved the analysis of process operations in imperfect real systems and included the critique of alternative approaches. A crucial step, the synthesis of a methodology for diagnosis required the selection of a modeling

framework and the development of an algorithm to perform inference on the resulting mathematical problem. The final step involved the implementation of the methodology in a computer program, tests on abstract problems and the development of an industrial application. The industrial implementation has contributed to the evaluation of the methodology and has suggested directions of further research.

6.5 Plant Description

The plants used in this project are unmanned computer controlled units that produce nitrogen with a proprietary membrane system using atmospheric air as the raw material. This project aims to extend the remote capabilities of the monitoring system by partially automating diagnosis and providing additional support to the expert operator.

Membrane plants constitute the least complex option when compared with other non-cryogenic processes such as pressure swing adsorption and vacuum pressure swing adsorption. The main advantages of membrane processes are compactness and simplicity, which imply a smaller capital cost, especially at smaller capacities and moderate purities.

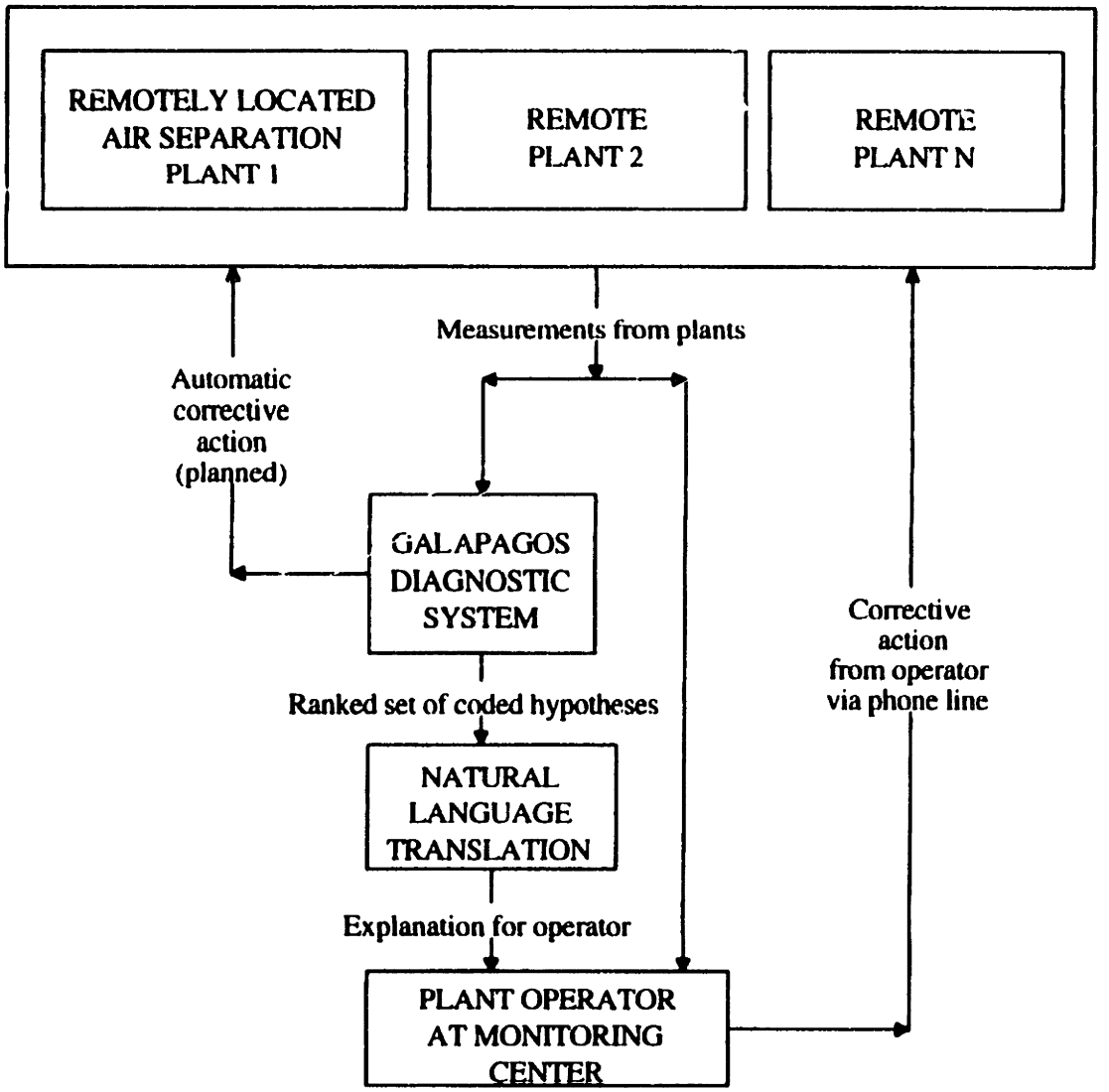


Figure 6-1. The probabilistic diagnostic system

The membrane systems are based on the selective permeability of air through composite polymeric membrane fibers. Most membranes are hollow asymmetric or composite fibers. The composite material is a dense membrane at the surface of the fiber supported by a porous fiber wall. The fibers are manifolded in a polymeric tube sheet and are assembled into cartridges which are then inserted into shells. Typically a set of membrane modules are installed in parallel. The separation is based on a higher solubility and diffusivity of oxygen in the composite material resulting in a faster permeation of oxygen through the membrane. Nitrogen remains as the retentate and air enriched in oxygen constitutes the

waste stream of the process as shown in Figure 6-2. The permeabilities for water and carbon dioxide are very high and accumulate in the permeate, leaving less than approximately 1 ppm of each in the retentate nitrogen product. Purities higher than 99.95% can be reached using a process whose main elements are a membrane and a compressor. By adding a catalytic deoxygenation system in series with the membrane, purities of 99.9995% of N_2 can be reached.

The separating capability of the system depends on the properties and geometry of the separating layer of the membrane, the process arrangement and the process conditions, mainly temperature and pressure. Process operating pressure and temperature are selected based on a tradeoff between power consumption and membrane area to achieve the minimum nitrogen product cost. Precise control, continuous monitoring, and accurate and fast diagnoses ensure a reliable and efficient operation.

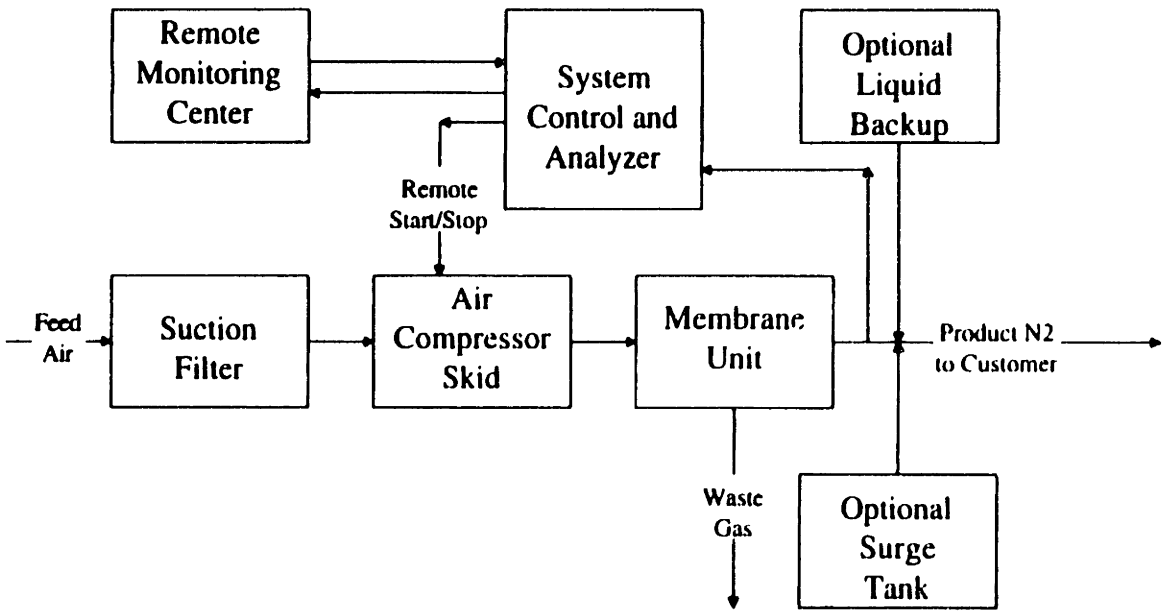


Figure 6-2. Flow diagram for a membrane separation plant

6.6 Model Construction

The probabilistic framework described in this paper simplifies the modeling task and enables the integration of knowledge from different sources. Knowledge about the process can be obtained from the plant flowsheet, the process and instrumentation diagram, the plant design, the troubleshooting guide, historical data under normal and abnormal operating conditions, and planned experiments. The troubleshooting guide is particularly useful since it focuses on abnormal symptoms and operating conditions and, by suggesting a corrective action, provides insight into the mechanism that starts with a root cause(s) and ends with observable symptoms and abnormal operation.

Model construction involves several steps, outlined below:

1. Identification of the most important variables in the system,
2. Definition of their discrete states
3. Identification of the direct probabilistic (often causal) relations
4. Identification of the prior and conditional probabilities

The Bayesian network framework constitutes a graphical communication tool that can be used throughout the modeling process. With the graphical model, experts can discuss and compare their understanding of causal relations, mechanisms and interactions among variables in the process. Within this framework it is possible to have an expert in the specific type of process build the model. The model construction does not require specific expertise on Bayesian networks or genetic algorithms, therefore avoiding the necessary loss of information that results when a knowledge engineer serves as the intermediate link between the expert and the model.

The construction of the model is an iterative process. Once a preliminary model is ready, it can be discussed with experts and its performance compared with known cases. The computer-generated diagnosis can be compared with a human-generated diagnosis using the same input data. Modifications to the Bayesian network model are straightforward because the nodes and arcs have a clean interpretation. When a node is added or deleted from the network the probability distribution of the values of a node changes, only a subset of its immediate neighbors in the graph requires updating.

When evaluating performance it is necessary to keep in mind that the "experts" we try to emulate may also be wrong sometimes. Human beings are subject to all manner of biases including a tendency to stick with their first hypothesis in the face of mounting contradictory evidence, to jump to conclusions with little or no evidence, to misjudge the importance of evidence, and to mistake cause and effect. For example, if an operator believes that a certain fault is present and therefore takes an action known to correct it, and subsequently the symptoms disappear, he may conclude that the hypothesized fault was indeed present. However, there may be other causes which are also corrected by the same corrective action, or the corrective action may have been unrelated to the disappearance of the symptoms. In either case, the operator may be led to the wrong conclusion, leading to a "bad" test case.

The first step in the model construction, the identification of the most important variables in the system, and the definition of their discrete states, is exemplified in Table 6-1 for a discrete variable (a fuse in the start circuitry) and a continuous variable (membrane pressure). In the case of continuous variables thresholds are selected for discretization. An initial model draft including only the topology of the Bayesian network is used to refine the node links that describe variable interactions and causal mechanisms. Additional steps include a discussion on the model draft and specific questions to the experts on areas

of the model that needed refinement. The definition of probabilistic distributions is the next step.

Table 6-1. States for continuous and discrete variables.

FUSE IN START CIRCUITRY
normal
abnormal: the fuse is blown and the contact is opened.
SYSTEM PRESSURE
normal
low
high

Provide a relative ranking of the following possible causes of GAS LEAKS.

	.....least likely ..... most likely
PIPE JOINTS	❑ ❑ ❑ ❑ ❑ ❑ ❑
VALVE 1 LEAK	❑ ❑ ❑ ❑ ❑ ❑ ❑
SEAL 1 LEAK	❑ ❑ ❑ ❑ ❑ ❑ ❑
VALVES 123 LEAK	❑ ❑ ❑ ❑ ❑ ❑ ❑

How certain are you about these estimates?

Low ❑ Medium ❑ High ❑

Figure 6-3. Knowledge acquisition form to rank competing root causes

What is the probability that the indicator X50 will be abnormal?

very low very high

❑ ❑ ❑ ❑ ❑ ❑ ❑

How certain are you about these estimates?

Low ❑ Medium ❑ High ❑

Fig. 6-4. Elicitation of prior probabilities

PROBABILITY ESTIMATES

Temperature 123 is affected by Temperature 124. Estimate the probabilities of values of Temperature 123 given the values of Temperature 124.

Assume that the TEMPERATURE CONTROL VALVE and the associated control loop are working normally.

T 124	T 123	Probability very seldom ... very often
normal	normal	❑ ❑ ❑ ❑ ❑ ❑ ❑
normal	low	❑ ❑ ❑ ❑ ❑ ❑ ❑
normal	high	❑ ❑ ❑ ❑ ❑ ❑ ❑
low	normal	❑ ❑ ❑ ❑ ❑ ❑ ❑
low	low	❑ ❑ ❑ ❑ ❑ ❑ ❑
low	high	❑ ❑ ❑ ❑ ❑ ❑ ❑
high	normal	❑ ❑ ❑ ❑ ❑ ❑ ❑
high	low	❑ ❑ ❑ ❑ ❑ ❑ ❑
high	high	❑ ❑ ❑ ❑ ❑ ❑ ❑

How certain are you about these estimates?

Low ❑ Medium ❑ High ❑

Figure 6-5. Form to elicit probabilistic distributions given a partially specified scenario.

6.7 Model Testing, Validation, and Maintenance

The GALGO system has been implemented as a computer program in the C++ object oriented language, and installed to work on-line using remote data available at the monitoring center. The program runs on a 486 66 Mhz desktop personal computer.

Testing involved comparing diagnostic conclusions from an expert operator with the diagnostic conclusions obtained by the computer system.

Additional benefits of the modeling framework include the simplicity of model maintenance and modification. Modeling is a continuous process, often called model maintenance, because it follows a moving target, necessary as plants are modified or their understanding improved. Modifications to the model are simpler than with other types of models thanks to the modularity and local interactions of nodes in the network. When a node is added or deleted from the network the probability distribution of only a subset of its immediate neighbors in the graph requires updating.

Since the systems under study are by nature uncertain, it is necessary to estimate their *diagnosability*. It is of interest to quantify the diagnostic accuracy of the computer-aided system and, how good can it be. The simplest diagnostic system would use only the prior probabilities of known faults. The diagnostic system using symptoms should obviously improve over the prior probability ranking. An upper bound can be estimated by statistically quantifying the stochastic nature of the process. Through Monte Carlo simulations it is possible to estimate the *diagnosability* of a Bayesian network. The Monte Carlo simulations use the Bayesian network model to stochastically generate plausible scenarios of operation. First, each root node (root causes of malfunctions) is assigned a state (either normal, or some abnormal state) according to its prior probability distribution of failure. Second, each variable in the system is assigned a state according to the conditional probability distribution of its malfunction. Note that for a variable to be stochastically assigned a value, all its parents must have previously been assigned a value. Each simulation starts with root nodes (nodes with no parents, typically root causes of failures) and ends with leaf nodes (nodes with no children, typically observable fault symptoms). Third, using only the observable symptoms generated in the scenario, the

diagnostic system is run to determine if the faults present in the scenario are diagnosed correctly. The fraction of cases accurately diagnosed will depend on the level of instrumentation of the plant and the stochastic nature of the process. This value constitutes an upper bound for the diagnostic system operating on-line at the chemical plant. If the accuracy of the diagnostic system on the case study data is higher than that estimated from the simulations, then the model has been overspecialized for a subset of the possible scenarios or the case study data is biased toward easily diagnosable scenarios. A confusion matrix can be used to summarize the diagnosability of the system.

Chapter 7

Conclusions

Probability theory constitutes the most complete and consistent framework currently available for dealing with uncertain knowledge, providing significant advantages over other methodologies. Recently developed methods for propagating probability information in the belief network structure have improved the ease with which probability data can be manipulated. These methods use distributed parallel computations in which probabilistic values are locally propagated between neighboring nodes in the belief network.

The Bayesian network approach has several advantages over other methodologies. In particular, it overcomes several limitations of rule-based expert systems. This approach is suitable for integrating knowledge from different sources, including the process flowsheet, historical data, behavioral descriptions and troubleshooting expertise from plant designers and operators. The probabilistic approach can represent and handle the uncertainty resulting from incomplete, noisy and conflicting data and can take advantage of additional evidence whenever available. The system diagnoses single or multiple faults, does not require a fixed set of evidence, and provides a probabilistically ranked set of diagnostic hypotheses. The modularity and local interactions in the network representation simplify knowledge acquisition, model construction and maintenance. The Multi-Stage Bayesian

Network constitutes an extension of Bayesian networks that has the additional capability of modeling temporal relations. Feedback loops become open spirals, and uncertain time delays can be modeled.

The most obvious criticism of this type of method is that it requires probability data. Along with many others, we believe that subjective estimates of probabilities which synthesize past experience, constitute a valid and valuable source of information. This assertion has been justified both from a philosophical as well as from a pragmatic point of view. The advantage of the probabilistic framework is that once subjective probabilities are selected, no further approximations or inaccuracies are introduced, which does not appear to be the case in most expert systems. The Bayesian network generates correct qualitative behaviors, even with uncertain probabilities.

Research is needed to evaluate the use of belief networks for large chemical processes. The interconnectedness of the resulting belief networks must be characterized. Knowledge engineering as well as issues such as handling dynamic systems, will be critical in determining whether this technology will prove practical for chemical process and related industrial applications.

The probabilistic framework proposed in this paper has the following capabilities:

1. Integrates probabilistic and deterministic process models of both normal and abnormal modes of operation. The behavioral knowledge can be obtained from a *deep formal* understanding of the behavior or from a *heuristic* understanding of behavior.

2. Captures the intrinsic uncertainty of complex and noisy measurements and systems by using a probabilistic approach.

3. Explicitly includes unknown modes of failure and therefore does not require knowledge of all possible failure modes.

4. Provides a theoretically solid quantification and selection of competing diagnostic system descriptions.

5. Provides the option to represent a system decomposed by its physical structure (component decomposition) or by its mathematical model structure (variable decomposition).

6. Covers all possible scenarios under which each component can behave without explicitly enumerating them. The reduction in complexity results from the modularity and local relations encoded in Bayesian belief networks. Hybrid models (discrete and continuous) can be built when the usage of continuous variables is advantageous.

7. Naturally captures interactions among components and quantifies them with probabilities removing the assumption that components fail independently.

8. Enables the construction of smaller networks than with the traditional belief network framework.

9. Takes advantage of existing algorithms to perform calculations in the belief network framework.

Chapter 8

Future Work

The present research work has developed a probabilistic framework based on Bayesian networks, capable of modeling dynamic relations, and integrating knowledge from several sources, including deterministic mathematical models, statistical data, and behavioral descriptions from experts. An extension to relate faults and equation residuals has also been developed. A graph-based genetic algorithm to solve the resulting mathematical problem has been proposed and has been experimentally characterized. In addition, the elements of the methodology have been successfully tested as a whole in an industrial implementation. However, there are still several questions that remain unanswered and others that have been formulated as a result of this work. This section outlines open questions, reviews previous related work, suggests possible solutions, and speculates on results.

The developments proposed in this thesis can be extended in several directions. This section sketches a diagnostic system with desirable additional features, and discusses problems to be solved to build it. The diagnostic system should be capable of integrating both continuous and discrete variables in a discretized time domain.

8.1 A Genetic Algorithm with a Marginalization Algorithm

Abductive inference on Bayesian networks can be performed with approximate algorithms such as the one described in this thesis and in (Rojas-Guzmán and Kramer, 1993, 1994). Diagnostic hypotheses are ranked and the best N ever generated are presented to the final user with their relative ranking (the absolute probability of the scenario comprised by the measured variables and the estimated variables). Additional information such as the value of the probability of the set of diagnoses given the measurements is desirable and can be calculated with any of a group of algorithms capable of calculating the probability of a subset of unknown variables given some evidence. An example of such methods is the nested dissection algorithm (Cooper, 1990) that is discussed in what follows. Nested dissection can evaluate the value of the conditional probability but lacks the capabilities to generate solutions other than systematically. Both algorithms can complement. The genetic algorithm can guide the search and, after convergence, the best diagnostic hypothesis can be evaluated with nested dissection.

Cooper (1990) proposed a general, exact algorithm to improve the efficiency of the calculation of the the probability of a hypothesis given the evidence in Bayesian belief networks. Even though the case of binary variables is discussed, the algorithm can handle variables with an arbitrary number of states. The algorithm can be used to evaluate and rank hypotheses by calculating the probability P(H1|H2) where H1 and H2 are disjoint sets of instantiated variables. Typically for diagnostic tasks H1 is the set of unknown variables of interest and H2 is the set of measured or observed variables. Cooper's algorithm efficiently calculates P(H1,H2) and P(H2). Finally P(H1|H2) is calculated according to Bayes theorem, P(H1|H2)=P(H1,H2)/P(H2). A proof of validity of the algorithm is included.

8.1.1 Limitations of nested dissection

The algorithm only improves the efficiency of the calculation of the probability P(H1|H2) of each hypothesis. The generation of hypotheses must be done beforehand using any other method. A suggested approach requires the exhaustive enumeration of all possible hypotheses (all possible joint instantiations of unobserved variables) and the determination of the probability of each hypothesis. An order for the calculations is suggested to reduce time required by avoiding redundant calculations (since the evidence set is the same for all hypotheses, P(H2) can be calculated only once for all of them).

8.1.2 Basic concept

- The algorithm calculates P(H1,H2) and P(H2) and then using Bayes theorem P(H1|H2)=P(H1,H2)/P(H2). The marginal probability is the sum of the probabilities of all possible joint instantiations of the set of variables which are not contained in the hypothesis.
- The algorithm consists of recursively bisecting a Bayesian belief network to create a binary tree.
- The dissection tree is the primary data structure used and plays a role that is analogous to the clique tree and the loop cutsets of other methods. Since finding a minimal-size vertex-separator set is NP-hard, a heuristic method for dissecting belief networks is used. Advances in research on vertex separators, an area of graph theory can be readily incorporated into this algorithm.
- A dissection is not unique, different dissections will have different efficiencies and which one is found depends on which node the algorithm begins with.
- The algorithm uses caches, the value returned by a frequently used internal function is stored in a table to avoid redundant calculations.

8.1.3 Time complexity

For an n-node *singly connected network* with a finite number of values per node, dissection can be constructed in O(nlogn) time and initialized in O(n) time. After initialization, a probability of the form P(H1|H2) can be computed in O((|H1|+|H2|)logn) time using a single processor.

8.1.4 Hybrid algorithms

- Dissection can be performed until a singly connected network is encountered and then Pearl's (1988) message passing algorithms can be used to answer the recursive call to the dissection algorithm.
- It is possible to combine this algorithm with clique-propagation.
- For highly connected networks it may be efficient to combine with a simulation algorithm.
- Dissection is not limited to bisection, but can be done with multiway dissections which allow dissections of degree m.

8.1.5 Complement for abductive inference algorithms

The nested dissection algorithm can be useful as a complement to the graph-based genetic algorithm implemented in the GALGO system. After finding a set of high probability hypotheses with GALGO, their probabilities conditioned on evidence can be calculated as additional information to be considered in the decision making process. The genetic algorithm ranks hypotheses by calculating the absolute probability of each scenario P(H1,H2). Since H1UH2=Ω, corresponding to a fully instantiated network model, P(H1,H2) can be calculated with a small computational cost. It is sufficient to use nested dissection for one term, P(H2) and divide any absolute probability P(H1,H2) by P(H2) to obtain P(H1|H2).

8.2 User interaction

Diagnostic conclusions are most useful when presented in a simple and concise form that enables the plant operator to quickly assimilate, evaluate, and respond to the process malfunction. It is necessary to selectively present the most relevant information to the operator and to provide interaction capabilities between the diagnostic system and the plant operator. This project is being extended in several directions, one of them is the construction of graphical user-friendly interfaces to simplify the model construction and program usage using a mouse, network elements (arcs and nodes) as objects, and a graphical display

8.2.1 Generation of explanations

Credibility by the operator is a crucial issue that can be partially addressed by presenting an explanation of how a certain diagnostic conclusion was reached. Any automated reasoning system should be able to provide to the human user an explanation of how its inference conclusions were achieved. This objective will be pursued by identifying causal paths in the process state description obtained by the inference algorithms. One approach to consider is to find causal paths by linking the most probable states of variables in the updated belief network. A more consistent explanation might be obtained by linking the states of adyacent variables in the best global explanation found by the abductive inference procedure. The presentation of the hypothesized causal mechanism can be a natural language summary or a graphical presentation of the Bayesian network model highlighting the suspected faults leading to the observed abnormal symptoms. The graphical constitutes one more advantage of the Bayesian network as a graphical communication tool.

8.2.2 Inclusion of additional observables

The flexibility to easily add measured, observed or hypothesized variables into the system (variable instantiation) to refine the diagnostic conclusion or to probabilistically simulate

8.2.3 Suggestion of corrective action

Often determining a corrective action(s) is simple once the diagnosis has been obtained. This is the case in the industrial implementation where the algorithm was tested. Through a simple association of root causes of faults with the repair activities required to fix them, it is technically simple to increase the usefulness of the computer tool.

8.2.4 Automated response on simple cases

When the probability of the diagnosed hypothesis is high or the corrective action to take is simple and does not increase the risk of the operation, automated action can be taken by the computer tool letting the operator focus on more complex cases.

8.2.5 Optimal test generation

The interaction with the user can be improved by developing techniques to generate tests which will guide the search for further evidence. This can be accomplished by identifying variables whose values will help to refine the diagnostic conclusion. Based on the selected variables, tests can be suggested to the system user. Once the topology of the belief network is known, key variables might be found to help to distinguish further between possible hypotheses and thereby refine the diagnostic conclusion. The selection of these variables may be done on a purely topological basis. Link cardinality should be considered as a criterium to be tested to select candidate variables. The optimal selection method is still to be explored.

Diagnosis can be substantially improved if the values of some subset of variables are known. The relations among variables are clearly depicted in the topology of the belief network and this information can be used to identify key variables.

8.3 A Hybrid Search Algorithm

The GALGO system utilizes a near-optimal graph-based genetic algorithm. Experimental characterization of the algorithm (Rojas-Guzmán and Kramer, 1994) on the ALARM network proposed by (Beinlich, 1989) suggests the usage of a hybrid algorithm to improve the performance of the genetic algorithm search with a low cost systematic search.

8.3.1 Previous work on hybrid algorithms

Several attempts have been made to develop hybrid algorithms that combine more than one approach to improve performance. For example, the Dynamic Hill Climbing algorithm (Yuret and de la Maza, 1993) combines three known optimization heuristics (1) remember the locations of local extrema and restart the optimization algorithm at a place distant from previously found local extrema, (2) adjust the size of probing steps, and (3) keep track of the directions of recent successes (to preferentially sample in the direction of most rapid ascent).

8.3.2 Experiments

To evaluate the feasibility and quantify the potential improvement of the hybrid algorithm, the PS899 parameter set was run on 5 different diagnostic cases, 10 times on each, and the Hamming distance between the estimated optimum and the best obtained hypothesis was measured.

8.3.3 Preliminary results

Results in a 37 node network show that convergence to the optimum can be accomplished in less than 30 seconds in an average of 80% of the runs. In an attempt to further increase the frequency of convergence to the optimum solution, a hybrid search might be used. In those cases in which the optimum is not found with the genetic algorithm, a local systematic search of bounded depth around the converged sub-optimal solution may improve overall performance at a low computational cost.

Results show that the Hamming distance is 1 only in 10% of the sub-optimal cases, in average it is 3.2 rendering the use of the local search too expensive.

Case	Optimum Frequency	Sub-optimal solution frequency	Hamming distances from sub-optimal solutions
300	9/10	1/10	2
310	8/10	2/10	1, 2
320	7/10	3/10	4, 4, 4
330	8/10	2/10	4, 4
340	8/10	2/10	3, 4

8.3.4 Discussion

If the Hamming distances were often 1, then a systematic search of depth 1 would be sufficient to find the optimum at a computational cost proportional to the number of genotype evaluations, E a UxA where U=number of unknown nodes in the network, A=average number of states per node. Since the cost grows exponentially when the depth of the search increases (changing simultaneously more than one value), systematic searches at deeper levels are not practical.

A bounded systematic search (of depth one) may still be feasible if the incorrectly estimated variables are conditionally independent of each other, encoded in the Bayesian network framework with the absence of a directed arc. A greedy search would be feasible and could be easily implemented by changing the value of one node at a time. The improvement in the phenotype would be immediate and independent of other new variable-value assignments. If the incorrectly estimated variables are clustered in the network in the sub-optimal solutions then the cost of the search increases because it is necessary to explore at deeper levels, each of which has a combinatorically growth.

8.4 Residual-based Probabilistic Diagnosis

The model-based approach can take advantage of the existence of a model (typically a deterministic mathematical model written as a set of algebraic or differential equations) which describes the behavior of the process (ideally under all conditions, but usually only for normal operation). Process faults can be detected by comparing the observed and the predicted behavior.

MSBNs offer a flexible probabilistic framework to model uncertainty in dynamics, functional form and parameters. With MSBNs it is also possible to integrate knowledge from different sources and to perform diagnosis with exact or approximate methods. This section discusses the probabilistic model-based approach to diagnosis using the Multi-Stage Bayesian Network (MSBN) framework.

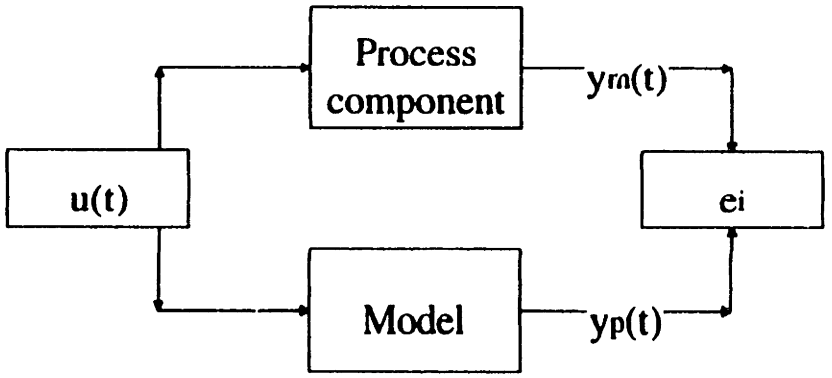


Figure 8-1. Process control inputs u(t) are used by the model to produce $y_p(t)$, predictions, and by the process to produce $y_m(t)$, measurements. Residuals e_i quantify the differences between $y_p(t)$ and $y_m(t)$.

De Kleer (1987) proposed a multiple-fault model-based diagnosis framework, General Diagnostic Engine (GDE), based on knowledge of the system structure and correct functionality of system components. A set of assumptions necessary for each component to behave normally is identified and used by an assumption-based truth maintenance system to isolate those assumptions inconsistent with the observations. In addition, next best measurements are suggested. *Sherlock*, an extension of the GDE framework (de Kleer, 1989) incorporates the notion of *modes of behavior* and requires the use of heuristics to avoid the resulting combinatorial explosion. Struss (1989) discusses the benefits of integrating fault models which should be able to explain abnormal observations and extends the framework of de Kleer (1987) with GDE+.

8.4.1 Limitations of previous approaches

Mathematical models are typically available only for normal operation. Furthermore, the description of normal behavior is often not sufficient for fault diagnosis, since the presence of faults usually invalidates some of the model assumptions (such as steady-state operation). In other cases, the model is simply not general enough to describe behavior outside of certain model parameters rendering extrapolation results unreliable.

When more than one set of faults explains the observations it is necessary to select the best or to rank the hypotheses. The ATMS approach has limited facilities to rank alternate hypotheses. Only prior probabilities of failure modes are encoded in Sherlock. Interactions among components are not modeled but do exist in many complex systems. In GDE, component states are assumed to remain at the same value independently of time but this assumption is not valid in dynamic systems.

8.4.2 Multi Stage Bayesian Networks

Multi-state Bayesian networks constitute a general framework that subsumes digraph and residual-based approaches (Rojas and Kramer, 1994). MSBNs extend the Bayesian network framework by using time-indexed variables which makes them capable of representing time delays and capturing other forms of dynamic behavior.

Nodes in the MSBN can represent propositions, system variables (measured or unmeasured), system components (at different levels of abstraction), and calculated quantities such as equation residuals. Several modeling approaches can be used:

(1) The most general use of MSBNs is similar to that of BNs where nodes represent arbitrary system variables, with the additional expressive capabilities resulting form time representation.

(2) A second approach for modeling with MSBNs assigns a system component to each node. Modes of operation of the component can be represented as discrete states of the node. A hierarchical system decomposition can help to perform diagnosis at different levels of abstraction depending on the level of detail required and the available computational resources and inference time. The architecture of a model-based diagnostic system can use a set of component models to compare predicted outputs with measured outputs, or it can substitute into the model a set of measured variables (including outputs and internal variables).

(3) A third approach takes advantage of the existence of traditional mathematical models of the system (i.e. sets of algebraic or ordinary differential equations) and provides a way to formally handle the ambiguity resulting form uncertain functional forms and model parameters.

8.4.3 Desired model properties

Models vary in at least two dimensions (1) the quality of the representation of the system, and (2) the resulting computational complexity. These dimensions are a function of the task to be performed, in this case diagnosis or abductive inference. Models can also be used to predict behavior, to gain insight in the process mechanisms, and fault detection. For all of them the model should be robust.

MSBN models should include all faults of interest (to be able to diagnose them) and a residual for each independent equation (to take advantage of all the modeling information). Ideally each fault or set of faults should produce a unique residual pattern. In the simplest case, the network is composed of a set of disconnected fault-residual subnetworks.

8.4.4 Network transformations

When using the belief network framework the modeler has the flexibility of including any variables in the model. It may be easier to build a model by including internal variables which are not measured in it. However, their inclusion is not always beneficial. Often, component models require only measurable variables and removing internal variables becomes necessary. The task of acquiring probabilistic parameters can be simplified by reducing the number of parameters or by acquiring a set of probabilities which is more intuitive to the expert or which can be more readily estimated from data.

Nodes can be removed using an inference algorithm (Shachter, 1986) for acyclic graphs in which any node can be eliminated from the graph through the elimination of a subset of the nodes and the reversal of arcs. The main disadvantage is that the new equivalent network may have additional non-intuitive reversed links, which usually render the estimation or calculation or probabilities difficult. Nodes can also be removed and the probabilistic distributions of a subset of the immediate neighbors of the node updated.

The resulting network is not unique. A preference criterion is therefore necessary to perform binary comparisons among models. With such a criterion it is possible, at least in principle, to generate and test models to search for the optimal model. In the worst case, a trial and error approach could be used. However, the insight gained from the present analysis may provide guidelines to systematically develop residual-based MSBN models. The key step is the optional construction of new residuals.

8.4.5 Problem representation

A set of residuals can be defined based on a set of equations, with one residual per equation. This set can also be represented as a two level Bayesian network relating faults

and residuals. The network, as any DAG (directed acyclic graph) can be represented by an adjacency matrix. This paper proposes the use of the adjacency matrix as an intermediate representation to assess the topological properties of the equivalent Bayesian network and to guide the extensions or reductions of the model.

Equations can be algebraically manipulated to construct a network of specified properties. Valid operations on the set of equations include linear combinations of equations, resulting in the elimination of an equation or the creation of redundant equations. The topology of the equations is evaluated with the adjacency matrix and at the end, the matrix is translated into the graph representation.

In the matrix representation it is simple to test some topological properties of Bayesian network models. Algorithms exist to detect the presence of loops (using depth first systematic search). In the special case of 2 level network relating faults and residuals loops may not exist, since residuals are always leaves in the graph.

It is simple to test whether each fault produces a unique residual pattern, by looking at whether all the faults have some effect by looking at the projection of all matrix rows, (each column should have at least one non-zero element). Also, each residual should include at least one fault, (each row should have at least one non-zero element).

By using the signs of the faulty terms (for those faults working only in one direction such as a leak) it is possible to add signs to elements of the matrix to detect (and try to avoid) compensating effects when more two faults appear in the same residual.

8.4.6 Summary

The representation of the relations between faults and equation residuals within the Bayesian network representation constitutes a promising option to perform fault diagnosis when mathematical models in the form of equations are available.

Chapter 9

References

Almond, R., J. Bradshaw, and D. Madigan, "Reuse and sharing of graphical belief network components," in *Selecting Models from Data, Artificial Intelligence and Statistics IV*, edited by P. Cheeseman and R. Oldford, Springer-Verlag, New York, NY (1993).

Andersen, S. K., "Book review: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference," *Artificial Intelligence*, **48**, pp. 117-124 (1991).

Bareiss, E., "PROTOS: A unified approach to concept representation, classification, and learning," Ph.D. dissertation, University of Texas, TX (1988).

Bareiss, E., B. Porter, and C. Wier, "PROTOS: An exemplar-based learning apprentice," *Int. Journal of Man Machine Studies*, **29**, pp. 549-561 (1988).

Becraft, W. R., D. Z. Guo, P. L. Lee, and R. B. Newell, "Fault diagnosis strategies for chemical plants: A review of competing technologies," in *Proc. of the 4th Int. Symposium on Process Systems Engineering PSE'91*, **II** (1991).

Beinlich, I. and E. Herskovits, "ERGO, A graphical environment for constructing Bayesian belief networks," in *Proc. Sixth Conf. on Uncertainty in Artificial Intelligence*, Cambridge, MA (1990).

Beinlich, I. A., H. J. Suermondt, R. M. Chavez, and G. F. Cooper, "The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks," in *Proc. of the Second European Conf. on Artificial Intelligence in Medicine*, London, England, pp. 247-256 (1989).

Ben-Bassat, M., R. W. Carlson, V. K. Puri, *et al*, "Pattern-based interactive diagnosis of multiple disorders: The Medas system," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **PAMI-2** (1980).

Ben-Bassat, M., K. L. Klove, and M. H. Weil, "Sensitivity analysis in Bayesian classification models: Multiplicative deviations," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **PAMI2** (1980).

Blum, M., "A stagewise parameter estimation procedure for correlated data," *Numer. Math.*, **3**, pp. 202-208 (1961).

Brailovsky, V. L., "Probabilistic approach to model selection: comparison with unstructured data set," in *Selecting Models from Data, Artificial Intelligence and Statistics IV*, edited by P. Cheeseman and R. Oldford, Springer-Verlag, New York, NY (1993).

Buchanan, B. G., and E. H. Shortliffe, *Rule-Based Expert Programs: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Reading, MA (1984).

Carnap, R., "The two concepts of probability", in *Logical Foundations of Probability*, University of Chicago Press, Chicago, IL (1950).

Cheeseman, P., "A method for computing generalized Bayesian probability values for expert systems," in *Proc. 8th Int. Joint Conf. on Artificial Intelligence*, Karlsruhe, West Germany, pp. 198-202 (1983).

Cheeseman, P. and R. W. Oldford, editors. "Selecting models from data," in *Artificial Intelligence and Statistics IV*, Springer-Verlag, New York, NY (1993).

Chow, E. Y., and A. S. Willsky, "Analytical redundancy and the design of robust failure detection systems," *IEEE Trans. on Automatic Control*, **AC-29**, pp. 603-614 (1984).

Cooper, G. F., "NESTOR: A computer-based medical diagnostic aid that integrates causal and probabilistic knowledge," Technical Report HPP-84-48, Stanford University, Stanford, CA, pp. 239 (1984).

Cooper, G. F., "Bayesian belief network inference using nested dissection," Report KSL 90-05, Medical Computer Science, Stanford University, Stanford, CA (1990).

Cooper, G. F., "Probabilistic inference using belief networks is NP-hard", Report KSL 87-27, Knowledge Systems Laboratory, Computer Science Department, Stanford University, Stanford, CA (1988).

Cooper, G. F., "Probabilistic inference using belief networks is NP-hard," *Artificial Intelligence*, **42**, pp. 393-405 (1990).

Cooper, G. F., and E. Herskovits, "A Bayesian method for the induction of probabilistic networks form data," *Machine Learning*, **9**, pp. 309-347 (1992).

Dagum, P., and M. Luby, "Approximating probabilistic inference in Bayesian belief networks is NP-hard," *Artificial Intelligence*, **60**, pp. 141-153 (1993).

Davidor, Y., "Epistasis variance: A viewpoint on GA-hardness," in *Foundations of Genetic Algorithms*, edited by G. J. E. Rawlings (1991).

Cormen, T. H., C. E. Leiserson and R. Rivest, *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Co., New York, NY (1990).

Davis, R, "Diagnostic reasoning based on structure and behavior," *Artificial Intelligence*, **24**, pp. 347-410 (1984).

Davis, R., "Reasoning from first principles in electronic troubleshooting," *Int. J. Man-Machine Studies*, **19**, pp. 403-23 (1983).

Davis, R., H. Shrobe, and P. Szolovits, "What is a knowledge representation? Broadening the perspective," Report, Artificial Intelligence Laboratory and Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA (1991).

De Finetti, B., "Foresight: Its logical laws, its subjective sources," in *Studies in Subjective Probability*, edited by H. E. Kyburg, Jr. and H. E. Smokler, Wiley, New York, NY (1964).

De Finetti, B., *Probability, Induction, and Statistics*, Wiley, New York, NY (1972).

De Kleer, J., and B. C. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, **32**, pp. 97-130 (1987).

De Kleer, J., and B. C. Williams, "Diagnosis with behavioral modes", in *Proc. 11th Int. Joint Conf. on Artificial Intelligence*, Detroit, MI, pp. 1324-1330 (1989).

De Jong, K., "Genetic algorithms: A 10 year perspective," in *Proc. Int. Conf. on Genetic Algorithms and their Applications*, Pittsburgh, PA, pp. 169-177 (1985).

De la Maza, M., and D. Yuret, "Dynamic hill climbing," *AI Expert*, **9**, no. 3, pp. 26-31 (1994).

Dubois, D. and H. Prade. An Introduction to Possibilistic and Fuzzy Logics. In *Non-Standard Logics for Automated Reasoning*, edited by P. Smets, *et al.* Academic Press, London (1988).

Duda, R. O. and P. E. Hart, *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York, NY (1973).

Duran, M. A., and I. E. Grossmann, "An outer-approximation algorithm for a class of mixed-integer nonlinear programs," *Math. Program*, **36**, pp. 307-339 (1986).

Finch, F. E., "Automated fault diagnosis of chemical process plants using model-based reasoning," Ph.D. dissertation, Chemical Engineering Department, Massachusetts Institute of Technology, Cambridge, MA (1989).

Finch, F. E., O. O. Oyeleye, and M. A. Kramer, "A robust event-oriented methodology for diagnosis of dynamic process systems," *Computers and Chemical Engineering*, **14**, pp. 1379-1396 (1990).

Frank, P., "Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy, A survey and some new results," *Automatica*, **26**, pp. 459-474.

Frank, P. M. and Xianchung Ding, "Frequency domain approach to optimally robust residual generation and evaluation for model-based fault diagnosis," *Automatica*, **30**, pp. 789-804 (1994).

Frank, P. M., "Robust model-based fault detection in dynamic systems," in *Proc. IFAC Symposium on On-Line Fault Detection and Supervision in the Chemical Process Industries*, Newark, DE, pp. 1-13 (1992).

Gelb, A., ed., J. F. Kasper, Jr., R. A. Nash, Jr., C. F. Price, A. A. Sutherland, Jr. *Applied Optimal Estimation.* The Analytic Sciences Corporation. MIT Press, Cambridge, MA (1974).

Geoffrion, A. M., "Generalized benders decomposition," *J. Optimization Theory Applic.* **10**, pp. 237-260 (1972).

Gertler, J., "A survey of model-based failure detection and isolation in complex plants," *IEEE Control Systems Mag.*, **3** (1988).

Gertler, J., "Structured residuals for fault isolation, disturbance decoupling and model error robustness," in *Proc. IFAC Symposium on On-Line Fault Detection and Supervision in the Chemical Process Industries*, Newark, DE, pp. 111-119 (1992).

Gertler, J., A. Bennani, and T. Phillips, "A parallel implementation of model based on-line failure detection and isolation," in *Proc. ACC*, Pittsburgh, PA (1989).

Gertler, J. and Q. Luo. Robust isolable models for failure diagnosis. *AIChE Journal*, **35**, No. 11, pp. 1856-1868 (1989).

Gertler, J., and D. Singer, " . new structural framework for parity equation-based failure detection and isolation," *Automatica*, **26**, pp. 381-388.

Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA (1989).

Grefenstette, J. J., "Optimization of control parameters for genetic algorithms," *IEEE Trans. on Systems, Man, and Cybernetics*, **16**, pp.122-128 (1986).

Grefenstette, J. J., "Conditions for implicit parallelism," in *Foundations of Genetic Algorithms*, edited by G. J. E. Rawlings (1991).

Grefenstette, J. J. and J. E. Baker, "How genetic algorithms work: A critical look at implicit parallelism," in *Proc. of the Third Int. Conf. on Genetic Algorithms*, George Mason University, pp. 20-27 (1989).

Gupta, O. K. and V. Ravindran, "Branch and bound experiments in convex nonlinear integer programming," *Mgmt. Sci.* **31**, pp. 1533-1546 (1985).

Hammond, K., and N. Hurwitz, "Extracting diagnostic features from explanations," in *Proc. of the DARPA Workshop on Case-Based Reasoning*, San Mateo, CA, pp. 169-178 (1988).

Hashimoto, Y., K. Kawahara, Y. Tanaka, A. Yoneya, and Y. Togari, "Fault diagnosis utilizing a three-layer directed graph," in *Proc. 4th Intl. Symposium on Process Systems Engineering*, **II.10** (1991).

Hamscher W., "Diagnosing circuits with state: an inherently underconstrained problem," in *Proc. American Association for Artificial Intelligence, AAAI-84*, pp. 142-147 (1984).

Henrion, M, *Towards Efficient Probabilistic Diagnosis in Multiply Connected Belief Networks, in Influence Diagrams, Belief Nets and Decision Analysis*, edited by Olivier and Smith, John Wiley & Sons Ltd., New York, NY (1990).

Henrion, M. "Propagating uncertainty in Bayesian networks by logic sampling," in *Uncertainty in Artificial Intelligence 2*, edited by J. F. Lemmer & L. N. Kanal, North-Holland, Amsterdam (1988).

Henrion, M., "An introduction to algorithms for inference in belief nets," in *Uncertainty in Artificial Intelligence 5*, edited by M. Henrion, *et al.*, Elsevier Science Publishers B. V. , pp. 129-138 (1990).

Herskovits, E. H., "Computer-based probabilistic-network construction," Ph.D. dissertation, Medical Information Sciences, Stanford University, Stanford, CA (1991).

Holland, J. H., *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, 2nd ed. MIT Press, Cambridge, MA (1992).

Horvitz, E. J., "Computations and action under bounded resources," Ph.D. dissertation, Stanford University, Stanford, CA (1990).

Howard, R. A., "Proximal decision analysis," *Management Sci.*, **17**, 9, (1971), pp. 507-541; reprinted in *The Principles and Applications of Decision Analysis, Vol. II*, edited by R. A. Howard, and J. E. Matheson, Strategic Decisions Group, Menlo Park, CA (1984).

Howard, R. A. and J. E. Matheson, "Influence diagrams," in *The Principles and Applications of Decision Analysis, Vol. II*, edited by Howard and Matheson, Strategic Decisions Group, Menlo Park, CA (1981).

Hunter, L., "Knowledge acquisition planning: Gaining expertise through experience," Ph.D. dissertation, Yale University, New Haven, CT (1989).

Iri, M., K. Aoki, E. O'Shima, and H. Matsuyama, "An algorithm for diagnosis of system failures in the chemical process," *Computers and Chemical Engineering*, **3**, pp. 489-493 (1979).

Jaynes, E. T., *Probability Theory -- The Logic of Science*, draft (June 1994).

Jaynes, E. T., *Where Do We Stand on Maximum Entropy, in The Maximum Entropy Formalism*, edited by R. D. Levin and M. Tribus, MIT Press, Cambridge, MA (1979).

Jensen, F. V., K. G. Olsen and S. K. Andersen, "An algebra of Bayesian belief universes for knowledge-based systems", *Networks*, **20**, pp. 637-660 (1990).

Kalman, R. E., "A new approach to linear filtering and prediction problems," *J. Basic Eng.*, pp. 35-46 (1960).

Kalman, R. E. and R. S. Bucy, "New results in linear filtering and prediction theory," *J. Basic Eng.*, pp. 95-108 (1961).

Kenley, C. R., "Influence diagram models with continuous variables," Ph.D. dissertation, EES Department, Stanford University, Stanford, CA (1986).

Keynes, J. M., *A Treatise on Probability*. Macmillan, London (1948).

Klug, W. S. and M. R. Cummings, *Concepts of Genetics*, 2nd ed. Scott, Foresman and Co. (1986).

Kokawa, M., S. Miyazaki, and S. Shingai, "Fault location using digraph and inverse direction search with application," *Automatica*, **19**, pp. 729-735 (1983).

Konolige, K, "Bayesian methods for updating probabilities," Final Report, Project 6415, SRI International (1979).

Koton, P., "Reasoning about evidence in causal explanations," in *Proc. of the DARPA Workshop on Case-Based Reasoning*. Morgan Kaufmann, San Mateo, CA, pp. 260-270 (1988).

Koutsoupias, E., and C. H. Papadimitriou, "On the greedy algorithm for satisfiability," *Information Processing Letters*, **43**, pp. 53-55 (1992).

Kramer, M. A., "Malfunction diagnosis using quantitative models with non-Boolean reasoning in expert systems," *AIChE Journal*, **33**, pp. 130-140 (1987).

Kramer, M. A., and R. S. H. Mah, "Model-based monitoring," in *Proc. Second Conference on Foundations of Computer Aided Process Operations*, Crested Butte, CO, pp. 1-24 (1993).

Kravaris, C. and P. Daoutides, "Nonlinear state feedback control of second-order nonminimum-phase nonlinear systems," *Computers and Chemical Engineering*, **14**, pp. 439-449 (1990).

Laplace, P. S. de, *A Philosophical Essay and Probabilities*. Dover, New York, NY (originally published in 1820) (1951).

Lauritzen, S. L., and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *J. Roy. Stat. Soc. B.*, **50**, pp. 157-224 (1988).

Lauritzen, S., B. Thiesson and D. J. Spiegelhalter, "Diagnostic systems by model selection: a case study," in *Selecting Models from Data, Artificial Intelligence and Statistics IV*, edited by P. Cheeseman and R. Oldford, Springer-Verlag, New York, NY (1993).

Lee, Y. and R. P. Lippmann, "Practical characteristics of neural network and conventional pattern classifiers on artificial and speech Problems," in *Proc. NIPS-89*, Denver, Colorado (1989).

Lemmer, J. F., and S. W. Barth, "Efficient minimum information updating for Bayesian inferencing in expert systems," in *Proc. 2nd Conf. on Artificial Intelligence*, Pittsburgh, pp. 424-427 (1982).

Leonard, J. A., and M. A. Kramer, "Solving dynamic fault diagnosis problems with modular neural networks, *IEEE Expert, Special Track on Diagnosis* (1991).

Li, Z., and B. D'Ambrosio, "An efficient approach for finding the MPE in belief networks," in *Proc. 9th Conf. in Uncertainty in Artificial Intelligence*, Washington, D.C., pp. 342-349 (1993).

Li, Z., and B. D'Ambrosio, "Efficient inference in Bayes networks as a combinatorial optimization problem", *Int. Journal of Approximate Reasoning*, **11**, pp. 55-81 (1994).

Lindley, D. V., A. Tversky, and R. V. Brown, "On the reconciliation of probability assessments," *J. Roy. Statist. Soc.*, A, pp. 146-180 (1979).

Lippmann, R. P., "An introduction to computing with neural nets, *IEEE ASSP Magazine*, **4**, pp. 4-22 (1987).

Lowrance, J. D., T. D. Garvey, and T. M. Strat, "A framework for evidential reasoning systems," in *Proc. 5th National Conf. on Artificial Intelligence (AAAI-86)*, Philadelphia, pp. 896-901 (1986).

Luenberger, D. G., "Observing the state of a linear system," *IEEE Trans. Mil. Electron.*, MIL-8, 74 (1964).

Luenberger, D. G., "An introduction to observers," *IEEE Trans. Autom. Control*, AC-16, 596 (1971).

Madigan, D., *et al*, "Strategies for graphical model selection," in *Selecting Models from Data, Artificial Intelligence and Statistics IV*, edited by P. Cheeseman and R. Oldford, Springer-Verlag, New York, NY (1993).

Measor, *Artificial Intelligence Review*, **5** (1991).

Mehra, R. K., *IEEE Trans. Aut. Cont.*, pp. 175-184 (1970).

Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York, NY (1992).

Micro Data Base Systems, Inc. *GURU, Reference Manual, Artificial Intelligence that Means Business*. Lafayette, IN (1985).

Miller, A. C., M. M. Merkhofer, R. A. Howard, J. E. Matheson, and T. R. Rice, *Development of Automated Aids for Decision Analysis*. Stanford Research Institute, Menlo Park, CA (1976).

Mohindra, S. and P. A. Clark, "Distributed fault diagnosis method based on digraph models: Steady-state analysis," *Computers and Chemical Engineering*, **17**, pp. 193-209 (1993).

Montanari, U., "Networks of constraints, fundamental properties and applications to picture processing," *Information Science*, **7**, pp. 95-132 (1974).

Moore, R. L., and M. A. Kramer, "Expert systems in on-line process control," in *Proc. of the Third Int. Conf. on Chemical Process Control*, Asilomar, CA (1986).

Narasimhan, S. and R. S. H. Mah, "Generalized likelihood ratio method for gross error identification," *AIChE J.*, **33**, pp. 1514-1521 (1987)

Neapolitan, R. E., *Probabilistic Reasoning in Expert Systems, Theory and Algorithms*. John Wiley & Sons, New York, NY (1990).

Nilsson, N. J., "Probabilistic logic," *Artificial Intelligence*, **28**, pp. 71-87 (1986).

Obeid and Turner, *Artificial Intelligence Review*, **5** (1991).

Paris J. B. and Venkovská, A., "On the applicability of maximum entropy to inexact reasoning," *Int. Journal of Approximate Reasoning*, **3**, pp. 1-34 (1989).

Pearl, J., "Fusion, propagation and structuring in belief networks," *Artificial Intelligence*, **29**, pp. 241-288 (1986).

Pearl, J., "Evidential reasoning using stochastic simulation of causal models," *Artificial Intelligence*, **32** (1987).

Pearl, J., "Distributed revision of composite beliefs," *Artificial Intelligence*, **33** (1987).

Pearl, J., *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., San Mateo, CA (1988).

Peng, Y. and J. A. Reggia, "A probabilistic causal model for diagnostic problem solving (parts I and II)", *IEEE Trans. on Systems, Man and Cybernetics*, **SMC-17** (1987).

Peng, Y. and J. A.Reggia, "A connectionist model for diagnostic problem solving," *IEEE Trans. on Systems, Man, and Cybernetics*, **19**, pp. 285-298 (1989).

Peng, Y. and J. A. Reggia, *Abductive Inference Models for Diagnostic Problem-Solving*. Springer-Verlag, New York, NY (1990).

Petti, T. F., J. Klein, and P. S. Dhurjati, "Diagnostic model processor: Using deep knowledge for process fault diagnosis," *AICHE Journal*, **36**, pp. 565-575 (1990).

Ponton, J. W., "Detecting inverse responses in process models," Technical Report 1994-01, University of Edinburgh, Edinburgh, Scotland, (1994).

Poole, D., "Average-case analysis of a search algorithm for estimating prior and posterior probabilities in Bayesian networks with extreme probabilities," in *Proc. Int. Joint Conf. on Artificial Intelligence*, Chambery, Savoie, France, pp. 606-612 (1993).

Poole, D. "The use of conflicts in searching Bayesian networks," in *Proc. Ninth Conf. on Uncertainty in Artificial Intelligence*, Washington, D.C., pp. 359-367 (1993).

Poole, D., "Normality and faults in logic-based diagnosis," in *Proc. Int. Joint Conf. on Artificial Intelligence*, Detroit, MI, pp. 1304-1310 (1989).

Poole, D. and G. M. Provan, "What is an optimal diagnosis?," *in Proc . Sixth Conf. on Uncertainty in Artificial Intelligence*, Cambridge, MA, pp. 46-53 (1990).

Popper, K. R., *Logic of Scientific Discovery*. Hutchinson & Co. (originally published in 1935) (1975).

Popper, K. R., *Realism and the Aim of Science*. Rowman & Littlefield, Totowa, New Jersey (1983).

Provan, G. M., "Model selection for diagnosis and treatment using temporal influence diagrams," in *Selecting Models from Data, Artificial Intelligence and Statistics IV*, edited by P. Cheeseman and R. Oldford, Springer-Verlag, New York, NY (1993).

Quesada, I. and I. E. Grossmann, "An LP/NLP based branch and bound algorithm for convex MINLP optimization problems," *Computers and Chemical Engineering*, **16**, pp. 937-947 (1992).

Ramesh, T. S., J. F. Davis, and G. M. Schwenzer, "CATCRACKER: An expert system for process and malfunction diagnosis in fluid catalytic cracking units," *AICHE Annual Meeting*, San Francisco, CA (1989).

Reiter, R., "A theory of diagnosis from first principles, *Artificial Intelligence*, **32**, pp. 57-95 (1987).

Rich, S. H. and V. Venkatasubramanian, "Model-based reasoning in diagnostic expert systems for chemical proces plants," *Computers and Chemical Engineering*, **11**, pp. 111-122 (1987).

Rojas-Guzmán, C., and M. A. Kramer, "Belief networks for knowledge integration and abductive inference in fault diagnosis," in *Proc. On-Line Fault Detection and*

Supervision in the Chemical Process Industries, IFAC Int. Symposium, Newark, DE, pp. 14-19 (1992).

Rojas-Guzmán, C., J. Tan, and M. A. Kramer, "Diagnostic inference using probabilistic reasoning," *Fifth Annual LISPE-Industry Symposium*, Massachusetts Institute of Technology, Cambridge, MA (1991).

Rojas-Guzmán, C. and M. A. Kramer, "Comparison of belief networks and rule-based expert systems for fault diagnosis of chemical processes," *Engng. Appl. Artif. Intell.* **6**, pp. 191-202 (1993a).

Rojas-Guzmán, C. and M. A. Kramer, "GALGO: A Genetic ALGOrithm decision support tool for complex uncertain systems modeled with Bayesian belief networks," in *Proc. Ninth Conf. on Uncertainty in Artificial Intelligence*, pp. 368-375 (1993b).

Rojas-Guzmán, C. and M. A. Kramer, "Multi-Stage Bayesian Networks subsume digraph and residual-pattern approaches to fault diagnosis," in *Proc. of the Fifth Int. Symposium on Process Systems Engineering*, Kyongju, Korea, pp. 947-952 (1994).

Rojas-Guzmán, C., and M. A. Kramer, "Diagnosis with genetic algorithms and Bayesian belief networks," *AIChE National Conference*, San Francisco, CA, (1994).

Rojas-Guzmán, C., and M. A. Kramer, "An evolutionary computing approach to probabilistic reasoning on Bayesian networks", for submission to *Evolutionary Computation*, (1994).

Romagnoli, J. A. and G. Stephanopoulos, "Rectification of process measurement data in the presence of gross errors," *Chem. Eng. Sci.*, **36**, pp. 1849 (1981).

Savage, L. J., *The Foundations of Statistics*. Wiley, New York, NY (1954).

Seroussi, B. and J. L. Golmard, "An algorithm directly finding the k most probable configurations in Bayesian networks", *Int. Journal of Approximate Reasoning*, **11**, pp. 205-233, (1994).

Shachter, R. D. and C. R. Kenley, "Gaussian influence diagrams," *Management Science*, **35**, pp. 527-550 (1989).

Shafer, G., *A Mathematical Theory of Evidence*. Princeton University Press, Princeton, New Jersey (1976).

Shafer, G. and J. Pearl, *Readings in Uncertain Reasoning*. Morgan Kaufmann Publishers, San Mateo, CA (1990).

Shafer, G., P. P. Shenoy, and K. Meloulli, "Propagating belief functions in qualitative Markov trees," Working Paper No. 190., School of Business, University of Kansas, KS.

Sheridan, F. K. J., "A survey of techniques for inference under uncertainty," *Artificial Intelligence Review*, **5**, pp. 89-119 (1991).

Shimodaira, H., "A new criterion for selecting models from partially observed data", in *Selecting Models from Data, Artificial Intelligence and Statistics IV*, edited by P. Cheeseman and R. Oldford, Springer-Verlag, New York, NY (1993).

Shimony, S. E., "Finding MAPs for belief networks is NP-hard," *Artificial Intelligence*, **68**, pp. 399-410 (1994).

Shimony, S. E., and E. Charniak, "A new algorithm for finding MAP assignments to belief networks," in *Proc. 6th Conf. on Uncertainty in Artificial Intelligence*, pp. 98-103 (1990).

Shiozaki, J., H. Matsuyama, E. O'Shima, and M. Iri, "An improved algorithm for diagnosis of system failures in the chemical process," *Computers and Chemical Engineering*, **9**, pp. 285-293 (1985).

Simon, H. A., "Search and reasoning in problem solving," *Artificial Intelligence*, **21**, pp. 7-29 (1983).

Slide, S., "Case-based reasoning: A research paradigm," *Artificial Intelligence Magazine*, pp. 42-55 (1991).

Spiegelhalter, D. J., "A statistical view of uncertainty in expert systems," in *Artificial Intelligence and Statistics*, edited by W. A. Gale, Addison-Wesley, Reading, MA (1985).

Srinivas, S. and J. Breese, "IDEAL: A software package for analysis of influence diagrams," in *Proc. of the Sixth Conf. on Uncertainty in Artificial Intelligence*, Cambridge, MA (1990).

Struss, P. and O. Dressler, "Physical negation - integrating fault models into the general diagnostic engine," in *Proc. 11th Int. Joint Conf. on Artificial Intelligence*, Detroit, MI, pp. 1318-1323 (1989).

Suermondt and G. Cooper, "Probabilistic inference in multiply connected belief networks using loop cutsets," *Int. Journal of Approximate Reasoning*, **4**, pp. 283-306 (1990).

Tamhane, A. C., and R. S. H. Mah, "Data reconciliation and gross error detection in chemical process networks," *Technometrics*, **27**, pp. 409 (1985).

Tamhane, A. C., C. Iordache and R. S. H. Mah, "A Bayesian approach to gross error detection in chemical process data, Part I: Model development," *Chemometrics and Intel. Lab. Sys.*, **4**, pp. 33-45 (1988).

Tamhane, A. C., C. Iordache and R. S. H. Mah, "A Bayesian approach to gross error detection in chemical process data. Part II: Simulation results," *Chemometrics and Intel. Lab. Sys.*, **4**, pp. 131-146 (1988).

Tsuge, Y., J. Shiozaki, H. Matsuyama, and E. O'Shima, in *Proc. Process Systems Engineering*, pp. 133-144 (1985).

Turner, R., "Organizing and using schematic knowledge for medical diagnosis," in *Proc. of the DARPA Workshop on Case-Based Reasoning*, pp. 435-446, Morgan Kaufmann (1988).

Viswanathan J. and I. E. Grossmann, "A combined penalty function and outer-approximation method for MINLP optimization," *Computers and Chemical Engineering*, **14**, pp. 769-782 (1990).

Von Mises, R., *Probability, Statistics and Truth.* (originally published in Vienna, 1928) George Allen & Unwin, London (1957).

Weatherford, R. *Philosophical Foundations of Probability Theory.* Routledge & Kegan Paul, London (1982).

Weissermel, K. and H.-J. Arpe, *Industrial Organic Chemistry, Important Raw Materials and Intermediates.* Verlag Chemie, New York, NY (1978).

Wen, W. X., "Directed cycles in belief networks," in *Proc. Uncertainty in Artificial Intelligence*, pp. 377-384 (1989).

Whitley, L. D., "Fundamental principles of deception in genetic search," in *Foundations of Genetic Algorithms*, edited by G. J. E. Rawlings.

Wilcox, N. A. and D. M. Himmelblau, "The possible cause and effect graphs (PCEG) model for fault diagnosis -- I. Methodology," *Computers and Chemical Engineering*, **18**, pp. 103-116 (1994).

Willsky, A. S. and H. L. Jones, "A generalized likelihood ratio approach to state estimation in linear systems subject to abrupt changes," in *Proc. IEEE Conf. Decisions and Control*, pp. 846 (1974).

Wise, B. P., "An experimental comparison of uncertain inference systems," Ph.D. dissertation, Engineering and Public Policy, Carnegie-Mellon University, Pittsburgh, PA, (1986).

York, J. C., and D. Madigan, "Markov chain Monte Carlo methods for hierarchical Bayesian expert systems," in *Selecting Models from Data, Artificial Intelligence and Statistics IV*, edited by P. Cheeseman and R. Oldford, Springer-Verlag, New York, NY (1993).

Yu, C. C. and C. Lee, "Fault diagnosis based on qualitative/quantitative process knowledge," *AIChE Journal*, **37**, pp. 617-628 (1991).

Yuret, D., and M. de la Maza, "Dynamic hill climbing: Overcoming the limitations of optimization techniques," in *Proc. of the Second Turkish Symposium on Artificial Intelligence and Artificial Neural Networks*, pp. 254-260 (1993).

Yuret, D., "From genetic algorithms to efficient optimization," M.S. dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA (1994).

Zadeh, L. A., "The role of fuzzy logic in the management of uncertainty in expert systems," *Fuzzy Sets and Systems*, **11**, pp. 199-228 (1983).

Appendix A

Prior and conditional distributions

These distributions correspond to the 5 node network shown in Figure 4-5.

P(plugging=none) = 0.70
P(plugging=moderate) = 0.20
P(plugging=severe) = 0.10

P(atomization_quality=poor | plugging=none, injector=broken) = 0.90
P(atomization_quality=normal | plugging=none, injector=broken) = 0.10
P(atomization_quality=poor | plugging=none, injector=not broken) = 0.05
P(atomization_quality=normal | plugging=none, injector=not broken) = 0.95
P(atomization_quality=poor | plugging=moderate, injector=broken) = 0.80
P(atomization_quality=normal | plugging=moderate, injector=broken) = 0.20
P(atomization_quality=poor | plugging=moderate, injector=not broken) = 0.55
P(atomization_quality=normal | plugging=moderate, injector=not broken) = 0.45
P(atomization_quality=poor | plugging=severe, injector=broken) = 1.00
P(atomization_quality=normal | plugging=severe, injector=broken) = 0.00
P(atomization_quality=poor | plugging=severe, injector=not broken) = 0.85
P(atomization_quality=normal | plugging=severe, injector=not broken) = 0.15

P(injector=broken) = 0.10
P(injector=not broken) = 0.90

P(product_quality=normal | plugging=none) = 0.95
P(product_quality=low | plugging=none) = 0.05
P(product_quality=normal | plugging=moderate) = 0.40
P(product_quality=low | plugging=moderate) = 0.60
P(product_quality=normal | plugging=severe) = 0.10
P(product_quality=low | plugging=severe) = 0.90

P(H2_production=normal | atomization_quality=poor) = 0.10
P(H2_production=high | atomization_quality=poor) = 0.90
P(H2_production=normal | atomization_quality=normal)= 0.80
P(H2_production=high | atomization_quality=normal)= 0.20

Appendix B

Exhaustive Systematic Enumeration

The following table contains the exhaustive enumeration of all possible system states for the 5 node network shown in Figure 4-5.

P	=Plugging	(1=normal, 2=moderate, 3=severe)
AQ	=Atomization quality	(1=poor, 2=normal)
IT	=Injector tip	(1=broken, 2=not broken)
PQ	=Product quality	(1=normal, 2=low)
HP	=Hydrogen production	(1=normal, 2=high)

STATE P[P,AQ,IT,PQ,HP] = State Probability

State 0: P[1, 1, 1, 1, 1] = 0.7 0.9 0.1 0.95 0.1 = 0.005985
State 1: P[1, 1, 1, 1, 2] = 0.7 0.9 0.1 0.95 0.9 = 0.053865
State 2: P[1, 1, 1, 2, 1] = 0.7 0.9 0.1 0.05 0.1 = 0.000315
State 3: P[1, 1, 1, 2, 2] = 0.7 0.9 0.1 0.05 0.9 = 0.002835
State 4: P[1, 1, 2, 1, 1] = 0.7 0.05 0.9 0.95 0.1 = 0.002992
State 5: P[1, 1, 2, 1, 2] = 0.7 0.05 0.9 0.95 0.9 = 0.026932
State 6: P[1, 1, 2, 2, 1] = 0.7 0.05 0.9 0.05 0.1 = 0.000157
State 7: P[1, 1, 2, 2, 2] = 0.7 0.05 0.9 0.05 0.9 = 0.001417
State 8: P[1, 2, 1, 1, 1] = 0.7 0.1 0.1 0.95 0.8 = 0.00532
State 9: P[1, 2, 1, 1, 2] = 0.7 0.1 0.1 0.95 0.2 = 0.00133
State 10: P[1, 2, 1, 2, 1] = 0.7 0.1 0.1 0.05 0.8 = 0.00028
State 11: P[1, 2, 1, 2, 2] = 0.7 0.1 0.1 0.05 0.2 = 7e-05
State 12: P[1, 2, 2, 1, 1] = 0.7 0.95 0.9 0.95 0.8 = 0.45486 <<
State 13: P[1, 2, 2, 1, 2] = 0.7 0.95 0.9 0.95 0.2 = 0.113715
State 14: P[1, 2, 2, 2, 1] = 0.7 0.95 0.9 0.05 0.8 = 0.02394
State 15: P[1, 2, 2, 2, 2] = 0.7 0.95 0.9 0.05 0.2 = 0.005985
State 16: P[2, 1, 1, 1, 1] = 0.2 0.8 0.1 0.4 0.1 = 0.00064
State 17: P[2, 1, 1, 1, 2] = 0.2 0.8 0.1 0.4 0.9 = 0.00576

State 18: P[2, 1, 1, 2, 1] = 0.2 0.8 0.1 0.6 0.1 = 0.00096
State 19: P[2, 1, 1, 2, 2] = 0.2 0.8 0.1 0.6 0.9 = 0.00864
State 20: P[2, 1, 2, 1, 1] = 0.2 0.55 0.9 0.4 0.1 = 0.00396
State 21: P[2, 1, 2, 1, 2] = 0.2 0.55 0.9 0.4 0.9 = 0.03564
State 22: P[2, 1, 2, 2, 1] = 0.2 0.55 0.9 0.6 0.1 = 0.00594
State 23: P[2, 1, 2, 2, 2] = 0.2 0.55 0.9 0.6 0.9 = 0.05346
State 24: P[2, 2, 1, 1, 1] = 0.2 0.2 0.1 0.4 0.8 = 0.00128
State 25: P[2, 2, 1, 1, 2] = 0.2 0.2 0.1 0.4 0.2 = 0.00032
State 26: P[2, 2, 1, 2, 1] = 0.2 0.2 0.1 0.6 0.8 = 0.00192
State 27: P[2, 2, 1, 2, 2] = 0.2 0.2 0.1 0.6 0.2 = 0.00048
State 28: P[2, 2, 2, 1, 1] = 0.2 0.45 0.9 0.4 0.8 = 0.02592
State 29: P[2, 2, 2, 1, 2] = 0.2 0.45 0.9 0.4 0.2 = 0.00648
State 30: P[2, 2, 2, 2, 1] = 0.2 0.45 0.9 0.6 0.8 = 0.03888
State 31: P[2, 2, 2, 2, 2] = 0.2 0.45 0.9 0.6 0.2 = 0.00972
State 32: P[3, 1, 1, 1, 1] = 0.1 1 0.1 0.1 0.1 = 0.0001
State 33: P[3, 1, 1, 1, 2] = 0.1 1 0.1 0.1 0.9 = 0.0009
State 34: P[3, 1, 1, 2, 1] = 0.1 1 0.1 0.9 0.1 = 0.0009
State 35: P[3, 1, 1, 2, 2] = 0.1 1 0.1 0.9 0.9 = 0.0081
State 36: P[3, 1, 2, 1, 1] = 0.1 0.85 0.9 0.1 0.1 =.000765
State 37: P[3, 1, 2, 1, 2] = 0.1 0.85 0.9 0.1 0.9 =.006885
State 38: P[3, 1, 2, 2, 1] = 0.1 0.85 0.9 0.9 0.1 =.006885
State 39: P[3, 1, 2, 2, 2] = 0.1 0.85 0.9 0.9 0.9 =.061965
State 40: P[3, 2, 1, 1, 1] = 0.1 0 0.1 0.1 0.8 = 0
State 41: P[3, 2, 1, 1, 2] = 0.1 0 0.1 0.1 0.2 = 0
State 42: P[3, 2, 1, 2, 1] = 0.1 0 0.1 0.9 0.8 = 0
State 43: P[3, 2, 1, 2, 2] = 0.1 0 0.1 0.9 0.2 = 0
State 44: P[3, 2, 2, 1, 1] = 0.1 0.15 0.9 0.1 0.8 = 0.00108
State 45: P[3, 2, 2, 1, 2] = 0.1 0.15 0.9 0.1 0.2 = 0.00027
State 46: P[3, 2, 2, 2, 1] = 0.1 0.15 0.9 0.9 0.8 = 0.00972
State 47: P[3, 2, 2, 2, 2] = 0.1 0.15 0.9 0.9 0.2 = 0.00243

Appendix C

Probability Distributions for Network BN1

N1	p(N1)
1	0.6
2	0.3
3	0.1

N3	p(N3)
1	0.1
2	0.9

N9	p(N9)
1	0.85
2	0.15

N2	N1	N3	p(N2 | N1,N3)
1	1	1	0.90
2	1	1	0.10
1	1	2	0.01
2	1	2	0.99
1	2	1	0.40
2	2	1	0.60
1	2	2	0.30
2	2	2	0.70
1	3	1	0.99
2	3	1	0.01
1	3	2	0.90
2	3	2	0.10

N4	N1	N9	p(N4 | N1,N9)
1	1	1	0.90
2	1	1	0.10
1	2	1	0.20
2	2	1	0.80
1	3	1	0.05
2	3	1	0.95
1	1	2	0.20
2	1	2	0.80
1	2	2	0.05
2	2	2	0.95
1	3	2	0.01
2	3	2	0.99

N5	N2	p(N5| N2)
1	1	0.10
2	1	0.90
1	2	0.95
2	2	0.05

N6	N5	p(N6 | N5)
1	1	0.90
2	1	0.10
1	2	0.10
2	2	0.90

N7	N2	p(N7 |N 2)
1	1	0.90
2	1	0.10
1	2	0.10
2	2	0.90

N8	N7	p(N8 | N7)
1	1	0.20
1	2	0.75
2	2	0.25
2	1	0.80

N10	N9	p(N10 | N9)
1	1	0.90
2	1	0.10
1	2	0.20
2	2	0.80

N11	N9	p(N11 | N9)
1	1	0.70
2	1	0.30
1	2	0.05
2	2	0.95

N12	N9	p(N12 | N9)
1	1	0.75
2	1	0.25
1	2	0.15
2	2	0.85

N13	N9	p(N13 | N9)
1	1	0.99
2	1	0.01
1	2	0.01
2	2	0.99

Appendix D

Comparison of Crossover Strategies

Table D-1. Inference time (t) and best phenotype found (Ph) for a crossover strategy using a constant expansion level equal to 4, heuristically determined as half of the longest directed path. Diagnostic cases C300, C310, C320, C330, and C340.

		C300	C300	C310	C310	C320	C320	C330	C330	C340	C340	Sum
		t	Ph	t	Ph	t	Ph	t	Ph	t	Ph	
PS922	Opt cnt		4		0		3		4		1	12
	Ave	14.10	3.31E-11	16.40	8.04E-14	14.70	9.87E-09	14.60	2.35E-11	14.80	4.25E-12	
	Stdev	3.31	2.42E-11	3.31	6.16E-14	3.92	1.02E-08	2.41	2.48E-11	2.86	8.21E-12	
PS910	Opt cnt		7		2		2		4		6	21
	Ave	26.30	4.97E-11	26.30	1.99E-13	27.40	1.01E-08	27.70	3.35E-11	25.60	1.91E-11	
	Stdev	3.77	9.9E-12	3.43	2.72E-13	4.55	9.16E-09	4.92	1.56E-11	0.52	1.06E-11	
PS906	Opt cnt		7		4		1		0		6	18
	Ave	14.90	4.79E-11	16.00	2.94E-13	12.00	6.31E-09	12.70	8.83E-12	14.70	1.64E-11	
	Stdev	4.31	1.73E-11	4.76	3.54E-13	2.87	8.04E-09	2.45	7.14E-12	3.27	1.25E-11	
PS921	Opt cnt		8		2		3		4		3	20
	Ave	16.90	4.52E-11	17.90	2..1E-13	14.20	1.16E-08	15.70	2.7E-11	17.00	1.5E-11	
	Stdev	2.60	2.26E-11	3.03	2.63E-13	3.49	0.000000001	2.50	2.27E-11	3.97	9.57E-12	
PS913	Opt cnt		1		0		2		2		0	5
	Ave	5.90	6.03E-12	6.10	1.29E-14	6.50	6.35E-09	6.10	1.41E-11	6.20	3.22E-12	
	Stdev	0.57	1.75E-11	1.10	3.86E-14	1.35	1.03E-08	1.37	2.06E-11	1.14	3.84E-12	
Total			27		8		11		14		16	76

Table D-2. Performance for a crossover strategy using a uniform probability distribution for expansion level between 1 and R(=7)

		C300	C300	C310	C310	C320	C320	C330	C330	C340	C340	Sum
		t	Ph	t	Ph	t	Ph	t	Ph	t	Ph	
PS722	Opt Cnt		5		3		4		2		0	14
	Ave	12.7	3.87E-11	13.5	2.78E-13	11.8	1.03E-08	11.9	2.20E-11	12.2	3.41E-12	
	Stdev	2.83	2.60E-11	4.67	3.11E-13	2.15	1.15E-08	1.52	2.13E-11	3.05	5.84E-12	
PS710	Opt Cnt		6		0		6		3		4	19
	Ave	25.7	4.58E-11	22.8	5.38E-14	22.1	1.85E-08	22.7	3.08E-11	22.1	1.53E-11	
	Stdev	2.41	1.77E-11	5.83	5.88E-14	4.02	7.20E-09	4.14	2.03E-11	4.01	1.02E-11	
PS706	Opt Cnt		5		2		3		1		3	14
	Ave	15.3	4.94E-11	16.2	1.94E-13	13.2	8.10E-09	13.38	1.66E-11	14.2	1.47E-11	
	Stdev	3.47	9.22E-12	2.74	2.76E-13	1.32	1.07E-08	3.02	1.64E-11	1.93	1.02E-11	
PS721	Opt Cnt		5		2		3		2		3	15
	Ave	17.2	2.90E-11	16.4	1.91E-13	17.8	1.48E-08	18.8	1.63E-11	16.1	1.35E-11	
	Stdev	2.74	2.84E-11	3.20	2.74E-13	3.08	8.11E-09	3.94	1.98E-11	1.52	1.11E-11	
PS713	Opt Cnt		0		1		1		1		0	3
	Ave	5.7	1.75E-12	5.5	7.69E-14	5.9	3.26E-09	6.2	7.00E-12	5.5	8.38E-13	
	Stdev	1.49	2.76E-12	1.18	2.21E-13	0.99	7.43E-09	1.23	1.60E-11	1.08	2.07E-12	
Total		21		8		17		9		10		65

Table D-3. Probability distribution of expansion level (L) with a maximum at L=4.

L	0	1	2	3	4	5	6	7	8
P(L)	0	0.1	0.15	0.20	0.25	0.15	0.1	0.05	0

Table D-4. Performance for a crossover strategy with a maximum at L=4.

		C300	C300	C310	C310	C320	C320	C330	C330	C340	C340	
		t	Ph	t	Ph	t	Ph	t	Ph	t	Ph	Sum
PS822	Opt Cnt		2		0		5		2		1	10
	Ave	11	2.42E-11	12.3	2.83E-14	11.1	1.40E-08	10.9	1.70E-11	12.2	5.10E-12	
	Stdev	3.62	2.45E-11	3.27	4.77E-14	1.60	1.09E-08	2.56	2.03E-11	3.71	7.88E-12	
PS810	Opt Cnt		6		2		3		5		3	19
	Ave	23.9	4.73E-11	25.4	2.24E-13	26.4	1.25E-08	23.8	2.80E-11	23.5	1.50E-11	
	Stdev	4.01	1.72E-11	4.84	2.69E-13	7.99	9.23E-09	3.79	2.53E-11	3.98	9.53E-12	
PS806	Opt Cnt		2		3		3		2		1	11
	Ave	14	2.56E-11	16.1	2.60E-13	14	8.29E-09	13.2	1.70E-11	14.3	7.50E-12	
	Stdev	3.56	2.28E-11	3.48	3.21E-13	2.05	1.07E-08	3.08	2.01E-11	4.30	9.04E-12	
PS821	Opt Cnt		9		3		4		4		4	24
	Ave	18.2	5.38E-11	18.8	2.64E-13	16.3	1.66E-08	15.9	2.60E-11	17.6	1.20E-11	
	Stdev	6.80	6.48E-12	3.94	3.06E-13	3.37	7.88E-09	3.48	2.35E-11	5.87	1.24E-11	
PS813	Opt Cnt		0		1		2		1		0	4
	Ave	5.8	7.87E-12	5.6	9.22E-14	5	7.08E-09	4.6	5.30E-12	5.7	1.90E-12	
	Stdev	1.40	1.39E-11	1.17	2.18E-13	1.05	1.01E-08	1.07	1.62E-11	1.95	4.48E-12	
Total			19		9		17		14		9	68

Appendix E

Node Removal

A Bayesian network of general topology can be reduced to a 2 level network relating only faults and symptoms without intermediate variables. The 2 level structure is motivated by the potential savings in computational costs and the equivalent performance expected for abductive inference tasks. Two operations can be performed, the removal of a node and the combination of a cluster of nodes.

Simplifying a network will often result in a loss of information. Two models of different complexity that can provide the same results can be considered functionally equivalent and consistent. The joint probability distribution of faults given symptoms should be the same for both networks (before and after removing the intermediate nodes).

Nodes can be removed to create a new network consistent with the previous model, as illustrated in Figure A.

When a node is removed, only a subset of its Markov blanket requires updating, specifically the parents of children, (*i.e.* node F in Figure A), do not require updating.

The topology of the network must be changed by removing the node and updating the links of its immediate neighbors.

(1) For parents of the removed node (*i.e.* A), the new set of children is

$$Ch(A) = Ch(A) \cup Ch(B) - B$$

(2) For children of the removed node, the new set of parents is

$$Pa(C) = Pa(C) \cup Pa(B) - B$$

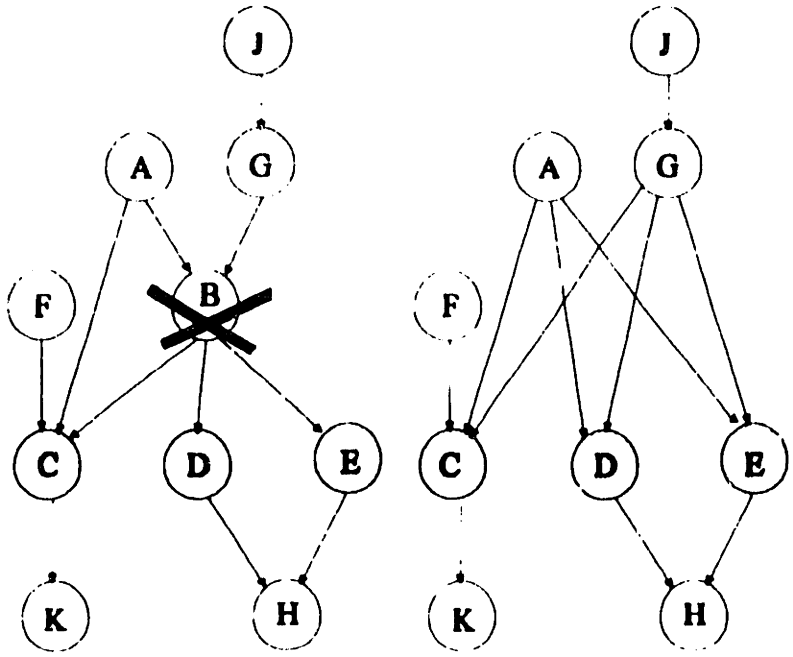


Figure. Belief network model before and after removal of node B.

Probabilistic parameters must be updated only for children of the removed node. Parameters for the new network can be calculated based only on parameters of the previous network in four simple steps:

(1) Order the nodes in a list such that for any two nodes X and Y, if Y is a successor of X, then Y is placed after X in the list. (*i.e.* [A,G,B,F], [A,F,G,B], or [G,A,B,F]).

(2) Write the general expression of the probability starting with the prior probability of the first element in the list and adding additional factors, one for each term in the list, with each factor conditioned on all the previous elements in the list. (i.e., for [A,G,B,F] the expression is p(C|A,G,B,F) = p(A)p(G|A)p(B|A,G)p(F|A,G,B).)

(3) Simplify each factor by removing, from the set of nodes on which it is conditioned, all those nodes which are not its immediate successors or parents. (*i.e.* p(C|A,G,B,F) = p(A)p(G)p(B|A,G)p(F).)

(4) Marginalize by summing over all possible states of the removed node. The following formula can be used to calculate the The table of conditional probabilities for node C would contain terms of the following form:

$$p(C_j|A_k G_m F_n) = p(A_k)p(G_m)p(F_n)\sum_{i=1}^{i=|B|} p(B_i|A_k G_m)$$

Since the number of parameters affects the complexity of the acquisition it is desirable to know the effect of the removal of a node on the total number of parameters. The parameter differential (change in the number of probabilities required) is:

$$\Delta N = N_2 - N_1 =$$

$$\sum_{i=1}^{i=|\Phi_R|} [S(c_i)\{\prod_{t=1}^{t=|\Pi_{c_i,r}|} S(p_t) - \prod_{u=1}^{u=|\Pi_{c_i,r}|} S(p_u)\}]$$

$$-S(R)\prod_{k=1}^{k=|\Pi_R|} S(p_k)$$

where

R=removed node,

P_R={p ' p is parent of R}

F_R={c ' c is a child of R'

S(n)=number of states of node n

$\prod_{r,c_i}$ ={p ' parents of node c_i at network r}

Appendix F

Alternative inference approaches: Mixed Integer Non-Linear Programming

Abductive inference in the Bayesian belief network framework can be viewed as an optimization problem in a large discrete multi-dimensional search space. The objective is to find the *best* set of discrete values for the unknown variables (given the values of the known variables), in other words, to maximize the probability of the inferential hypothesis given the evidence. Commonly used optimization techniques such as integer and mixed integer programming techniques have been successfully used in optimization problems involving discrete variables. However, MINLP techniques are not adequate for inference with Bayesian networks for reasons which are discussed in what follows.

The optimization for abductive inference must be performed at a global level to find the most probable, globally consistent hypothesis. Due to the interactions that exist among variables within the network, a decomposition in subproblems to be solved in sequence is, in general, not possible. This requirement of dynamic programming is not satisfied in Bayesian networks.

Several algorithms exist for Mixed Integer Non-Linear Programming (MINLP) problems. One class of MINLP algorithms is based on a branch and bound search (Gupta and Ravindran, 1985). Methods which improve the efficiency of convex MINLP problems in which the bottleneck lies in the combinatorial search for the binary variables have been proposed (Quesada 1992). Geoffrion (1972) proposed the Generalized Benders decomposition. The outer approximation algorithm (Duran and Grossmann, 1986) initially limited to convex problems was extended to nonconvex MINLP problems by Viswanathan and Grossmann (1990).

Variables in a Bayesian network are not restricted to being binary. It is therefore necessary to preprocess the model to an equivalent formulation which contains only binary variables. The size of the problem consequently grows.

Some approaches to MINLP problems handle integer variables as if they were continuous and round the optimal solution to the nearest integer value. The Branch and Bound search involves the relaxation of discrete conditions of the binary variables. If an integral solution exists, the optimum has been found, otherwise, it constitutes a lower bound (when performing minimization). In the next step, a systematic search is performed on a tree whose nodes represent subproblems where a subset of the discrete conditions have been relaxed. Integer solutions at these nodes constitute upper bounds. (Quesada 1992). Relaxation would work for continuous variables represented as discrete variables. For example, temperature can be quantified by discrete states which are intrinsically ordered (0=very low, 1=low, 2=normal, 3=high, 4=very high). The relaxation of discrete conditions is not meaningful for variables whose discrete states are not intrinsically ordered. Other discrete variables have states which can not be ordered in a meaningful way, such as the states of a valve, (0=open, 1=closed, 2=stuck open, 3=stuck closed). Handling the valve states as a continuous variable is meaningless, and

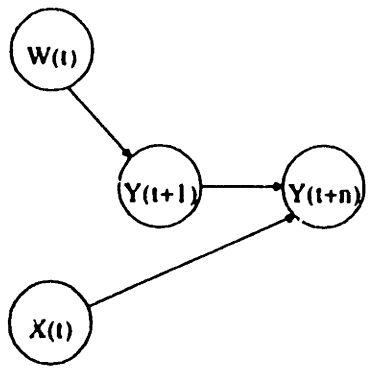
The complexity of branch and bound search in the discrete space usually renders most BN problems intractable. Bounded search approaches have been proposed in Bayesian networks (Poole, 1993). The algorithm is efficient only with probability distributions with extreme probabilities (close to 0 or 1), however the algorithm is very inefficient when the distributions become less extreme.

Appendix G

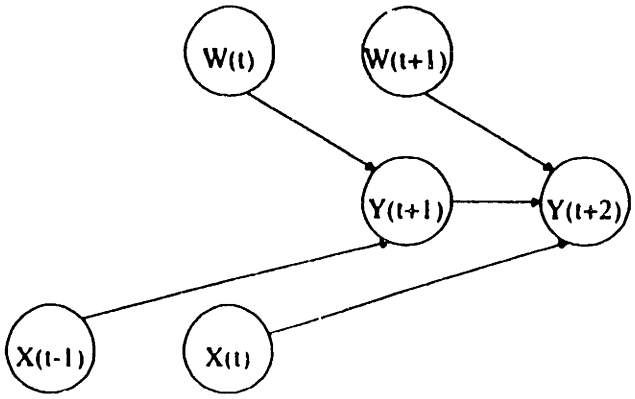
Inverse response modeling with MSBNs

Inverse response behavior is of interest when modeling and controlling dynamic systems and results from the combined effects of at least two competing mechanisms with at least two of them having effects of opposite directions and different timescales.

MSBNS can model inverse response behavior using time indexed discretized variables. In the example, Y(t+1) the short term response is probabilistically dependent on W(t). The same variable Y at a later time, Y(t+2), the long term response is a function of X(t). The interval [t,t+n] refers to the long term response, and [t,t+1] refers to the short term response. The value of n quantifies the relative magnitude of both intervals.



This simple model assumes that once W and X change at time t, they remain at the same value. Additional complexity would result if W(t+1) were different from W(t). The modeling is still possible by adding a node for W(t+1) and redefining the probability distribution of Y(t+2) to reflect the simultaneous combined effect of W(t+1) and X(t). A more general representation is:

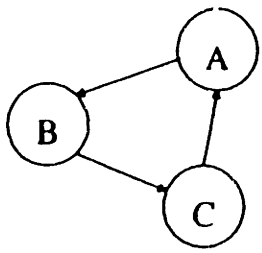


Multivariable systems pose a more complex problem which can still be modeled in the MSBN framework. Note that inverse response behavior may be incorrectly detected when a control system exhibits underdamped behavior (Finch, 1989).

Appendix H

Directed cycles in Bayesian networks

Bayesian networks are, by definition directed acyclic graphs (DAGs), which means that directed loops are not permitted. When using message passing algorithms on networks with directed loops, the messages may be sent indefinitely, or the converged results may be incorrect. The hypothesis that algorithms which do not rely on message passing are insensitive to the presence of directed loops, such as the GA is proven incorrect with the following two counterexamples.



For simplicity assume A, B, and C are discrete binary variables, and the following notation: P(B=A)=1 means P(B=0|A=0)=1, P(B=1|A=1)=1, P(B=1|A=0)=0 P(B=0|A=1)=0.

Example 1

P(B=A)=1, P(C=E)=1, P(A=C)=1 is a consistent system description. Only two system states, (0,0,0) and (1,1,1) are possible with P>0, the rest have P=0. All impossible scenarios are correctly assigned to 0, but the probability of each possible state is calculated to be 1, making the sum of the probabilities of the possible states > 1! In the absence of any other information, each possible state would have P=1/2. Nevertheless, the relative probability of each system state is still correct.

Specify any of the 3 variables and the other 2 are determined.

Specify 2 with the same value and the 3rd is determined.

Specify 2 with different value and all possible assignments for the 3rd have P=0, indicating an inconsistency.

Specify 3 with one having a different value and the probability of the fully instantiated scenario is calculated to be 0.

Example 2

If P(B|A)=1/2, P(C|B)=1/2, and P(A|C)=1/2, a consistent system description, all possible system states have the same probability of 0.5^3=0.125. The sum of P over all states is 8*0.125=1. However, note that a partial system measurement does not help to estimate unmeasured variables.

In general, if P(B|A)=P(C|B)=P(A|C)=y, then P(A=a,B=b,C=c)=y^3, for any set of values of {a, b, c}, and every full instantiation will have the same probability. The sum, however, of p for all scenarios will not be one except for the case y=1/2.

Example 3

P(B=A)=1, P(C=B)=1, P(A=C)=0 is an inconsistent distribution.

0, 1, 2 or 3 variables can be specified and all the scenarios will give P=0.

Results obtained form the GA are based on a comparison of P(full instantiation). If the network distribution is inconsistent, or a scenario inconsistent with the distribution is proposed, a hypotheses with non-zero probabilities will not be found because they do not exist. The calculation of the metric of a hypothesis will have the same computational cost.

Wise (1986) discusses the use of linear constraints for handling directed cycles of length two and Wen (1989) proposed a method to calculate the initial network distribution by writing conditional probabilities as linear equality constraints and then solving the Maximum Entropy/Minimum Cross Entropy problem using the Lagrange multiplier method.

Appendix I

Diagnostic System: User's Guide

GALGO ©

Genetic Algorithm Approach to Abductive Inference on Graphs for Complex Systems

Diagnostic System

User's Guide

Carlos Rojas-Guzmán
Mark A. Kramer
M.I.T.
1994.

CONTENTS

1 Introduction

1.1 System and User's Guide description

1.2 Theoretical background

2 Installation instructions

2.1 Hardware and software requirements

2.2 Project installation

3 Usage instructions

3.1 Model

3.1.1 Topology

3.1.2 Probabilistic parameters

3.2 Symptoms (Plant Data)

3.3 Diagnosis

3.4 Optimization options

4 System description

- Project Files

- Software Options

- Architecture

5 References

1 Introduction

1.1 System and User's Guide description

This *User's Guide* describes GALGO, a computer program designed to perform on-line diagnosis for industrial plants (or nuclear plants, or financial systems, or medical systems) given remotely available plant data (or financial data, or patient symptoms). This document explains how to install the program and how to use it. In addition, this manual describes the steps necessary to modify and create new models and discusses the diagnostic output which can optionally include useful data for development purposes.

Section 1.2 briefly describes the theory behind the diagnostic system. In Section 2 detailed instructions are provided to either install a copy of the program to be immediately run or to install all the source code to enable modification and compilation of the software. Section 3 describes the conventions on the model of M-15 plants to enable the user to modify the existing model or to create new models. Usage instructions are given and illustrated step by step with an existing model and a sample set of plant output data. Section 4 includes information on project files, and options for the compiler and Section 5 includes some references to the research work on which the GALGO system is based.

1.2 Theoretical background

The availability of accurate real-time on-line computer-aided diagnosis can significantly contribute to safe and efficient process operations. The computer program described in this manual implements a methodology based on a probabilistic reasoning approach to monitor and diagnose faults, process disturbances and equipment failures based on the interpretation of local sensor data. GALGO is an object oriented implementation of the diagnostic methodology.

Diagnostic or abductive inference involves finding the best (which in this context means the most probable) globally consistent description of the state of the system given an incomplete, noisy and often conflicting subset of the measurements.

Uncertainty is intrinsic to fault diagnosis of chemical processes. The need to model uncertainty derives primarily from engineering limitations both in process modeling and measurement. Consequently, deterministic models for the effects of faults cannot always be constructed, and measurements to diagnose every fault uniquely may be missing, noisy or inaccurate. GALGO is part of a research project whose goal is to develop and exploit the advantages of a probabilistic framework capable of modeling and handling uncertain, incomplete and conflicting information to perform fault diagnosis in complex processes.

Fault diagnosis involves two basic tasks, (1) modeling the system behavior and (2) performing inference using the model. Bayesian networks (also called belief networks) have been suggested as a method of representing and solving chemical engineering diagnosis problems (Rojas-Guzmán and Kramer, 1992). The Bayesian network (Pearl, 1988) is a formal graphical representation combined with quantitative parameters, capable of modeling interactions in complex systems where variables are represented by discrete

states. Belief networks have a sound theoretical basis, are consistent with probability theory, use results from graph theory, and constitute a powerful tool in probabilistic reasoning.

Algorithms exist to perform diagnostic inference on these graphs rigorously, using the laws of probability, resulting in the most probable assignment of all unobserved variables. In a Bayesian network, any variables can be observed or unobserved, giving the method great flexibility in dealing with partially-measured systems. Recently developed methods to propagate probability information use distributed parallel computations (Pearl, 1988). Approximate solution techniques which reduce calculation time and generate rankings of possible hypotheses have also been introduced (Rojas-Guzmán and Kramer, 1993a). A genetic algorithm has been developed and implemented by posing inference as search in a large discrete multi-dimensional space where the metric is the probability of each diagnostic hypothesis. Potential hypotheses are iteratively selected from an initially random set, and combined to improve the quality of the solution set.

The modularity and local interactions in the network representation simplify knowledge acquisition, model construction and maintenance. The implementation described in this manual is part of an effort to test and validate the system and a model on a non-cryogenic high purity N_2 plant. The model was constructed based on the process flowsheet, historical data from similar plants, behavioral descriptions and troubleshooting expertise from plant designers and operators. The system performs on-line remote monitoring and diagnosis of single or multiple faults on an unmanned plant with local computer control.

2 Installation instructions

2.1 Hardware and software requirements

GALGO has been designed to run on a PC IBM-compatible machine on the WINDOWS system or directly on the DOS system. The source code has been written in the C++ object oriented language. C++ is a highly portable language and the program was written minimizing the use of functions specific to the compiler used. However, it is recommended to modify and maintain the program on the same combination of software and platform used for its development.

GALGO was developed and tested on the following system:

Hardware		
	Computer	PC from Gateway2000
	Processor	486 DX2-66 Mhz
	Memory (RAM)	8 Megabytes
Software		
	Compiler	Borland C++ V. 3.1
	Operating System	Windows 3.1 DOS 5.0 or 6.2

Before installing the GALGO software, the following should already be installed:

(1) DOS 5.0 or 6.2

(2) Windows 3.1

(3) Borland C++ 3.1 for Windows or Microsoft Visual C++

GALGO produces output text files in formats ready to be used by a text editor and a spreadsheet. The following programs have been used.

(4) Notepad (part of the Windows 3.1 operating system)

(5) Excel (version 4.0)

2.2 Project installation

Before starting any installation procedures make a backup copy of the distribution disks. Make a complete working copy of the distribution disks immediately after you receive them, then store the original disks away in a safe place.

2.2.1 Option 1: Install executable only

A compiled and executable file is contained on Disk 1, labelled as *Executable*. To run the executable called *galgo.exe*,

(1) Create a new directory in the hard disk.

(2) Copy all the files contained in disk 1 into the new directory in the hard disk.

(3) Open File Manager.

(4) Click twice on the icon named *galgo.exe* and a new window should appear.

(5) Follow the instructions from the new window (*Section 3. Usage*, on this document has additional useful information).

2.2.2 Option 2: Installation for development and maintenance

Disk 2 labelled as *Source Code* is necessary for installation, it contains source files and has four directories, (1) *include*, (2) *output*, (3) *project*, and (4) *source*. Since GALGO is a multiple file system a *project* can be ***installed*** within the Borland C++ system by performing the following steps:

(1) Create a directory in your hard disk where you would like to keep the source code files. Call it *c:\delivery*. If you want to use a different name, follow instructions on OPTION 3.

(2) Copy all the files contained in the *source* directory of Disk 2, labelled *Source Code*, into the newly created *delivery* directory.

Files in this set have the **.cpp* extension and contain C++ code.

(3) Copy all the files contained in the *include* directory of Disk 2, labelled *Source Code*, into the newly created *delivery* directory.

Files in this set have the **.h* extension and contain header files.

(4) Copy all the files contained in the *project* directory of Disk 2, labelled *Source Code,* into the *C:\borlandc\bin* directory on your hard disk.

(5) Open the Windows version of Borland C++ 3.1 by clicking on its icon from *Program Manager.*

(6) Click on *Project* on the top menu, and select the *Open Project* option. A small window will appear with available projects, if *galgo2.prj* appears there, select it. If *galgo2.prj* does not appear, the type in the *File name* section *c:\borlandc\bin\galgo2.prj* and click on the *OK* button or press *Enter.*

(7) To compile GALGO, select *Compile* from the top menu, and click on the *Build all* option. (Files with the **.obj* extension should be created and an executable will be produced after linking them with the name *galgo2.prj*)

(8) To run GALGO, select the *Run* option from the top menu.

(9) A new window will be created. Follow the instructions that will appear in the window. More information on how to use it is included in Section 3 of this manual, *Usage instructions*.

2.2.3 Option 3: Installation for development and maintenance

Disk 2 labelled as *Source Code* is necessary for installation, it contains source files and has four directories, (1) *include*, (2) *output*, (3) *project*, and (4) *source*. GALGO is a multiple file system and a *project* can be ***created*** within the Borland C++ system by performing the following steps:

(0) Consider using OPTION 2 first. OPTION 3 is recommended only a) if you want a customized installation, b) if you have a special configuration of your hard drive (*i.e.* the name of the hard drive is not C:\) or c) if you have any problem with OPTION 2.

(1) Create a directory in your hard disk where you would like to keep the source code files. Give it a name, for example *c:\delivery* but you can use any other name.

(2) Copy all the files contained in the *source* directory of Disk 2, labelled *Source Code*, into the newly created *delivery* directory.

Files in this set have the **.cpp* extension and contain C++ code.

(3) Copy all the files contained in the *include* directory of Disk 2, labelled *Source Code*, into the newly created *delivery* directory.

Files in this set have the **.h* extension and contain header files.

(4) Open the Windows version of Borland C++ 3.1 by clicking on its icon from *Program Manager*.

(5) Click on *Project* on the top menu, and select the *Open Project* option. Create a new project by typing a name with the **.prj* extension in the small window

that will appear with available projects. The default directory where you keep your projects will be used unless you specify another one. By using the default directory, the next time you select the *Open Project* option, the project created in this step will appear among the options.

(6) All the files with the **.cpp* extension which were copied in step 2 into the *c:\delivery* directory should now be added to the project. Click on the icon with the green plus sign or select the *Project* option, and then the *Add item* option. Type *c:\delivery* at the filename section and press *Enter*, all the GALGO **.cpp* files should appear. Select one at a time and click on the *Add* button. When you finish click with the mouse on the *Done* button.

(8) In order to set the options for the project select *Options* from the top menu and select the options which are detailed in Section 4.

(9) Close the project to save it and open it again.

(7) To compile GALGO, select *Compile* from the top menu, and click on the *Build all* option. (Files with the **.obj* extension should be created and an executable will be produced after linking them.)

(8) To run GALGO, select the *Run* option from the top menu.

(9) A new window will be created. Follow the instructions that will appear in the window. More information on how to use it is included in Section 3 of this manual, *Usage instructions.*

3 Usage instructions

Once the system has been installed according to instructions contained in Section 2, it can be run with an example.

(1) From the Borland C++ compiler, select Project and open the *galgo2.prj*

(2) Select Run and a small window with a simple temporary interface will appear with instructions.

(3) At the prompt type the name of a set of symptoms, for example *caset5.txt*

(4) Type *0* if you agree with the default options shown for the algorithm. For a first run it is recommended to accept the defaults.

(5) If you would like to run the system with a different set of symptoms (computer plant output) create or modify the Symptoms file (this one called *caset5.txt*) and run it again.

The following subsection describe in more detail the model input files and output files, however, reading the rest of the section is not necessary to run the system under routine conditions.

For plant monitoring and diagnosis a new case file with symptoms should be created based on the plant computer data. The case file is the one on which the diagnostic conclusions will be based. The following sections explain the contents of the case files which are the ones to be routinely created and constitute the input to the program. An explanation of

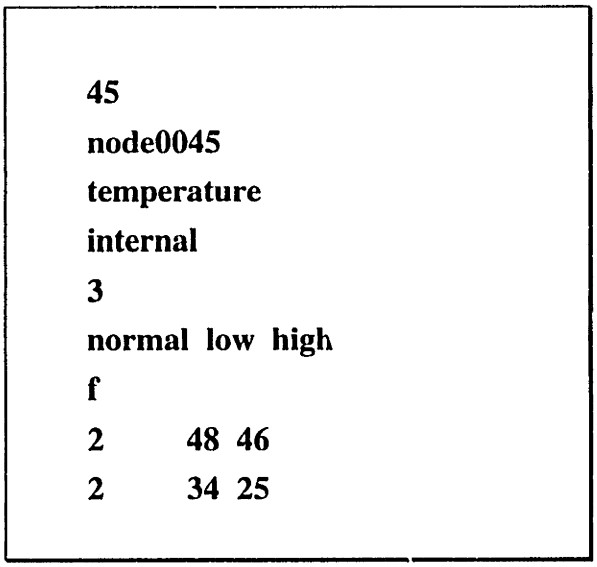
the model files is included to enable their modification which can be part of a continuing development and maintenance effort.

3.1 Model

The GALGO system describes a chemical plant with a probabilistic model called a Bayesian belief network. The model has two main components, the network topology and the network parameters which are described in the following subsections. The model is described by independent text files which can be easily modified or created in a simple text editor such as *Notepad* which is included in the Windows operating system. The files which contain the model are contained both in Disk 1 and on Disk 2. The file *topology.txt* describes the topology and the file *probabil.txt* describes the parameters. For a summary on the semantics and theory behind GALGO refer to Section 1.2 *Theoretical Background* or find more details by looking at the references listed in Section 5.

3.1.1 Topology

A Bayesian belief network, also referred to as Bayesian network is a graph with nodes and arcs. The description of the topology for the model contains a graphical representation of the network and the file *topology.txt* which formally characterizes the network. The file begins with the number of nodes (or variables in the model), and a paragraph follows for each node. The components of a typical node descriptions are described below:



45	This is the variable number.
node0045	This is the node name.
temperature	This is the variable name.
internal	This describes a node as root,internal or leaf.
3	This is the number of discrete states of the node.
normal low high	These are the names of the states of the node:
f	*f* is used if the value of the variable is unknown *t* indicates that the value is known in advance
2 48 46	The first number shows the number of parents of the node (2 in this case), and the node numbers of the parents follow (parents are nodes 48 and 46 in this case)
2 34 25	The first number shows the number of children of the node (2 in this case), and the node numbers of the children follow (34 and 25 in this case).

An similar paragraph is used for each node. Variable names should include only letters and underscore symbols with no spaces in between. Variable names should be separated by at least one space. All variables should have an unknown value (*f*) by default. Root nodes have no parents, therefore a 0 should be included in the corresponding line and, of course, no node numbers will follow. For leaf nodes a 0 should appear in the last line with no node numbers following it. For clarity at least one empty line is left between paragraphs.

3.1.2 Probabilistic parameters

The second component for the model is a set of prior and conditional probabilities. A typical probabilities table follows with an explanation of the convention used in it.

Node 32, for example is a root node with 2 states. Since it is a root node a set of prior probabilities is associated with it, one for each of the discrete states it can assume. The prior probability of variable 32 being in its first state (state 0) is 0.95, and that for its second state (state 1) is 0.05. Note that states are numbered starting with 0 in this file.

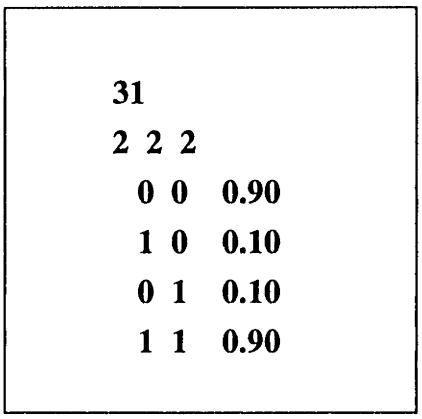
```
32
1 2
0  0.95
1  0.05
```

The first line contains the node number (32 in this case), the next indicates the number of dimensions in the array, which is equal to the number of parents plus 1. Root nodes have no parents, therefore a 1 dimensional array is necessary. The following numbers in the same line are the sizes of the dimensions of the array which indicate the number of states of the nodes involved (node 32 and all its parents, if any).

The following paragraph describes the parameters for node 31 which has 1 parent, (therefore the number of dimensions in the array is 2, and consequently the line begins with a 2). The second number in the line (a 2 also) indicates the number of states of node 31, and the third number shows the number of states of its parent (which is 2 also). The next 4 lines describe each one of the elements of the array, which are conditional probabilities since node 32 is not a root node. The third line in the paragraph (0 0 0.90) says that

$$P(\text{Node32}=0 \mid \text{Parent1}=0)=0.90,$$

in other words, the probability of node 31 being in state 0, given that its parent node is in state 0 is 0.90.



3.2 Symptoms (Plant Data)

The GALGO system uses as one of its input files, a text file with variable names and, if their value is known, then their discrete state value. A typical section of an input file (which contains symptoms or evidence) is printed below. The first number in the text file is the number of variables in the model (in this case 66). Each of the subsequent lines start with the node number followed by at least one space, and one letter (*f* if the variable is unknown, or *t* if the variable is known). The name of each variable is contained in the file describing the topology. In the example that follows, variable 1 is known to be in its discrete state 1 (in other words, the value of variable 1 is equal to 1), the value of variable 2 is unknown, and the value of variable 17 is known to be in state 2.

```
66

1 t 1
2 f
3 f
4 f
...
15 f
16 t 1
17 t 2
...
64 f
65 f
66 f
```

3.3 Diagnosis

The GALGO diagnostic system generates several output files during each run. These files describe the diagnostic conclusions and their ranking index, a measure of their relative probabilities. Values closer to 0 are less probable, given the available evidence. Other files contain a description of the run including intermediate values of algorithm parameters and useful data for development purposes. The files created at each run are automatically numbered.

For run 127 for example, the following files were created (**.txt* files can be read in a simple text editor and **.xls* files can be read in a spreadsheet such as Excel 4.0 by selecting *File*, then *Open*, clicking on the *Text* button and selecting as *Column delimiter* a comma).

bestn127.txt

Contains an English description of the top N diagnostic conclusions. This is the most useful output file for routine use. The following box shows part of the file with the first two diagnostic conclusions from a run. The best description appears first, followed by the second best. The number of top hypotheses to store can be specified in the program. Each paragraph contains the probabilistic index of the hypothesis, the coded hypothesis and its translation to English, focusing on a subset of variables considered important and showing them if they are in an abnormal state. This is a simple temporary user interface.

Evidence file used: caset2.txt.

Top 5 from among the best at each generation.

Run identifier: 070.

Consider the following ranked scenarios:

The following hypothesis has a probabilistic ranking index of 1.626605e-10

1111111112121131312111111111211112311222111111111111111113212122 11

The true O2 concentration is suspected to be high.

Internal fouling in the aftercooler is high.

The system pressure drop is suspected to be high.

The following hypothesis has a probabilistic ranking index of 1.536238e-10

1111111112121131312111111111211112312222111111111111111113212122 11

The true O2 concentration is suspected to be high.

Which might result from low ambient temperature

Distribution of customer flow demand: with_spikes.

Internal fouling in the aftercooler is high.

The system pressure drop is suspected to be high.

The following files are useful for development purposes:

topnp127.xls Top n probabilistic ranking indices generated

topng127.xls Top n hypotheses (coded)

bestp127.xls Locally best indices at each generation.

bestg127.xls Locally best hyotheses (coded) at each generation

genot127.xls Set of ranking indices available at end of run

pheno127.xls Set of coded hypotheses at end of run

histo127.xls Recorded history of algorithm progress containing the following information which can be read as an Excel file.

column0=generation number
column1=evolving probability mass
column2=breeding probability mass
column3=num improved mutations in this generation (positive mutations)
column4=number of negative mutations
column5=accumulated mutations in this generation
column6=positive mutation fraction in this generation
column7=accumulated offspring
column8=positive offspring
column9=negative offspring
column10=positive offspring fraction
column11=best phenotype in this generation
column12=best phenotype found so far (off-line performance)
column13=average phenotype in this generation (on-line performance)
column14=average evolving probability
column15=average breeding probability
column16=attempted mutations on instantiated gene
column17=exchanged block size from semantically close crossover_3 bn
column18=average exchanged block size from crossover_3
column19=convergence ratio

summa127.txt Summary of parameters used for the run. This file is useful to reproduce a run, using the same parameters and the same random number seed. Store this file, along with the topology, probabilistic parameters and evidence file to report a bug, or reproduce a run.

3.4 Optimization options

Among performance criteria of a diagnostic system are the diagnostic accuracy and the response time. Real-time operation is desirable when monitoring simultaneously or sequentially several plants, and is required when an immediate decision will be made based on the diagnostic conclusion.

Running time has decreased significantly and can be further reduced by using the following options.

Option 1:

A comparison of configurations was performed using different operating systems. These results show a significant reduction when running on DOS instead of on Windows.

Option 2:

Run the software on a Pentium PC-computer, commercially available for less than 2000 dollars (*i.e.* A Gateway2000, Model P5-60, 60 Mhz Pentium CPU, 8 MB RAM, 424 MB hard drive, costs $2300 US dlls. as of 10/94).

Option 3:

Reduced record keeping during a run also saves time. A more efficient approach makes all record keeping activities optional and has been implemented.

Option 4:

Hierarchical model decomposition. The use of several smaller models can be constructed.

Option 5:

A refined model not only reduces the search space but, more importantly, is expected to accelerate convergence by providing a coherent landscape.

Option 6:

Through the measurement of more variables, the search space is reduced and the quality of diagnosis should improve and should require less time.

Option 7:

Parameter optimization of the genetic algorithm should continue to reduce the number of parallel runs required.

4 System description

Part 1: Project Files

The GALGO project is made up of the following filesThe following header files are included in Disk 2 *Source Code* inside the directory *include.*

bn_math.h	basic combinatorial math functions
mat_2d.h	dynamically allocated two-dimensional matrix object
dyn_str.h	dynamically allocated strings
vect.h	dynamically allocated vector object
multidim.h	dynamically allocated multidimensional array
int_list.h	dynamically allocated list of integers
nd_queue.h	queue containing pointers to void (node)
node_cls.h	basic type for graph algorithms
node_lst.h	list of pointers to nodes
nd_vect.h	one dim array of pointers to node
bel_net.h	generic Bayesian belief network class made up of nodes
maximum.h	max value,ties,and positions of dvect
err_mess.h	general error handling functions
vect_set.h	set of vectors
syst_inf.h	hypotheses ranking through exhaustive search
qcksrt.h	quicksort sorting algorith
piksort.h	sorting algorithm
galgo.h	approximate inference genetic algorithm
int_face.h	elementary interface

The GALGO project is made up of the following files which are included in Disk 2 *Source Code* under directory *source*.

md_alloc.cpp	md_elem.cpp	piksort.cpp	bn_main.cpp
err_mess.cpp	galgo.cpp	int_list.cpp	maximum.cpp
nd_queue.cpp	node_cls.cpp	node_lst.cpp	int_face.cpp
qcksrt.cpp	vect.cpp	vect_set.cpp	mat_2d.cpp
syst_inf.cpp	bn_math.cpp	bel_net.cpp	

4 System description

Part 2: Architecture

The GALGO system can be best described by a set of components organized as classes of objects. Classes are the basis for abstract data types and object oriented programming. Encapsulation mechanisms package the internal implementation details of the type and the externally available operations and functions that can act on objects of that type. Function prototypes allow functions to be completely checked as to number of arguments and type.

All the tasks of the GALGO diagnostic system are coordinated by one main function (file *bn_main.cpp*) that provides a user interface, reads a model definition, reads process measurements, creates an evolutionary world object, and initiates world evolution with user specified options.

The following are the main classes of the system:

1. Evolving world class

A *world* object has attributes, data structures and functions which support evolutionary programming capabilities. One of the key functions of the *world* object is an implementation of a graph-based genetic algorithm that guides the evolution of the world with the purpose of obtaining the most likely diagnostic hypothesis. (Files *galgo.cpp* and *galgo.h*).

2. Belief network class

A belief network object contains a description of the network including its size, node members, topology and functions to define and modify itself. (Files *bel_net.cpp* and *bel_net.h*). Additional functions to calculate the metric of a given diagnostic hypotheses or to count the size of the search space are available (File *metric.cpp*).

3. Node class

Nodes are the building blocks of belief networks. Nodes keep track of their relations with other nodes (parents and children), and the discrete probabilistic distributions which quantify their interactions with their parent nodes. (File *node_cls.cpp* and *node_cls.h*).

4. Vector class

Two classes of vectors are available. One class, vect, contains *long int* and the other, dvect, contains *double* elements. Pointers fit in the *long int* version of vectors and are often used to store pointers to nodes or other objects. (Files *vect.cpp* and *vect.h*).

5. Node list class

A node list is a linked list whose elements are object nodes. (Files *node_lst.cpp* and *node_lst.h*).

6. Matrix class

Two dimensional arrays which store *integers* or *doubles* which encapsulate checking bounds for indices and have useful error messages to identify the source of attempts to illegally access their elements.

7. Multiple dimension array

N-dimensional arrays which store *double* numbers. The main role of these arrays is to store and handle the probabilistic values of the belief network models. (Files *md_alloc.cpp* and *md_elem.cpp*).

8. Auxiliary functions

Additional support functions include error-message handling (file *err_mess.cpp*), temporary interface functions (file *int_face.cpp*), auxiliary math functions (file *bnmath.cpp*), sorting routines (*piksort.cpp*).

5 References

Holland, J. H. *Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, 2nd ed. MIT Press, Cambridge, MA (1992).

Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA (1988).

Rojas-Guzmán, C., "An evolutionary programming approach to probabilistic model-based fault diagnosis of chemical processes", Ph.D. dissertation, Department of Chemical Engineering, Massachusetts Institute of Technology, Cambridge, MA, (Feb. 1995).

Rojas-Guzmán, C., and M. A. Kramer, "An evolutionary computing approach to probabilistic reasoning on Bayesian networks", for submission to *Evolutionary Computation*, (1995).

Rojas-Guzmán, C., M. A. Kramer, "A probabilistic methodology for monitoring and fault diagnosis," *AIChE Fall National Meeting*, Session on Monitoring, Detection and Data Interpretation in Process Operations, San Francisco, CA (1994).

Rojas-Guzmán, C., and M. A. Kramer, "Multi-Stage Bayesian Networks subsume digraph and residual-pattern approaches to fault diagnosis," in *Proc. Fifth Int. Symposium on Process Systems Engineering*, Kyongju, Korea, pp. 947-952, (1994).

Rojas-Guzmán, C. and M. A. Kramer, "GALGO: A Genetic ALGOrithm decision support tool for complex uncertain systems modeled with Bayesian belief networks," in *Proc. Ninth Conf. on Uncertainty in Artificial Intelligence*, Washington, D.C. pp. 368-375 (1993a).

Rojas-Guzmán, C. and M. A. Kramer, "Comparison of belief networks and rule-based expert systems for fault diagnosis of chemical processes," *Engineering Applications of Artificial Intelligence*, **6**, pp. 191-202 (1993b).

Rojas-Guzmán, C. and M. A. Kramer, "Belief networks for knowledge integration and abductive inference in fault diagnosis of chemical processes," in *Proc. of the Int. Federation of Automatic Control Int. Symposium: On Line Fault Detection and Supervision in the Chemical Process Industries*, Newark, DE, pp. 14-19 (1992).

Appendix J

GALGO©: Source code

This appendix contains the source code for the GALGO system implementing a graph-based genetic algorithm to perform abductive inference in Bayesian networks. The following files are listed:

bn_main.h	bn.cpp
bel_net.h	bel_net.cpp
node_cls.h	node_cls.cpp
galgo.h	galgo.cpp
node_lst.h	node_lst.cpp
multidim.h	md_alloc.cpp
md_elem.cpp	
vect.h	vect.cpp
metric.h	metric.cpp
mat_2d.h	mat_2d.cpp
inf_face.h	int_face.cpp
bn_math.h	bn_math.cpp
err_mess.h	err_mess.cpp

GALGO©: Source code: bn_main.cpp

```
////////////////////////////////////////  bn_main.cpp  /////
//
//                                  GALGO© Copyright 1992,1993,1994
//                                    Written by Carlos Rojas-Guzman
//                              as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                    Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  PROBABILISTIC REASONING USING BAYESIAN BELIEF NETWORKS
//
//  PURPOSE
//  The GALGO program performs probabilistic diagnostic
//  inference for chemical processes. It is intended to be
//  used in real-time as an on-line system.
//
//  SYSTEM ARCHITECTURE
//
//  The main components of the diagnostic system are the
//  algorithm and the probabilistic model.
//  The GALGO system can be best described by a set of
//  components organized as classes of objects.  Classes are
//  the basis for abstract data types and object oriented
//  programming.  Encapsulation mechanisms package the internal
//  implementation details of the type and the externally
//  available operations and functions that can act on objects
//  of that type.  Function prototypes allow functions to be
//  completely checked as to number of arguments and type.
//
//  Interphase tasks of the GALGO diagnostic system are
//  coordinated by one main function (contained in this file,
//  bn_main.cpp) that provides a user interface, reads a model
//  definition, reads process measurements, creates an
//  evolutionary world object, and initiates world evolution
//  with user specified options.
//
//
//  SYSTEM CLASSES
//
//  (1) Evolving world class
```

```
//      A world object has attributes, data structures and
//      functions which support evolutionary programming
//      capabilities.  One of the key functions of the world
//      object is an implementation of a graph-based genetic
//      algorithm that guides the evolution of the world with
//      the purpose of obtaining the most likely diagnostic
//      hypothesis. (Files galgo.cpp and galgo.h).
//
//  (2) Belief network class
//      A belief network object contains a description of the
//      network including its size, node members, topology and
//      functions to define and modify itself.  (Files
//      bel_net.cpp and bel_net.h).
//      Additional functions to calculate the metric of a given
//      diagnostic hypotheses or to count the size of the search
//      space are available (File metric.cpp).
//
//  (3) Node class
//      Nodes are the building blocks of belief networks.  Nodes
//      keep track of their relations with other nodes (parents
//      and children), and the discrete probabilistic
//      distributions which quantify their interactions with
//      their parent nodes. (File node_cls.cpp and node_cls.h).
//
//  (4) Vector class
//      Two classes of vectors are available.  One class, vect,
//      contains long int and the other, dvect, contains double
//      elements. Pointers fit in the long int version of vectors
//      and are often used to store pointers to nodes or other
//      objects. (Files vect.cpp and vect.h).
//
//  (5) Node list class
//      A node list is a linked list whose elements are object
//      nodes. (Files node_lst.cpp and node_lst.h).
//
//  (6) Matrix class
//      Two dimensional arrays which store integers or doubles.
//      They encapsulate checking bounds for indices and have
//      useful error messages to identify the source of attempts
//      to illegally access their elements.
//
//  (7) Multiple dimension array
//      N-dimensional arrays which store double numbers.
//      The main role of these arrays is to store and handle
//      the probabilistic values of the belief network models.
//      (Files md_alloc.cpp and md_elem.cpp).
//
//
//  AUXILIARY FUNCTIONS
//
//  Additional support functions include error-message handling
//  (file err_mess.cpp), temporary interface functions (file
//  int_face.cpp), auxiliary math functions (file bnmath.cpp),
```

```
//  sorting routines (piksort.cpp).


////////////////////////////////////////////////////////////////
//
//  HEADER FILES
//  The file bn.h contains all header files for the program.
//  Files with the .cpp extension contain C++ code with function
//  definitions.  Files with the .h extension are header files
//  and contain function prototypes, classes definitions and
//  sometimes inline functions.  The file "bn.h" contains all
//  the header file names for the GALGO program.

#include "bn.h"

////////////////////////////////////////////////////////////////
//
//  GLOBAL VARIABLES
//  Usually used only during the development stage for testing.

//  int general_symbol_generator;
//  double tol_eq=1e-6;
//  global temporary variables:
//  long int count_vect, count_vect_arg;
//  long int dvect_count, dvect_res_count;
//  long int delete_vect;
//  long int node_list_counter=0, node_list_counter_destructor=0;

////////////////////////////////////////////////////////////////
//
//   MAIN
//
//  This is the main funcition for the GALGO system.
//  Interphase tasks of the GALGO diagnostic system are
//  coordinated by one main function (contained in this file,
//  bn_main.cpp) that provides a user interface, reads a model
//  definition, reads process measurements, creates an
//  evolutionary world object, and initiates world evolution
//  with user specified options.
//
//  crg 10/24/93 added initialize_states() to store state names
//      Run with executable and input files in the same
//      directory,
//      or to run from BCPP put input files in d:/borlandc/bin
//  crg 3/16/94 Added arguments to main()
//      argv[1]=topology filename  (i.e. tname_66.txt)
//  argv[2]=probabilities filename  (i.e. prob_66.txt)
//  argv[3]=output filename
//  argv[4]=input data filename
//  argv[5]=options flag
//  argv[6]=storage filename (for run reproduction) optional
//  crg 10/11/93 FILE *ifp3 added for evidence
//  crg 2/10/94  added idi for output files identifier
```

```
//  crg 3/23/94  FILE *lang_ofp moved from galgo.cpp
//  crg 10/14/94: Conditional compilation of windows specific
//  instructions


void main(int argc, char *argv[]) {
  char c, data[64], id1='0', id2='0', id3='7';
  belief_network  bnet;
  node *p, *parent, *child;
  vect *carrier, *dimension_sizes, *indices;
  multidim *prob_array;
  int node_number, initial_network_size=0,
  initial_network_size2=0;
  int initial_network_size3=0, state_number=0;
  int index, k, k2, k3, k4, k5, k6, x, y, i, j;
  int number_of_dimensions, number_of_elements, dim;
  float probabilistic_parameter;

  //Interface variables:
  int evol_pop, tot_gen, crossover, sampling, mutation;
  double mutation_freq, selectivity, life_t;
  int parent_sel, exp_levels, transf, first_interface_loop, run_id=0;
  FILE *ifp, *ifp2, *ifp3;
  FILE *ofp, *idn, *lang_ofp;

  //timer added crg 2/15/94
  time_t first, second;
  //key(); //password protection
  first = time(NULL);//for profiling purposes

  introduction();
  cout << "\nENTRANCE TO MAIN PROGRAM ";

  if ((ifp=fopen(argv[1],"r"))==NULL) {
    cout<<"\nTopology input file "<< argv[1] << " not opened\n";
    bnerror("Unavailable file.");
  }
  else cout<<"\nInput file " << argv[1] << " successfully opened.";

  if ((ifp2=fopen(argv[2],"r"))==NULL) {
    cout<<"\nProbabilities input file "<<argv[2]<<" not opened\n";
    bnerror("Unavailable file.");
  }
  else cout<<"\nInput file " << argv[2] << " successfully opened.";

  if ((lang_ofp=fopen(tmp_fname,"w"))==NULL) {
    cout<<"\nOutput diagnoses file "<<tmp_fname<<" not opened\n";
    bnerror("Unavailable file.");
  }
  else cout<<"\nOutput diagnoses file "<<argv[3]
           << " successfully opened.";

  if ((ifp3=fopen(argv[4],"r"))==NULL) {
```

```
    cout<<"\nEvidence file " << argv[4] << " not opened\n";
    bnerror("Unavailable file.");
  }
  else cout<<"\nEvidence file "<<argv[4]<<" successfully opened.";

  if ((ofp=fopen(argv[6],"w"))==NULL) {
    cout<<"\nStorage file " << argv[6] << " not opened\n";
    bnerror("Unavailable file");
  }
  else cout<<"\nStorage file "<<argv[6]<<" successfully opened.";

////////////////////////////////////////////////////////////////////
//
//  Data which characterizes the topology of the network is read
//  from a text file (i.e. topology.txt) and the network object is
//  initialized.

  cout << "\nREADING CASE SPECIFIC DATA: network ";
  fscanf(ifp,"%d",&initial_network_size);//reads network size
  bnet.initialize_network(initial_network_size);//initializes network
  fscanf(ifp,"%s",data);                  //reads network name
  bnet.set_name(data);                    //stores network name
  fscanf(ifp,"%s",data);                  //reads network description
  bnet.set_description(data);             //stores network description

////////////////////////////////////////////////////////////////////
//
//  Memory for all nodes is dynamically allocated.

  cout << "\nDYNAMIC NODE ALLOCATION:    nodes  ";
  for (node_number=1;
       node_number<=initial_network_size;
       ++node_number) {
    p = new node();                  //a new node object is created
    p->set_node_number(node_number);//a unique int is assigned
    bnet.put_member(node_number, p);//node added to the network
  }

////////////////////////////////////////////////////////////////////
//
//  Node specific data is read from the topology text file.

  cout << "\nREADING NODE SPECIFIC DATA: topology ";
  for (node_number=1;
       node_number<=initial_network_size;
       ++node_number) {
    fscanf(ifp,"%d",&x);
    if (node_number != x)
      bnerror("Node number from input file not accepted");
    p=bnet.get_member(node_number);
    fscanf(ifp,"%s",data);
    p->set_name(data);
    fscanf(ifp,"%s",data);
```

```
  p->set_node_description(data);
  fscanf(ifp,"%s",data);
  if (data[0] == 'r') p->set_kind(root);
  else if (data[0] == 'i') p->set_kind(internal);
  else if (data[0] == 'l') p->set_kind(leaf);
  else bnerror("Node kind not recognized");
  fscanf(ifp,"%d",&x);
  p->set_number_of_states(x);
  p->initialize_states(x);     //vect (long int) for (char*) names
  for (k6=1; k6<=x; ++k6) {    //for each state
    fscanf(ifp,"%s",data);     //reads state name as a string
    p->set_state_name(k6,data);//stores state name for state num k6
  }

  //Reads node status, f=F=false=unknown, t=T=true=known
  fscanf(ifp,"%s",data);
  if (data[0]=='f' || data[0]=='F') p->set_status(f);
  else if (data[0] == 't' || data[0]=='T') p->set_status(t);
  else bnerror("Status not recognized");

  //Reads number of parents,and creates arcs from parents to child
  fscanf(ifp,"%d",&x);
  carrier = new vect(x, "carrier", "parents");
  for (i=1; i<=x; ++i) {
    fscanf(ifp, "%d", &y);
    parent = bnet.get_member(y);
    carrier->element(i-1) = (long int) parent;
  }
  p->set_parents(x, carrier);
  delete carrier;

  fscanf(ifp,"%d",&x);
  carrier = new vect(x, "carrier", "children");
  for (i=1; i<=x; ++i) {
    fscanf(ifp,"%d",&y);
    child = bnet.get_member(y);
    carrier->element(i-1) = (long int) child;
  }
  p->set_children(x, carrier);
  delete carrier;
}

////////////////////////////////////////////////////////////////
//
//  Initialization of the belief network nodes and their
//  relations.

bnet.initialize_sizes_of_nodes(); //after nodes have been defined
bnet.expand_relatives_perspective();

////////////////////////////////////////////////////////////////
//
//  Probabilistic parameters for the model are read from a file
```

```
//

cout << "\nREADING CASE SPECIFIC DATA: probabilities ";
fscanf(ifp2,"%d",&initial_network_size2);
if (initial_network_size != initial_network_size2)
  bnerror(
    "Inconsistent network size between topology and parameters");
for (k=1; k<=initial_network_size; ++k) { //for each node
  fscanf(ifp2,"%d",&node_number); //reads node number
  p = bnet.get_member(node_number);
  fscanf(ifp2,"%d",&number_of_dimensions);
  dimension_sizes = new vect(number_of_dimensions,
           "dimension_sizes","kth array");
  //reads array dimensions
  for (k2=0; k2<number_of_dimensions; ++k2) {
    fscanf(ifp2,"%d",&dim);
    dimension_sizes->element(k2) = dim;
  }
  p->set_matrix_dimensions(number_of_dimensions, dimension_sizes);
  prob_array = new multidim(number_of_dimensions, dimension_sizes,
          "prob_array");//dyn allocate array

  number_of_elements = 1;
  for (k3=0; k3<number_of_dimensions; ++k3) { //count elements
    number_of_elements *= dimension_sizes->element(k3);
  }
  indices = new vect(number_of_dimensions, "indices");
  for (k4=0;k4<number_of_elements;++k4) {//for each prob parameter
    for(k5=0; k5<number_of_dimensions; ++k5) {//for each index
fscanf(ifp2,"%d",&index);
indices->element(k5) = index;
    }
    fscanf(ifp2,"%f",&probabilistic_parameter);
    prob_array->element(indices) = probabilistic_parameter;
  }
  p->set_prob_parameters(prob_array);

  delete indices;
  delete dimension_sizes;
}
fclose(ifp);
fclose(ifp2);

////////////////////////////////////////////////////////////////
//
//  Evidence is read from a text file (i.e. evidence.txt)
//  The first line of the input evidence file is read and written
//  into the natural language output file as its first line.
//  The second line is written to the screen.

  while((c=getc(ifp3)) != '\n') {
    putc(c,lang_ofp);
  }
```

```
  cout << "\n";
  while((c=getc(ifp3)) != '\n') {
    cout << c;
  }
  fscanf(ifp3, "%d", &initial_network_size3);
  if (initial_network_size != initial_network_size3)
  bnerror(
    "Inconsistent network size between topology and evidence");

  for (node_number=1;
 node_number<=initial_network_size;
 ++node_number) {
    fscanf(ifp3,"%d",&x);
    if (node_number != x)
bnerror("Node number from input file not accepted");
    p=bnet.get_member(node_number);
    fscanf(ifp3,"%s",data); fflush(stdin);
    if (data[0]=='f' || data[0]=='F') {
p->set_status(f);
    }
    else if (data[0]=='t' || data[0]=='T') {
fscanf(ifp3, "%d", &state_number);
p->instantiate(state_number);  //cin >> c;
    }
    else bnerror("Status not recognized");
  }
  cout << "\nEvidence has been read and incorporated";


////////////////////////////////////////////////////////////
//
//  TEMPORARY USER INTERFACE FOR GALGO
//
//  Interface to enable quick changes in parameter values
//  Default values for interface:

evol_pop = 130;      //instead of 75
tot_gen = 200;//170  //instead of 200
crossover = 3;
sampling = 1;
mutation = 1;
mutation_freq = 0.35; //instead of 0.25
selectivity = 0.9;    //instead of 0.7
life_t = 3.0;         //instead of 5
parent_sel = 3;
exp_levels = 3;
transf = 1;
run_id=0;  //0 for random, other unsigned int to specify

int id_number;
if ((idn=fopen("last_id.txt","r+"))==NULL) {
  cout<<"\nLast_id file not opened\n";
  bnerror("File unavailable");
```

```
}
else cout<<"\nLast_id file successfully opened.";
fscanf(idn,"%c",&id1);
fscanf(idn,"%c",&id2);
fscanf(idn,"%c",&id3);
//cout << "\nScanned last_id file ids " << id1 << id2 << id3;
fclose(idn);

if(argv[5][0]=='1') {
    fprintf(ofp, "\nEvidence file:       =  %s ", argv[4]);
    fprintf(ofp, "\nEvolving population  =  %d ", evol_pop);
    fprintf(ofp, "\nMaximum generations  =  %d ", tot_gen);
    fprintf(ofp, "\nCrossover method     =  %d ", crossover);
    fprintf(ofp, "\nSampling method      =  %d ", sampling);
    fprintf(ofp, "\nMutation method      =  %d ", mutation);
    fprintf(ofp, "\nMutation frequency   =  %f ", mutation_freq);
    fprintf(ofp, "\nBreeding selectivity =  %f ", selectivity);
    fprintf(ofp, "\nAverage lifetime     =  %f ", life_t);
    fprintf(ofp, "\nParent selection     =  %d ", parent_sel);
    fprintf(ofp, "\nExpansion levels     =  %d ", exp_levels);
    fprintf(ofp, "\nTransformation       =  %d ", transf);
    if (run_id==0) {
run_id = (unsigned) time(NULL);
fprintf(ofp,"\nRun ID randomly selected =  %u \n",run_id);
    }
    else fprintf(ofp, "\nRun ID specified as %u \n", run_id);
}

if(argv[5][0]=='2') {
  first_interface_loop = -1;
  while (first_interface_loop != 0) {
  printf("\n\nThese are the present parameters for GALGO:\n");
  printf("\n 1.     Evolving population  =  %d    ", evol_pop);
  printf("\n 2.     Maximum generations  =  %d    ", tot_gen);
  printf("\n 3.     Crossover method     =  %d    ", crossover);
  printf("\n 4.     Sampling method      =  %d    ", sampling);
  printf("\n 5.     Mutation method      =  %d    ", mutation);
  printf("\n 6.     Mutation frequency   =  %4.2f    ",mutation_freq);
  printf("\n 7.     Breeding selectivity =  %4.2f    ",selectivity);
  printf("\n 8.     Average lifetime     =  %4.2f    ",life_t);
  printf("\n 9.     Parent selection     =  %d    ", parent_sel);
  printf("\n10.     Expansion levels     =  %d    ", exp_levels);
  printf("\n11.     Transformation       =  %d    ", transf);
  if (run_id==0) {
    run_id = (unsigned) time(NULL);
    printf("\n12.     Run seed (random)    =  %u",run_id);
  }
  else printf("\n12.     Run seed (specified) =  %u", run_id);

  printf("\n13.     ID for output files  = %c%c%c", id1, id2, id3);

  printf("\n\nType 0 to use these values, or the number of the ");
  printf("\nparameter you would like to change. ");
```

```
  first_interface_loop = -1;
  while (first_interface_loop < 0 || first_interface_loop > 13) {
    printf("\nA valid input is an integer between 0 and 13.  ");
    scanf("%d", &first_interface_loop);
    fflush(stdin);
  }

  switch(first_interface_loop) {
    case 0:
printf("\nEvolution will proceed with specified parameters");

fprintf(ofp, "\nEvidence file:       =  %s ", argv[4]);
fprintf(ofp, "\nEvolving population  =  %d ", evol_pop);
fprintf(ofp, "\nMaximum generations  =  %d ", tot_gen);
fprintf(ofp, "\nCrossover method     =  %d ", crossover);
fprintf(ofp, "\nSampling method      =  %d ", sampling);
fprintf(ofp, "\nMutation method      =  %d ", mutation);
fprintf(ofp, "\nMutation frequency   =  %f ", mutation_freq);
fprintf(ofp, "\nBreeding selectivity =  %f ", selectivity);
fprintf(ofp, "\nAverage lifetime     =  %f ", life_t);
fprintf(ofp, "\nParent selection     =  %d ", parent_sel);
fprintf(ofp, "\nExpansion levels     =  %d ", exp_levels);
fprintf(ofp, "\nTransformation       =  %d ", transf);
fprintf(ofp, "\nRun_id               =  %u ", run_id);
fprintf(ofp, "\nID for output files  =  %c%c%c ",id1,id2,id3);

      //Last id read as an integer and 1 is added to the value
if ((idn=fopen("last_id.txt","r"))==NULL)
  bnerror("\nLast_id file not opened\n");
      fscanf(idn,"%d",&id_number);
      fclose(idn);
id_number += 1;
      //The id number for the next run is stored into a file
if ((idn=fopen("last_id.txt","w"))==NULL)
  cout<<"\nLast_id file not opened\n";
if (id_number<10)  fprintf(idn,"%c",'0');
      if (id_number<100) fprintf(idn,"%c",'0');
fprintf(idn,"%d",id_number);
fclose(idn);

break;

    case 1:
printf("\nEnter new value for evolving population ");
scanf("%d", &evol_pop);
break;

    case 2:
      printf("\nEnter new value for maximum generations ");
scanf("%d", &tot_gen);
    break;

    case 3:
```

```
printf("\nEnter new value for crossover ");
scanf("%d", &crossover);
    break;

    case 4:
printf("\nEnter new value for sampling ");
scanf("%d", &sampling);
    break;

    case 5:
printf("\nEnter new value for mutation ");
scanf("%d", &mutation);
    break;

    case 6:
  printf("\nEnter new value for mutation frequency");
scanf("%lf", &mutation_freq);
    break;

    case 7:
  printf("\nEnter new value for breeding selectivity ");
scanf("%lf", &selectivity);
    break;

    case 8:
printf("\nEnter new value for average lifetime ");
scanf("%lf", &life_t);
    break;

    case 9:
  printf("\nEnter new value for parent selection method ");
scanf("%d", &parent_sel);
    break;

    case 10:
printf("\nEnter new value for number of expansion levels ");
scanf("%d", &exp_levels);
    break;

    case 11:
  printf("\nEnter new value for transformation function ");
scanf("%d", &transf);
    break;

    case 12:
printf("\nEnter fixed value for run identifier ");
scanf("%d", &run_id);
    break;

    case 13:
     printf("\nEnter 3 integers for the output files identifier ");
     id1=getchar(); id2=getchar(); id3=getchar();
```

```
      default:
       printf("\nInvalid input: Type an integer between 0 and 13. ");
  } //close switch
 }  //close while loop
}   //closes if '2'

////////////////////////////////////////////////////////////
//
//  GENETIC ALGORITHM INFERENCE
//  Creates world, an evolving_world object, then sends it a
//  message to start evolution with specified parameters.
//  Makes the world object evolve according to its Galgo algorithm

  //cout << "\nGA inference starting";
  evolving_world world;   //creates world object
  world.galgo(&bnet, evol_pop, tot_gen, crossover, sampling,
        mutation, mutation_freq, selectivity, life_t,
        parent_sel, exp_levels, transf, run_id, lang_ofp,
        id1, id2, id3, argv[5][0]);


  fcloseall();
  cout << "\nExit from GALGO inference system";

  second=time(NULL);
  printf("\nTotal time = %f ",difftime(second,first));

// crg 10/14/94: Conditional compilation of windows specific
instructions
#ifdef WINDOWS_COMPILATION
  _wsetexit(_WINEXITNOPERSIST);
#endif
//$$$END
}

////////////////////////////////////////////////////////////////
//
//  TEMPORARY INTERFACE FOR BORLAND C++ WINDOWS COMPILER

#ifdef BCPP_COMPILER
  bnet.set_number_of_possible_states();
  generate_all(&bnet);
  evolving_world world;//creates world object
  world.galgo(&bnet);  //performs inference
  world.print();       //after solutions evolve,prints new population
  cout << "\nExit from GALGO inference system";
  return;
#endif
```

GALGO©: Source code: bn.h

```
////////////////////////////////////////////////////////////////
//
//                              GALGO© Copyright 1992,1993,1994
//                                Written by Carlos Rojas-Guzman
//                          as part of the following Ph.D. thesis:
//         An Evolutionary Programming Approach to Probabilistic
//             Model-based Fault Diagnosis of Chemical Processes
//                                Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//
//  BAYESIAN BELIEF NETWORKS FOR ABDUCTIVE INFERENCE
//  USAGE: FAULT DIAGNOSIS OF CHEMICAL PROCESSES
//
//  Sep. 1992 Translated to Microsoft C++ 7.0 on MS-DOS/Windows
//  Oct. 1992 Translated to Borland C++ 3.1 on Windows by crg
//  Nov. 1992 Wrote code for compatibility among the compilers
//  May  1994 Code developed for Borland C++ 3.1 and MS C++ 7.0

////////////////////////////////////////////////////////////////
//
//  GLOBAL DEFINITIONS

enum boolean  {f, t};                              //false=0, true=1
enum xboolean {false,true,inconsistent,unknown}; //false=0,unknown=3
enum message_type {lambda, pi};
enum node_kind {root, internal, leaf, isolated};
enum color {white,gray,black,red,green,blue,yellow,unknown_color};

////////////////////////////////////////////////////////////////
//
//  HEADER FILES
//    BCPP  : BORLAND C++ 3.1
//    MSCPP : MICROSOFT C++ 7.0
//    GNU   : GCC from GNU for UNIX
//  crg 10/14/94:  Added WINDOWS_COMPILATION variable

#undef BCPP_COMPILER
#undef MSCPP_COMPILER
#undef GNU_COMPILER
#undef MESSAGES_ALG
#undef DEVELOPMENT
```

```
#undef WINDOWS_COMPILATION

////////////////////////////////////////////////////////////////
//
//  COMPILATION MODE
//  Select development = 1 to compile additional functions

#define DEVELOPMENT 1


////////////////////////////////////////////////////////////////
//
//  OPERATING SYSTEM FOR COMPILATION
//  Select WINDOWS_COMPILATION = 1 to compile additional functions
//  specific to the windows environment such as _wyield, _wsetexit,
//  _winexitnopersist.  Use '0' to compile for DOS.
//  These windows functions are necessary for operation within the
//  monitoring system at PA, however, the windows version runs
//  slower than the DOS version.  For experimental purposes (in
//  order to run *.bat files directly on DOS) it is preferable
//  to avoid the compilation of those functions.
//  crg 10/14/94: The variable windows_compilation was added

//#define WINDOWS_COMPILATION 1


////////////////////////////////////////////////////////////////
//
//  COMPILER
//  Define only one of the following compiler options
//  5/94:  The code has been tested on the Borland Compiler.
//         Porting to Microsoft should, in principle, be simple.

//  #define BCPP_COMPILER 1
    #define MSCPP_COMPILER 2
//  #define GNU_COMPILER 3
//  #ifdef GNU_COMPILER
//  #include <stream.h>
//  #endif

////////////////////////////////////////////////////////////////
//
//  Microsoft Visual C++ Compiler

#ifdef MSCPP_COMPILER
#include <iostream.h>
#include <process.h>
//#include <conio.h>
#include <io.h>
#endif

////////////////////////////////////////////////////////////////
//
```

```
//  Borland 3.1 C++ Compiler

#ifdef BCPP_COMPILER
#include <iostream.h>
#include <conio.h>
//for interface:
#include <ctype.h>
#endif

////////////////////////////////////////////////////////////////
//
//  STANDARD header files

#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <limits.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

////////////////////////////////////////////////////////////////
//
//  GALGO header files

#include "bn_math.h"   // basic combinatorial math functions
#include "mat_2d.h"    // dyn alloc 2 dim matrix
#include "vect.h"      // dynamically allocated vectors
#include "multidim.h"  // dynamically allocated multi dim array
#include "int_list.h"  // dyn alloc list of integers
//#include "nd_queue.h"// queue containing pointers to void (node)
#include "node_cls.h"  // basic type for graph algorithms
#include "node_lst.h"  // list of pointers to nodes
#include "bel_net.h"   // generic class made up of nodes
#include "err_mess.h"  // general error handling functions
#include "metric.h"    // count num of states and calculate metric
#include "piksort.h"   // piksort sorting algorithm
#include "galgo.h"     // approximate inference genetic algorithm
#include "int_face.h"  // crg 11/3/93 interface functions

////////////////////////////////////////////////////////////////
```

GALGO©: Source code: bel_net.h

```
//////////////////////////////////////////////  bel_net.h  /////
//
//                              GALGO© Copyright 1992,1993,1994
//                                Written by Carlos Rojas-Guzman
//                          as part of the following Ph.D. thesis:
//         An Evolutionary Programming Approach to Probabilistic
//             Model-based Fault Diagnosis of Chemical Processes
//                                Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//
//  BAYESIAN NETWORK CLASS
//
//
//  crg 4/24/94: possible_states and absolute_prob changed to be
//               pointers (previously objects). Functions which use
//               these pointers must be updated by creating and
//               deleting objects with new and delete.
//  Members is a vector object whose elements are long integers
//  Counted, Generated and Ranked are flags to avoid redundant
//               calculations when the network properties have not
//               changed and would therefore yield the same result.
//  crg 3/20/93: added optional arg to initialize network.
//  crg 3/10/92: get_member(int i) was missing and rewritten.

class belief_network {
  char* name_of_network;       //network name, a unique identifier
  char* network_description;   //any appropriate information
  int number_of_nodes;         //total number of nodes
  int message_counter;         //counts messages sent
  vect *members;               //vector stores pointers to nodes
  long double number_of_possible_states;//search space size
  matrix_2dim *possible_states;//all possible combinations of values
  dmatrix_2dim *absolute_prob; //probability & id of each hypothesis
  boolean counted,
    generated,
    ranked;                    //flags avoid redundant calculations
public:
  belief_network();
  void initialize_network(int initial_network_size=0);
  void set_name(char* n);
  char* get_name();           //new version
```

```
  void set_description(char* n);
  char* get_description();    //new version
  void set_size(int s);
  int  get_size();
  void add_members(int cnt,...);
  void put_member(int node_number, node* p);//node_number in [1,n]
  vect* get_members();
  node* get_member(int i);
  void print(int level=1);
  char* check_network_consistency();
  void initialize_sizes_of_nodes();
  int revise_topology();
  void expand_relatives_perspective();
  void take_snapshot(vect* snapshot, int option=1);
  void put_snapshot(vect* snapshot, int option=1);
  long double set_number_of_possible_states();
  long double get_number_of_possible_states();
  matrix_2dim* get_possible_states();
  dmatrix_2dim* get_absolute_prob();
  void set_counted(boolean value=t) { counted=value; }
  boolean test_counted() { return counted; }
  void set_generated(boolean value=t) { generated=value; }
  boolean test_generated() { return generated; }
  void set_ranked(boolean value=t) { ranked=value; }
  boolean test_ranked() { return ranked; }
};
```

GALGO©: Source code: bel_net.cpp

```
//////////////////////////////////////////  bel_net.cpp  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                            as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  The BAYESIAN_NETWORK or belief network class describes the
//  network object that consists of a set of nodes each containing
//  its own links, and its own probabilistic parameters (either
//  prior probabilities or conditional probabilities).
//
//  A belief network object contains a description of the network
//  including its size, node members, topology and functions to
//  define and modify itself.  (Files bel_net.cpp and bel_net.h).
//
//  Additional functions to calculate the metric of a given
//  diagnostic hypotheses or to count the size of the search space
//  are available (File metric.cpp).

#include "bn.h"

////////////////////////////////////////////////////////////////////////
//
//  BAYESIAN_NETWORK constructor
//  The constructor is not overloaded and it does not require
//  any parameters.  The specific attributes of the object are
//  associated after the object is created with functions which
//  belong to this class.
//
//  Note the initialization syntax of members (a vect object).
//  This is conventional for Microsoft C++ constructors within
//  constructors.
//  crg 4/16/94: The creation of members is unnecessary and the use
//  of new is preferred.
//  Three flags reduce redundant calculations
//  (1) counted (T = size of the search space has been calculated)
//  (2) generated (T = systematic generation of hypotheses has
```

```
//      been conducted)
//  (3) ranked (T = after a systematic generation of hypotheses,
//      a sorting algorithm was used to rank them according to their
//      probabilities).

belief_network::belief_network()
             //:members(0,"members","belief_network by constructor")
             {
  members = NULL;//members initialized to the NULL pointer crg 4/16/94

  if ( ( name_of_network = new char[32]) == NULL ) {
    bnerror(
      "Not enough memory to allocate buffer for name_of_network");
  }
  strcpy(name_of_network, "anonymous network");

  if ( ( network_description = new char[64]) == NULL ) {
    bnerror(
      "Not enough memory to allocate buffer for network description");
  }
  strcpy(network_description, "unavailable description");

  number_of_nodes = -1;
  counted=f;
  generated=f;
  ranked=f;
}
////////////////////////////////////////////////////////////////////////
//
//  INITIALIZE_NETWORK
//  Initializes the belief network object.  Default names and
//  descriptions are given.  The size of the network is assigned
//  using the only argument the function takes.  The members set
//  is created using a vector object.  Elements of members are
//  pointers to nodes (each fits in the space of a long integer,
//  32 bit signed integer.)

void belief_network::initialize_network(int initial_network_size) {

  strcpy(name_of_network, "initialized network");
  strcpy(network_description, "initialized without description");
  number_of_nodes = initial_network_size;
  members = new vect(initial_network_size,"members of belief network",
            "by initialize_network");
  counted=f;
  generated=f;
  ranked=f;
}

////////////////////////////////////////////////////////////////////////
//
//  SET_NAME and GET_NAME
//  Memory for the network name has already been allocated by the
```

```
//  constructor.

void belief_network::set_name(char* n) {
  strcpy(name_of_network, n);
}

char* belief_network::get_name() {
  return (name_of_network);
}

char* belief_network::name() {
    bnerror("belief_network::name(), substitute with get_name");
    return (name_of_network);
}

////////////////////////////////////////////////////////////////////////
//
//  SET_DESCRIPTION and GET_DESCRIPTION
//  Assign and retrieve network attributes.

void  belief_network::set_description(char* n) {
  strcpy(network_description,n);
}

char* belief_network::get_description() {
  return network_description;
}

char* belief_network::description() {
  bnerror(
    "belief_network::description,substitute with get_description");
  return (network_description);
}

////////////////////////////////////////////////////////////////////////
//
//  SET_SIZE and GET_SIZE
//  The size of the network is one of the object attributes.

  void  belief_network::set_size(int s) {
    number_of_nodes = s;
  }

  int belief_network::get_size() {
    return (number_of_nodes);
  }

////////////////////////////////////////////////////////////////////////
//
//  ADD_MEMBERS
//  The members of the Bayesian network object are nodes.  The set of
//  nodes that belong to the network are stored as members.
```

```
void belief_network::add_members(int cnt, ...) {
  int i;
  node* p;
  va_list ap;

  delete members; //if a previous set exists, it is deleted
  members = new vect(cnt,"members of ", name_of_network);
  number_of_nodes = cnt;
  va_start (ap,cnt);
  for (i=0; i<cnt; ++i) {
    p = va_arg(ap,node*);
    members->element(i) = (long int) p;
  }
  va_end(ap);
}

////////////////////////////////////////////////////////////////////////
//
//  PUT_MEMBER
//
//  When adding a new member of the network at a later time, its
//  perspectives are expanded so that it knows how its children
//  perceive it (the order in which the node appears in the set of
//  parents of its children).
//
//  crg 3/20/93: function written to add a node assuming vector has
//  enough memory allocated (otherwise the vect object will complain).
//  Members has indices in [0,n-1].
//  Put_member(...) takes node_number in the interval [1,n].

void belief_network::put_member(int node_number, node* p) {
  int i;

  members->element(node_number-1) = (long int) p;
}

////////////////////////////////////////////////////////////////////////
//
//  GET_MEMBERS
//  Returns the pointer to the members vector.  The main benefit of
//  this approach is the significant reduction in function calls and
//  the resulting increase in code speed.  The pointer should be used
//  only to read the pointers to nodes but should not be used to
//  change the pointer values.  Often the usage of this pointer will
//  be for iteration of all elements to read some information of each.

vect* belief_network::get_members() {
  return(members);
}

////////////////////////////////////////////////////////////////////////
//
//  GET_MEMBER
```

```
//
//  Returns the pointer to one of the members of the network.
//  The node is selected according to node_number.
//  crg 3/10/92 function missing and rewritten
//  Note that i is an element of [1,n].

node* belief_network::get_member(int i) {
  return (node*) members->element(i-1);
}

////////////////////////////////////////////////////////////////////////
//
//  PRINT
//  The Bayesian network prints itself to the screen.
//  Each node member receives a message to print itself.
//  Level (of detail) has a default value of 1 in the prototype.

void belief_network::print(int level) {
  int i=0, iterator;
  node* n;

  cout << "\n\nNetwork Name:  " << name_of_network;
  cout << "\nNumber of nodes: " << number_of_nodes;
  cout << "\nDescription: " << "\n    " << network_description;
  cout << "\nMember nodes are:";

  members->start(iterator);
    while (members->ok(iterator)) {
      n = (node*) members->next(iterator);
      cout << "\n\nMember number " << ++i << " of the network: ";
      n->print(level);
    }
}

////////////////////////////////////////////////////////////////////////
//
//  CHECK_NETWORK_CONSISTENCY
//
//  Checks that if root then 0 parents, if leaf 0 children
//  Checks that if internal node then at least one parent
//  Checks that probability distributions sum to one.
//  For each node in the network, checks each of its links
//  this means that each child of node N should recognize N as its
//  parent, and each parent of N should recognize N as its child.
//  Checks that number of nodes is consistent by comparing the size
//  attribute of the network and the number of valid elements in the
//  members vector.

char* belief_network::check_network_consistency()  {

  char* consistency_analysis;

  if ( ( name_of_network = new char[128]) == NULL ) {
```

```
    bnerror(
    "Not enough memory to allocate buffer for consistency analysis");
  }

  strcpy(consistency_analysis,
      "belief_network::check_consistency() is not yet available");
  return consistency_analysis;
}

////////////////////////////////////////////////////////////////////////
//
//  INITIALIZE_SIZES_OF_NODES
//  For each network member the node number is set and its sizes are
//  initialized.

void belief_network::initialize_sizes_of_nodes()  {
  char c;
  int i;
  node* p;

  members->start(i);
  while (members->ok(i)) {
    p = (node*) members->next(i);
    p->set_node_number(i);
    p->initialize_sizes();
  }
}

////////////////////////////////////////////////////////////////////////
//
//  REVISE_TOPOLOGY
//
//  Tests if the network is singly connected, returns 1 or 0.
//  This can be used later to trigger simplification algorithms
//  such as conditioning or clustering.

int belief_network::revise_topology() {
  cout << "\nNetwork topology revision";
  bnerror("belief network::revise_topology unavailable");
  int answer=1; //while being implemented, set to 1 temporarily
  return(answer);
}

////////////////////////////////////////////////////////////////////////
//
//  EXPAND_RELATIVES_PERSPECTIVES
//  For each member of the Bayesian network, the children perspectives
//  and the parents perspectives are set.  These perspectives ensure
//  that a child recognizes all of its parents and a parent node
//  recognizes all of its children.  The redundancy of links recorded
//  in both directions increases efficiency when testing whether a
//  probabilistic node exists between an arbitrary pair of nodes.
```

```
void belief_network::expand_relatives_perspective() {
  node* p;
  int c;
  members->start(c);
  while (members->ok(c)) {
    p = (node*) members->next(c);
    p -> set_children_perspective();
    p -> set_parents_perspective();
  }
}

////////////////////////////////////////////////////////////////////////
//
//  TAKE_SNAPSHOT
//  Takes a snapshot of the states of all nodes in the network.
//  Option = 1 by default.
//  The order of state values in the returned vect* is the same of
//  the members vector.

void belief_network::take_snapshot(vect* snapshot, int option) {
  int i;
  node *np;

  //cout << "\nEntrance to take_snapshot ";
  members->start(i);
  while (members->ok(i)) {
    np = (node*) members->next(i);
    snapshot->element(i-1) = np->get_state();
  }
  if (option>1) {
    bnerror("Snapshots at level>1 ...implement here if necessary");
  }
}

////////////////////////////////////////////////////////////////////////
//
//  PUT_SNAPSHOT
//  Assigns one value to each variable based on a given snapshot of
//  the network.  This function is useful when at the middle of other
//  calculations it is necessary to make changes or tests which
//  require altering the network and also keeping the option to put
//  the original state back.
//  Option = 1 by default.

void belief_network::put_snapshot(vect* snapshot, int option) {
  int i;
  node *the_node;

  //cout << "\nEntrance to put_snapshot";
  snapshot->print(2);
  if (option==1) {
    while (members->ok(i)) {
      the_node = (node*) members->next(i);
```

```
      //cout << "\nfound node : i= " << i << " " << the_node-
>get_name();
      the_node->set_state(snapshot->element(i-1));
    }
  }
  else if (option>2)
    bnerror("Presently, the only valid option is 1");
}

////////////////////////////////////////////////////////////////////
//
//  SET_NUMBER_OF_POSSIBLE_STATES
//  The number of possible system states is the number of possible
//  state combinations of the nodes and corresponds to the size of
//  the search space of the network.  A long double is used to be
//  able to handle very large spaces (a long double requires 80 bits
//  and can store numbers from 3.4*10^-4932 to 1.1*10^4932 while the
//  max for a double is only 10^308 and for a float 10^38.)

long double belief_network::set_number_of_possible_states() {
  counted=t;
  return ( number_of_possible_states=count_possible_states(this) );
}

////////////////////////////////////////////////////////////////////
//
//  GET_NUMBER_OF_POSSIBLE_STATES
//  The number of possible system states is calculated ONLY if it has
//  not been calculated before.  The value of the 'counted' flag is
//  changed to TRUE if previously FALSE, otherwise it remains as TRUE.

long double belief_network::get_number_of_possible_states() {
  if (test_counted() == t)
    return number_of_possible_states;
  else
    return set_number_of_possible_states();
}

////////////////////////////////////////////////////////////////////
//
//  GET_POSSIBLE_STATES
//  All possible system states are stored in a two dimensional
//  array. Note that due to memory available on any computer the
//  maximum search space size will be limited.
//  crg 4/24/94: Changed possible_states to be a pointer
//               to matrix_2dim (previously an object).

matrix_2dim* belief_network::get_possible_states() {
  return possible_states;
}

////////////////////////////////////////////////////////////////////
//
```

```
//  GET_ABSOLUTE_PROB
//  Returns the absolute probabilities corresponding to all possible
//  system states.
//  crg 4/24/94: Changed absolute_prob from an object to
//               a pointer to dmatrix_2dim

dmatrix_2dim* belief_network::get_absolute_prob() {
  return absolute_prob;
}
```

GALGO©: Source code: node_cls.h

```
//////////////////////////////////////////  node_cls.h  /////
//
//                              GALGO© Copyright 1992,1993,1994
//                                Written by Carlos Rojas-Guzman
//                          as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//
//  NODE class

class node {
   char* name_of_node;     //unique identifier or description
   char* node_description; //comments and notes about the node variable
   //int symbol_generator; //to generate automatically new unique names
   node_kind local_topology;//kind of node: root, internal or leaf
   int number_of_states;   //this definition is for discrete variables
   boolean status;         //status is a boolean variable: known = 1
   int topological_order;  //number from topological sorting
   //queue *message_queue; //available pointer to access external queue
   int node_number;        //from the order given in input files
   //color mark;           //used to mark nodes in graph algorithms,
                           //colors are white, gray, black, red, unknown

   int  state;             //discrete var,finite no. of states, s:[1,ns]
   int  number_of_parents; //an integer
   int  number_of_children;//an integer
   vect *states_names;     //vect of long ints (pointers to char)
//int  max_states_per_parent;//an integer
   vect *parents;          //array of long int ..pointers
   vect *parents_sizes;    //array of long int
   vect *children;         //array of long int ..pointers
   vect *children_perspective;//indices children use to refer to this n.
   vect *parents_perspective; //indices parents use to refer to this n
   double *probabilities;//either prior or conditional probabilities
   multidim* parameters;//prior or conditional in multidim array
   vect *argument_list; //prob parameters access for variable args
   vect *sequence_length_pi;//used in option generation
   int combinations;    //no.possibilities in comparison    //int
   lambda_combinations; //no.items in comparison to create lambda mess.
```

```
   vect *dimensions;    //size of each dimension in the prob matrix
   dvect *prob_value;   //size no. of states of node //change to pointer
   vect *star_states;   //states of variable in most prob. explanation

public:
   node();
   void      set_name(char* n);
   char*     get_name();
   void      set_node_description(char* d);
   char*     get_node_description();

   void      set_kind(node_kind  k);
   node_kind get_kind();
   void      set_number_of_states(int n);
   int           get_number_of_states();

   void      set_node_number(int n);
   int       get_node_number();

   void      set_status(boolean s);
   boolean   get_status();
   void      set_state(int i);
   int       get_state();
   void      set_number_of_parents(int n); //used by bn_maker
   int       get_number_of_parents();
   void      set_number_of_children(int n);//used by bn_maker
   int       get_number_of_children();

   void      set_children_perspective();
   void      set_child_perspective(int child_number);
   int       get_child_perspective(int child_number);

   void      set_parents_perspective();
   void      set_parent_perspective(int parent_number);
   int       get_parent_perspective(int parent_number);

   void      set_parents(int cnt, ...);
   void      set_parents(int cnt, vect* parents_transporter);
   node*     get_parent(int index); //where 1 <= index <= no_of_parents
   int       set_max_states_per_parent();
   void      set_children(int cnt, ...);
   void      set_children(int cnt, vect* children_transporter);
   node*     get_child(int index); //where 1 <= index <= no_of_children

   void    print_parents();
   void    print_children();
   void    print_iter(vect* vector); //experimental iterator

   void    set_probabilities(void* matrix_address);//ok, overload later
   double  get_probability(int cnt, ...);        //variable arguments
   void    set_prob_parameters(multidim* prob_parameters); //multidim
   double  get_probability(vect* ); //overloaded as best option
           //arguments are (number_of_following_args, i,j,k,...),i [1,n]
```

```
   void    set_matrix_dimensions(int cnt, ...);
   void    set_matrix_dimensions(int num_of_dims, vect* v);
   void    auto_set_mat_dim();
   int     get_matrix_dimension(int dimension_index);

   void    print(int level=1);
   char*   check_consistency();
   void    initialize_sizes();
   void    set_parents_sizes();
   void    initialize_states(int num_of_states);
   void    set_state_name(int state_number, char* name);
   char*   get_state_name(int x);

   //experimental iterator enhances modularity and OOP, 2/3/92
   //not tested yet (check correct argument passing and overhead)
   //void start_parents_iteration(int& i) const { parents.start(i); }
   //int  ok_parents(int& i) const  { return parents.ok(i); }
   //node*  next_parent(int& i) const { return ((node*)
parents.next(i));}

   //changes to known state,sends,stores:
   void    instantiate(int state, int di_opt=2);

#if defined MESSAGES_ALG
   void    create_and_send_lambda_message(int parent_number,
                              boolean selective=f,
                              int di_opt=2);
   void    create_and_send_sum_lambda_message(int parent_number,
                              boolean selective=f,
                              int di_opt=2);
   void    create_and_send_pi_message(int child_number);
   void    send_pi_messages_to_all_children();
   void    send_pi_messages_to_other_children(int isolated_child);
   void    send_lambda_messages_to_all_parents(int di_opt=2);
   void    send_lambda_messages_to_other_parents(int isolated_parent,
                                 int di_opt=2);
   void    respond_to_lambda_message(int child_sender, int di_opt=2);
   void    respond_to_pi_message(int parent_sender, int di_opt=2);
   void    update_lambda_value();    //computes lambda v and updates
node
   void    update_pi_value_sum(int di_opt=2);//computes pi val, updates
node
   void    update_pi_value_max(int di_opt=2);//computes pi val, updates
node
   void    update_prob_value();      //computes and updates prob value
   int     select_best_state();      //selects most prob. state in
explan.
#endif

};
```

GALGO©: Source code: node_cls.cpp

```
//////////////////////////////////////////  node_cls.cpp  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                            as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  NODE class
//
//
//  Nodes are the building blocks of belief networks.  Nodes
//  keep track of their relations with other nodes (parents and
//  children), and the discrete probabilistic distributions
//  which quantify their interactions with their parent nodes.
//  (File node_cls.cpp and node_cls.h).
//  Nodes can be used as if they were a basic type of the language.

#include "bn.h"

////////////////////////////////////////////////////////////////////
//
//  NODE constructor
//

node::node()   {
  if ( (name_of_node = (char *) malloc(64)) == NULL ) {
    bnerror(
      "Not enough memory to allocate buffer for name_of_node");
  }
  strcpy(name_of_node, "anonymous node");

  if ( (node_description = (char *) malloc(128)) == NULL ) {
    bnerror(
      "Not enough memory to allocate buffer for node_description");
  }
  strcpy(node_description, "unavailable description");

  number_of_parents = 2;
```

```
  number_of_children = 2;
}

////////////////////////////////////////////////////////////////////
//
//  SHORT MEMBER FUNCTIONS
//  These functions assign or retrieve values of object attributes.

void    node::set_name(char* n)          {strcpy(name_of_node, n);}
char*   node::get_name()                 {return (name_of_node); }
void    node::set_node_description(char* d){
            strcpy(node_description, d); }
char*   node::get_node_description()          {return
node_description;}
void    node::set_kind(node_kind  k)          {local_topology = k; }
node_kind node::get_kind()               {return (local_topology);}
void    node::set_number_of_states(int n)  {number_of_states = n; }
int     node::get_number_of_states()          {return
number_of_states;}
void    node::set_status(boolean s)      {status = s; }
boolean   node::get_status()             {return (status); }
void    node::set_state(int i)           {state = i; }
int     node::get_state()                {return (state); }
void      node::set_number_of_parents(int n) {number_of_parents = n; }
int     node::get_number_of_parents()      {return number_of_parents;}
void      node::set_number_of_children(int n){number_of_children = n; }
int     node::get_number_of_children()     {return number_of_children;}
void    node::set_node_number(int n)          {node_number = n; }
int     node::get_node_number()          {return node_number; }

////////////////////////////////////////////////////////////////////
//
//  SET_CHILDREN_PERSPECTIVE
//  Iteratively sets the perspective of all its children.

void node::set_children_perspective() {
  int i;

  if(number_of_children != 0) {
    for (i=1; i<=number_of_children; ++i) {
      set_child_perspective(i);
    }

  }
}

////////////////////////////////////////////////////////////////////
//
//  SET_CHILD_PERSPECTIVE
//  By iterating through all the parents of the child, this function
//  identifies the number that the child has assigned to this parent.
//  The parent (this node) then has information of how it is
//  perceived by its child.  The benefit of this cross reference is
```

```
//  to make access more efficient from either side of the
//  probabilistic link.
//
//  The variable child has indices in [1,num_of_ch]

void node::set_child_perspective(int child_number) {
  int p, viewpoint=0, parents_set_cardinality;
  node *child, *parent;
  child = (node*) children->element(child_number-1);
  parents_set_cardinality = child -> get_number_of_parents();
  for(p=1; p<=parents_set_cardinality; ++p) {
    //cout << "\n    p = " << p;
    parent =  child->get_parent(p);
    if ( parent -> get_name() == name_of_node) {
      viewpoint = p;
      break; //once found, it is unnecessary to continue the search
    }
  }
  children_perspective->element(child_number-1) = viewpoint;
}

////////////////////////////////////////////////////////////////////////
//
//  GET_CHILD_PERSPECTIVE
//

int node::get_child_perspective(int child_number) {
  return children_perspective->element(child_number-1);
}

////////////////////////////////////////////////////////////////////
//
//  SET_PARENTS_PERSPECTIVE
//  Sets the perspective with which parents see this node.

void node::set_parents_perspective() {
  int i;

  if(number_of_parents != 0) {
    for (i=1; i<=number_of_parents; ++i) {
      //cout << "\nParent to be analyzed: "
      //cout << get_parent(i)->get_name();
      set_parent_perspective(i);
    }
  }
}

////////////////////////////////////////////////////////////////////
//
//  SET_PARENT_PERSPECTIVE
//  The function looks at a parent of THIS node.
//  The objective is to determine the child number that the
```

```
//  parent has assigned to THIS node. This makes access more
//  efficient by enabling immediate identification from both
//  sides of the probabilistic link.
//  The variable parent has indices in [1,number_of_parents].

void node::set_parent_perspective(int parent_number) {
  int c, viewpoint=0, children_set_cardinality;
  node *parent, *child;// several pointer declarations
  parent = get_parent(parent_number);
  children_set_cardinality = parent -> get_number_of_children();
  for(c=1; c<=children_set_cardinality; ++c) {
    //cout << "\n    c = " << c;
    child =  parent->get_child(c);
    if (child -> get_name() == name_of_node) {
      viewpoint = c;
      break; //once found, it is unnecessary to continue searching
    }
  }
  parents_perspective->element(parent_number-1) = viewpoint;

}

////////////////////////////////////////////////////////////////////
//
//  GET_PARENT_PERSPECTIVE
//

int node::get_parent_perspective(int parent_number) {
  return parents_perspective->element(parent_number-1);
}

////////////////////////////////////////////////////////////////////
//
//  SET_PARENTS - overloaded
//
//  Arguments are &nodes.
//  Change the test to cnt>=0 and modify vect to accept argument=0.
//  What the vector should do is to store a -1 in the only location.
//  This will avoid wasting memory for root nodes.
//  crg 2/24/92: Did a similar process to the set_children function
//  crg 3/22/93: function replaced to handle node numbers, not
//               pointeres as a result of required code
//               modifications to handle huge networks.

void node::set_parents(int cnt, ...) {
  number_of_parents=cnt;
  parents = new vect(number_of_parents,"parents of",name_of_node);
  int i;
  long int  abs_address;
  node* p;
  va_list ap;

  va_start (ap,cnt);
```

```
  for (i=0; i < cnt; ++i) {
    p = va_arg(ap,node*);
    abs_address = (long int) p;
    parents->element(i) = abs_address;
  }
  va_end(ap);
}

////////////////////////////////////////////////////////////////////
//
//  SET_PARENTS - overloaded
//  Note that both parents and transporter begin at index=0

void node::set_parents(int cnt, vect* parents_transporter) {
  int i;

  number_of_parents=cnt;
  parents = new vect(cnt, "parents of", name_of_node );
  if (number_of_parents != 0) parents->fill(-1);
  for (i=0; i < cnt; ++i) {
    parents->element(i) = parents_transporter->element(i);
  }
}

////////////////////////////////////////////////////////////////////
//
//  GET_PARENT
//  parent_index in [1, np]

node* node::get_parent(int parent_index)  {
  //cout<<"\nget_parent:getting parent number="<<parent_index;
  return ( (node*) parents->element(parent_index-1) );
}

////////////////////////////////////////////////////////////////////
//
//  SET_CHILDREN - overloaded
//


void node::set_children(int cnt, ...)  {
  number_of_children=cnt;
  children = new vect(cnt, "children of", name_of_node);
  int i;
  long int abs_address;
  node* p;
  va_list ap;

  va_start (ap,cnt);
  for (i=0; i<cnt; ++i) {
    p = va_arg(ap,node*);
    abs_address = (long int) p;
    children->element(i) = abs_address;
```

```
  }
  va_end(ap);
}

////////////////////////////////////////////////////////////////////
//
//  SET_CHILDREN - overloaded
//  transporter contains long int (corresponding to node pointers)

void node::set_children(int cnt, vect* children_transporter) {
  int i;

  number_of_children=cnt;
  children = new vect(cnt, "children of", name_of_node );
  if (number_of_children != 0) children->fill(-1);
  for (i=0; i < cnt; ++i) {
    children->element(i) = children_transporter->element(i);
  }
}

////////////////////////////////////////////////////////////////////
//
//  GET_CHILD
//  In this function,  1 <= index <= no_of_parents

node* node::get_child(int index)  {
  return ( (node*) children->element(index-1) );
}

////////////////////////////////////////////////////////////////////
//
//  PRINT_PARENTS
//  Prints parents to the screen.

void node::print_parents() {
  long int abs_address;
  int i;
  node* np;

  cout << "\nThe PARENTS of the node " << name_of_node << " are: ";
  for ( i=0; i<number_of_parents; ++i ) {
    abs_address = parents->element(i);
    cout << "\nAddress of the parent is ---" << abs_address << " ";
    np = (node*) abs_address;
    cout << "\n" <<  np -> get_name();
  }
}

////////////////////////////////////////////////////////////////////
//
//  PRINT_CHILDREN
//  Prints children to the screen.
//  Note the experimental use of the alternate printing function
```

```
//  an equivalent and tested function to iteratively print elements
//  Vect iterator:
//  tested 12/8/91,with 1 element,
//  tested 2/1/92 with 4, ok
//  working properly 2/92

void node::print_children() {
  long int abs_address;
  int i;
  node* np;

  cout << "\nThe CHILDREN of the node " <<name_of_node<< " are: ";
  for ( i=0; i<number_of_children; ++i ) {
    abs_address = children->element(i);
    np = (node*) abs_address;
    cout << "\n" << np->get_name();
  }
}

////////////////////////////////////////////////////////////////////
//
//  PRINT_ITER
//

void node::print_iter(vect* vector) {
  int var;
  node* np;

  vector->start(var);
  while (vector->ok(var)) {
    np = (node*) vector->next(var);
    cout <<  np->get_name() << "\n";
  }
}

////////////////////////////////////////////////////////////////////
//
//  SET_PROBABILITIES
//  To be overloaded soon, to enable adding elements one by one.

  void node::set_probabilities(void* matrix_address) {
    probabilities = (double*) matrix_address;
  }

////////////////////////////////////////////////////////////////////
//
//  SET_PROB_PARAMETERS
//  Sets probabilistic parameters of the Bayesian belief network
//  for this node.

 void node::set_prob_parameters(multidim* prob_parameters) {
   parameters = prob_parameters;
 }
```

```
////////////////////////////////////////////////////////////////////////
/
//
//  GET_PROBABILITY - overloaded
//
//  This implementation does not test using cnt, it uses
number_of_parents.
//  The number of dimensions in the matrix is: no of parents + 1.
//  The maximum size of each dimension is: the number of states of
//  each variable, always beginning with the node of interest, i.e.:
//  P(a1 | p1 p2 p3), 4 dimensions, order as set up in vector list.
//  Note that vect (the constructor) takes n as argument (n elements).
//  4/16/92 changed to take args [1,n], not [0,n-1]
//  Tested g1 crg 3/92, 5/25/92.
//  crg 3/4/93: included bnerror notice to determine when it is used
//  The reason for tracing it is that it may be discontinued in the
event
//  of an emergency situation due to problems in direct memory
allocation
//  which might arise when changing to a different memory model in the
//  intel processor, where pointers may be large or huge and where the
//  matrix (if it is too large) may be stored in different blocks. The
//  described scenario might render this algorithm useless. See the
//  vector argument approach in the overloaded version which can handle
//  both options: direct memory access and conventional matrix access.

double node::get_probability(int cnt, ... ) {
    int sum = 0;
    double* initial = probabilities;
    //cout << "\nFirst element of matrix = " << *probabilities;

    cout << "\nUsing var arg -get_probability- with cnt = " <<
cnt<<"\n";
    bnerror("\nGET_PROBABILITY with (int cnt, ...) is being used ");

    int dim = 1; //begins with the second element of vector (dim = 1,
not 0)
    va_list ap;
    va_start(ap, cnt);  //note efficiency in the for loop calculations
    //cout << "\n\nStarting probability search";
      for (sum = va_arg(ap,int)-1; dim <= number_of_parents; ++dim ) {
            sum *= dimensions->element(dim);
            sum += va_arg(ap,int)-1;
      }
    va_end(ap);
    return *(initial+sum);
}

////////////////////////////////////////////////////////////////////////
/
//
//  GET_PROBABILITY - overloaded
```

```
//
//  The number of dimensions in the matrix is: no of parents + 1.
//  The maximum size of each dimension is: the number of states of
//  each variable, always beginning with the node of interest as in
//  P(a1 | p1 p2 p3): 4 dimensions, order as set up in the vector list.
//  Note that vect (the constructor) takes n as argument (n elements).
//  This OVERLOADED function takes the argument list as an int vector of
//  variable size containing: v[0] = number of following arguments,
//  which is the number_of_parents+1
//  and     v[1],...,v[n] = the function arguments (the indices)
//  4/16/92 changed to take args in [1,n], not [0,n-1]
//  5/25/92 revised and tested
//  Has DIRECT MEMORY ACCESS to a matrix element
//  by using absolute addresses:
//  The algorithm was written by crg 10/91, based
//  on factorizing the product.
//  i.e. 4 dimensions  a[i][j][k][i], with
//  dimensions in each size m,n,o,s
//  initial_address = a[0][0][0][0]
//  address = initial_address + nosi + osj + sk + 1
//  and by taking factors, the last 4 terms
//  can be efficiently computed as:
//  ...+i *n +j *o +k *s +1 with each operator
//  applied to the sum so far as
//  ((((((i)*n)+j)*o)+k)*s)+1 . It can be
//  easily implemented recursively.
//  Note that m (the first dimension is
//  not used), it starts with n,o,s,...
//
//  crg 3/4/93: When working with large
//  matrices, or with different memory
//  models in the intel processor, direct
//  memory access might have problems.
//  One possible case is when the matrix is stored in separate pieces.
//  In that case, it will be necessary to
//  implement a case by case (based
//  on the number of dimensions of the matrix)
//  algorithm as an alternative.
//  The access will still be correct but slightly less efficient.  This
//  complexity results from the need to
//  handle matrices of arbitrary sizes
//  and with an arbitrary number of dimensions,
//  with an arbitrary size in
//  each dimension. Direct memory acces saves the space of working out
//  every specific case (matrix num of dimensions, and size of each dim)
//  and/or specifying beforehand the size and
//  thus wasting valuable space.
//
//  crg 3/4/93: It is advisable to complete algorithm 2 (it might be
//  needed later).
//  crg 3/15/93: This function (1) works with BCPP, not with MSC++
//  crg 3/31/93: Extension to handle multidimensional array objects
```

```
double node::get_probability(vect* v) {
  int sum=0, i=2, dim=1; //dim should begin with 2nd element of vector
  int array_dimensions, k;
  double* initial = probabilities;
  double probability;
  int number_of_dimensions=0, algorithm_option=2;
  double r=0.0; //for 2
  double* p; //for 2
  double** pp;
  double*** ppp;
  vect *argument_list;
  char c;

  algorithm_option = 2;//1=direct memory access,2=multidim access

  if  (algorithm_option==1) {
    //note efficiency in the for loop calculations
    for (sum =(v->element(1))-1; dim <= number_of_parents; ++dim,++i ) {
      sum *= dimensions->element(dim);
      sum += (v->element(i))-1;
    }
    return *(initial+sum);
  }

  //arguments which are indices are in [1,n]
  else if (algorithm_option == 2) {
    array_dimensions = v->element(0); //num par + 1,
    //next line commented 4/15/94 (worked fine, just testing)
    argument_list = new vect(array_dimensions,
                "arg list for get_prob","get_prob(*v) algorithm=2");
    //vect argument_list(array_dimensions, "arg list for get_prob");
    for(k=1; k<=array_dimensions; ++k) {
      argument_list->element(k-1) = v->element(k)-1; //from [1,n] to
[0,n-1]
    }
    probability = (double) parameters->element(argument_list);
    delete argument_list; //commented 4/15/94
    return probability;
  }

  else if (algorithm_option == 3) {
    bnerror("get_probab avoiding direct memory access being
implemented");
    cout << "\nTesting alg option 2";
    number_of_dimensions = v->element(0);
    switch(number_of_dimensions) {
    case 1: p = probabilities;
          r = p[v->element(1)]; break;
    case 2: pp = (double**) probabilities;
          r = pp[v->element(1)][v->element(2)]; break;
    case 3: ppp = (double***) probabilities;
          r = ppp[v->element(1)][v->element(2)][v->element(3)]; break;
```

```
    default: cout <<"\nUnavailable ndims in get_prob
"<<number_of_dimensions;
    exit(1);
    }
    return r;
  }
  else      {
    cout << "\nWrong algorithm option in node_class::get_probability";
    return 0.0;
  }
}

////////////////////////////////////////////////////////////////////
//
//  SET_MATRIX_DIMENSIONS - overloaded  1 of 2
//  Uses variable arguments

void node::set_matrix_dimensions(int cnt, ...) {
  //cout << "\nnode::set_matrix_dimensions (overloaded 1) uses reset";
  dimensions = new vect(cnt,"dims of cond prob array",name_of_node);
  va_list ap;
  int i;
  va_start (ap, cnt);         //note it begins with i=0
  for (i=0; i < cnt; ++i) {  //instead of cnt, can use node properties
      dimensions->element(i)=va_arg(ap,int);
  }
  va_end(ap);
}

////////////////////////////////////////////////////////////////////
//
//  SET_MATRIX_DIMENSIONS - overloaded  2 of 2
//  Uses an int and a vector as an argument

void node::set_matrix_dimensions(int num_of_dims, vect* v) {
  //cout<<"\nnode::set_matrix_dimensions (overloaded 2) uses reset";
  dimensions = new vect(num_of_dims,
              "dims of cond prob array",name_of_node);

  for (int i=0; i < num_of_dims; ++i) {
    dimensions->element(i) = v->element(i);
  }
}


////////////////////////////////////////////////////////////////////
//
//  AUTO_SET_MAT_DIM

void node::auto_set_mat_dim() {
  cout << "\nauto_set_mat_dim()  to be implemented";
}
```

```
////////////////////////////////////////////////////////////////
//
//  GET_MATRIX_DIMENSION

int node::get_matrix_dimension(int dimension_index) {
  //note that index is contained in [1,n] and not [0,n-1]
  return(  dimensions->element(dimension_index-1)  );
}

////////////////////////////////////////////////////////////////
//
//  PRINT
//  level=1 by default

void node::print(int level) {
  cout << "\nNODE: " << name_of_node ;
  cout << "\n" << node_description;
  cout << "\nIt is a";
  switch(get_kind()) {
    case 0: cout << " root node "; break;
    case 1: cout << "n internal node "; break;
    case 2: cout << " leaf node "; break;
    default:cout << " node of unknown type";
  }
  cout <<"with " << number_of_states << " states";
  cout <<"\nIt directly depends on "<<number_of_parents<<" parents";
  cout <<" and directly affects "<<number_of_children<<" children";
  //cout << "\nThe status of the node is " << status;
  cout << "\nThe value of this variable is presently ";
  switch(status) {
    case f: cout << "unknown"; break;
    case t: cout << "known to be in state " << state; break;
    default: cout << " in an erroneous ambiguous status";
  }
  print_children();
  children_perspective->print();
  print_parents();
  parents_perspective->print();
  cout << "\nThe sizes of the parents are ";parents_sizes->print();
  //additional information is printed using the optional parameter
}

////////////////////////////////////////////////////////////////
//
//  CHECK_CONSISTENCY
//  Checks consistency among node attributes.

char* node::check_consistency() {
  char* summary;

  if ( (summary = (char *) malloc(128)) == NULL ) {
    bnerror("Not enough memory to allocate buffer for node summary");
  }
```

```
  strcpy(name_of_node,"node:check_consistency: unavailable");

  bnerror( "\nnode::check_consistency - to be implemented -");
  cout << "\nWill return explanations (a concatenated string)";
  cout << "\nwill describe inconsistencies, will check cardinality";
  cout << "\nof lists, size of matrices, values of probabilities";
  return summary;
}

////////////////////////////////////////////////////////////////
//
//  INITIALIZE_SIZES
//  'Parents' and 'children' are initialized at set_parents,
//  set_children
//  'Dimensions' is initialized when the probability matrix is set
//  The pi_message is a 2 dim matrix with rows of different sizes

void node::initialize_sizes()  {
  int np,k,p;
  parents_sizes = new vect(number_of_parents,
                           "parents_sizes of", name_of_node);
  prob_value = new dvect(number_of_states,
                         "dvect prob_value of",name_of_node);
  star_states= new vect(number_of_states,
                        "star_states set of", name_of_node);
  children_perspective = new vect(number_of_children,
                           "children_perspective of",name_of_node);
  parents_perspective = new vect(number_of_parents,
                             "parents_perspective",name_of_node);
  sequence_length_pi= new vect(number_of_parents,
                           "sequence_length_pi", name_of_node);
  set_parents_sizes();

  state=-1;
  //Calculation of the number of values being compared in
update_pi_value:
  combinations = 1;
    for (p=1; p<=number_of_parents; ++p) {
      combinations *= parents_sizes->element(p-1);
    }
  //Calculation of size for lambda_tie 2d_mat...to be tested...oct 30 92
  int min_states_in_receiver; //potential receivers are the node parents
  int x;
  int min=10000;  //any 'large' real number to approximate infinity
  node *parent;
  for (np=1; np<=number_of_parents; ++np) {
    parent = get_parent(np);
    x = parent->get_number_of_states();
    if (x < min)
      min = x;
  }
  min_states_in_receiver = min;
  //Calculation of possible ties in create_and_send_lambda_message:
```

```
  //comm 4/21/94
  #if defined MESSAGES_ALG

lambda_combinations=combinations/min_states_in_receiver*number_of_states
;
  #endif
  //Calculation of factor to fill in sequence_length_pi:
  int factor;
  factor = combinations;
  for (p=1; p<=number_of_parents; ++p) {
    sequence_length_pi->element(p-1) = factor/
                                       (get_parent(p)-
>number_of_states);
    factor = sequence_length_pi->element(p-1);
  }
}

////////////////////////////////////////////////////////////////////////
/
//
//  SET_PARENTS_SIZES

void node::set_parents_sizes() {
  //cout<<"\nStarting set_parents_sizes";
  int np;
  node *parent;
  for (np=1; np<=number_of_parents; ++np) {
    parent = get_parent(np);
    parents_sizes->element(np-1) = parent -> get_number_of_states();
  }
}

////////////////////////////////////////////////////////////////////
//
//  INITIALIZE_STATES
//
//  crg 10/24/93 Initializes the vect containing long ints which will
//  store the (char*) corresponding to the names of the states of the
//  node and allocates memory for the pointers to the names.

void  node::initialize_states(int num_of_states) {
  if (num_of_states < 1 || num_of_states != number_of_states)
    bnerror("Node::initialize_states:inconsistent num of states");
  states_names=new vect(num_of_states,"state names of",name_of_node);
  states_names->fill(-1);
}

////////////////////////////////////////////////////////////////////
//
//  SET_STATE_NAME
//  crg 10/24/93: Sets the name of state number 'state_number'
//  Note state_number is an element of [1,n]
//  The char* is stored as a long int as an element of a vect.
```

```
//  crg 4/16/94: Replaced malloc by new char[32]

void node::set_state_name(int state_number, char* name) {

  char* temporary_name_pointer;

  if (state_number<1)
    bnerror("NEGATIVE OR 0 STATE NUMBER:node::set_state_name");

  if ( ( temporary_name_pointer = new char[32]; == NULL ) {
    bnerror("Not enough memory to allocate buffer for state_name");
  }
  strcpy(temporary_name_pointer, name);
  states_names->element(state_number-1) =
    (long int) temporary_name_pointer;
}

////////////////////////////////////////////////////////////////////
//
//  GET_STATE_NAME
//  crg 10/24/93: Returns the name assigned to state number x.
//  Note argument x is an element of [1,n]

char* node::get_state_name(int x) {
  if (x<1)
    bnerror("INVALID STATE NAME (<=0):node::get_state_name");
  return (char*) states_names->element(x-1);
}

////////////////////////////////////////////////////////////////////
//
//  SET_QUEUE
//
//  Used by belief network when distributing queue.

/* commented 4/21/94 since queues are not used for Galgo
void node::set_queue(queue* q) {
  message_queue = q;
}
*/

////////////////////////////////////////////////////////////////////
//
//  GET_QUEUE
//
//  Retrieves queue pointer for internal access

/*
queue* node::get_queue() {
  return message_queue;
}
*/
```

```
////////////////////////////////////////////////////////////////////
//
//  INSTANTIATE
//  Assigns a value to a variable within the Bayesian network.
//  di_opt=2 by default

void node::instantiate(int known_state, int di_opt) {
  //cout << "\nNode " << name_of_node << " being instantiated to state
";
  //cout << known_state;
  set_state(known_state);
  set_status(t);//the value of the node is known
}
```

GALGO©: Source code: galgo.h

```
//////////////////////////////////////////////  galgo.h  /////
//
//                              GALGO© Copyright 1992,1993,1994
//                                Written by Carlos Rojas-Guzman
//                          as part of the following Ph.D. thesis:
//         An Evolutionary Programming Approach to Probabilistic
//             Model-based Fault Diagnosis of Chemical Processes
//                                Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//
//  EVOLVING_WORLD class
//  An approximate genetic inference algorithm for belief networks.
//  Documentation for this class is contained in galgo.cpp.
//  crg 11/1/93: Added convergence_test as an independent function.
//  crg 11/4/93: Added storage best individual at each generation.
//  crg  2/9/94: Added top_n_size and best_local_size.
//  crg 2/12/94: Generalized natural_language_output declaration.

class evolving_world  {
  char* name_of_world;        //object identifier
  belief_network* bnet;       //encodes probabilistic model
  long double total_population;//size of the complete search space
  matrix_2dim* genotypes;     //code specifying an n-dim point in the
space
  dmatrix_2dim* phenotypes;   //the metric is the probability of the
point
  dmatrix_2dim* transformed_phenotypes;//transformation for parent
selection
  matrix_2dim* best_local_genotypes;//contains best genotypes at each
gen
  dmatrix_2dim* best_local_phenotypes;//contains best phenotypes at each
gen
  matrix_2dim* top_n_genotypes;//contains best n genotypes of all
evolution
  dmatrix_2dim* top_n_phenotypes;//best n phenotypes of all evolution
  vect* specimen;       //temporary storage for a solution candidate
  int number_of_traits;       //number of dimensions in the search space
  int total_generations;      //number of iterations in simulated
evolution
  int generation;       //iteration index
  double evolving_fraction;   //fraction of individuals sampled
```

```
  int evolving_population;    //number of individuals sampled from the
space
  double breeding_selectivity;//fraction of evolving population for
breeding
  int breeding_population;    //number of set with best individuals
  double lifetime;            //average lifetime of any given individual
  int births_per_generation;  //number of replacement individuals
  int pairs_per_generation;   //number of pairs of replacement
individuals
  int crossover_method;       //option for combining best individuals
  int block_size;       //compact block size for inheritance
  int sampling_method;        //option for sampling
  int mutation_method;        //option for mutation method
  double mutation_frequency;  //mutations per generation
  double initial_mutation_frequency;//initial value which may be changed
  //double evolving_probability_mass;//monitors p mass of evolving
population
  //double average_evolving_probability_mass;//average
  //double breeding_probability_mass;//monitors p mass of breeding
population
  //double average_breeding_probability_mass;//average
  dmatrix_2dim* history;      //keeps records of the simulated evolution
  double phenotype1;          //phenotype of parent1 (for comparison)
  double phenotype2;          //phenotype of parent2 (for comparison)
  double best_phenotype;      //best phenotype in current generation
  int g_index1;         //index contained in pt pointing to genotypes
  int g_index2;         //index contained in phenotypes pointing to gt
  int parent1;                //selected parent1 for breeding
  int parent2;                //selected parent2 for breeding
  int parent_selector;        //method for parents selection
  int expansion_levels;       //for compact block construction
  int transformation;         //function for phenotypes transformation
  int diagnosed_state;        //variable for state filtering crg
10/12/93
  int top_n_size;             //cardinality of set containing best
hypotheses
  int best_local_size;        //cardinality of set containing best at
each gen
  FILE *lang_ofp;             //pointer to natural language output file
  char id1, id2, id3;         //identifiers for output file names crg
2/10/94
  char option_flag;           //options for program runs 1=on-
line,2=development

public:
  evolving_world();
  void reset();
  ~evolving_world() { release(); }
  void galgo(belief_network* bn,
          int evol_pop = 100,
          int tot_gen = 25,
          int crossover = 3,
          int sampling = 1,
```

```
            int mutation = 1,
            double mutation_freq = 0.15,
            double selectivity = 0.30,
            double life_t = 5.0,
            int parent_sel = 2,
            int exp_levels = 2,
            int transf = 1,
            unsigned int run_id = 0,
            FILE *lang_ofp = NULL,
            char id_arg1 = '0',
            char id_arg2 = '0',
            char id_arg3 = '7',
            char option_flag_arg = '2');
  int  convergence_test(double* previous_phenotype);
  void natural_language_output(dmatrix_2dim* phenotypes_set,
                          matrix_2dim* genotypes_set,
                          int phenotype_position,
                          int output=0,
                          FILE* language=NULL);//0=screen,1=file
  void save_evolution_records();
  void create_territory();
  void delete_territory();
  void create_initial_population(unsigned int run_id=0);//0=random
  void start_evolution();
  void individual_evaluation();
  void select();
  void combine();
  void mutate();
  //void evaluate_offspring();
  void replace();
  void monitor_evolution();
  void print(int level=1);
  void parameters_adjustment(double *previous_phenotype,int strategy=1);
  void release() {
  //crg 2/10/94 objects deletion caused a crash that required ctrl alt
del to fix
  //        objects deletion has been temporarily removed
  //        deletion functions must be revised
  //        determine whether it is necessary to delete objects or
whether it is
  //        done automatically when the program exits.

              cout << "\nEntering world::release"; cout.flush();
                  /*
              delete genotypes; cout << "\nAfter deleting genotypes";
cout.flush();
              delete phenotypes;cout << "\nAfter deleting
phenotypes";cout.flush();
              //added next line 5/4/93:
              delete transformed_phenotypes;cout << "\nAfter del
transf_phen";cout.flush();
              delete specimen; cout << "\nAfter deleting
specimen";cout.flush();
```

```
                delete history;  cout << "\nAfter deleting
history";cout.flush();
                //added 2/10/94
                delete best_local_phenotypes; cout << "\nAfter del
best_local_ph";cout.flush();
                delete best_local_genotypes;  cout << "\nAfter del
best_local_ge";cout.flush();
                delete top_n_genotypes;   cout << "\nAfter del
top_n_genotypes";cout.flush();
                delete top_n_phenotypes;  cout << "\nAfter del
top_n_phenotypes";cout.flush();
                //crg 4/14/94:
                //note that only pointers to objects were declared, when
used,
                //they should be created with new, and deleted within the
code,
                //as necessary.
                */
                cout << "\nExit world::release\n"; cout.flush();
  }
  boolean block_test(int trait, int block_method);
  void crossover_1();
  void crossover_2();
  void crossover_3();
  void select_parents(int parent_sel);
  void summarize();
};
```

GALGO©: Source code: galgo.cpp

```
//////////////////////////////////////////////  galgo.cpp  /////
//
//                                  GALGO© Copyright 1992,1993,1994
//                                    Written by Carlos Rojas-Guzman
//                              as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                    Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  GALGO
//  An approximate genetic inference algorithm for Bayesian belief
//  networks. Probabilistic reasoning is motivated by the needs of
//  diagnostic and predictive activities in complex and uncertain
//  systems.  The graph-based genetic algorithm implemented in the
//  GALGO system is a function of the class evolving_world.

#include "bn.h"

////////////////////////////////////////////////////////////////////
//
//  EVOLVING_WORLD
//
//  A world with evolutionary capabilities is created.
//  A world object has attributes, data structures and functions
//  which support evolutionary programming capabilities.  One of
//  the key functions of the world object is an implementation
//  of a graph-based genetic algorithm (function called 'galgo'
//  that guides the evolution of the world with the purpose of
//  obtaining the most likely global diagnostic hypothesis.
//  (Files galgo.cpp and galgo.h).

////////////////////////////////////////////////////////////////////
//
//  REFERENCES
//
//  The probabilistic diagnostic methodology on which this system
//  is based is described in the following references.  Bayesian
//  belief networks are described by Pearl (1988).  Holland (1992)
//  discusses adaptive systems and the basic concepts behind
//  genetic algorithms   The rest of the papers by Rojas-Guzman
```

```
//  and Kramer discuss the basis for this diagnostic system,
//  including advantages of the probabilistic approach, suitability
//  of the genetic algorithm techniques to Bayesian networks, a
//  comparison with traditional rule-based systems, details on the
//  algorithm, experimental results on abstract Bayesian networks,
//  an industrial application experience, and theoretical extensions
//  to the methodology.
//
//  Holland, J. H. (1992) Adaptation in Natural and Artificial
//  Systems, An Introductory Analysis with Applications to Biology,
//  Control, and Artificial Intelligence. 2nd ed. MIT Press,
//  Cambridge, MA.
//
//  Pearl, J. (1988) Probabilistic Reasoning in Intelligent Systems:
//  Networks of Plausible Inference.  Morgan Kaufmann, San Mateo,CA.
//
//  Rojas-Guzmán, Carlos, Mark A. Kramer and Carl Chang (1994) A
//  Probabilistic Methodology for Monitoring and Fault Diagnosis.
//  AIChE Fall National Meeting, Session on Monitoring, Detection and
//  Data Interpretation in Process Operations, San Francisco, CA.
//
//  Rojas-Guzmán, Carlos, and Mark A. Kramer (1994) Multi-Stage
//  Bayesian Networks Subsume Digraph and Residual-Pattern Approaches
//  to Fault Diagnosis.  Fifth International Symposium on Process
//  Systems Engineering PSE-94, Kyongju, Korea.
//
//  Rojas-Guzmán, C. and M. A. Kramer (1993a). GALGO: A Genetic
//  ALGOrithm Decision Support Tool for Complex Uncertain Systems
//  Modeled with Bayesian Belief Networks. Proc. Ninth Conf. on
//  Uncertainty in Artificial Intelligence,Washington,D.C.pp.368-375.
//
//  Rojas-Guzmán, Carlos and Mark A. Kramer (1993b)  Comparison of
//  Belief Networks and Rule-Based Expert Systems for Fault
//  Diagnosis of Chemical Processes.  Engineering Applications of
//  Artificial Intelligence, Vol. 6, No.3, pp. 191-202.
//
//  Rojas-Guzmán, Carlos and Mark A. Kramer (1992)  Belief Networks
//  for Knowledge Integration and Abductive Inference in Fault
//  Diagnosis of Chemical Processes. Proc. of the IFAC International
//  Symposium: On Line Fault Detection and Supervision in the
//  Chemical Process Industries,  International Federation of
//  Automatic Control, Newark, Delaware, U.S.A.
//
//  crg 11/4/93:  Added storage of top N individuals
//  crg  2/5/94:  Extended code comments
//  crg 2/10/94:  Added run identifiers id1, id2, id3
//  crg 5/10/94:  Added references and extended class description

evolving_world::evolving_world() {

  if ( (name_of_world = (char *) malloc(64)) == NULL ) {
    bnerror(
      "Not enough memory to allocate buffer for name of world");
```

```
}
strcpy(name_of_world, "world");
total_population = 0.0;

//crg: 11/4/93 added 4 arrays
//crg: 2/22/94 made object creation optional
//     Objects within the world object are ready in case they
//     are required. Memory will be saved when they are not filled.
//     The object creation has a minimal memory usage
//crg 4/14/94: it is not necessary to create the objects at this
//     point. If then objects are reset, the automatic destructor
//     may not perform correctly.
//     New objects are therefore not created:
//genotypes = new matrix_2dim;
//phenotypes = new dmatrix_2dim;
//transformed_phenotypes = new dmatrix_2dim;
//specimen = new vect;
//history = new dmatrix_2dim;
//best_local_genotypes = new matrix_2dim;
//best_local_phenotypes = new dmatrix_2dim;
//top_n_genotypes = new matrix_2dim;
//top_n_phenotypes = new dmatrix_2dim;

//pointers initialization to NULL to avoid usage by mistake:
genotypes = NULL;
phenotypes = NULL;
transformed_phenotypes = NULL;
specimen = NULL;
history = NULL;
best_local_genotypes = NULL;
best_local_phenotypes = NULL;
top_n_genotypes = NULL;
top_n_phenotypes   NULL;

number_of_traits = 0;
total_generations = 0;
generation = 0;
evolving_fraction = 0.0;
evolving_population = 0;
breeding_selectivity = 0.0;
breeding_population = 0;
lifetime = 0.0;
births_per_generation = 0;
pairs_per_generation = 0;
crossover_method = 1;
block_size = 0;
sampling_method = 1;
mutation_method = 1;
mutation_frequency = 0.0;
initial_mutation_frequency = 0.0;
phenotype1 = 0.0;
phenotype2 = 0.0;
best_phenotype = 0.0;
```

```
  parent_selector = 1;
  expansion_levels = 2;
  transformation = 1;
  lang_ofp = NULL;
  id1='0'; id2='0'; id3='7';
  option_flag = '2';
}

////////////////////////////////////////////////////////////////
//
//  RESET
//  By resetting an object properties, data structures associated
//  with the object are kept but all information contained in them
//  is erased. (Changing memory allocation to name_of_world is not
//  necessary.)

void evolving_world::reset() {

  genotypes = NULL;
  phenotypes = NULL;
  transformed_phenotypes = NULL;
  specimen = NULL;
  history = NULL;
  best_local_genotypes = NULL;
  best_local_phenotypes = NULL;
  top_n_genotypes = NULL;
  top_n_phenotypes = NULL;

  name_of_world = "anonymous on reset";
  total_population = 0.0;
  number_of_traits = 0;
  total_generations = 0;
  generation = 0;
  evolving_fraction = 0.0;
  evolving_population = 0;
  breeding_selectivity = 0.0;
  breeding_population = 0;
  lifetime = 0.0;
  births_per_generation = 0;
  pairs_per_generation = 0;
  crossover_method = 1;
  block_size = 0;
  sampling_method = 1;
  mutation_method = 1;
  mutation_frequency = 0.0;
  initial_mutation_frequency = 0.0;
  phenotype1 = 0.0;
  phenotype2 = 0.0;
  double best_phenotype = 0.0;
  parent_selector = 2;
  expansion_levels = 2;
  transformation = 1;
  lang_ofp = NULL;
```

```
  id1='0'; id2='0'; id3='7';
  option_flag = '2';
}

////////////////////////////////////////////////////////////////
//
//  GALGO
//  Main function for the inference algorithm.
//  Evolution parameters are specified when the galgo function
//  is called, otherwise default values are used.
//  By default all methods select option 1.
//
//  crg 10/10/93:Selective printing of generation index crg
//  crg 2/10/94: Added numbering for output files

void evolving_world::galgo(belief_network* bn,
            int evol_pop,
            int tot_gen,
            int crossover,
            int sampling,
            int mutation,
            double mutation_freq,
            double selectivity,
            double life_t,
            int parent_sel,
            int exp_levels,
            int transf,
            unsigned int run_id,
            FILE *lang_ofp,
            char id_arg1,
            char id_arg2,
            char id_arg3,
            char option_flag_arg) {

  int hypothesis, i;
  //$$$NEW3B
  int min_gen = 0; //minimum number of generations
  double previous_phenotype=1e-100;

  bnet = bn;
  number_of_traits = bnet->get_size();
  total_generations = tot_gen;
  evolving_population = evol_pop;
  crossover_method = crossover;
  sampling_method = sampling;
  mutation_method= mutation;
  mutation_frequency = mutation_freq;
  initial_mutation_frequency = mutation_freq; //crg 11/4/93 added
  breeding_selectivity = selectivity;
  lifetime = life_t;
  parent_selector = parent_sel;
  expansion_levels = exp_levels;
  transformation = transf;
```

```
  id1=id_arg1; id2=id_arg2; id3=id_arg3;
  option_flag = option_flag_arg;

  create_territory();       //allocate space in memory
  create_initial_population(run_id);//randomly sample from population
  start_evolution();        //evaluate evolving population
//$$$NEW
// function _wyield() releases control back to windows

  for (generation=1; generation<=total_generations ; ++generation) {
    select();       //rank individuals in evolving_population

#ifdef WINDOWS_COMPILATION
    _wyield();
#endif

    if (generation%10 == 0) {
      cout<<"\nGeneration "<<generation; cout.flush();
      if ( convergence_test(&previous_phenotype) )
       //break; //....temporarily OFF to ENSURE MAX GENERATIONS ...
      //if(generation > 100 && option_flag == '2')
      if(generation > 1 && option_flag == '1')  // print English
explanation
       {
        natural_language_output(phenotypes,
                          genotypes,evolving_population,0);

#ifdef WINDOWS_COMPILATION
        _wyield();
#endif
       }
    }
    combine();  //create new indiv based on top 'breeding_population'

#ifdef WINDOWS_COMPILATION
    _wyield();
#endif

    mutate(); //introduce random gene variations, evaluate mutants

#ifdef WINDOWS_COMPILATION
    _wyield();
#endif

    if(option_flag=='2')
     {
      monitor_evolution();//keeps track of evolution performance

#ifdef WINDOWS_COMPILATION
      _wyield();
#endif

     }
```

```
    if(option_flag=='2')
     {
      parameters_adjustment(&previous_phenotype);//evaluate and adapt

#ifdef WINDOWS_COMPILATION
      _wyield();
#endif

     }
//$$$END
  }
  cout << "\nPost-evolutionary state of the world ";
  generation -= 1;

  //OPTIONAL  (can remain optional without time effects crg 3/27/94)
  if(option_flag == '2') {
    summarize();
    save_evolution_records();
  }

  //print(2);

  //Summarize top n hypotheses in natural language. crg 2/13/94
  fprintf(lang_ofp,"\nOUTPUT DIAGNOSIS FROM GALGO");
  fprintf(lang_ofp,
          "\nThis file contains the top %d hypotheses ", top_n_size);
  fprintf(lang_ofp,
          "\nselected from among the best at each generation.");
  fprintf(lang_ofp,"\nRun identifier: %c%c%c.", id1, id2, id3);
  for (hypothesis=top_n_size; hypothesis>=1;  hypothesis) {
    //$$$NEW3B CR 8/16/94 // both options to use top_n
      natural_language_output(top_n_phenotypes,
                  top_n_genotypes,hypothesis,1,lang_ofp);
  }
  delete_territory();
}

////////////////////////////////////////////////////////////////////
//
//  CREATE_TERRITORY
//  Allocates memory for the initial population, the offspring and
//  the mutated solutions.
//  Since the explanation set is a linked list, only the pointer is
//  necessary. Maximum size of evolving population set to 250.
//
//  crg 1/5/94: create territory for best local hypotheses and top n
//       the number of hypotheses kept in each set can be passed as
//       an argument and defined by the user.
//
//       The best_local_size should be ideally equal to the number of
//  generations in the evolution process, in practice the best is
//  to make it larger to ensure the complete record is kept (The
//       number of generations is not known in advance). If necessary
```

```
//      the array can be dynamically allocated again when more space
//      is required.This last extension is left for the next version.
//
//  Top_n_size is the number of hypotheses that should be kept,
//      the top n generated throughout the evolutionary process.
//      Create territory for best local hypotheses and top n.

void evolving_world::create_territory() {
  double evaluations = 0.0;
  double mutants_per_generation = 0.0;
  int i,j;

  //cout<<"\nCreating territory for network "<<bnet->get_name();
  best_local_size=total_generations;//depends on max generations
  top_n_size=5; //changed from 10 to 5 on 2/17/94
  total_population = bnet->get_number_of_possible_states();
  //an even amount of evolving population individuals is determined:
  evolving_population = 2 * ( floor(evolving_population/2.0) );
  cout << "\nEvolving_population                    =  "
       << evolving_population;
  //sets max evolving_population and calculates subpopulation sizes
  if (evolving_population > 1000) {
    cout<<"\nEvolving_population is too big: "<<evolving_population;
    cout<<" -  automatically set at 250";
    evolving_population = 250;
  }

  evolving_fraction = evolving_population / total_population;
  breeding_population=floor(evolving_population*breeding_selectivity);

  births_per_generation = evolving_population/lifetime;
  pairs_per_generation = floor(births_per_generation/2.0);
  births_per_generation = 2.0 * pairs_per_generation;
  //cout << "\nPairs per generation = " << pairs_per_generation;
  cout << "\nBirths per generation =                "
       << births_per_generation;

  //The value for mutants is calculated again in mutate()
  mutants_per_generation =
     floor(evolving_population * mutation_frequency);
  if (mutants_per_generation < 1.0) mutants_per_generation=1.0;
  cout << "\nMutants per generation =               "
       << mutants_per_generation;

  //The maximum total number of evaluations is calculated
  evaluations = evolving_population +
    total_generations *
    (births_per_generation+mutants_per_generation);

  //An upper bound on accepted number of evaluations is set
  cout << "\nThe size of the search space is        "
       << total_population;
  cout << "\nMaximum number of specimen evaluations in GALGO is "
```

```
    << evaluations;
if (evaluations >= total_population) {
  cout<<"\nWith due respect, systematic enumeration is recommended";
  bnerror(
   "\nInefficient approach attempted-extend for user interaction");
}

//Dimensions are assigned to data structures for population.
genotypes = new matrix_2dim(evolving_population+1,
          number_of_traits+1,
                              "Genotypes of", name_of_world);
genotypes->fill(0);
for (j=1;j<=number_of_traits;++j) {
  genotypes->element(0,j-1)=j;
}

specimen = new vect(number_of_traits+1,
                    "specimen of", name_of_world);
specimen->fill(-1);

phenotypes = new dmatrix_2dim(evolving_population + 1, 3,
                              "Phenotypes of", name_of_world);
phenotypes->fill(0.0);

transformed_phenotypes = new dmatrix_2dim( evolving_population+1,3,
           "Transformed_Phenotypes of",
           name_of_world);
transformed_phenotypes->fill(0.0);

top_n_genotypes = new matrix_2dim(top_n_size+1, number_of_traits+1,
    "Top_n_genotypes of", name_of_world);
top_n_genotypes->fill(-1);
top_n_phenotypes = new dmatrix_2dim(top_n_size+1, 3,
    "Top_n_phenotypes of", name_of_world);
for (j=1;j<=number_of_traits;++j) top_n_genotypes->element(0,j-1)=j;
top_n_phenotypes->fill(-1.0);

//Next line initializes pointers to the corresponding top_n_genotypes
for (i=1;i<=top_n_size; ++i) top_n_phenotypes->element(i,0)=i;

//NOT OPTIONAL
//The lack of these lines slows down running time by a factor of 7.
//if (option_flag=='2') {
  history = new dmatrix_2dim(total_generations+1,20,
                              "evolution history of",name_of_world);
  history->fill(0.0);
  best_local_genotypes = new matrix_2dim(best_local_size+1,
                              number_of_traits+1,
           "Best_local_genotypes of",
           name_of_world);
  best_local_phenotypes = new dmatrix_2dim(best_local_size+1,3,
      "Best_local_phenotypes of",
      name_of_world);
```

```
    best_local_genotypes->fill(-1);
    for (j=1;j<=number_of_traits;++j)
      best_local_genotypes->element(0,j-1)=j;
    best_local_phenotypes->fill(-1.0);
  //}

  //cout << "\nExit from CREATE TERRITORY"; cout.flush();
}

////////////////////////////////////////////////////////////////////
//
//  DELETE_TERRITORY
//  Deletes objects from memory created with 'new' inside the
//  function create_territory.
//  The alternative to having creation and deletion handled
//  automatically by constructors is not used to enable dynamic
//  creation of objects within the evolving_world object.  The
//  dimensions of objects is unknown in advance.

void evolving_world::delete_territory() {
  delete genotypes;
  delete phenotypes;
  delete transformed_phenotypes;
  delete history;
  delete specimen;
  delete best_local_genotypes;
  delete best_local_phenotypes;
  delete top_n_genotypes;
  delete top_n_phenotypes;
}

////////////////////////////////////////////////////////////////////
//
//  CREATE_INITIAL_POPULATION
//
//  Randomly creates system states which will evolve towards the
//  best hypothesis.
//  A description of individual i, is stored in row i of genotypes.
//  The performance metric is stored in row i of phenotypes.
//  Uses rand(), a pseudo random number generator.
//  A potential efficiency increase can be obtained by not
//  retrieving the number of states of nodes for each individual;
//  the information can be stored in a vector.  The increase would
//  be small.
//  crg  5/3/93: Modified gene assignment to avoid possible 0 value.
//  crg 1/10/94: Stores seed into 'suma000.txt' summary file.
//  crg 4/25/94: Stores seed into argv[6] file

void evolving_world::create_initial_population(unsigned int run_id){
  int individual, var, gene, number_of_states;
  vect* traits;
  node* np;
```

```
  //cout<< "\nCreation of initial population"; cout.flush();
  traits = bnet -> get_members();
  srand(run_id); //provides a seed for random number generation

  for(individual=1;individual<=evolving_population;++individual) {
    traits->start(var);
    while (traits->ok(var)) {
      np = (node*) traits->next(var);
      if (np->get_status() == f ) { //0 or f means unknown state
  number_of_states = np -> get_number_of_states();//gene=[1,n]
  gene=1+(int)((double) number_of_states*rand()/(32767+1.0));
      }
      else { //value of node is known since it has been instantiated
  gene = np->get_state();
      }
      specimen->element(var-1) = gene;
      genotypes->element(individual,var-1) = gene;
    }
    specimen->element(var) = individual;
    genotypes->element(individual, var) = individual;
    phenotypes->element(individual, 0) = individual;
  }
}

////////////////////////////////////////////////////////////////
//
//  START_EVOLUTION
//  The phenotype of the evolving population is initially evaluated
//  and stored to be used later as a basis for comparison.
//  Specimen is an int vect with traits from 0 to t-1, index in t.

void evolving_world::start_evolution() {
  int individual, trait;
  double fitness;

  for (individual=1; individual<=evolving_population; ++individual) {
    //copy row into specimen
    for (trait=1; trait<=number_of_traits; ++trait) {
      specimen->element(trait-1) =
        genotypes->element(individual, trait-1);
    }
    specimen->element(number_of_traits)=individual;
    //evaluate specimen and store metric in phenotypes
    fitness = calculate_metric(bnet,specimen);
    phenotypes->element(individual,1) = fitness;
  }
}

////////////////////////////////////////////////////////////////
//
//  INDIVIDUAL_EVALUATION
//  Evaluates the phenotype of any given genotype.
//  Based on a variation of start_evolution.
```

```
//
//  crg 5/2/93 Added function to test hypotheses individually
//  crg 1/5/94 Revise before using, identify args properly.

void evolving_world::individual_evaluation() {
  int individual, trait, given_trait;
  double fitness;
  char c;

  //cout << "\nEntrance to individual evaluation "; cout.flush();
  for (individual=1; individual>=1; ++individual) {

    for (trait=1; trait<=number_of_traits; ++trait) {
      cout <<"\nInput value for trait number " << trait << " ";
      cin >> given_trait;
      specimen->element(trait-1) = given_trait;
    }
    specimen->element(number_of_traits)=individual;
    cout << "\nRequested specimen is: "; specimen->print(3);
    //evaluate specimen
    fitness = calculate_metric(bnet,specimen);
    phenotypes->element(individual,1) = fitness;
    cout << "\nIts phenotype is: " << fitness; cin >> c;
  }
  //cout << "\nExit from individual evaluation "; cout.flush();
}

////////////////////////////////////////////////////////////////
//
//  SELECT
//  Erasing individuals with lowest metric is unnecessary since
//  the corresponding genotype storage can be overwritten.
//  The number of individuals to be replaced is births_per_generation.
//  Other methods are available besides quicksort.
//  Note that the general version of quicksort has its worst case
//  performance when the array is already ordered, but the version
//  used here is modified and introduces a random reorganization to
//  avoid that case.
//  Straight insertion is an N^2 sorting algorithm recommended only
//  for small N, (i.e. N<50), piksrt from Numerical recipes uses it.
//  For 50<N<1000, roughly, Shell's method, goes as N^(3/2) in the
//  worst case but is usually faster.  For randomly ordered data the
//  operations count goes approximately as N^(1.27) at least for
//  N<60,000 according to Press, et al, Numerical Recipes in C.
//  Best phenotype and genotype are stored for historical purposes.
//  Selection is done before crossover and mutation.
//  crg 11/4/93: Added storage of the best case for each generation.
//  crg 1/6/94: Added storage of local best and top n best
//    and documentation on data structures used for it.
//
//  Best_local_phenotypes: stores best phenotype of each generation.
//  All elements of the array are initialized with -1.
//  The first row of the array is presently not being used.
```

```
//  The first column (index=0) contains the generation number.
//  The 2nd column (index=2) contains the highest phenotype at that
//  generation.
//  The selection of the highest is done at the beginning of the
//  generation, Immediately after the population is sorted and
//  before mutations or crossover occur. Since the population is
//  not sorted after each mutation and crossover, the phenotypes
//  are not guaranteed to be ordered except immediately after
//  sorting.
//
//  Best_local_genotypes: stores the best hypothesis of each
//  generation. All elements of the array are initialized with -1.
//  The first row of the array (index=0) is presently not being used.
//  The 2nd row of the array (index=1) contains generation number 1.
//  The first column (index=0) stores the value of the first gene.
//  Gene N is stored in column (gene-1).
//  The last column (index=number_of_traits) contains the
//  generation number.
//
//  The size of the array (best_local_size) that stores these
//  hypotheses should be large enough to hold one genotype per
//  generation.  Since convergence is not known in advance, an
//  estimate is necessary (or a larger than expected size should
//  be used).  In a future version, this size can be changed
//  whenever necessary (i.e. every time M specimens are filled, a
//  new array of M+N specimens can be dynamically allocated and
//  the old one copied into the new one.

void evolving_world::select() {

  int best_genotype_position=-1;
  int gene, hypothesis, g;
  int hypothesis_already_in;
  int position_of_nth, genotype_position_in_top_n_set;

  //cout << "\nEntrance to selection"; cout.flush();
  //qcksrt(evolving_population, phenotypes);
  piksort(evolving_population, phenotypes);

  best_phenotype = phenotypes->element(evolving_population,1);//used
  best_genotype_position = phenotypes->element(evolving_population,0);

  //OPTIONAL RECORD KEEPING (can remain optional 3/27/94)

  //$$$NEW3B, added option 1 to test
  if(option_flag=='2' || option_flag=='1') {
    best_local_phenotypes->element(generation,1) = best_phenotype;
    best_local_phenotypes->element(generation,0) = generation;
    for (gene=1;gene<=number_of_traits;++gene) {
      best_local_genotypes->element(generation, gene-1) =
        genotypes->element(best_genotype_position,gene-1);
    }
    best_local_genotypes->element(generation,number_of_traits)
```

```
      = generation;
  }

//  Top_n_phenotypes and top_n_genotypes ever found are stored in a
//  small predefined-size matrix (store the solution and its p).
//  If the best phenotype in this generation is better than the
//  worst in the top_n list, then consider including it in the set.
//  Consider checking to make sure that the new hypothesis is not
//  already in the set - the obvious drawback is the computational
//  cost of this comparison - which perhaps can be made negligible
//  by using a small n for the size of the top_n list. Check for
//  differences, and the first obvious difference is the phenotype;
//  the probability of 2 hypotheses being different and having the
//  same phenotype is small, if both have the same phenotype, then
//  check genes until the 1st difference is found.
//
//  In top_n_phenotypes, after sorting the worst phenotype is stored
//  at (1,1) and the best at (top_n_size,1). The row at which the
//  corresponding genotypes are stored in top_n_genotypes is stored
//  at (1,0) and (top_n_size,0).
//
//  Substitute worst in top n with best_phenotype and then sort the
//  top_n set. After accepting a hypothesis for addition, it is
//  necessary to substitute both the phenotype and the genotype of
//  the nth (worst) hypothesis in the set.  If at least one
//  hypothesis matches, then the new hypothesis is not unique and
//  its addition is not necessary.  The worst among the top n is
//  stored at row 1. The genotype position of top_n_phenotypes
//  (row=1,col=0) remains constant. Save time by not doing the
//  top_n comparisons for the first Y generations because those
//  genotypes will be discarded anyway. Determine Y as a fraction
//  of the total expected generations.


  //OPTIONAL SECTION FOR TOP N RECORDING
  //$$$NEW3B: added option 1
  if(option_flag=='2' || option_flag=='1') {
  //cout << "\nStarting top N"; cout.flush();
  hypothesis_already_in = 0; //initial value
  if (best_phenotype>top_n_phenotypes->element(1,1)){//if > nth
    for (hypothesis=1; hypothesis<=top_n_size; ++hypothesis) {
      if (hypothesis_already_in == 1) {
  break; //if previous hypothesis has no difference,
         //already in, don't search/add
      }
      hypothesis_already_in = 1;
      if (best_phenotype == top_n_phenotypes->element(hypothesis,1)){
  genotype_position_in_top_n_set
    = top_n_phenotypes->element(hypothesis,0);
  for (gene=1; gene<=number_of_traits; ++gene) {
    if (genotypes->element(best_genotype_position,gene-1) !=
        top_n_genotypes->element(genotype_position_in_top_n_set,
              gene-1) ) {
```

```
      hypothesis_already_in = 0;
      //cout << "\nAt least one gene has a different allele";
      break;//after a difference is found, tries next hypothesis
          }
  }//closes for gene
  //break jumps to here
      }//closes if
      else {
        //cout << "\nAt least the phenotype is different";
  hypothesis_already_in = 0;
      }
    }//closes for(hypothesis=1...

    if (hypothesis_already_in == 0) {
      //cout<<"\nDifferent top local hypothesis - will be added";
      top_n_phenotypes->element(1,1) = best_phenotype;
      //position_of_nth is the row for the corresponding genotype
      position_of_nth = top_n_phenotypes->element(1,0);
      //cout << "\nposition of nth = " << position_of_nth << "\n";
      for (g=1; g<=number_of_traits; ++g) {
        top_n_genotypes->element(position_of_nth,g-1) =
            genotypes->element(best_genotype_position,g-1);
      }
      piksort(top_n_size, top_n_phenotypes);//sorts only if new added
    }
  }//closes if statement that filters admission to top_n
  else
    cout<<"\nBest at this Gen not high enough to enter top_n_set";
 }//closes if option_flag==2

  //print();
  //cout << "\nExit select (sorting) of evolving population";
  //cout.flush();
}

////////////////////////////////////////////////////////////////
//
//  COMBINE
//  Takes the best individuals 'breeding_population' and creates
//  'births_per_generation' new individuals using a crossover method.
//  1. random parents and random positions, 2 parents->1 child
//  2. random parents and random positions, 2 parents->2 children
//  3. semantically close nodes form blocks in network structure
//  4. interleaving-no blocks are inherited (expected low performance)
//  5. randomly chooses blocks or interleaving and uses both methods
//  6. combines subnetworks of belief networks keeping building blocks
//     as compact units with close semantic relations.
//  number of breeding individuals: breeding_population
//  number of new individuals: births_per_generation (crossover 1)
//  number of new individuals: 2 x pairs_per_generation(crossover 2)

void evolving_world::combine() {
  //cout << "\nEntrance to COMBINE"; cout.flush();
```

```
  if (crossover_method == 1)
    crossover_1();
  else if (crossover_method == 2)
    crossover_2();
  else if (crossover_method == 3)
    crossover_3();
  else bnerror("\nGALGO: combine: crossover_method not defined");
  //cout << "\nExit from COMBINE"; cout.flush();
}


////////////////////////////////////////////////////////////////
//
//  MUTATE
//  Mutates and evaluates mutants.
//
//  Method 1: Mutation on one position of the string if
//  non-instantiated.
//  Risked_index refers to a row in the phenotypes matrix.
//  The row number at which the corresponding genotype is stored
//  in the genotype matrix can be found in
//  phenotypes(row=risked_index,col=0)
//  Risked individual corresponds to a row in the genotypes matrix
//  A gene (a position in the genotype) is selected and changed.
//  The phenotype for the new genotype is calculated and stored.
//
//  Method 2: Mutation is avoided for the breeding population since
//  p of improvement is much smaller than p of degeneration.
//  ... if the mutation is on a breeding element, then do not
//  substitute, create a new individual ... or always substitute on
//  lowest in the rank
//
//  crg 3/4/92: The minimum number of mutations will be ONE. If the
//              mutation frequency is too low, it is upgraded to 1.
//  crg 5/4/93: Changed rand scaling for risked_index,
//              mutation_position and mutated state.
//  crg 10/12/93:Commented 'attempted mutation on fixed gene' message
//  crg   2/5/94:Updated 'phenotypes' with post_mutation_metric after
//               mutation takes place.

void evolving_world::mutate() {
  int mutant, number_of_mutants, risked_individual, risked_index;
  int mutation_position, mutated_state, number_of_states, t;
  double pre_mutation_metric, post_mutation_metric;
  vect* traits;
  node* trait;

  //cout << "\nStarting mutation for generation " << generation;
  //cout.flush();
  //traits is a pointer to a vector of members
  traits = bnet -> get_members();
  if (mutation_method == 1) {
    //cout << "\nEntrance to mutation_method 1";
```

```
  number_of_mutants = floor(evolving_population*mutation_frequency);
  if (number_of_mutants<1) number_of_mutants=1; //MINIMUM=1

  for(mutant=1; mutant<=number_of_mutants; ++mutant) {

    //OPTIONAL RECORD KEEPING
    //accumulated mutations
    if(option_flag=='2') ++history->element(generation,5);

    //risked_index=ceil((double)evolving_population*rand()/32767.0);
    risked_index = 1 +
      (int)((double)evolving_population*rand()/(32767+1.0));//[1,n]
    risked_individual = phenotypes->element(risked_index, 0);
    mutation_position = 1 +
      (int)((double)number_of_traits*rand()/(32767+1.0));//[1,n]
    trait=(node*)traits->element(mutation_position-1);//ind=[0,n-1]
    if (trait->get_status() == f) {
//cout<<"\n\nMutation on individual "<<risked_individual;
//cout << "\nMutation position = " << mutation_position;
number_of_states = trait->get_number_of_states();
mutated_state = 1 +
  (int)((double)number_of_states*rand()/(32767+1.0));//[1,n]
      //the genotype is changed and the change is stored
genotypes->element(risked_individual,mutation_position-1) =
  mutated_state;

//evaluate the mutant by copying it into the 'specimen' array
for (t=1; t<=number_of_traits; ++t) {
  specimen->element(t-1) =
    genotypes->element(risked_individual,t-1);
}
specimen->element(number_of_traits)=risked_individual;

pre_mutation_metric = phenotypes->element(risked_index,1);
post_mutation_metric = calculate_metric(bnet,specimen);
//the phenotype has changed, and the change is stored:
phenotypes->element(risked_index,1)=post_mutation_metric;
//cout << "\nMutated specimen is: "; specimen->print(3);
//cout << "\nMetric before mutation: " << pre_mutation_metric;
//cout << "\nMetric after mutation: " << post_mutation_metric;

      //OPTIONAL RECORD KEEPING
      if(option_flag=='2') {
  if (post_mutation_metric > pre_mutation_metric) {
    history->element(generation,3) += 1.0;
    //cout << "\nPositive mutation";
  }
  else if(post_mutation_metric < pre_mutation_metric) {
    history->element(generation,4) += 1.0;
    //cout << "\nNegative mutation";
  }
  else {
    //cout << "\nNeutral mutation";
```

```
    }
        }

      }
      else {
        //cout<<"\nAttempted mutation on fixed (instantiated) gene";
  if(option_flag=='2') ++history->element(generation,16);
      }
    }     //close for m=1...
  }       //close if method==1
  else bnerror("\nGALGO: Mutate: mutation_method unavailable");
  //cout << "\nExit from mutation"; cout.flush();
}

////////////////////////////////////////////////////////////////
//
//  EVALUATE_OFFSPRING
//  To keep historical records of evolution, new individuals are
//  evaluated immediately after their creation.  By waiting until
//  this point to evaluate them, the effects of mutation and
//  crossover would be combined and indistinguishable.  However,
//  keeping the evaluation in one single  nction here can
//  accomplish some reduction in the size of the code by avoiding
//  repetition.
//  crg 12/18/92: Each individual is evaluated immediately after
//                being created or mutated, consequently this
//                function is not necessary anymore.
//
//void evolving_world::evaluate_offspring() {
//  int individual, trait;
//  cout << "\nEntrance to evaluate_offspring"; cout.flush();
//  cout << "\nExit from evaluate_offspring"; cout.flush();
//}

////////////////////////////////////////////////////////////////
//
//  MONITOR_EVOLUTION
//  History is a record of the evolution process and is encoded as
//  a dmatrix_2dim with one row summarizing each generation:
//
//  column0=generation
//  column1=evolving probability mass
//  column2=breeding probability mass
//
//  column3=num improved mutations in this generation (positive
mutations)
//  column4=number of negative mutations
//  column5=accumulated mutations in this generation
//  column6=positive mutation fraction in this generation
//
//  column7=accumulated offspring
//  column8=positive offspring
//  column9=negative offspring
```

```
//  column10=positive offspring fraction
//  column11=best phenotype in this generation
//  column12=best phenotype found so far (off-line performance)
//
//  column13=average phenotype in generation(on-line performance)
//  column14=average evolving probability
//  column15=average breeding probability
//  column16=attempted mutations on instantiated gene
//  column17=exchanged block size from crossover_3 bn
//  column18=average exchanged block size from crossover_3
//  column19=convergence ratio(added crg 11/2/93,removed 3/27/94)

void evolving_world::monitor_evolution() {
  int selection;
  double evolving_probability_mass = 0.0;
  double average_evolving_probability_mass = 0.0;
  double breeding_probability_mass = 0.0;
  double average_breeding_probability_mass = 0.0;

  //cout << "\nEntrance to monitor_evolution"; cout.flush();
  for (selection=1; selection<=evolving_population; ++selection) {
    evolving_probability_mass += phenotypes->element(selection,1);
  }
  average_evolving_probability_mass = evolving_probability_mass /
    evolving_population;
  for (selection = evolving_population - breeding_population;
       selection <= evolving_population;
       ++selection ) {
    breeding_probability_mass += phenotypes->element(selection,1);
  }
  average_breeding_probability_mass = breeding_probability_mass /
     breeding_population;

  history->element(generation,0) = generation;
  history->element(generation,1) = evolving_probability_mass;
  history->element(generation,2) = breeding_probability_mass;
  if (history->element(generation,5)==0)
    bnerror(
      "Monitor_evolution: division by 0, accumulated mutations");
  history->element(generation,6) = history->element(generation,3) /
           history->element(generation,5);
  history->element(generation,11) = best_phenotype;
  if (best_phenotype > history->element(generation-1,12) )
    history->element(generation,12) = best_phenotype;
  else
    history->element(generation,12)=history->element(generation-1,12);

  history->element(generation,14) = average_evolving_probability_mass;
  history->element(generation,15) = average_breeding_probability_mass;
  if (history->element(generation,7)==0)
    bnerror("Monitor_evolution: division by 0, accumulated offspring");
  history->element(generation,10) = history->element(generation,8) /
            history->element(generation,7);
```

```
  //cout << "\nExit from monitor_evolution"; cout.flush();
}

////////////////////////////////////////////////////////////////
//
//  PARAMETERS ADJUSTMENT
//  After evaluating its own performance, the algorithm changes
//  the parameters of the simulated evolution to optimize the
//  search.
//  Consider the following extension:
//  From records stating percentage of network selected by expansion
//  calculate the percentage in each side,
//  if size of clusters is not half, move in right direction by one
//
//  VARIABLES:
//  frequency_of_adjustment: Parameters are adjusted when the
//       generation number is a multiple of this variable.
//  ratio: Average phenotype of breeding population at present
//       divided by the average phenotype of breeding population
//       four generations before. A value of 1 indicates convergence.
//  scaling factor: Proportionality constant for mutation_frequency
//       adjustment.
//  close_to_convergence: value below which mutation is adjusted.
//
//  crg 11/2/93: mutation_frequency adjusted as a function of
//               convergence_ratio.
//  crg 3/29/94: removed dependence on historical data.

void evolving_world::parameters_adjustment(double *previous_phenotype,
             int strategy) {
  int frequency_of_adjustment = 10;
  double ratio=100.0, scaling_factor=0.20, close_to_convergence=5.0;
  double max_mutation_frequency=0.5;

  if (generation % frequency_of_adjustment == 0) {
    if (generation > 5) {
      ratio=phenotypes->element(evolving_population,1)/
                        (*previous_phenotype);
      //ratio = phenotypes->element(evolving_population,1)/
                          //history->element(generation-4,15);
      if ( ratio < close_to_convergence ) {
        mutation_frequency *= 1+scaling_factor*(1/ratio);
        if (mutation_frequency >= max_mutation_frequency)
    mutation_frequency = max_mutation_frequency;
        cout<<"\nAdjusted mutation frequency = "<<mutation_frequency;
      }
      else mutation_frequency = initial_mutation_frequency;
    }
  }
}

////////////////////////////////////////////////////////////////
//
```

```
//  PRINT
//  Prints at different levels of detail 1,...,n
//  By default level=1

void evolving_world::print(int level) {
  cout << "\nEvolving world object: " << name_of_world;

  if(level>=2) {
    cout << "\nRegulated by belief network: ";
    cout << bnet->get_name();
    cout << "\nTotal population       =  " << total_population;
    cout << "\nTotal generations      =  " << total_generations;
    cout << "\nNumber of traits       =  " << number_of_traits;
    cout << "\nEvolving fraction      =  " << evolving_fraction;
    cout << "\nEvolving population    =  " << evolving_population;
    cout << "\nBreeding selectivity   =  " << breeding_selectivity;
    cout << "\nBreeding population    =  " << breeding_population;
    cout << "\nLifetime               =  " << lifetime;
    cout << "\nCrossover method       =  " << crossover_method;
    cout << "\nParent selector        =  " << parent_selector;
    cout << "\nBirths per generation  =  " << births_per_generation;
    cout << "\nPairs per generation   =  " << pairs_per_generation;
    cout << "\nSampling method        =  " << sampling_method;
    cout << "\nMutation method        =  " << mutation_method;
    cout << "\nMutation frequency     =  " << mutation_frequency;
  }

  if(level>=2) {
    cout << "\nEvolutionary historical records: \n";
    history->print(2);
  }
  cout << "\nGeneration " << generation;
  genotypes->print(2);
  cout << "\n";
  phenotypes->print(2);
  transformed_phenotypes->print(2);
  cout << "\n";
  //specimen->print(3);
  //cout << "\nPrinting done";
}

////////////////////////////////////////////////////////////////
//
//  BLOCK_TEST
//  Given a string position (or node number), the test determines
//  whether the gene corresponding to the string position
//  is an element of the 'compact block' or not.
//  By changing this function, different criteria can be easily
//  implemented.  Simple extensions are:
//  (1) size of the block can be limited to a fixed number,
//  (2) block size can be determined according to a given size
//  distribution (i.e. Gaussian, N(x,s)),
//  (3) distance from the center without considering number of
```

```
//  elements in the block
//  By default block_method=1
//  block_method=1: all immediate neighbors constitute the block
//  block_method=2: in development
//  block_method=3: in development

boolean block_test(int trait, int block_method) {

  if (block_method == 1) {
    bnerror("\nBlock_test number 1 - Not implemented yet");
  }
  else if (block_method == 2) {
    bnerror("\nBlock_test number 2 - Not implemented yet");
  }
  bnerror("\nUnknown block_method identifier");
  return(f);
}

////////////////////////////////////////////////////////////////
//
//  CROSSOVER_1 METHOD
//  2 parents, 1 newborn
void evolving_world::crossover_1() {
{
  bnerror("Crossover_1 method has been substituted by Crossover_3");
}

////////////////////////////////////////////////////////////////
//
//  CROSSOVER_2 METHOD:
//  2 parents, 2 newborn individuals
//  2 random positions in the string guide crossover
//  NEEDS CORRECTION OF G_INDEX, C_INDEX...


void evolving_world::crossover_2() {
}

////////////////////////////////////////////////////////////////
//
//  CROSSOVER_3 METHOD: 2 PARENTS, 2 NEWBORN INDIVIDUALS
//
//  Semantically close genes make up compact blocks in the BN
//  structure. Two parents generate two children.
//  The size of an exchange block is a function of expansion
//  levels. When the number of expansion levels is equal to 2,
//  the exchange block contains the center node an all its
//  immediate neighbors in the graph (verify 0 or 1)
//
//  Assumes position on string corresponds to node number in bn.
//  Nodes are added at the head of the waiting list (left), but
//  the counter or iterator continues moving to the right until
//  the start_iteration function is called again.
```

```
void evolving_world::crossover_3() {
  int newborn, traits, trait, t, temp, parent;
  int pair, local_block_size, level;
  int child, child1, child2, center;
  int p_num, c_num, num_parents, num_children;
  int index, c_index1, c_index2;
  double new_phenotype;
  char c;

  node *p_node, *c_node, *center_node, *expanding_node;
  node_list *exchanged, *waiting_list;

  //cout << "\nSTARTING CROSSOVER 3"; cout.flush();
  //genotypes->print(2);
  //phenotypes->print(2);

  for (pair=1; pair <= pairs_per_generation; ++pair) {
    exchanged = new node_list;
    waiting_list = new node_list;
    //cout << "\nBefore select_parents" << cout.flush();
    select_parents(parent_selector);
    //cout << "\nAfter select_parents" << cout.flush();
    //phenotypes already calculated in select_parents()
    //phenotype1 = phenotypes->element(g_index1, 1);
    //phenotype2 = phenotypes->element(g_index2, 1);
    //center = ceil(rand()/(double)32767.0*number_of_traits);//0?!
    //cout << "\nNumber of traits = " << number_of_traits;

    center = 1 +
      (int)((double)number_of_traits*rand()/(32767+1.0));//[1,n]
    //cout << "\nCENTER = " << center;
    if (center==0)
  bnerror(
    "CENTER in CROSSOVER3 has been incorrectly assigned to 0");

    c_index1 = pair*2 - 1;//cout << "\nCHILD INDEX 1 = " << c_index1;
    c_index2 = pair*2;    //cout << "\nCHILD INDEX 2 = " << c_index2;
    child1 = phenotypes->element(c_index1,0);
    child2 = phenotypes->element(c_index2,0);

    //BN BLOCK CONSTRUCTION: start by copying parents into children
    for (trait=1; trait<=number_of_traits; ++trait) {
      genotypes->element(child1,trait-1) =
        genotypes->element(parent1,trait-1);
      genotypes->element(child2,trait-1) =
        genotypes >element(parent2,trait-1);
    }
    //exchange compact building blocks
    center_node = (node*) bnet->get_member(center);
    num_parents = center_node->get_number_of_parents();
    num_children = center_node->get_number_of_children();
    local_block_size = num_parents + num_children;
```

```
  //history->element(generation,17) = num_parents + num_children;

  //cout << "\nWaiting list before adding first element: ";
  //waiting_list->print(); cin >> c;
  waiting_list->add(center_node); //only one element so far
  //cout << "\nWaiting list after adding first element";
  //waiting_list->print(); cin >> c;

  //cout << "\nExpansion levels = " << expansion_levels;
  for (level=1; level<=expansion_levels; ++level) {
    //cout<<"\nStart wl iteration for new expansion level "<<level;
    waiting_list->start_iteration();
    while (waiting_list->ok()) {
//cout << "\nStarting expansion loop again ";
//cout << "on next waiting list element";
expanding_node = waiting_list->next();
//check that all instances are removed
waiting_list->remove(expanding_node);
p_num = expanding_node->get_node_number();
//cout<<"\np_num for expanding node is " << p_num;
//cout<<"Node "<<expanding_node->get_name();
//cout<<" being exchanged ";
//cin>>c;
genotypes->element(child1,p_num-1) =
  genotypes->element(parent2,p_num-1);
genotypes->element(child2,p_num-1) =
  genotypes->element(parent1,p_num-1);
exchanged->add(expanding_node);
//cout << "\nAFTER exchanging an allele";
//genotypes->print(2);
//phenotypes->print(2);

//if ( ++local_block_size < block_size) {
num_parents = expanding_node->get_number_of_parents();
num_children = expanding_node->get_number_of_children();
for (parent=1; parent<=num_parents; ++parent) {
  p_node = expanding_node->get_parent(parent);
  //cout << "\nParent " << parent;
  //cout << " out of " << num_parents << ": ";
  //cout << p_node->get_name();
        //the following 2 tests can be combined in one single test
  if (exchanged->presence_test(p_node) == f) {
    //cout<<"\nPresence test shows node has not been exchanged";
    p_num = p_node->get_node_number();

    //genotypes->element(child1,p_num-1) =
    //  genotypes->element(parent2,p_num-1);
    //genotypes->element(child2,p_num-1) =
    //  genotypes->element(parent1,p_num-1);
    //exchanged->add(p_node);
    //test to add to wl(may be in wl, but not yet exchanged)
          if (waiting_list->presence_test(p_node) == f ) {
      //waiting_list->print(); cin >> c;
```

```
      waiting_list->add(p_node);
      //cout << "\nP Node " << p_node->get_name();
      //cout << " added to wl";
      //waiting_list->print(); cin >> c;
    }
  }
}
for (child=1; child<=num_children; ++child) {
  c_node = expanding_node->get_child(child);
  //cout << "\nChild " << child ;
  //cout << " out of " << num_children << ": ";
  //cout << c_node->get_name();
  if (exchanged->presence_test(c_node) == f) {
    //cout << "\nPresence test shows the node ";
    //cout << has not been exchanged";
    c_num = c_node->get_node_number();

    //cout << "\nC Node " <<c_node->get_name() ;
    //cout << " being added to wl";
    //genotypes->element(child1,c_num-1)=
    //  genotypes->element(parent2,c_num-1);
    //genotypes->element(child2,c_num-1)=
    //  genotypes->element(parent1,c_num-1);
    //exchanged->add(c_node);
    //test to add to wl
    if (waiting_list->presence_test(c_node) == f ) {
      //waiting_list->print(); cin >> c;
      waiting_list->add(c_node);
      //cout<<"\nC Node "<<c_node->get_name()<<" added to wl";
      //waiting_list->print(); cin >> c;
    }
  }
}
//}old if
    }
  }
  //cout << "\n\nAfter the exchange of all newborns";
  //genotypes->print(2);
  //phenotypes->print(2);

  //calc phenotypes of newborn individuals for historical purposes
  for (index=c_index1; index<=c_index2; ++index) {
    //cout << "\nC_indices being evaluated ";
    //cout << c_index1 << " and " << c_index2;
    //cout.flush();
    child = phenotypes->element(index, 0);
    for (t=1; t<=number_of_traits; ++t) {
specimen->element(t-1)=genotypes->element(child, t-1);
    }
specimen->element(number_of_traits)=child; //unnecessary (?)
//cout << "\n"; specimen->print(3); //cout.flush();
new_phenotype = calculate_metric(bnet,specimen);
//cout << "\nNew phenotype measured is " << new_phenotype;
```

```
  //cout.flush();
  phenotypes->element(index,1) = new_phenotype;
  //cout<<"\nPHENOTYPE 1 = "<<phenotype1;
  //cout<<" PHENOTYPE 2 = "<<phenotype2;
  //cout.flush();
   /cout << "\n";  specimen->print(3); //cout.flush();
  //cout<<"\nNEW phenotype for child "<<child;
  //cout<<" = "<<new_phenotype;
  //cout.flush();

  //OPTIONAL RECORD KEEPING
        if(option_flag=='2') {
    ++history->element(generation,7); //accumulated offspring
    if ((new_phenotype>phenotype1)&&(new_phenotype>phenotype2)){
      ++history->element(generation,8); //positive offspring
      //cout << "\nIMPROVED offspring phenotype";
      //cout.flush();
    }
    else if
      ((new_phenotype<phenotype1) && (new_phenotype<phenotype2)){
      ++history->element(generation,9); //negative offspring
      //cout << "\nLOWER phenotype than both parents";
      //cout.flush();
    }
    else { //cout << "\nINTERMEDIATE offspring"; //cout.flush();
    }
  }

      }
      //to ensure memory is deallocated:
      delete exchanged;
      delete waiting_list;
      //cin >> c;
    }
    //cout << "\nExit from CROSSOVER_3"; cout.flush();
    //genotypes->print(2);
    //phenotypes->print(2);
    //delete exchanged, waiting_list;  //moved on 3/4/93
}


////////////////////////////////////////////////////////////////
//
//  SELECT_PARENTS
//  (1) Probability of being selected is 1/breeding_population for
//      those individuals within the breeding population, and 0
//      for others.
//  (2) The probability of being selected is proportional to the
//  individual's phenotype, and 0 for those outside breeding
//      population
//  (3) The probability of being selected is proportional to the
//      ranking within all the evolving population
//
```

```
//  g_index1 and g_index2 are indices which indicate positions in
//  the phenotype matrix. Higher indices correspond to better points
//  The location of the genotype which originated the phenotype
//  contained in the g_indexi position is located in a row of the
//  genotype matrix. The row numbers can be found in the phenotype's
//  column 0.
//  crg   5/4/93: extended to use any transformed phenotypes
//                for selection
//  crg 10/10/93: revised method 3, suspected of giving out of
//                bounds index

void evolving_world::select_parents(int parent_sel) {
  char c;

  // FILE *temp_ofp;
  //  if ((temp_ofp=fopen("temp_out.txt","a"))==NULL)
  //    bnerror("Unavailable temp file.");

  ////////////////////////////////////////////////////////////////
  //
  //  Parent selection - Method 1
  //  Randomly choose 2 parents from breeding population

  if (parent_sel == 1) {
    //cout <<"\nEntrance to select_parents-Method 1";cout.flush();
    g_index1 =
      ceil(evolving_population-rand()/32767.0*breeding_population);
    g_index2 =
      ceil(evolving_population-rand()/32767.0*breeding_population);
    parent1 = phenotypes->element(g_index1,0);
    parent2 = phenotypes->element(g_index2,0);
    //cout << "\n\nPARENT 1 = " << parent1;
    //cout << "\nPARENT 2 = " << parent2;
    phenotype1 = phenotypes->element(g_index1, 1);
    phenotype2 = phenotypes->element(g_index2, 1);
  }

  ////////////////////////////////////////////////////////////////
  //
  //  Parent selection - Method 2

  else if (parent_sel == 2) {
    int i, interval_size, r1, r2;
    double phenotype_sum = 0.0;
    //cout <<"\nEntrance to select_parents - Method 2";cout.flush();
    //find the sum of all breeding population phenotypes
    for (i = evolving_population - breeding_population + 1;
   i <= evolving_population;
   ++i)  {
      phenotype_sum += phenotypes->element(i,1);
    }
    cout << "\nPhenotype sum = " << phenotype_sum;
```

```
    //calculate normalized fraction and interval for each phenotype
    for (i=evolving_population-breeding_population+1;
   i<=evolving_population;
   ++i) {
      interval_size = phenotypes->element(i,1)/phenotype_sum*32767.0;
      //cout << "\nInterval size = " << interval_size;
      phenotypes->element(i,2) =
        phenotypes->element(i-1,2) + interval_size;
    }
    //cout << "\nLast interval upper bound before adjustment";
    //cout << phenotypes->element(evolving_population,2);
    phenotypes->element(evolving_population,2) = 32767.0;
    //cout << "\nLast interval upper bound after adjustment";
    //cout << phenotypes->element(evolving_population,2);
    //phenotypes->print(2);
    r1= rand(); r2 = rand();

    for (i=evolving_population-breeding_population+1;
   i<=evolving_population;
   ++i)  {
      if (r1 < phenotypes->element(i,2) )
  break; //interval i contains the random number r1
    }
    g_index1 = i;
    //cout << "\ng_index1 = " << g_index1;
    parent1 = phenotypes->element(g_index1, 0);
    for (i = evolving_population - breeding_population + 1;
   i <= evolving_population;
   ++i)  {
      if (r2 < phenotypes->element(i,2) )
  break; //i contains the interval number
    }
    g_index2 = i;
    //cout << "\ng_index2 = " << g_index2;
    parent2 = phenotypes->element(g_index2, 0);
    cout << "\n\nPARENT 1 = " << parent1;
    cout <<   "\nPARENT 2 = " << parent2;
    phenotype1 = phenotypes->element(g_index1, 1);
    phenotype2 = phenotypes->element(g_index2, 1);
    cout << "\nPhenotype 1 = " << phenotype1;
    cout << "\nPhenotype 2 = " << phenotype2;
    //cin>>c;
  }

////////////////////////////////////////////////////////////////
//
//  Parent selection - Method 3
//  Generalizes to any f(phenotype) or f(rank)
//
//  Apply transformation to phenotypes set
//
//  ...random quick thoughts for personal use only:
//  test when it is necessary to do the calculations again, it is
```

```
//  not necessary to do it every time parents are selected, perhaps
//  every generation, but then a few individuals are changed and
//  will require updating (in phenotypes already done) and in
//  transformed_phenotypes; temporarily this will do it before every
//  parent selection, but this function should be extended to make
//  it efficient.
//  transformation=1 uses natural log, or 'log'
//  ..do it only for the breeding population, not from 1, and since
//  the breeding population  is not substituted (except for
//  mutations), it is not necessary but to update once per generation

  else if (parent_sel == 3) {
    int i, j, interval_size, r1, r2;
    double transformed_phenotypes_sum = 0.0;
    double original_ph,transformed_ph;

    //cout <<"\nEntrance to select_parents - Method 3"; cout.flush();
    if (transformation == 1) {
      //cout << "\nEntrance to transformation 1";
      for (j = evolving_population - breeding_population + 1;
           j <= evolving_population;
     ++j)  {
  original_ph = phenotypes->element(j,1);
  if (original_ph == 0)
    transformed_ph = 1e-300;
  else {
    transformed_ph = (1/log(original_ph))*(1/log(original_ph));
        }
        transformed_phenotypes->element(j,1) = transformed_ph;
      }
    }
    else if (transformation == 2) {
      bnerror("Phenotype transformation 2 not implemented yet");
    }
    else if (transformation == 3) {
      bnerror("Phenotype transformation 3 not implemented yet");
    }
    else bnerror("Unknown phenotype transformation");

    //find the sum of all breeding population transformed_phenotypes
    for (i = evolving_population - breeding_population + 1;
   i <= evolving_population;
   ++i)  {
      transformed_phenotypes_sum +=
        transformed_phenotypes->element(i,1);
    }
    //cout << "\nTransformed_phenotypes_sum = ";
    //cout << transformed_phenotypes_sum;
    //calculate normalized fraction and interval for each phenotype
    for (i=evolving_population-breeding_population+1;
   i<=evolving_population;
   ++i) {
      interval_size =
```

```
transformed_phenotypes->element(i,1) /
  transformed_phenotypes_sum*32767.0;

    //cout << "\nInterval size = " << interval_size;
    transformed_phenotypes->element(i,2) =
      transformed_phenotypes->element(i-1,2) + interval_size;
  }
  //cout << "\nLast interval upper bound before adjustment";
  //cout << phenotypes->element(evolving_population,2);
  transformed_phenotypes->element(evolving_population,2) = 32767.0;
  //cout << "\nLast interval upper bound after adjustment";
  //cout << phenotypes->element(evolving_population,2);
  //phenotypes->print(2); cin >> c;
  //transformed_phenotypes->print(2);  cin >> c;
  r1= rand(); r2 = rand();
  for (i=evolving_population-breeding_population+1;
 i<=evolving_population;
 ++i)  {
    if (r1 < transformed_phenotypes->element(i,2) )
break; //interval i contains the random number r1
  }
  g_index1 = i;
  //check for valid bounds on g_index1 and g_index2, crg 10/10/93
  if (g_index1 > evolving_population ) {
    //OPTIONAL DEVELOPMENT MESSAGE
    if(option_flag=='2') {
cout<<"\nWARNING:parent_selector3:g_index1 ";
cout<<" > evolving_population";
      cout<<"\nAssigned value g_index1 = "<< g_index1;
cout<<"\nWill set g_index1 = adjusted evolving_population";
    }
    g_index1 = evolving_population;
  }
  parent1 = phenotypes->element(g_index1, 0);
  for (i = evolving_population - breeding_population + 1;
 i <= evolving_population;
 ++i)  {
    if (r2 < transformed_phenotypes->element(i,2) )
break; //i contains the interval number
  }
  g_index2 = i;
  //cout << "\ng_index2 = " << g_index2;
  if (g_index2 > evolving_population ) {
    cout<<"\nWARNING:parent_selector3:g_index2>evolving_population";
    cout<<"\nAssigned value g_index2 = "<< g_index2;
    cout<<"\nWill set g_index2 = adjusted evolving_population";
    g_index2 = evolving_population; //evolv_pop has been adjusted
    //cin >> c;
  }
  parent2 = phenotypes->element(g_index2, 0);
  //cout << "\n\nPARENT 1 = " << parent1;      //crg 10/10/93
  //cout <<   "\nPARENT 2 = " << parent2;
  phenotype1 = transformed_phenotypes->element(g_index1, 1);
```

```
    phenotype2 = transformed_phenotypes->element(g_index2, 1);
    //cout << "\nTransformed Phenotype 1 = " << phenotype1;
    //cout << "\nTransformed Phenotype 2 = " << phenotype2;
    //cin>>c;
  }
  //else if (parent_sel==4){//this would be a special case of 3...
    //
    //only the first time: (this is the key benefit, only once)
    //  find sum of all breeding pop ranks
    //  for each phenotype, find its rank/sum as its factor
    //  calculate the upper bound by sequential addition
    //every time the function is called:
    //  find rand()
    //  determine position and return lucky parent
    //
  //}
  else bnerror("\nInvalid Parent selector method");
  //cout << "\nExit from select parents"; cout.flush();
  //fclose(temp_ofp);
}

////////////////////////////////////////////////////////////////////
//
//  SUMMARIZE
//  Summarizes evolution history, including:
//  1. average block size
//  2.

void evolving_world::summarize() {
  int i, average_block_size=0, total=0;

  for (i=1; i<=total_generations; ++i) {
    total += history->element(i,17);
  }
  average_block_size = total / total_generations;
  history->element(total_generations,18) = average_block_size;
}

////////////////////////////////////////////////////////////////////
//
//  CONVERGENCE_TEST()
//  Test for convergence - fast, moderate, slow or stable.
//  Convergence is assessed with the ratio of the best
//  phenotype at the present generation and the value it had
//  four generations before.
//
//  The best phenotype of the latest generation is stored at
//  phenotypes->element(evolving_population,1)
//  and the best phenotype 4 generations before is stored at
//  history->element(generation-4,11)
//
//  Ratio discretization and screen output:(note change)
//      ratio > 10          rapid
```

```
//      2 < ratio < 10      moderate
//      1.01 < ratio < 2    slow
//      ratio < 1.01        stable
//
//  crg 11/1/93: Added function.
//  If generation <= 5, the test returns 0 (not converged yet),
//  since not enough historical information exists.
//  crg 2/10/94: changed interval limit from 1.01 to 1.20


int  evolving_world::convergence_test(double* previous_phenotype) {
  int stability = 0;  //0=not stable, 1=convergence
  double convergence_ratio;

  if (generation > 10)  {
    convergence_ratio =
      phenotypes->element(evolving_population,1) /
      (*previous_phenotype);
    if(option_flag=='2') {
      cout<<"\nNPH="<<phenotypes->element(evolving_population,1);
      cout<<" OPH="<< *previous_phenotype;
      cout<<" Convergence ratio = " << convergence_ratio;
    }
    //discretization of ratio and output to user
    cout<<"\nThe genetic algorithm ";
    if (convergence_ratio > 10)
      cout << "is converging rapidly.";
    else if (convergence_ratio > 2 && convergence_ratio < 10)
      cout << "is converging at a moderate speed.";
    else if (convergence_ratio > 1.05  && convergence_ratio < 2)
      cout << "is converging slowly.";
    else if (convergence_ratio < 1.05) {
      cout << "is converging very slowly.";
      stability = 1;
    }
  }
  *previous_phenotype=phenotypes->element(evolving_population,1);
  return stability;
}

////////////////////////////////////////////////////////////////
//
//  NATURAL_LANGUAGE_OUTPUT()
//
//  crg 10/13/93: Added functionality
//  crg 10/31/93: Wrote independent function using states_names
//
//  Postprocessing of genotypic information.
//  Preliminary version of natural language description of the
//  genotype.  Variables of special interest have been selected.
//
//  A subset of the variable is selected at run-time,
//  only those in an abnormal state will be printed out.
```

```
//  State names (as opposed to numbers) are used for clarity.
//  crg 2/12/94: Changed to take a phenotype set, a genotype set,
//     and a position in the phenotype set and provide
//     a natural language output.
//     Output is sent to the screen or to a file.
//     Optional argument:output: 0=screen by default, 1=file
//     File before these changes in galgo.bk4
//  crg 3/27/94: Reduced code by switching over output streams.

void evolving_world::natural_language_output(
                              dmatrix_2dim* phenotypes_set,
             matrix_2dim* genotypes_set,
             int phenotype_position,
             int output,
             FILE* language)  {
int genotype_location, i;
node *np;
FILE *destination;

//Selects output destination, the screen or a text file
if (output==0)
  destination=stdout;
else if(output==1)
  destination=language;

//Finds location of genotype as the element with best phenotype
genotype_location = phenotypes_set->element(phenotype_position,0);
fprintf(destination,
  "\n\nThe hypothesis has a probabilistic ranking index of %e",
  phenotypes_set->element(phenotype_position,1));

//$$$NEW_CRG_7_29_94
//OPTIONAL printing of the coded genotype
if(option_flag=='2') {   //crg 4/14/94 temporary change...
  fprintf(destination,"\n");
  for (i=0; i<number_of_traits; ++i)
    fprintf(destination,"%d",
            genotypes_set->element(genotype_location,i));
}

//  VARIABLE 1
diagnosed_state = genotypes_set->element(genotype_location,181-1);
if (diagnosed_state!=1) {
  np = bnet->get_member(181);
  fprintf(destination,"\nVariable 1 is diagnosed to be in state ");
  fprintf(destination, "%s.",np->get_state_name(diagnosed_state) );
}

//  VARIABLE 2
diagnosed_state = genotypes_set->element(genotype_location,86-1);
if (diagnosed_state!=1) {
  np = bnet->get_member(86);
  fprintf(destination,"\nVariable 1 is diagnosed to be in state ");
```

```
  fprintf(destination,"%s.", np->get_state_name(diagnosed_state) );
}
}


////////////////////////////////////////////////////////////////
//
//  SAVE EVOLUTION RECORDS
//  Evolution partial results and parameters useful for development
//  are optionally recorded and temporarily stored into several
//  arrays.  This function writes the contents of the arrays into
//  text files for a-posteriori analysis.

void evolving_world::save_evolution_records() {

  char histo_name[13];
  char genot_name[13], pheno_name[13];
  char bestg_name[13], bestp_name[13];
  char topng_name[13], topnp_name[13];

  strcpy(histo_name, "histo000.xls");
  strcpy(pheno_name, "pheno000.xls");
  strcpy(genot_name, "genot000.xls");
  strcpy(bestg_name, "bestg000.xls");
  strcpy(bestp_name, "bestp000.xls");
  strcpy(topng_name, "topng000.xls");
  strcpy(topnp_name, "topnp000.xls");

  histo_name[5]=id1; histo_name[6]=id2; histo_name[7]=id3;
  pheno_name[5]=id1; pheno_name[6]=id2; pheno_name[7]=id3;
  genot_name[5]=id1; genot_name[6]=id2; genot_name[7]=id3;
  bestg_name[5]=id1; bestg_name[6]=id2; bestg_name[7]=id3;
  bestp_name[5]=id1; bestp_name[6]=id2; bestp_name[7]=id3;
  topng_name[5]=id1; topng_name[6]=id2; topng_name[7]=id3;
  topnp_name[5]=id1; topnp_name[6]=id2; topnp_name[7]=id3;


  history->write(histo_name);      //history of evolution
  phenotypes->write(pheno_name);   //phenotypes of final population
  genotypes->write(genot_name);    //genotypes of final population
  best_local_genotypes->write(bestg_name);//best genotype from each gen
  best_local_phenotypes->write(bestp_name);//best phenotype from each
gen
  top_n_genotypes->write(topng_name);//top N genot from best at each gen
  top_n_phenotypes->write(topnp_name);//top N phenot from best at each
gen
}
```

GALGO©: Source code: node_lst.h

```
//////////////////////////////////////////  node_lst.h  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                           as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  NODE LIST


struct node_list_elem {
    node*       data;
    node_list_elem*  next;
};

class node_list {
    node_list_elem* h;
    node_list_elem* iterator;
    node_list_elem* iterator2;
    node_list_elem* tmp;
    node_list_elem* tmp2;
public:
    node_list();
    ~node_list() { node_list_release(); }
    void  add(node* p);
    void  del();
    void remove(node* n);
    node_list_elem* head_pointer() { return (h); }
    node* first_data() { return ( h -> data ); }
    boolean test_empty() { return boolean( h == 0 ); }

    //the following iteration scheme has the advantage of being
    //autonomous when being called, it is not necessary to provide
    //a variable as index; if a nested loop exists, a different
    //iterator should be used
    void start_iteration()  { iterator  = h; }
    void start2_iteration() { iterator2 = h; }
```

```
    node* next() { tmp=iterator; iterator=iterator->next; return tmp-
>data;}
    node* next2(){ tmp2=iterator2;
                   iterator2=iterator2->next;
                   return tmp2->data; }
    int ok()  { return (iterator  != 0); }
    int ok2() { return (iterator2 != 0); }

    void copy_to(node_list* rec);
    void print();
    int count();//counts the number of elements in the list
    void put_into_vector(vect* receiver);
    void node_list_release();
    boolean presence_test(node* n);
};
```

GALGO©: Source code: node_cls.cpp

```
//////////////////////////////////////////  node_lst.cpp  /////
//
//                                  GALGO© Copyright 1992,1993,1994
//                                    Written by Carlos Rojas-Guzman
//                             as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                    Thesis advisor: Mark A. Kramer
//                                                 M.I.T. Feb. 1995
//
//
//  NODE LIST
//
//  Node_list is a singly linked node_list data type.
//  Dynamic self-referential data structure with pointers
//  which refer to elements of its own type.
//  The elements in each node element are pointers to node.

#include "bn.h"

//////////////////////////////////////////////////////////////////////
//
// NODE_LIST is the constructor of the class 'node_list'.

node_list::node_list() {
  //extern long int node_list_counter;
  //++node_list_counter;
  //if ( (node_list_counter%10) == 0 )
  //  cout << "\nNODE_LIST() count = " << node_list_counter;

   //cout << "\nInitializing node_list";
   h = 0;
   //cout << "\nNode list has been initialized";
}

//////////////////////////////////////////////////////////////////////
//
// ADD adds a (node pointer) to the node_list.  It also adds children
// and parents to the respective lists in the node_list  ... verify

void node_list::add(node* p) {
    int i;
```

```
    if (p==0) bnerror("node_list:add: argument is NULL pointer!");
    node_list_elem* temporary = new node_list_elem;
    temporary -> next = h;
    temporary -> data = p;
    h = temporary;

    //cout << "\nAfter add(node* p)";
}

////////////////////////////////////////////////////////////////////////
//
//  DEL
//  Deletes the head element of the node list.  Note that only one
//  element is deleted.
//  Node list elements are destroyed to avoid destroying nodes
//  pointed at by pointers stored inside the node list elements.

void node_list::del() {
  node_list_elem* temporary = h;

  //cout << "\nEntrance to node_list::del() ";
  h = h -> next;
  delete temporary;
  //cout << "\nReturning from node_list::del() ";
}

////////////////////////////////////////////////////////////////////////
//
//  REMOVE is used to remove a node from a node_list.
//  All appearances of node n are removed.
//  Note that the pointers to the node are compared.
//  Two nodes are considered equal if they occupy the same space
//  in memory.

void node_list::remove(node* n) {
  node_list_elem *temporary, *shadow, *removable;
  //removable added 4/24/94

  //bnerror("Monitoring:  Entrance to node_list::remove(node* n)");
  //cout << "\nEntrance to node_list::remove(node* n)";
  //cout << "\nBefore removal";
  //print();
  while ( ( h != 0 ) && (h->data == n) ) {
    //cout << "\nDeleting first element of the list";
    del();
    //cout << "\n h = " << h;
  }
  if (h!=0) {
    temporary = h->next;
    shadow = h;
    while (temporary != 0) {
      if ( (temporary->data) == n) {
```

```
      removable = temporary;
      shadow -> next = temporary -> next;
      temporary = temporary -> next;
      delete removable;
      }
      else {
      shadow = temporary;
      temporary = temporary -> next;
      }
    }
  }
}

////////////////////////////////////////////////////////////////////
//
//  COPY_TO copies a list into the specified list 'rec'.
//  Nodes are not duplicated, pointers are created.

void node_list::copy_to(node_list* rec) {
  node_list mirror;
  node_list_elem* temporary = h;

  while (temporary != 0) {
    mirror.add( temporary -> data );
    temporary = temporary -> next;
  }
  temporary = mirror.head_pointer();
  while (temporary != 0) {
    rec->add( temporary -> data );
    temporary = temporary -> next;
  }
}

////////////////////////////////////////////////////////////////////
//
//  PRINT prints the names of all the elements of the node_list.
//  Any other property of the node objects could be used
//  alternatively as an identifier of the nodes.

void node_list::print() {
    node_list_elem* temporary = h;
    cout << "<head>  ";
    while (temporary != 0) {
        //next line prints (pointer -> name) which is a string
        //cout << form("%s <> ", (temporary -> data) -> get_name());
      //cout <<  (temporary -> data) -> get_name();
        cout << (temporary->data)->get_name() << " <> ";
        temporary = temporary -> next;
      }
    cout << "<tail>\n";
}

////////////////////////////////////////////////////////////////////
```

```
//
//  COUNT

int node_list::count() {
    int counter = 0;
    node_list_elem* temporary = h;

    while (temporary != 0) {
      temporary = temporary -> next;
        ++counter;
    }
    //cout << "\nnode_list contains " << counter << " elements. ";
    return counter;
}

////////////////////////////////////////////////////////////////////
//
//  PUT_INTO_VECTOR
//  The function assumes that the vector has the right size.
//  Elements are copied into the vect starting at element 0.

void node_list::put_into_vector(vect* receiver) {
    int i = 0, list_size = 0, vect_size = 0;
    node* np;
    node_list_elem* temporary = h;

    list_size = this->count();
    //cout << "\nPut_into_vector:list size = " << list_size;
    vect_size = receiver->get_size();
    if (receiver->test_empty() == t)
      vect_size = 0;
    //cout << "\nPut_into_vector:vector size = " << vect_size;
    if (list_size != vect_size)
      bnerror("put_into_vector: failed due to incompatible sizes");

    while (temporary != 0) {
      //cout<<"\nCopying "<<(temporary->data)->get_name();
      //cout<<" into vect.";
        np = temporary->data;
        receiver->element(i) = (long int) np;
      temporary = temporary -> next;
        ++i;
    }
}

////////////////////////////////////////////////////////////////////
//
//  NODE_LIST_RELEASE is the destructor for the node_list class.
//  This function is used for dynamic allocation of free memory.
//  Node_list_release() is used to return used space to free memory.

void node_list::node_list_release() {
  //extern long int node_list_counter_destructor;
```

```
  //++node_list_counter_destructor;
  //if ( (node_list_counter_destructor%100) == 0 ) {
  //  cout << "\nNODE_LIST() destructor = ";
  //  cout << node_list_counter_destructor;
  //}
   while (h != 0) {
     del();
   }
}

////////////////////////////////////////////////////////////////////
//
//  PRESENCE_TESTS returns either a 1 or a 0 depending on whether
//  the node n is a member or not in a node_list.

boolean node_list::presence_test(node* n) {
  node_list_elem* temporary = h;
  while (temporary != 0) {
    if ( (temporary->data) == n)
      return t; //indicates that n exists in the list
    temporary = temporary -> next;
  }
  return f; //if n is not a member of the list returns f=false
}
```

GALGO©: Source code: multidim.h

```
//////////////////////////////////////////// multidim.h /////
//
//                              GALGO© Copyright 1992,1993,1994
//                                Written by Carlos Rojas-Guzman
//                          as part of the following Ph.D. thesis:
//         An Evolutionary Programming Approach to Probabilistic
//             Model-based Fault Diagnosis of Chemical Processes
//                                Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//
//  MULTI-DIMENSIONAL ARRAY class
//

////////////////////////////////////////////////////////////////
//
//  MULTIDIM class
//  N dimensional array which stores floats
//  An arbitrary number of dimensions can be dynamically allocated,
//  with sizes which may be different for each dimension.
//  Implemented to handle 15 dimensions
//  Indices start always with zero
//  Each array has only one identifier (32 bytes)

class multidim {
  float* p1;
  float** p2;
  float*** p3;
  float**** p4;
  float***** p5;
  float****** p6;
  float******* p7;
  float******** p8;
  float********* p9;
  float********** p10;
  float*********** p11;
  float************ p12;
  float************* p13;
  float************** p14;
  float*************** p15;

  int number_of_dimensions;
```

```
  vect* sizes;
  char* name1;

public:
  vect* upper_bounds;
  multidim(int number_of_dimensions,vect* lengths,char*
id1="anonymous");
  float& element(vect* i);
  ~multidim();

  //void print(int level=1);
  //void write(int level=1);  //writes a matrix into a file to be
exported
  //void fill(float filling);
};
```

```
//////////////////////////////////////////  md_alloc.cpp  /////
//
//                              GALGO© Copyright 1992,1993,1994
//                                Written by Carlos Rojas-Guzman
//                          as part of the following Ph.D. thesis:
//         An Evolutionary Programming Approach to Probabilistic
//             Model-based Fault Diagnosis of Chemical Processes
//                                Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//
//  MULTIDIM class
//
//  Multi-dimensional array whose elements are floats.
//  This file contains the constructor function.

#include "bn.h"

////////////////////////////////////////////////////////////////////
//
//  MULTIDIM
//  Constructor for integer N dimensional array.
//  The number of dimensions is the first argument,
//  the size of the vector which carries the sizes of the n
//  dimensions is not used to determine the number of dimensions.
//  The use of float instead of double saves memory.
//  Sizes and upper_bounds start with element 0.
//  Length is copied into sizes to let lengths be assigned
//  anything later
//
//  Sizes of dimensions must be non-negative integers
//  Indices used for access using 'element' use numbers in [1,n]

multidim::multidim(int num_of_dimensions,
       vect* lengths, char* id1) {

  int i,k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,k11,k12,k13,k14;

  number_of_dimensions = num_of_dimensions;
  if ( (name1 = (char *) malloc(64)) == NULL ) {
    bnerror(
     "Not enough memory to allocate buffer for name1 of multidim
array");
  }
  strcpy(name1, id1);

  if (number_of_dimensions < 1)
    bnerror(
     "Attempt to construct multidim array with number of dimensions <
1");

  if (lengths->get_size() < number_of_dimensions)
```

```
    bnerror(
     "Multidim array constructor: size of lengths < num of dimensions");


  for (i=0; i<number_of_dimensions; ++i) {
    if(lengths->element(i)<0) {
      cout << "Error when constructing multidim array object " << name1;
      cerr<<"illegal array size "<<lengths->element(i)
          <<" in dimension "<<i;
      exit(1);
    }
  }

  sizes = new vect(number_of_dimensions, "sizes of multidim array",
id1);
  upper_bounds = new vect(number_of_dimensions,
                          "upper bounds of multidim array",id1);

  for (i=0; i<number_of_dimensions; ++i) {
    sizes->element(i) = lengths->element(i);
    upper_bounds->element(i) = lengths->element(i) - 1;
  }

  switch(number_of_dimensions) {
    case 1:
      p1 = (float*) new float[(int) sizes->element(0)];
      break;

    case 2:
      p2 = (float**) new float*[(int)sizes->element(0)];//cast
unnecessary
      if (p2==0)
        bnerror("Multidim array constructor: insufficient memory");
      for (k1=0; k1 < sizes->element(0); ++k1) // i < s1
  p2[k1] = (float*) new float[(int)sizes->element(1)];// s2
      break;


    case 3:
      p3 = new float**[(int)sizes->element(0)];
      if (p3==0)
        bnerror("Multidim array constructor: insufficient memory");
      for (k1=0; k1 < sizes->element(0); ++k1) {
  p3[k1] = new float*[(int)sizes->element(1)];
  for (k2=0; k2 < sizes->element(1); ++k2) {
    p3[k1][k2] = new float[(int)sizes->element(2)];
        }
      }
      break;

    case 4:
    p4 = new float***[(int)sizes->element(0)];
    if (p4==0)
```

```
      bnerror("Multidim array constructor: insufficient memory");
    for (k1=0; k1 < sizes->element(0); ++k1) {
      p4[k1] = new float**[(int)sizes->element(1)];
      for (k2=0; k2 < sizes->element(1); ++k2) {
  p4[k1][k2] = new float*[(int)sizes->element(2)];
  for (k3=0; k3 < sizes->element(2); ++k3) {
    p4[k1][k2][k3] = new float[(int)sizes->element(3)];
  }
      }
    }
    break;


    case 5:
    p5 = new float****[(int)sizes->element(0)];
    if (p5==0)
      bnerror("Multidim array constructor: insufficient memory");
    for (k1=0; k1 < sizes->element(0); ++k1) {
      p5[k1] = new float***[(int)sizes->element(1)];
      for (k2=0; k2 < sizes->element(1); ++k2) {
  p5[k1][k2] = new float**[(int)sizes->element(2)];
  for (k3=0; k3 < sizes->element(2); ++k3) {
    p5[k1][k2][k3] = new float*[(int)sizes->element(3)];
    for (k4=0; k4 < sizes->element(3); ++k4) {
      p5[k1][k2][k3][k4] = new float[(int)sizes->element(4)];
          }
  }
      }
    }
    break;

case 6:
  p6 = new float*****[(int)sizes->element(0)];
  if (p6==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0; k1 < sizes->element(0); ++k1) {
    p6[k1] = new float****[(int)sizes->element(1)];
    for (k2=0; k2 < sizes->element(1); ++k2) {
      p6[k1][k2] = new float***[(int)sizes->element(2)];
      for (k3=0; k3 < sizes->element(2); ++k3) {
  p6[k1][k2][k3] = new float**[(int)sizes->element(3)];
  for (k4=0; k4 < sizes->element(3); ++k4) {
    p6[k1][k2][k3][k4] = new float*[(int)sizes->element(4)];
    for (k5=0; k5 < sizes->element(4); ++k5) {
      p6[k1][k2][k3][k4][k5] = new float[(int)sizes->element(5)];
          }
        }
      }
    }
  }
  break;
```

```
case 7:
  p7 = new float******[(int)sizes->element(0)];
  if (p7==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0; k1 < sizes->element(0); ++k1) {
    p7[k1] = new float*****[(int)sizes->element(1)];
    for (k2=0; k2 < sizes->element(1); ++k2) {
      p7[k1][k2] = new float****[(int)sizes->element(2)];
      for (k3=0; k3 < sizes->element(2); ++k3) {
  p7[k1][k2][k3] = new float***[(int)sizes->element(3)];
  for (k4=0; k4 < sizes->element(3); ++k4) {
    p7[k1][k2][k3][k4] = new float**[(int)sizes->element(4)];
    for (k5=0; k5 < sizes->element(4); ++k5) {
      p7[k1][k2][k3][k4][k5] = new float*[(int)sizes->element(5)];
      for (k6=0; k6 < sizes->element(5); ++k6) {
        p7[k1][k2][k3][k4][k5][k6] = new float[(int)sizes->element(6)];
            }
          }
        }
      }
    }
  }
  break;


  case 8:
  p8 = new float*******[(int)sizes->element(0)];
  if (p8==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0;k1<sizes->element(0);++k1) {
    p8[k1]=new float******[(int)sizes->element(1)];
    for (k2=0;k2<sizes->element(1);++k2) {
      p8[k1][k2]=new float*****[(int)sizes->element(2)];
      for (k3=0;k3<sizes->element(2);++k3) {
  p8[k1][k2][k3]=new float****[(int)sizes->element(3)];
  for (k4=0;k4<sizes->element(3); ++k4) {
    p8[k1][k2][k3][k4]=new float***[(int)sizes->element(4)];
    for (k5=0;k5<sizes->element(4);++k5) {
      p8[k1][k2][k3][k4][k5]=new float**[(int)sizes->element(5)];
      for (k6=0;k6<sizes->element(5);++k6) {
        p8[k1][k2][k3][k4][k5][k6]=new float*[(int)sizes->element(6)];
        for (k7=0;k7<sizes->element(6);++k7) {
  p8[k1][k2][k3][k4][k5][k6][k7]=new float[(int)sizes->element(7)];
        }
      }
          }
        }
      }
    }
  }
  break;
```

```
  case 9:
  cout << "\nEntrance to case 9"; cout.flush();
  p9 = new float********[(int)sizes->element(0)];
  if (p9==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0;k1<sizes->element(0);++k1) {
    p9[k1]=new float*******[(int)sizes->element(1)];
    for (k2=0;k2<sizes->element(1);++k2) {
      p9[k1][k2]=new float******[(int)sizes->element(2)];
      for (k3=0;k3<sizes->element(2);++k3) {
  p9[k1][k2][k3]=new float*****[(int)sizes->element(3)];
  for (k4=0;k4<sizes->element(3); ++k4) {
    p9[k1][k2][k3][k4]=new float****[(int)sizes->element(4)];
    for (k5=0;k5<sizes->element(4);++k5) {
      p9[k1][k2][k3][k4][k5]=new float***[(int)sizes->element(5)];
      for (k6=0;k6<sizes->element(5);++k6) {
        p9[k1][k2][k3][k4][k5][k6]=new float**[(int)sizes->element(6)];
        for (k7=0;k7<sizes->element(6);++k7) {
    p9[k1][k2][k3][k4][k5][k6][k7]=new float*[(int)sizes->element(7)];
    for (k8=0;k8<sizes->element(7);++k8) {
      p9[k1][k2][k3][k4][k5][k6][k7][k8]=new float[(int)sizes-
>element(8)];
                }
        }
      }
          }
        }
      }
    }
  }
  break;

  case 10:
  cout << "\nEntrance to case 10"; cout.flush();
  p10 = new float*********[(int)sizes->element(0)];
  if (p10==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0;k1<sizes->element(0);++k1) {
    p10[k1]=new float********[(int)sizes->element(1)];
    for (k2=0;k2<sizes->element(1);++k2) {
      p10[k1][k2]=new float*******[(int)sizes->element(2)];
      for (k3=0;k3<sizes->element(2);++k3) {
  p10[k1][k2][k3]=new float******[(int)sizes->element(3)];
  for (k4=0;k4<sizes->element(3); ++k4) {
    p10[k1][k2][k3][k4]=new float*****[(int)sizes->element(4)];
    for (k5=0;k5<sizes->element(4);++k5) {
      p10[k1][k2][k3][k4][k5]=new float****[(int)sizes->element(5)];
      for (k6=0;k6<sizes->element(5);++k6) {
        p10[k1][k2][k3][k4][k5][k6]=new float***[(int)sizes-
>element(6)];
        for (k7=0;k7<sizes->element(6);++k7) {
    p10[k1][k2][k3][k4][k5][k6][k7]=new float**[(int)sizes->element(7)];
    for (k8=0;k8<sizes->element(7);++k8) {
```

```
      p10[k1][k2][k3][k4][k5][k6][k7][k8]=new float*[(int)sizes-
>element(8)];
      for (k9=0;k9<sizes->element(8);++k9) {
        p10[k1][k2][k3][k4][k5][k6][k7][k8][k9]=new float[(int)sizes-
>element(9)];
                    }
          }
        }
      }
          }
        }
      }
    }
  }
  cout << "\nExit form case 10"; cout.flush();
  break;

case 11:
  p11 = new float**********[(int)lengths->element(0)];
  if (p11==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0;k1<sizes->element(0);++k1) {
    p11[k1]=new float*********[(int)sizes->element(1)];
    for (k2=0;k2<sizes->element(1);++k2) {
      p11[k1][k2]=new float********[(int)sizes->element(2)];
      for (k3=0;k3<sizes->element(2);++k3) {
  p11[k1][k2][k3]=new float*******[(int)sizes->element(3)];
  for (k4=0;k4<sizes->element(3); ++k4) {
    p11[k1][k2][k3][k4]=new float******[(int)sizes->element(4)];
    for (k5=0;k5<sizes->element(4);++k5) {
      p11[k1][k2][k3][k4][k5]=new float*****[(int)sizes->element(5)];
      for (k6=0;k6<sizes->element(5);++k6) {
        p11[k1][k2][k3][k4][k5][k6]=new float****[(int)sizes-
>element(6)];
        for (k7=0;k7<sizes->element(6);++k7) {
    p11[k1][k2][k3][k4][k5][k6][k7]=new float***[(int)sizes-
>element(7)];
    for (k8=0;k8<sizes->element(7);++k8) {
      p11[k1][k2][k3][k4][k5][k6][k7][k8] =
          new float**[(int)sizes->element(8)];
      for (k9=0;k9<sizes->element(8);++k9) {
        p11[k1][k2][k3][k4][k5][k6][k7][k8][k9] =
      new float*[(int)sizes->element(9)];
        for (k10=0;k10<sizes->element(9);++k10) {
          p11[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10] =
        new float[(int)sizes->element(10)];
        }
      }
          }
        }
      }
          }
        }
```

```
      }
    }
  }
  break;


  case 12:
  p12 = new float**********[(int)sizes->element(0)];
  if (p12==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0;k1<sizes->element(0);++k1) {
    p12[k1]=new float*********[(int)sizes->element(1)];
    for (k2=0;k2<sizes->element(1);++k2) {
      p12[k1][k2]=new float********[(int)sizes->element(2)];
      for (k3=0;k3<sizes->element(2);++k3) {
  p12[k1][k2][k3]=new float*******[(int)sizes->element(3)];
  for (k4=0;k4<sizes->element(3); ++k4) {
    p12[k1][k2][k3][k4]=new float******[(int)sizes->element(4)];
    for (k5=0;k5<sizes->element(4);++k5) {
      p12[k1][k2][k3][k4][k5]=new float*****[(int)sizes->element(5)];
      for (k6=0;k6<sizes->element(5);++k6) {
        p12[k1][k2][k3][k4][k5][k6]=new float****[(int)sizes-
>element(6)];
        for (k7=0;k7<sizes->element(6);++k7) {
    p12[k1][k2][k3][k4][k5][k6][k7]=new float***[(int)sizes-
>element(7)];
    for (k8=0;k8<sizes->element(7);++k8) {
      p12[k1][k2][k3][k4][k5][k6][k7][k8] =
          new float**[(int)sizes->element(8)];
      for (k9=0;k9<sizes->element(8);++k9) {
        p12[k1][k2][k3][k4][k5][k6][k7][k8][k9] =
      new float*[(int)sizes->element(9)];
        for (k10=0;k10<sizes->element(9);++k10) {
          p12[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10] =
        new float[(int)sizes->element(10)];
          for (k11=0;k11<sizes->element(10);++k11) {
            p12[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11] =
         new float[(int)sizes->element(11)];
          }
                    }
      }
          }
        }
          }
        }
    }
  }
  break;

//$$$NEW
```

```
// this code is commented out because it needs to be restructured before
it
// can be compiled.
/*

  case 13:
  p13 = new float***********[(int)sizes->element(0)];
  if (p13==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0;k1<sizes->element(0);++k1) {
    p13[k1]=new float**********[(int)sizes->element(1)];
    for (k2=0;k2<sizes->element(1);++k2) {
      p13[k1][k2]=new float*********[(int)sizes->element(2)];
      for (k3=0;k3<sizes->element(2);++k3) {
  p13[k1][k2][k3]=new float********[(int)sizes->element(3)];
  for (k4=0;k4<sizes->element(3); ++k4) {
    p13[k1][k2][k3][k4]=new float*******[(int)sizes->element(4)];
    for (k5=0;k5<sizes->element(4);++k5) {
      p13[k1][k2][k3][k4][k5]=new float******[(int)sizes->element(5)];
      for (k6=0;k6<sizes->element(5);++k6) {
        p13[k1][k2][k3][k4][k5][k6]=new float*****[(int)sizes-
>element(6)];
        for (k7=0;k7<sizes->element(6);++k7) {
    p13[k1][k2][k3][k4][k5][k6][k7]=new float****[(int)sizes-
>element(7)];
    for (k8=0;k8<sizes->element(7);++k8) {
      p13[k1][k2][k3][k4][k5][k6][k7][k8] =
          new float***[(int)sizes->element(8)];
      for (k9=0;k9<sizes->element(8);++k9) {
        p13[k1][k2][k3][k4][k5][k6][k7][k8][k9] =
      new float**[(int)sizes->element(9)];
        for (k10=0;k10<sizes->element(9);++k10) {
          p13[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10] =
        new float*[(int)sizes->element(10)];
          for (k11=0;k11<sizes->element(10);++k11) {
            p13[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11] =
          new float[(int)sizes->element(11)];
      for (k12=0;k12<sizes->element(11);++k12) {
              p13[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11][k12] =
            new float[(int)sizes->element(12)];
            }
          }
                  }
      }
          }
        }
          }
        }
          }
        }
    }
  }
  break;
```

```
  case 14:
  cout << "\nEntrance to case 14";
  p14 = new float*************[(int)sizes->element(0)];
  if (p14==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0;k1<sizes->element(0);++k1) {
    p14[k1]=new float************[(int)sizes->element(1)];
    for (k2=0;k2<sizes->element(1);++k2) {
      p14[k1][k2]=new float************[(int)sizes->element(2)];
      for (k3=0;k3<sizes->element(2);++k3) {
  p14[k1][k2][k3]=new float**********[(int)sizes->element(3)];
  for (k4=0;k4<sizes->element(3); ++k4) {
    p14[k1][k2][k3][k4]=new float*********[(int)sizes->element(4)];
    for (k5=0;k5<sizes->element(4);++k5) {
      p14[k1][k2][k3][k4][k5]=new float********[(int)sizes->element(5)];
      for (k6=0;k6<sizes->element(5);++k6) {
        p14[k1][k2][k3][k4][k5][k6]=new float*******[(int)sizes-
>element(6)];
        for (k7=0;k7<sizes->element(6);++k7) {
    p14[k1][k2][k3][k4][k5][k6][k7]=new float******[(int)sizes-
>element(7)];
    for (k8=0;k8<sizes->element(7);++k8) {
      p14[k1][k2][k3][k4][k5][k6][k7][k8] =
          new float*****[(int)sizes->element(8)];
      for (k9=0;k9<sizes->element(8);++k9) {
        p14[k1][k2][k3][k4][k5][k6][k7][k8][k9] =
      new float****[(int)sizes->element(9)];
        for (k10=0;k10<sizes->element(9);++k10) {
          p14[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10] =
        new float***[(int)sizes->element(10)];
          for (k11=0;k11<sizes->element(10);++k11) {
            p14[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11] =
          new float**[(int)sizes->element(11)];
      for (k12=0;k12<sizes->element(11);++k12) {
              p14[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11][k12] =
            new float*[(int)sizes->element(12)];
        for (k13=0;k13<sizes->element(12);++k13) {

p14[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11][k12][k13] =
            new float[(int)sizes->element(13)];
        }
      }
          }
                    }
      }
          }
        }
      }
          }
        }
      }
    }
```

```
  }
  cout << "\nExit from case 14";
  break;


 case 15:
  p15 = new float**************[(int)sizes->element(0)];
  if (p15==0)
    bnerror("Multidim array constructor: insufficient memory");
  for (k1=0;k1<sizes->element(0);++k1) {
    p15[k1]=new float*************[(int)sizes->element(1)];
    for (k2=0;k2<sizes->element(1);++k2) {
      p15[k1][k2]=new float************[(int)sizes->element(2)];
      for (k3=0;k3<sizes->element(2);++k3) {
  p15[k1][k2][k3]=new float***********[(int)sizes->element(3)];
  for (k4=0;k4<sizes->element(3); ++k4) {
    p15[k1][k2][k3][k4]=new float**********[(int)sizes->element(4)];
    for (k5=0;k5<sizes->element(4);++k5) {
      p15[k1][k2][k3][k4][k5]=new float*********[(int)sizes-
>element(5)];
      for (k6=0;k6<sizes->element(5);++k6) {
        p15[k1][k2][k3][k4][k5][k6]=new float********[(int)sizes-
>element(6)];
        for (k7=0;k7<sizes->element(6);++k7) {
    p15[k1][k2][k3][k4][k5][k6][k7]=new float*******[(int)sizes-
>element(7)];
    for (k8=0;k8<sizes->element(7);++k8) {
      p15[k1][k2][k3][k4][k5][k6][k7][k8] =
          new float******[(int)sizes->element(8)];
      for (k9=0;k9<sizes->element(8);++k9) {
        p15[k1][k2][k3][k4][k5][k6][k7][k8][k9] =
      new float*****[(int)sizes->element(9)];
        for (k10=0;k10<sizes->element(9);++k10) {
          p15[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10] =
        new float****[(int)sizes->element(10)];
          for (k11=0;k11<sizes->element(10);++k11) {
            p15[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11] =
          new float***[(int)sizes->element(11)];
      for (k12=0;k12<sizes->element(11);++k12) {
              p15[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11][k12] =
            new float**[(int)sizes->element(12)];
        for (k13=0;k13<sizes->element(12);++k13) {

p15[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11][k12][k13] =
            new float*[(int)sizes->element(13)];
          for (k14=0;k14<sizes->element(13);++k14) {

p15[k1][k2][k3][k4][k5][k6][k7][k8][k9][k10][k11][k12][k13][k14] =
            new float[(int)sizes->element(14)];
          }
        }
      }
          }
```

```
                    }
        }
            }
          }
        }
            }
          }
        }
      }
    }
    break;
*/
//$$$END

    default: bnerror("Unavailable multidim array object size");
    }
}
```

GALGO©: Source code: md_elem.cpp

```
//////////////////////////////////////////  md_elem.cpp  /////
//
//
//                                  GALGO© Copyright 1992,1993,1994
//                                    Written by Carlos Rojas-Guzman
//                             as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                    Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  MULTIDIM class
//  Multi-dimensional array whose elements are floats.
//  This file contains the element access function used to
//  store and retrieve elements from multidimensional arrays.

#include "bn.h"

////////////////////////////////////////////////////////////
//
//  ELEMENT
//

float& multidim::element(vect* i) {
  int k;

  //cout << "\nNumber of dimensions = " << number_of_dimensions;
  cout.flush();

  if (i->get_size() < number_of_dimensions)
    bnerror(
     "Multidim::element: arg list smaller than number of dimensions");

  for (k=0; k<number_of_dimensions; ++k) {
    if (i->element(k)<0 || i->element(k) > upper_bounds->element(k)){
      cout<<"\nERROR in multidim::element in object "<<name1;
      cout<<"\nillegal index: "
          <<i->element(k)<<" in dimension[1,n]: "<<(k+1);
      bnerror(
        "illegal index in multidimensional array::element access");
    }
```

```
  }
  switch(number_of_dimensions) {
  case 1:
    return p1[(int)i->element(0)];
    //break;  //unnecessary since return precedes

  case 2:
    //cout << "\nCase 2 in element";
    return p2[(int)i->element(0)][(int)i->element(1)];
    //break;

  case 3:
    return p3[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)];
    //break;

  case 4:
    return p4[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
              [(int)i->element(3)];
    //break;

  case 5:
    return p5[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
              [(int)i->element(3)][(int)i->element(4)];
    //break;

  case 6:
    return p6[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
              [(int)i->element(3)][(int)i->element(4)][(int)i-
>element(5)];
    //break;

  case 7:
    return p7[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
        [(int)i->element(3)][(int)i->element(4)][(int)i->element(5)]
        [(int)i->element(6)];
    //break;

  case 8:
    return p8[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
        [(int)i->element(3)][(int)i->element(4)][(int)i->element(5)]
        [(int)i->element(6)][(int)i->element(7)];
    //break;

  case 9:
    return p9[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
        [(int)i->element(3)][(int)i->element(4)][(int)i->element(5)]
```

```
      [(int)i->element(6)][(int)i->element(7)][(int)i->element(8)];
    //break;

  case 10:
    return p10[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
        [(int)i->element(3)][(int)i->element(4)][(int)i->element(5)]
        [(int)i->element(6)][(int)i->element(7)][(int)i->element(8)]
        [(int)i->element(9)];
    //break;

  case 11:
    return p11[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
        [(int)i->element(3)][(int)i->element(4)][(int)i->element(5)]
        [(int)i->element(6)][(int)i->element(7)][(int)i->element(8)]
        [(int)i->element(9)][(int)i->element(10)];
    //break;

  case 12:
    return p12[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
        [(int)i->element(3)][(int)i->element(4)][(int)i->element(5)]
        [(int)i->element(6)][(int)i->element(7)][(int)i->element(8)]
        [(int)i->element(9)][(int)i->element(10)][(int)i->element(11)];
    //break;

  case 13:
    return p13[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
        [(int)i->element(3)][(int)i->element(4)][(int)i->element(5)]
        [(int)i->element(6)][(int)i->element(7)][(int)i->element(8)]
        [(int)i->element(9)][(int)i->element(10)][(int)i->element(11)]
        [(int)i->element(12)];
    //break;

  case 14:
    return p14[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
        [(int)i->element(3)][(int)i->element(4)][(int)i->element(5)]
        [(int)i->element(6)][(int)i->element(7)][(int)i->element(8)]
        [(int)i->element(9)][(int)i->element(10)][(int)i->element(11)]
        [(int)i->element(12)][(int)i->element(13)];
    //break;

  case 15:
    return p15[(int)i->element(0)][(int)i->element(1)][(int)i-
>element(2)]
        [(int)i->element(3)][(int)i->element(4)][(int)i->element(5)]
        [(int)i->element(6)][(int)i->element(7)][(int)i->element(8)]
        [(int)i->element(9)][(int)i->element(10)][(int)i->element(11)]
        [(int)i->element(12)][(int)i->element(13)][(int)i->element(14)];
    //break;
```

```
  default:
    bnerror("Unavailable multidim array object size");
//$$$NEW
//  the compiler requires a valid return
    return p1[(int)i->element(0)];    // needed to return something
//$$$END
  }
}

////////////////////////////////////////////////////////////
//
//  DESTRUCTOR
//  One destructor for each size of array

multidim::~multidim() {
/*
  for (int i=0; i<= ub1; ++i)
    delete p[i];
  delete p;
*/
}
```

GALGO©: Source code: vect.h

```
//////////////////////////////////////////////  vect.h  /////
//
//                              GALGO© Copyright 1992,1993,1994
//                                Written by Carlos Rojas-Guzman
//                          as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//  VECT.H
//  For all vector classes:  index in [0,size-1].
//  A description for each function is included in vect.cpp with
//  each function definition.

////////////////////////////////////////////////////////////////
//
//  VECT class
//

class vect {
   long int* p;
   int size;
   char* name1;
   char* name2;
   boolean empty;
public:
   vect(); //signals error only for n<0; n=0 sets 1 element to -1
   vect(int n, char* id1="anonymous", char* id2="anonymous");
   void reset(int n, char* id1="anonymous", char* id2="anonymous");
   ~vect() {
     //extern long int delete_vect;
     //extern long int count_vect_arg;
     //++delete_vect;
     //if ( (delete_vect%5000) == 0 )
     //  cout<<"\n~vect() count="<<delete_vect<<count_vect_arg;
   #if defined DEVELOPMENT
     delete name1;
     delete name2;
   #endif
     delete p;
```

```
    }

    long int& element(int i);
    int ub;  // upper bound = size - 1
    void set_size_index(int s);//doesn't change the size,only the index
    int get_size();
    void set_name(char* s1,char* s2) {name1 = s1; name2=s2;}
    char* get_name(int i=1);
    boolean test_empty() { return empty; }
    void print(int format=1);
    void write(char* file_name="vect_out.txt");
    void start(int& i) const { i = 0; }
    int ok(int& i)    const { return i<=ub; }
    long int next(int& i) const { return p[i++]; }
    int presence_test(long int);//returns index if found, -1 otherwise
    void fill(long int filling);
    long int normalize(long int total=1);
};


////////////////////////////////////////////////////////////////////
//
//  DVECTOR class
//

class dvect {
    double* p;
    int size;
    char* name1;
    char* name2;
    boolean empty;
public:
    int ub;//upper bound = size - 1
    dvect() {
      cerr<<("Warning: dvect:dvect() used");//remove if necessary
      extern long int dvect_count;
      cout << "\nDVECT::DVECT() used " << ++dvect_count;
      size = 2;
      p = new double[size];
      ub = size - 1;
      if ( ( name1 = new char[32]) == NULL ) {
        cerr<<
          "Not enough memory to allocate buffer for name1 of dvect()";
      }
      strcpy(name1, "anonymous");

      if ( ( name2 = new char[32]) == NULL ) {
        cerr<<
          "Not enough memory to allocate buffer for name2 of dvect()";
      }
      strcpy(name2, "anonymous");
    }
```

```
   dvect(int n, char* id1="anonymous",char* id2="anonymous");
   void reset(int n, char* id1="anonymous",char* id2="anonymous");
   //dvect(double* p, int size); //define this one later
   ~dvect() {delete p; delete name1; delete name2; }
   double& element(int  i);
   int get_size();
   boolean test_empty() { return empty; }
   void set_names(char* n1, char* n2) { name1 = n1;  name2 = n2;}
   char* get_name(int i=1);
   void print();
   void write(char* file_name="dvec_out.txt");
   void fill(double filling);
   double normalize(double total=1.0);
};

//////////////////////////////////////////////////////////////////////
//
//  DOUBLE MALLOC VECTOR function
//

double *mdvector(int nl,int nh);

//////////////////////////////////////////////////////////////////////
//
//  INT MALLOC VECTOR function
//

int *mivector(int nl,int nh);

//////////////////////////////////////////////////////////////END OF FILE
```

GALGO©: Source code: vect.cpp

```
//////////////////////////////////////////////  vect.cpp  /////
//
//                                  GALGO© Copyright 1992,1993,1994
//                                    Written by Carlos Rojas-Guzman
//                            as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                    Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  VECT and DVECT classes
//  These two vector classes are one-dimensional arrays.
//  VECT can store long integers and DVECT doubles.
//  The VECT class uses long int (in order to be able to handle any
//  pointer size within the Borland C++ or Microsoft C++ compilers).
//  Memory is dynamically allocated for both classes.
//  This file includes also memory allocation functions using malloc
//  for double vectors from nrutil.c
//  Future extensions: A class called pvect will handle void*.

#include "bn.h"

////////////////////////////////////////////////////////////////////////
//
//  VECT
//  Overloaded constructor.
//  Vect() is the default constructor for the class vect.
//  Memory is allocated for vector identifiers.
//  If operator new() cannot allocate storage it will return 0
//  according to B. Stroustrup, the C++ Programming Language pp.577
//  Instead of using this default constructor it is recommended to
//  use the overloaded version which takes arguments.  The reason is
//  the reduction in the number of function calls required to create
//  a vector.
//  The default size is 2 elements.

vect::vect() {
  bnerror("Warning: vect::vect() used");

  size = 2;
  p = new long int[size];
```

```
  ub = size - 1;

  //cout << "\nConstructor vect() creating a default size vector";
  if ( ( name1 = new char[32]) == NULL ) {
    bnerror(
      "Not enough memory to allocate buffer for name1 of vect()");
  }
  strcpy(name1, "anonymous");
  if ( ( name2 = new char[32]) == NULL ) {
    bnerror(
      "Not enough memory to allocate buffer for name2 of vect()");
  }
  strcpy(name2, "anonymous");
  empty = t;
}

////////////////////////////////////////////////////////////////////////
//
//  VECT
//  Overloaded constructor
//  Allows dynamically allocated arrays and can create 0-size vectors.
//  crg 4/24/94: Suggestion: Make constructor an inline function given
//               the frequency with which it is used.  Running time is
//               expected to decrease.
//  crg 4/25/94: Memory allocation for names is compiled only for
//               development purposes.

vect::vect(int n, char* id1, char* id2) {

#if defined DEVELOPMENT
  if ( ( name1 = new char[32]) == NULL ) {
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    bnerror(
      "Not enough memory to allocate buffer for name1 vect(n,n1,n2)");
  }
  strcpy(name1, id1);

  if ( ( name2 = new char[32]) == NULL ) {
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    bnerror(
      "Not enough memory to allocate buffer for name2 vect(n,n1,n2)");
  }
  strcpy(name2, id2);
 #endif

  empty = t;
  if (n < 0) {
    cerr << "illegal vect size "<< n <<" in "
         << name1 << " " << name2 << "\n";
    exit(1);
  }
```

```
  if (n > 0) {
    //cout << "\n---Construction of a size " << n << " vector: ";
    //cout << name1 << " " << name2;
    size = n;
    empty=f;
    if (  (p = new long int[size]) == NULL ) {
      cout << "\nName1 = " << id1 << "\n";
      cout << "\nName2 = " << id2 << "\n";
      bnerror(
        "Not enough memory to allocate buffer for vect elements");
    }
    ub = size - 1;
  }
  if (n == 0) {
    //cout << "\n-*-Construction of a size 0 vector: ";
    //cout << name1 << " " << name2;
    size = 1;
    empty=t;
    if (  (p = new long int[size]) == NULL ) {
      cout << "\nName1 = " << id1 << "\n";
      cout << "\nName2 = " << id2 << "\n";
      bnerror(
        "Not enough memory to allocate buffer for vect elements");
    }
    ub = size - 1;
    p[0]= -1;
  }
}

////////////////////////////////////////////////////////////////////////
//
//  RESET same function as vect(), with a different name to initialize
//  vect objects at any time.  It will be necessary to verify whether
//  the construction of these objects does not disable the destructor.
//  RESET does not allocate memory, memory allocation can be
//  accomplished by declaring an identifier as a vect:
//       vect myvect;
//  or by using new:
//       myvect = new vect(..here any arguments for initialization)
//  When only name values are changed (as in name1) it is unnecessary
//  to free and allocate memory again.
//  crg 4/14/94: as suspected, objects reconstruction with reset
//       interferes with the normal operation of the destructor. It
//       seems that the pointer has to be the one initially created,
//       even though it is part of the object.
//  crg 4/94: Recommendation: do not use the vect->reset function.
//       Use a declaration or new with parameters.

void vect::reset(int n, char* id1, char* id2) {
  bnerror("vect::reset is not an acceptable practice anymore");
  cout << "\nResetting vector " << id1 << " " << id2;
  strcpy(name1, id1);
  strcpy(name2, id2);
```

```
  empty = t;

  delete p;//crg 4/14/94 substitutes the use of free

  //new memory allocation according to new object dimensions
  if (n < 0) {
    cerr << "illegal vect size "<<n<<" while resetting "
         << name1 << " " << name2 << "\n";
    exit(1);
  }
  if (n > 0) {
    //cout << "\n---Construction of a size " << n << " vector: ";
    //cout << name1 << " " << name2;
    size = n;
    empty=f;
    if ( (p = new long int[size]) == NULL) {
      cout << "\nName1 = " << name1;
      cout << "\nName2 = " << name2;
      bnerror(
        "Not enough memory to allocate buffer for vect elements");
    }
    ub = size - 1;
  }
  if (n == 0) {
    //cout << "\n-*-Construction of a size 0 vector: ";
    //cout << name1 << " " << name2;
    size = 1;
    empty = t;
    if ( (p = new long int[size]) == NULL) {
      cout << "\nName1 = " << name1;
      cout << "\nName2 = " << name2;
      bnerror(
        "Not enough space to allocate buffer for vector elements");
    }
    ub = size - 1;
    p[0]= -1;
  }
}

////////////////////////////////////////////////////////////////////////
//
//  ELEMENT
//  Retrieves an element of the vector object.
//  crg 10/24/93: The argument to vect::element is in [0,n-1]

long int& vect::element(int i) {
  if (i < 0 || i > ub) {
     cerr << "\nillegal vect index " << i << " in object: "
          << name1 << " " << name2 << "\n";
     exit(1);
   }
   return (p[i]);
}
```

```
////////////////////////////////////////////////////////////////////
//
//  GET_SIZE
//  Retrieves the size attribute of the vector object.

int vect::get_size() {
  return size;
}

////////////////////////////////////////////////////////////////////
//
//  GET_SIZE
//  This function changes the index of the vector.
//  It is used by presence_test when the vector is not full and
//  only the first s elements are to be compared.
//  A size value can be assigned only if it is smaller than the real
//  allocated size (to avoid impossible retrievals).

void vect::set_size_index(int s) {
  if (s<=size)
    size = s;
  else
    bnerror("Invalid attempt to increase size index in vect object");
}

////////////////////////////////////////////////////////////////////
//
//  PRINT
//  Prints the contents of a vector object to the screen.

void vect::print(int format) {
  int i;
  if (empty == t) {
    cout << "\nThe vector " << name1 << " " << name2 << " is empty.";
    return;
  }
  if (format==1) {
    //cout << "\nPrinting a vector with format 1";
    cout << "\nInteger vector " << name1 << " " << name2 << " :";
    for (i=0; i<size; ++i) {
      cout << "\nelement(" << i << ") = " << element(i);
      }
    cout << "\n";
  }
  //format 2 does not print element 0,
  //it prints horizontally as [1, 32, 15]
  else if (format==2) {
    //cout << "\nPrinting a vector with format 2";
    cout << "\nInteger vector " << name1 << " " << name2 << " :";
    cout << "\n  [" << element(1);
    for (i=2; i<size; ++i)
      cout << ", " << element(i);
```

```
    cout << "] \n";
  }
  //format 3 prints all elements including 0,
  //horizontally as [5,3,2,7,22]
  else if (format==3) {
    //cout << "\nPrinting a vector with format 3";
    cout << "[" << element(0);
    for (i=1; i<size; ++i)
      cout << ", " << element(i);
    cout << "]";
  }
}

////////////////////////////////////////////////////////////////////////
//
//  WRITE
//  Writes the contents of a vector object to a text file.

void vect::write(char* file_name) {
  int i;
  FILE *ofp;

  if ( (ofp=fopen(file_name,"w+"))==NULL )
    cout << "Output file " << file_name
         << " for writing vect not opened\n";
  else
    cout << "Output file " << file_name
         << " opened for writing vect\n";

  cout << "\nLong int vector: "
       << name1 << " " << name2 << " contents: ";

  for (i=1; i<=size; ++i) {
    fprintf(ofp,"\ne(%d) = %d", i, element(i-1));
    //cout << "\nelement(" << i << ") = " << element(i-1);
  }
  fprintf(ofp, "\n");
  fclose(ofp);
}

////////////////////////////////////////////////////////////////////////
//
//  PRESENCE_TEST
//  Tests for the presence of a long integer element in the vector.
//  Returns index if found, -1 otherwise.

int vect::presence_test(long int x) {
  if (empty == t)
    return (-1);
  else {
    for (int index=0; index<size; ++index) {
      if (element(index) == x)
      return index;
```

```
    }
    return (-1);
  }
}

////////////////////////////////////////////////////////////////////
//
//  GET_NAME
//  i=1 by default
//  crg 1/14/93: Temporarily modified.
//  crg 3/11/93: Dummy allocated.
//  crg 4/27/94: Consider removing the dummy leaving only bnerror.

char* vect::get_name(int i) {
  char* dummy;

  if (i==1) return name1;
  if (i==2) return name2;
  else {
    if ( (dummy = (char *) malloc(4)) == NULL ) {
      bnerror(
        "Not enough memory to allocate buffer for dummy of vect");
    }
    strcpy(dummy, "dummy");
    bnerror("Wrong argument in vect::get_name");
    return dummy;
  }
}

////////////////////////////////////////////////////////////////////
//
//  FILL
//  Fills a vect object with the long int specified as the argument.
//  This function ensures the contents of the data structure are not
//  random values and serves as an initialization procedure.

void vect::fill(long int filling) {
  int i;

  if (empty == f) {
    for(i=0;i<size;++i) {
      element(i)=filling;
    }
  }
  else
    cout << "\nvect::fill:attempted fill on empty vect"
         << name1 << " " << name2;
}

////////////////////////////////////////////////////////////////////
//
//  NORMALIZE
//  Normalizes the elements of the vector to have a sum equal to TOTAL
```

```
//  TOTAL =1 by default.
//  Returns the sum of all elements originally present in the vector.
//  An empty vector is not normalized (an empty vector has one element,
//  which contains -1 and its property 'empty' is true).

long int vect::normalize(long int total) {
  int i;
  long int sum=0;

  if (empty == f) {
    for (i=0; i<size; ++i)
      sum += element(i);
    for (i=0; i<size; ++i)
      element(i) = element(i) * total / sum;
  }
  else
    cout << "\nvect::normalize: attempted on empty vector"
         << name1 << " " << name2;
  return sum;
}


////////////////////////////////////////////////////////////////////////
//
//   DVECTOR class
//   Allows dynamically allocated arrays and can create 0 size vectors

dvect::dvect(int n, char* id1, char* id2 ) {
  //extern long int dvect_count;
  //cout << "\nDVECT::DVECT(n,id1,id2) used "<<++dvect_count;

  if ( ( name1 = new char[32]) == NULL ) {
    bnerror("Not enough memory to allocate buffer for name1 of
dvect()");
  }
  strcpy(name1, "anonymous");

  if ( ( name2 = new char[32]) == NULL ) {
    bnerror("Not enough memory to allocate buffer for name2 of
dvect()");
  }
  strcpy(name2, "anonymous");

   if (n < 0) {
     cerr << "illegal vect size " << n << "in "
          << name1 << " " << name2 << "\n";
     exit(1);
   }
   if (n > 0) {
     //cout << "\n---Construction of a size " <<n << " dvect: ";
     //cout << name1 << " " << name2;
     size = n;
     empty=f;
```

```
    p = new double[size];
    ub = size - 1;
  }
  if (n == 0) {
    //cout << "\n-*-Construction of a size 0 dvect: ";
    //cout << name1 << " " << name2;
    size = 1;
    empty=t;
    p = new double[size];
    ub = size - 1;
    p[0]= -1.0;
  }
}

////////////////////////////////////////////////////////////////////
//
//  RESET
//  Modifies the properties of the object, but is not a constructor.
//  Allows dynamically allocated arrays and can create 0-size vectors
//  crg 3/11/93:  Before resetting, pointers should be deleted.
//  crg 4/25/94:  It is recommended not to use this function due to
//                interference with the destructor function.  Instead,
//                use the constructor with new and parameters.

void dvect::reset(int n, char* id1, char* id2) {
  bnerror("dvect::reset(n,id1,id2) is not acceptable anymore");
  //extern long int dvect_res_count;
  //cout << "\nDVECT::RESET(n,id1,id2) used" << ++dvect_res_count;
  strcpy(name1, id1);
  strcpy(name2, id2);

  delete p; //4/16/94
  if (n < 0) {
    cerr << "illegal vect size " << n << "in "
         << name1 << " "<<name2<<"\n";
    exit(1);
  }
  if (n > 0) {
    //cout << "\n---Construction of a size " << n << " dvect: ";
    //cout << name1 << " " << name2;
    size = n;
    empty=f;
    p = new double[size];
    ub = size - 1;
  }
  if (n == 0) {
    //cout << "\n-*-Construction of a size 0 dvect: ";
    //cout << name1 << " " << name2;
    size = 1;
    empty=t;
    p = new double[size];
    ub = size - 1;
    p[0]= -1.0;
```

```
    }
}

////////////////////////////////////////////////////////////////////////
//
//  ELEMENT
//  Retrieves an element of the dvect object.

double& dvect::element(int i) {
  if (i < 0 || i > ub) {
     cerr << "illegal dvect index " << i << " in object: "
          << name1 << " " << name2 << "\n";
     exit(1);
    }
    return (p[i]);
}

////////////////////////////////////////////////////////////////////////
//
//  GET_SIZE
//  Returns the size of a dvect object (the number of elements).

int dvect::get_size() {
  return(size);
}

////////////////////////////////////////////////////////////////////////
//
//  PRINT
//  Prints the contents of a dvect object to the screen.

void dvect::print() {
  int i;
  cout << "\nDouble vector: " << name1 << " " << name2 << ": ";
  for (i=1; i<=size; ++i) {
      //printf("\ne(%d) = %f",i,element(i-1));
      cout << "\nelement(" << i << ") = " << element(i-1);
    }
      //printf("\n");
      cout << "\n";
}

////////////////////////////////////////////////////////////////////////
//
//  WRITE
//  Writes the contents of a dvect object to a text file.

void dvect::write(char* file_name) {
  int i;
  FILE *ofp;

  if ( (ofp=fopen(file_name,"w+"))==NULL )
    cout << "Output file " << file_name
```

```
        << " for writing dvect not opened\n";
  else
    cout << "Output file " << file_name
         << " opened for writing dvect\n";

  cout << "\nDouble vector: " << name1 << " "
       << name2 << " contents: ";

  for (i=1; i<=size; ++i) {
    fprintf(ofp,"\ne(%d) = %f", i, element(i-1));
    //cout << "\nelement(" << i << ") = " << element(i-1);
  }
  fprintf(ofp, "\n");
  fclose(ofp);
  //cout << "\n";
}

////////////////////////////////////////////////////////////////////////
//
//  GET_NAME
//  Returns one of the names for the dvect object.

char* dvect::get_name(int i) {


  char* dummy;

  //char* dummy=0;
  if (i==1) return name1;
  if (i==2) return name2;
  else {
    if ( (dummy = (char *) malloc(4)) == NULL ) {
      bnerror(
        "Not enough memory to allocate buffer for dummy of vect");
    }
    strcpy(dummy, "dummy");
    bnerror("Wrong argument in vect::get_name");
    return dummy;
  }
}

////////////////////////////////////////////////////////////////////////
//
//  FILL
//  Fills the dvect object with the value specified as argument.

void dvect::fill(double filling) {
  int i;

  if (empty == f) {
    for(i=0; i<size; ++i) {
      element(i)=filling;
    }
```

```
  }
  else
    cout << "\ndvect::fill: attempted on empty dvector "
         << name1 << " " << name2;
}

////////////////////////////////////////////////////////////////////
//
//  NORMALIZE
//  Normalizes the elements of the vector to have a sum equal to TOTAL
//  TOTAL = 1 by default.
//  Returns the sum of all the elements originally present in the vect
//  An empty vector is not normalized (an empty vector has one element,
//  which contains -1 and its property 'empty' is true).

double dvect::normalize(double total) {
  int i;
  double sum=0.0;

  if (empty == f) {
    for (i=0; i<size; ++i)
      sum += element(i);
    for (i=0; i<size; ++i)
      element(i) = element(i) * total / sum;
  }
  else
    cout << "\ndvect::normalize: attempted on empty vector"
       << name1 << " " << name2;
  return sum;
}

////////////////////////////////////////////////////////////////////
//
//  MALLOC DOUBLE VECTOR
//  Allocate a double vector with subscript range v[nl..nh]
//  From nrutil.c of numerical recipes

double *mdvector(int nl,int nh)  {
      double *v;
        v=(double *)malloc((unsigned) (nh-nl+1)*sizeof(double))-nl;
        if (!v) bnerror("allocation failure in dvector()");
        return v;
}

////////////////////////////////////////////////////////////////////
//
//  MALLOC INT VECTOR
//  Allocate an int vector with subscript range v[nl..nh]
//  From nrutil.c of numerical recipes

int *mivector(int nl,int nh)  {
      int *v;
        v=(int *)malloc((unsigned) (nh-nl+1)*sizeof(int))-nl;
```

```
        if (!v) bnerror("allocation failure in ivector()");
        return v;
}

//////////////////////////////////////////////////////////////////////
//
//  TESTS AND ILLUSTRATION
//

/*
main() {
dvect a,b;

a.dvect(4);
b.dvect(6);
a.element(0)=0;
b.element(0)=0;
a.element(3)=2.0/3.0;
b.element(2)=5.0/3;
printf("\n%f",a.element(3));
printf("\n%f",b.element(2));
a.print();
b.print();
}

The vector must be declared before used.
Indices are valid from 0 to size-1.
The size of the vector can be changed in the middle of the program.
When the size is changed, old information is lost.
*/
```

GALGO©: Source code: metric.h

```
//////////////////////////////////////////////  metric.h  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                           as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  METRIC.H


long double count_possible_states(belief_network* network);

double calculate_metric(belief_network* network, vect* arg_list);
```

GALGO©: Source code: metric.cpp

```
//////////////////////////////////////////////  metric.cpp  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                             as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  Auxiliary functions to (1) count the number of possible
//  states in the Bayesian network (number of possible combinations
//  of variable states) and to (2) calculate the metric (the
//  probability of any given system state (or diagnostic hypothesis).
//
//  crg 3/29/94: Count_possible_states and calculate_metric are
//  kept in this module.

#include "bn.h"

////////////////////////////////////////////////////////////////////
//
//  COUNT_POSSIBLE_STATES
//  Auxiliary function to count the number of possible
//  states in the Bayesian network (number of possible combinations
//  of variable states).  This calculation provides the basis for
//  a decision regarding the algorithm to be used.  Since the
//  number of states in the system shows combinatorial explosion,
//  some algorithms will not be feasible.
//
//  s does not require memory allocation

long double count_possible_states(belief_network* network) {
  int number_of_nodes, number_of_states;
  long double product=1.0;
  int var;
  vect* all_nodes;
  char* s;
  node* np;

  //cout << "\n\nCounting possible system states: ";
```

```
  number_of_nodes = network -> get_size();
  all_nodes = network -> get_members();
  all_nodes->start(var);
     while (all_nodes->ok(var)) {
       np = (node*) all_nodes->next(var);
       number_of_states = np -> get_number_of_states();
       s = np -> get_name();
       //cout <<"\nNode "<< s <<" can assume " << number_of_states;
       //cout << " discrete states.";
       product *= number_of_states;
       //cout << "\nAccumulated product is " << product;
     }
  cout <<"\nThe number of possible states of the system is ";
  cout <<product<<".";
  network->set_counted();//to avoid redundant calculations
  //cout << "\nExit from Count_possible_states"; cout.flush();
  return product;
}


////////////////////////////////////////////////////////////
//
//  CALCULATE_METRIC
//
//  Calculates the metric (probability) corresponding to any given
//  system state.  A 'system state' is completely specified by
//  indicating the value of each of the variables in the system.
//  The probability of a system state is the product of N terms,
//  where N is the number of nodes in the network.  Each term is
//  either a prior probability (for root nodes) or a conditional
//  probability (for leaf and internal nodes).  Prior probabilities
//  are only a function of the state of the variable, while
//  conditional probabilities depend on the state of the variable
//  and on the states of the its parents.


double calculate_metric(belief_network* network, vect* arg_list) {
  int var, size, parents_num, r;
  double relative_metric = 1.0, factor;
  //vect *nodes_set, prob_arg; //commented on 4/14/94
  vect *nodes_set, *prob_arg;  //added on 4/14/94
  node *np, *parent;
  //char *input; seems unnecessary, commented 3/11/93
  //cout << "\nEntering calculate_metric";
  nodes_set = network->get_members();
  size = network->get_size();

  //cout<<"\nSetting states according to arg_list:";
  //cout<<" one of the system states";
  for (int counter=0; counter<size; ++counter) {
    np = (node*) nodes_set->element(counter);
    np->set_state(arg_list->element(counter));
  }
```

```
  //cout << "\nObtaining the conditional probabilities for each node";
  for(int k=0; k<size; ++k) {
    //cout << "\nInside for loop of cond prob";
    np = (node*) nodes_set->element(k);
    //cout << "\nFinding cond prob of node " << np->get_name();
    parents_num = np->get_number_of_parents();
    //cout << "\nNumber of parents for this node is "<<parents_num;
    prob_arg = new vect(parents_num+2,"prob_arg","calculate_metric");
    //cout << "\nSize of prob_arg is: " << prob_arg.get_size();
    prob_arg->element(0) = parents_num + 1;
    prob_arg->element(1) = np->get_state();  // P(n|p1,p2,...)
    //cout << "\nState of focused node is " << prob_arg.element(1);
    for( r=1; r<=parents_num; ++r) {
      parent = np->get_parent(r); //later merge with next lines...
      //cout << "\nParent  " << r << " is " << parent->get_name();
      //prob_arg->element(r) = arg_list->element(i?);//out
      prob_arg->element(r+1) = parent->get_state();
      //cout<<"\nState of parent number "<<r<<" is " ;
      //cout<< prob_arg.element(r+1);
    }
  //cout << "\nBefore getting probability";
  //cout << "\nThe probability argument is ";
  //prob_arg.print();
  factor = np->get_probability(prob_arg);
  //cout << "\nFactor (from get_prob) = " << factor;
  relative_metric *= factor;
  //cout << "\nAccumulated product = " << relative_metric;
  delete prob_arg;//deletes the vect object(one created for each node).
  }
  //cout<<"\nRelative metric is = "<<relative_metric;//temporarily off
  return relative_metric;
}
```

GALGO©: Source code: mat_2d.h

```
//////////////////////////////////////////////  mat_2d.h  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                            as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  MATRIX_2DIM class and DMATRIX_2DIM class
//  Two versions exist, one for int, and one for doubles.
//  Both are dynamically allocated.

////////////////////////////////////////////////////////////////////
//
//  MATRIX_2DIM
//  Two dimensional matrix which stores integers

class matrix_2dim {
   long int** p;
   int   s1, s2;
   char* name1;
   char* name2;
public:
   int ub1,ub2;
   matrix_2dim(char* id1="anonymous", char* id2="anonymous");
   matrix_2dim(int d1, int d2,
               char* id1="anonymous", char* id2="anonymous");
   void  reset(int d1, int d2,
               char* id1="anonymous", char* id2="anonymous");
   ~matrix_2dim();
   long int& element(int i, int j);
   void print(int level=1);
   void write(char* file_name="history.xls", int level=1);
   void fill(long int filling);
};

////////////////////////////////////////////////////////////////////
//
//  DMATRIX_2DIM
```

```
//  Two dimensional matrix which stores doubles.

class dmatrix_2dim {
  double **p;               //to handle large arrays
  int s1, s2;
  char* name1;
  char* name2;
public:
  int ub1,ub2;
  dmatrix_2dim(char* id1="anonymous", char* id2="anonymous");
  dmatrix_2dim(int d1, int d2,
               char* id1="anonymous", char* id2="anonymous");
  void reset(int d1, int d2,
             char* id1="anonymous", char* id2="anonymous");
  ~dmatrix_2dim();
  double& element(int i, int j);
  void print(int level=1);
  double** get_pointer() { return p; }
  //writes a matrix into a file to be exported
  void write(char* file_name="history.xls", int level=1);
  void fill(double filling);
};

////////////////////////////////////////////////////////////////
//
//  PMATRIX_2DIM
//  Two dimensional matrix which stores pointers to void
//  This class can be used for lambda_tie and pi_tie.
//  To be implemented if necessary.
```

GALGO©: Source code: mat_2d.cpp

```
//////////////////////////////////////////  mat_2d.cpp  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                             as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  MAT_2DIM class
//  TWO DIMENSIONAL MATRIX CLASS
//
//  Two dimensional arrays which store integers (matrix_2dim) or
//  doubles (dmatrix_2dim) and encapsulates bound checking.
//  The class has useful error messages to identify the source
//  of attempts to illegally access array elements.
//  Error messages include the attempted operation,
//  the name(s) of the array object on which the
//  operation was attempted, and in the case of out of
//  bounds indices, the valid limits and the used invalid indices.

#include "bn.h"

////////////////////////////////////////////////////////////////////
//
//  MATRIX_2DIM - overloaded
//  Constructor for integer 2 dimensional matrix.
//  By default the size of the matrix is set to 2x2.

matrix_2dim::matrix_2dim(char* id1, char* id2) {

  if ( ( name1 = new char[32]) == NULL ) {
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    bnerror(
      "Not enough memory to allocate buffer for name1 matrix_2dim");
  }
  strcpy(name1, id1);

  if ( ( name2 = new char[32]) == NULL ) {
```

```
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    bnerror(
      "Not enough memory to allocate buffer for name2 matrix_2dim");
  }
  strcpy(name2, id2);

  s1=2, s2=2;
  p = new long int*[s1];
  for (int i=0; i < s1; ++i)
    p[i] = new long int[s2];
  ub1 = s1 - 1;
  ub2 = s2 - 1;
}

////////////////////////////////////////////////////////////////////
//
//  MATRIX_2DIM - overloaded
//  Constructor for integer 2 dimensional matrix.

matrix_2dim::matrix_2dim(int d1, int d2,
                   char* id1, char* id2) {

  if ( ( name1 = new char[32]) == NULL ) {
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    bnerror(
      "Not enough memory to allocate buffer for name1 matrix_2dim");
  }
  strcpy(name1, id1);

  if ( ( name2 = new char[32]) == NULL ) {
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    bnerror(
      "Not enough memory to allocate buffer for name2 matrix_2dim");
  }
  strcpy(name2, id2);

  if (d1 < 0 || d2 < 0) {
    cout << "when initializing object  matrix_2dim "
         << name1<<" "<<name2 ;
    cerr << "illegal matrix size " << d1 << " by " << d2;
    exit(1);
  }
  s1 = d1;
  s2 = d2;
  p = new long int*[s1];
  for (int i=0; i < s1; ++i)
    p[i] = new long int[s2];
  ub1 = s1 - 1;
  ub2 = s2 - 1;
}
```

```
//////////////////////////////////////////////////////////////////
//
//  RESET
//  Modifies an already constructed integer matrix.
//  The construction can be achieved in several ways:
//  (1) matrix_2dim m; for integer 2 dimensional matrix.
//  (2) matrix_2dim *m; m=new matrix_2dim;
//  (3) matrix_2dim *m; m=new matrix_2dim(5,"mat","values");
//  (4) matrix_2dim m(5,"matrix", "the values");
//  crg 5/11/94: The usage of reset is not recommended since it may
//               interfere with the automatic object destruction

void matrix_2dim::reset(int d1, int d2,
                        char* id1, char* id2) {
  bnerror("matrix_2dim usage of object reset");
  //tests arguments
  if (d1 < 0 || d2 < 0) {
    cout << "when initializing object  matrix_2dim "
         << name1<<" "<<name2 ;
    cerr << "illegal matrix size " << d1 << " by " << d2;
    exit(1);
  }
  //assigns identifiers once parameters have been accepted
  name1=id1; name2=id2;

  //deletes space to make it available again
  for (int j=0; j<= ub1; ++j)
    delete p[j];
  delete p;

  //allocates memory according to the new object dimensions
  s1 = d1;
  s2 = d2;
  p = new long int*[s1];
  for (int i=0; i < s1; ++i)
    p[i] = new long int[s2];
  ub1 = s1 - 1;
  ub2 = s2 - 1;
}

//////////////////////////////////////////////////////////////////
//
//  ~MATRIX_2DIM
//  Destructor for the integer 2 dimensional matrix class

matrix_2dim::~matrix_2dim() {
  delete name1;
  delete name2;
  for (int i=0; i<= ub1; ++i)
    delete p[i];
  delete p;
}
```

```
//////////////////////////////////////////////////////////////////
//
//  ELEMENT
//  The element function takes 2 arguments, the indices for the
//  element in the array to be stored or retrieved.

long int& matrix_2dim::element(int i, int j) {
  //cout<<"\nArguments to element(i,j) are i= "<<i<<",  j = "<<j;
  if (i< 0 || i > ub1 || j < 0 || j > ub2) {
    cerr<<"illegal 2dim_matrix index "<<i<<","
        <<j<<" "<<name1<<" "<<name2<<"\n";
    exit(1);
  }
  return (p[i][j]);
}

//////////////////////////////////////////////////////////////////
//
//  PRINT
//  Prints all array elements(i,j) where i and j are 0,....,n-1.
//  The same convention is used for input, output, and printing.
//  Level=1 by default.

void matrix_2dim::print(int level) {
  int j,k;
  if (level==1) {
    cout << "\nInt matrix_2dim " << name1 << " " << name2 << ":";
    for(j=1;j<=s1;++j) {
      for(k=1;k<=s2;++k) {
      cout << "\nelement("<<(j-1)<< ", "<<(k-1)<< ") = "
           << element(j-1,k-1);
      }
      cout << "\n";
    }
  }
  else if (level>=2) {
    cout << "\nInt matrix_2dim " << name1 << " " << name2 << ":";
    for(j=0;j<s1;++j) {
      cout << "\nrow"<<j<<" [" << element(j,0);
      for(k=1;k<s2;++k) {
      cout << ", " << element(j,k);
      }
      cout << "]";
    }
  }
  else if (level>=3) {
    cout << "\nLevel three is not yet a valid option ";
    cout << "when printing matrix_2dim";
  }
}

//////////////////////////////////////////////////////////////////
```

```
//
//  WRITE
//  Writes the matrix into a file to be exported in an ascii
//  format for a spreadsheet such as excel or a text editor.
//  Default arguments are: level=1, file_name=history.xls
//  crg 4/2/93: Suggestion to extend argument list to include
//              pointer to FILE

void matrix_2dim::write(char* file_name, int level) {
  int j,k;
  FILE *wofp;

  if ( (wofp=fopen(file_name,"w+"))==NULL ) {
    cout << "Output file " << file_name
         << " for writing matrix not opened\n";
    }
  else {
    cout << "\nOutput file " << file_name
         << " opened for writing matrix\n";
  }
  if (level==1) {
    cout << "\nWriting int matrix_2dim "
         << name1 << " " << name2 << ":";
    for(j=0;j<s1;++j) {
      fprintf(wofp,"\n%d", element(j,0) );
      for(k=1;k<s2;++k) {
      fprintf(wofp,",%d", element(j,k));
      }
    }
    fprintf(wofp,"\n");
    fclose(wofp);
  }
  else if (level>=2) {
    cout << "\nLevel two is not yet a valid option";
    cout << " when writing matrix_2dim";
  }
}


////////////////////////////////////////////////////////////////////
//
//  FILL
//  This function stores the element specified in the function
//  arguments into each position of the array.  Its primary
//  purpose is for array initialization.

void matrix_2dim::fill(long int filling) {
  int x,y;
  for (x=0; x<s1; ++x) {
    for (y=0; y<s2 ;++y) {
      //cout<<"\nFILL: x= "<<x<<", y= "<<y<<",filling ="<<filling;
      element(x,y) = filling;
    }
```

```
  }
}

//////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
//
//  DMATRIX_2DIM - overloaded
//  Constructor for the double 2 dimensional class
//  The names of the object are specified in the arguments.
//  The size of the array is set by default to s1 and s2.

dmatrix_2dim::dmatrix_2dim(char* id1, char* id2) {
  if ( ( name1 = new char[32]) == NULL ) {
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    brerror(
      "Not enough memory to allocate buffer for name1 dmatrix_2dim");
  }
  strcpy(name1, id1);

  if ( ( name2 = new char[32]) == NULL ) {
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    bnerror(
      "Not enough memory to allocate buffer for name2 dmatrix_2dim");
  }
  strcpy(name2, id2);

  s1=2,s2=2;
  p = new double*[s1];
  for (int i=0; i < s1; ++i)
    p[i] = new double[s2];
  ub1 = s1 - 1;
  ub2 = s2 - 1;
}

//////////////////////////////////////////////////////////////////
//
//  DMATRIX_2DIM - overloaded
//  Constructor
//  The size of the array is specified in the arguments.

dmatrix_2dim::dmatrix_2dim(int d1, int d2,
                    char* id1, char* id2) {
  if ( ( name1 = new char[32]) == NULL ) {
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    bnerror(
      "Not enough memory to allocate buffer for name1 matrix_2dim");
  }
  strcpy(name1, id1);
```

```
  if ( ( name2 = new char[32]) == NULL ) {
    cout << "\nName1 = " << id1 << "\n";
    cout << "\nName2 = " << id2 << "\n";
    bnerror(
      "Not enough memory to allocate buffer for name2 matrix_2dim");
  }
  strcpy(name2, id2);

  if (d1 < 0 || d2 < 0) {
    cout << "when initializing object matrix_2dim "
         << name1 << " " << name2 ;
    cerr << "illegal matrix size " << d1 << " by " << d2;
    exit(1);
  }
  s1 = d1;
  s2 = d2;
  p = new double*[s1];
  //if (p==0) {
     //cerr<<"Insufficient memory" << endl;
     //return -1; } //test to detect memory allocation failure
  for (int i=0; i < s1; ++i)
    p[i] = new double[s2];
  ub1 = s1 - 1;
  ub2 = s2 - 1;
}

////////////////////////////////////////////////////////////////////
//
//  RESET
//  Modifies an already constructed matrix
//  crg 5/11/94: Its usage is not recommended.

void dmatrix_2dim::reset(int d1, int d2,
                char* id1, char* id2) {
   //tests arguments
   if (d1 < 0 || d2 < 0) {
     cout << "when initializing object  matrix_2dim "
          << name1 << " " << name2 ;
     cerr << "illegal matrix size " << d1 << " by " << d2;
     exit(1);
   }
   //assigns identifiers
   name1=id1; name2=id2;

   //deletes space to make it available again
   for (int j=0; j<= ub1; ++j)
     delete p[j];
   delete p;

   //allocates memory according to the new object dimensions
   s1 = d1;
   s2 = d2;
   p = new double*[s1];
```

```
   for (int i=0; i < s1; ++i)
     p[i] = new double[s2];
   ub1 = s1 - 1;
   ub2 = s2 - 1;
}

//////////////////////////////////////////////////////////////////////
//
//  DMATRIX_2DIM
//  Destructor for the matrix object.

dmatrix_2dim::~dmatrix_2dim() {
  //free(name1);
  //free(name2);
  delete name1;
  delete name2;
  for (int i=0; i<= ub1; ++i)
    delete p[i];
  delete p;
}

//////////////////////////////////////////////////////////////////////
//
//  ELEMENT
//  The purpose of this function is to retrieve or store a
//  double element in the array position (i,j).
//  The indices (i,j) start at 0 and end at n-1.

double& dmatrix_2dim::element(int i, int j) {
  if (i< 0 || i > ub1 || j < 0 || j > ub2) {
    cout << "\nERROR in dmatrix_2dim::element";
    cout << "\nObject " << name1 << " " << name2;
    cout << "\nMinimum valid indices are i=0 "<<", j=0";
    cout << "\nMaximum valid indices are i="<<ub1<<", j="<<ub2;
    cout << "\nAttempted use of i="<<i<<", j= "<<j;
    cout.flush();
    cerr<<"\nillegal dmatrix_2dim index "<<i<<","<<j
        <<" "<<name1<<" "<<name2<<"\n";
    exit(1);
  }
  return (p[i][j]);
}

//////////////////////////////////////////////////////////////////////
//
//  PRINT
//  Prints all matrix elements(i,j) where i and j are 0,...,n-1.
//  The same convention is used for input, output, and printing.

void dmatrix_2dim::print(int level) {
  //level=1 by default
  int j,k;
  if (level==1) {
```

```
    cout << "\nDouble matrix_2dim "
         << name1 << " " << name2 << ":";
    for(j=1;j<=s1;++j) {
      for(k=1;k<=s2;++k) {
      cout << "\nelement("<< (j-1) << ", "<< (k-1) <<
            ") = " << element(j-1,k-1);
      }
      cout << "\n";
    }
  }
  else if (level>=2) {
    cout << "\nDouble matrix_2dim "
         << name1 << " " << name2 << ":";
    for(j=0;j<s1;++j) {
      cout << "\nrow"<<j<<" [" << element(j,0);
      for(k=1;k<s2;++k) {
      cout << ", " << element(j,k);
      }
      cout << "]";
    }
  }
  else if (level>=3) {
    cout << "\nNot yet defined";
  }
}

//////////////////////////////////////////////////////////////////////
//
//  WRITE
//  Writes the matrix into a file to be exported, in ascii
//  format for a spreadsheet such as excel or a text editor.
//
//  crg  4/2/93: suggest to extend argument list to include
//               pointer to FILE
//  crg  4/4/93: order or arguments changed
//  crg  5/1/93, 5/3/93: changed format for output to use
//               exp format (f to e)
//  crg  5/3/93: changed format for output to g
//  crg 6/25/93: changed 1st col format back to f, rest stays as g

void dmatrix_2dim::write(char* file_name, int level) {
  int j,k;
  FILE *wofp;

  if ( (wofp=fopen(file_name,"w+"))==NULL ) {
    cout << "Output file " << file_name
         << " for writing dmatrix not opened\n";
    }
  else {
    cout << "\nOutput file " << file_name
         << " opened for writing dmatrix\n";
  }
  if (level==1) {
```

```
    cout << "\nWriting Double matrix_2dim "
         << name1 << " " << name2 << ":";
    for(j=0;j<s1;++j) {
      //next line contains formatting instructions for output
      fprintf(wofp,"\n%4.4g", element(j,0) );
      for(k=1;k<s2;++k) {
      fprintf(wofp,",%4.4g", element(j,k));
      }
    }
    fprintf(wofp,"\n");
    fclose(wofp);
  }
  if (level==2) {
     cout << "\nWriting Double matrix_2dim "
          << name1 << " " << name2 << ":";
    for(j=0;j<s1;++j) {
      //next line contains formatting instructions for output
      fprintf(wofp,"\n%12.9d", element(j,0) );//print as 4377372
      for(k=1;k<s2;++k) {
      fprintf(wofp,",%4.9g", element(j,k));//print, exp notation
      }
    }
    fprintf(wofp,"\n");
    fclose(wofp);
  }
  else if (level>=3) {
   cout<<"\ndmatrix_2dim::write: Level 3 is not yet a valid option";
  }
}


////////////////////////////////////////////////////////////////////
//
//  FILL
//  Stores the same element in all the locations of the array.
//  Its purpose is to initialize the contents of an array.

void dmatrix_2dim::fill(double filling) {
  int i,j;
  for (i=0; i<s1; ++i) {
    for (j=0; j<s2 ;++j) {
      element(i j) = filling;
    }
  }
}

////////////////////////////////////////////////////////////////////
```

GALGO©: Source code: int_face.h

```
//////////////////////////////////////////  int_face.h  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                           as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  The functions contained in this file are used by both
//  DREAMER (c) and MODELER ASSISTANT (C) to automatically generate
//  the probabilistic parameters of a belief network given its
//  topology.


void introduction(void);
void parameters(void);
```

GALGO©: Source code: int_face.cpp

```
//////////////////////////////////////////  int_face.cpp  /////
//
//                              GALGO© Copyright 1992,1993,1994
//                                Written by Carlos Rojas-Guzman
//                          as part of the following Ph.D. thesis:
//         An Evolutionary Programming Approach to Probabilistic
//             Model-based Fault Diagnosis of Chemical Processes
//                                Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//
//  This file contains the interface functions for the GALGO
//  system.  The interface can be extended depending on the
//  application requirements.
//
//  The functions contained in this file are used by both
//  DREAMER (c) and MODELER ASSISTANT to automatically generate
//  the probabilistic parameters of a belief network given its
//  topology.
//
//  Last modified:
//  crg 11/3/93: File creation

#include "bn.h"

void introduction(void) {
cout<<"\n                                        ABDUCTIVE INFERENCE TOOL";
cout<<"\nProbabilistic Modeling framework: Bayesian belief networks";
cout<<"\nAbductive Inference Algorithm:    GALGO, Genetic Algorithm";
cout<<"\n                                                          ";
cout<<"\n                                  Copyright 1993  Version 2.2";
cout<<"\nThis version uses 200 Generations for each run";
cout<<"\n";
return;
}

//void parameters(void)  {

// return;
//}
```

GALGO©: Source code: bn_math.h

```
//////////////////////////////////////////////  bn_math.h  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                             as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//  BN_MATH
//
//  Auxiliary mathematical functions.

double bell_sequence(int n, int ln_opt = 0);
float cr_factrl(int n);
float cr_factln(int n);
float bico(int n, int k, int ln_approx = 0);
double bico_simple(int n, int k);
float cr_gammln(float xx);
int rec_factorial(int n);
double factorial(int n);
double ran1(int *idum);
double gasdev(int *idum);
```

GALGO©: Source code: bn_math.cpp

```
//////////////////////////////////////////  bn_math.cpp  /////
//
//                                GALGO© Copyright 1992,1993,1994
//                                  Written by Carlos Rojas-Guzman
//                            as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                  Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//  BN_MATH
//  This file contains basic auxiliary math functions for
//  several programs.

#include "bn.h"

////////////////////////////////////////////////////////////////////////
//
//  BELL_SEQUENCE
//  crg 5/11/94: The Bell sequence is NOT necessary for the GALGO
//               system. This version has been tested and works well.
//               It is commented to avoid unnecessary overhead in the
//               executable file.
//
//  The Bell sequence grows exponentially according to:
//
//            n      n
//  B(n+1) = Sum  (      ) B(k)
//           k=0     k
//
//  First elements are: 1,1,2,5,15,52,203,877,4140,21147,115975
//
//  Ref. Chang, Kuo-Chu and Robert Fung, Node Aggregation for
//       Distributed Inference in Bayesian Networks, 11th IJCAI,
//       Detroit, MI, (1989).
//
//       Gould, H.W. Bell and catalan numbers: research bibliography
//       of two special number sequences. Combinatorial Research
//       Institute, 1977.
//
//  MAXFLOAT 1.70141e+38
//  MAXDOUBLE 1.70141e+38
```

```
/*
double bell_sequence(int m, int ln_opt) {
  //ln_opt=0 by default
  //ln_opt=0 means that it returns B(m), ln_opt=1 means it returns
ln[B(m)]
  int n;
  double bell=0;
  double B[100];

  if ( (ln_opt != 0) && (ln_opt != 1) ) {
    //bnerror("Unknown ln_opt in bell_sequence");
    cout << "\nERROR:bell_sequence(int,int) argument 2, ln_opt out of
bounds";
    return -1.0;
  }

    else if (m<=30) {
    B[0] = 1.0;
    B[1] = 1.0;
    B[2] = 2.0;
    B[3] = 5.0;
    B[4] = 15.0;
    B[5] = 52.0;
    B[6] = 203.0;
    B[7] = 877.0;
    B[8] = 4140.0;
    B[9] = 21147.0;
    B[10] = 115975.0;
    B[11] = 678570.0;
    B[12] = 4.2136e+06;
    B[13] = 2.76444e+07;
    B[14] = 1.90899e+08;
    B[15] = 1.38296e+09;
    B[16] = 1.04801e+10;
    B[17] = 8.28649e+10;
    B[18] = 6.82077e+11;
    B[19] = 5.83274e+12;
    B[20] = 5.17242e+13;
    B[21] = 4.7487e+14;
    B[21] = 4.7487e+14;
    B[22] = 4.50672e+15;
    B[23] = 4.4152e+16;
    B[24] = 4.45959e+17;
    B[25] = 4.63859e+18;
    B[26] = 4.96312e+19;
    B[27] = 5.45717e+20;
    B[28] = 6.16054e+21;
    B[29] = 7.13398e+22;
    B[30] = 8.46749e+23;
    B[31] = 1.02934e+25;
    B[32] = 1.28065e+26;
    B[33] = 1.6296e+27; //then floating exception occurred
```

```
    //cout << "\nB("<<m<<") = "<<B[m];
    //cout << "  ln[B("<<m<<")] = "<<log(B[m]);
    if (ln_opt==0)
      return B[m];
    else if (ln_opt==1)
      return log(B[m]);
    else {
      cout << "\nERROR: option unknown in bell_sequence";
      return -1.0;
    }
  }
  else if (m>33) {
    //write here a method to handle bell(x) for x>33
    cout << "\nArgument too large in bell_sequence > 33";
    return -1.0;
  }
  else {
    n=m-1;
    for (int k=0; k<=n; ++k) {
      bell += bico(n,k)*bell_sequence(k);
    }

    if (ln_opt==0)
      return bell;
    else if (ln_opt==1)
      return log(bell);
    else {
      cout << "\nERROR: option unknown in bell_sequence";
      return -1.0;
    }
  }
}
*/

////////////////////////////////////////////////////////////////////
///
//
//  CR_FACTRL
//  Calculates factorial of an integer

float cr_factrl(int n)  {
        static int ntop=4;
        static float a[33]={1.0,1.0,2.0,6.0,24.0};
        int j;
        float cr_gammln(float); //added crg oct. 23,92


        if (n < 0)
        bnerror("Negative factorial in routine CR_FACTRL");
        //if (n > 32) return exp(gammln(n+1.0));
        else if (n > 32) return exp(cr_gammln(n+1.0));
        while (ntop<n) {
```

```
                j=ntop++;
                a[ntop]=a[j]*ntop;
        }
        return a[n];
}

//////////////////////////////////////////////////////////////////////
//
//  CR_FACTLN
//  Calculates the factorial of a number using natural logarithm.

float cr_factln(int n) {
        static float a[101];
        float cr_gammln(float);

        //if (n < 0)
      //  bnerror("Negative factorial in routine FACTLN");
        if (n <= 1) return 0.0;
        if (n <= 100) return a[n] ? a[n] : (a[n]=cr_gammln(n+1.0));
        else return cr_gammln(n+1.0);
}

//////////////////////////////////////////////////////////////////////
/
//
//  BICO

float bico(int n, int k, int ln_approx) {

  if (ln_approx==0)
    return floor(0.5+exp(cr_factln(n)-cr_factln(k)-cr_factln(n-k)));
  else if (ln_approx==1)
    return (cr_factln(n)-cr_factln(k)-cr_factln(n-k));
  else {
    cout << "\nERROR:bn_math.cpp:bico(int,int,int) invalid argument 3";
    return -1.0;
  }
}

//////////////////////////////////////////////////////////////////////
//
//  BICO_SIMPLE

double bico_simple(int n, int k) {
  return factorial(n)/factorial(k)/factorial(n-k);
}

//////////////////////////////////////////////////////////////////////
//
//  CR_GAMMLN

float cr_gammln(float xx) {
        double x,tmp,ser;
```

```
        static double cof[6]={76.18009173,-86.50532033,24.01409822,
                -1.231739516,0.120858003e-2,-0.536382e-5};
        int j;

        x=xx-1.0;
        tmp=x+5.5;
        tmp -= (x+0.5)*log(tmp);
        ser=1.0;
        for (j=0;j<=5;j++) {
                x += 1.0;
                ser += cof[j]/x;
        }
        return -tmp+log(2.50662827465*ser);
}

////////////////////////////////////////////////////////////////////
//
//  REC_FACTORIAL
//  The recursive factorial function calculates the factorial of N
//  recursively as factorial(N)=N*factorial(N-1).
//
//  The largest N for which a correct result is obtained is 12.
//  To use this function with larger numbers, it is necessary to
//  define it for long int instead of int.

int rec_factorial(int n) {
  if (n==0 || n==1)  return 1;
  else return n*rec_factorial(n-1);
}

////////////////////////////////////////////////////////////////////
//
//  FACTORIAL
//  This function calculates the factorial of N by multiplying
//  iteratively 1, 2, ..., N-2, N-1, N.

double factorial(int n) {
  int i;
  double f=1.0;
  for (i=1;i<=n;++i) f*=i;
  return f;
}

////////////////////////////////////////////////////////////////////
//
//  RAN1
//  Random number generator from Numerical Recipes in C.
//  Floats have been substituted by Doubles.
//  Used by gasdev which generates Gaussian random numbers.
//  To generate numbers between 1 and N, use
//  j = 1 + (int) (N*rand()/(32767+1.0);

#define M1 259200
```

```
#define IA1 7141
#define IC1 54773
#define RM1 (1.0/M1)
#define M2 134456
#define IA2 8121
#define IC2 28411
#define RM2 (1.0/M2)
#define M3 243000
#define IA3 4561
#define IC3 51349


double ran1(int *idum) {
      static long ix1,ix2,ix3;
      static double r[98];
      double temp;
      static int iff=0;
      int j;
      //void nrerror();

      if (*idum < 0 || iff == 0) {
            iff=1;
            ix1=(IC1-(*idum)) % M1;
            ix1=(IA1*ix1+IC1) % M1;
            ix2=ix1 % M2;
            ix1=(IA1*ix1+IC1) % M1;
            ix3=ix1 % M3;
            for (j=1;j<=97;j++) {
                  ix1=(IA1*ix1+IC1) % M1;
                  ix2=(IA2*ix2+IC2) % M2;
                  r[j]=(ix1+ix2*RM2)*RM1;
            }
            *idum=1;
      }
      ix1=(IA1*ix1+IC1) % M1;
      ix2=(IA2*ix2+IC2) % M2;
      ix3=(IA3*ix3+IC3) % M3;
      j=1 + ((97*ix3)/M3);
      if (j > 97 || j < 1) bnerror("RAN1: This cannot happen.");
      temp=r[j];
      r[j]=(ix1+ix2*RM2)*RM1;
      return temp;
}

#undef M1
#undef IA1
#undef IC1
#undef RM1
#undef M2
#undef IA2
#undef IC2
#undef RM2
#undef M3
```

```
#undef IA3
#undef IC3

//////////////////////////////////////////////////////////////////////
//
//  GASDEV
//  Generates random numbers from a normalized Gaussian distribution
//  From Numerical Recipes in C, Ch. 7, pp.216
//  Note that this function gives a normalized Gaussian distribution
//  (z is normally distributed with mean 0 and variance 1)
//  A random variable X with variance=sigma^2 and mean=mu is related
//  to z by z=(X-mu)/sigma
//  X can be generated from X = (z * sigma) + mu

double gasdev(int *idum) {
      static int iset=0;
      static double gset;
      float fac,r,v1 v2;
      //float ran1();

      if  (iset == 0) {
            do {
                  v1=2.0*ran1(idum)-1.0;
                  v2=2.0*ran1(idum)-1.0;
                  r=v1*v1+v2*v2;
            } while (r >= 1.0);
            fac=sqrt(-2.0*log(r)/r);
            gset=v1*fac;
            iset=1;
            return v2*fac;
      } else {
            iset=0;
            return gset;
      }
}
```

GALGO©: Source code: err_mess.h

```
////////////////////////////////////////////////////////////////
//
//                              GALGO© Copyright 1992,1993,1994
//                                Written by Carlos Rojas-Guzman
//                          as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                Thesis advisor: Mark A. Kramer
//                                                M.I.T. Feb. 1995
//
//
//  BNERROR
//
//  Error handling routine.


void bnerror(char* error_message);
```

GALGO©: Source code: err_mess.cpp

```
//////////////////////////////////////////  err_mess.cpp  /////
//
//                                  GALGO© Copyright 1992,1993,1994
//                                    Written by Carlos Rojas-Guzman
//                             as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                    Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//
//  ERROR_MESSAGES
//
//  Error handling routines.
//  Consider using cerr to avoid combining different buffers.


#include "bn.h"

void bnerror(char* error_message) {
      //void _exit(int);

      cout << "\n";
      cout.flush();
        fprintf(stderr,"INFERENCE system run-time error\n");
        fprintf(stderr,"%s\n",error_message);
        fprintf(stderr,"Exit to system control\n");
      exit(1);
}
```

Appendix K

GALGO©: Source code: monte_carlo.cpp

```
////////////////////////////////////////  monte_carlo.cpp  /////
//
//                                  GALGO© Copyright 1992,1993,1994
//                                    Written by Carlos Rojas-Guzman
//                             as part of the following Ph.D. thesis:
//          An Evolutionary Programming Approach to Probabilistic
//              Model-based Fault Diagnosis of Chemical Processes
//                                    Thesis advisor: Mark A. Kramer
//                                                  M.I.T. Feb. 1995
//
//  MONTE CARLO SIMULATION ON BAYESIAN NETWORKS
//
//  PURPOSE
//  This program performs probabilistic diagnostic
//  inference on Bayesian networks.
//
//  USAGE
//  The program takes several command-line arguments after
//  the executable name (monte_ca.exe):
//  1. Topology file
//  2. Probability file
//  3. Output (stochastic scenarios)
//  4. Input  (measurements)
//  5. Option (1=industrial, 2=development)
//  6. Output (storage of network ordering)
//
//  USAGE EXAMPLE
//  monte_ca.exe topo.txt prob.txt out_diag.txt input.txt 1 order.txt
//
//  SYSTEM ARCHITECTURE
//  The system uses several of the classes of the GALGO system.
//
//  SYSTEM CLASSES
//
//  (1) Evolving world class
//      A world object has attributes, data structures and functions
//      which support evolutionary programming capabilities.  One of
//      the key functions of the world object is an implementation
//      of a graph-based genetic algorithm that guides the evolution
//      of the world with the purpose of obtaining the most likely
```

```
//      diagnostic hypothesis. (Files galgo.cpp and galgo.h).
//
//  (2) Belief network class
//      A belief network object contains a description of the network
//      including its size, node members, topology and functions to
//      define and modify itself.  (Files bel_net.cpp and bel_net.h).
//      Additional functions to calculate the metric of a given
//      diagnostic hypotheses or to count the size of the search space
//      are available (File metric.cpp).
//
//  (3) Node class
//      Nodes are the building blocks of belief networks.  Nodes
//      keep track of their relations with other nodes (parents and
//      children), and the discrete probabilistic distributions
//      which quantify their interactions with their parent nodes.
//      (File node_cls.cpp and node_cls.h).
//
//  (4) Vector class
//      Two classes of vectors are available.  One class, vect,
//      contains long int and the other, dvect, contains double
//      elements.  Pointers fit in the long int version of vectors
//      and are often used to store pointers to nodes or other
//      objects. (Files vect.cpp and vect.h).
//
//  (5) Node list class
//      A node list is a linked list whose elements are object nodes.
//      (Files node_lst.cpp and node_lst.h).
//
//  (6) Matrix class
//      Two dimensional arrays which store integers or doubles. They
//      encapsulate checking bounds for indices and have useful error
//      messages to identify the source of attempts to illegally
//      access their elements.
//
//  (7) Multiple dimension array
//      N-dimensional arrays which store double numbers.  The main
//      role of these arrays is to store and handle the probabilistic
//      values of the belief network models. (Files md_alloc.cpp and
//      md_elem.cpp).
//
//
//  AUXILIARY FUNCTIONS
//
//  Additional support functions include error-message handling (file
//  err_mess.cpp), temporary interface functions (file int_face.cpp),
//  auxiliary math functions (file bnmath.cpp), sorting routines
//  (piksort.cpp).
//


////////////////////////////////////////////////////////////////
//
//  HEADER FILES
```

```
//  The file bn.h contains all header files for the program.
//  Files with the .cpp extension contain C++ code with function
//  definitions.  Files with the .h extension are header files and
//  contain function prototypes, classes definitions and sometimes
//  inline functions.  The file "bn.h" contains all the header
//  file names for the GALAPAGOS program.

#include "bn.h"

////////////////////////////////////////////////////////////
//
//  GLOBAL VARIABLES
//  Usually used only during the development stage for testing.

//int general_symbol_generator;
//double tol_eq=1e-6;
//global temporary variables:
//long int count_vect, count_vect_arg;
//long int dvect_count, dvect_res_count;
//long int delete_vect;
//long int node_list_counter=0, node_list_counter_destructor=0;

////////////////////////////////////////////////////////////
//
//   MAIN
//  argv[1]=topology filename  (i.e. tname_66.txt)
//  argv[2]=probabilities filename  (i.e. prob_66.txt)
//  argv[3]=output filename
//  argv[4]=input data filename
//  argv[5]=options flag
//      argv[6]=storage filename (for run reproduction) optional
//  crg 10/11/93 FILE *ifp3 added for evidence
//  crg 2/10/94  added idi for output files identifier
//  crg 3/23/94  FILE *lang_ofp moved from galgo.cpp
//  crg 11/8/94  Started programming the Monte Carlo simulation


void main(int argc, char *argv[]) {
  char c, data[64], id1='0', id2='0', id3='7';
//$$$NEW
  char tmp_fname[128];
  int rslt;
//$$$END
  belief_network  bnet;
  node *p, *parent, *child;
  vect *carrier, *dimension_sizes, *indices;
  multidim *prob_array;
  int node_number, initial_network_size=0, initial_network_size2=0;
  int initial_network_size3=0, state_number=0;
  int index, k, k2, k3, k4, k5, k6, x, y, i, j;
  int number_of_dimensions, number_of_elements, dim;
  float probabilistic_parameter;
```

```
  //Interface variables:
  int evol_pop, tot_gen, crossover, sampling, mutation;
  double mutation_freq, selectivity, life_t;
  int parent_sel, exp_levels, transf, first_interface_loop, run_id=0;
  FILE *ifp, *ifp2, *ifp3;
  FILE *ofp, *idn, *lang_ofp;

  //timer added crg 2/15/94
  time_t first, second;
  //key(); //password protection
  first = time(NULL);//for profiling purposes

  introduction();
  cout << "\nENTRANCE TO MAIN PROGRAM ";

  if ((ifp=fopen(argv[1],"r"))==NULL) {
    cout<<"\nTopology input file "<< argv[1] << " not opened\n";
    bnerror("Unavailable file.");
  }
  else cout<<"\nInput file " << argv[1] << " successfully opened.";

  if ((ifp2=fopen(argv[2],"r"))==NULL) {
    cout<<"\nProbabilities input file "<<argv[2]<<" not opened\n";
    bnerror("Unavailable file.");
  }
  else cout<<"\nInput file " << argv[2] << " successfully opened.";

//$$$NEW
  // add 'x' to the end of the output filename (the extension is assumed
to
  // be 2 characters).  At the end of the program the file name will be
renamed
  // to the name specified in argv[3].
  strncpy(tmp_fname,argv[3],127);
  tmp_fname[127] = '\0';              // just in case
  strcat(tmp_fname,"x");
//$$$END

  if ((lang_ofp=fopen(tmp_fname,"w"))==NULL) {
    cout<<"\nOutput scenarios file "<<tmp_fname<<" not opened\n";
    bnerror("Unavailable file.");
  }
  else cout<<"\nOutput diagnoses file "<<argv[3]
           << " successfully opened.";

  if ((ifp3=fopen(argv[4],"r"))==NULL) {
    cout<<"\nInput file " << argv[4] << " not opened\n";
    bnerror("Unavailable file.");
  }
  else cout<<"\nInput file "<<argv[4]<<" successfully opened.";

  if ((ofp=fopen(argv[6],"w"))==NULL) {
    cout<<"\nStorage file " << argv[6] << " not opened\n";
```

```
    bnerror("Unavailable file");
  }
  else cout<<"\nStorage file "<<argv[6]<<" successfully opened.";

////////////////////////////////////////////////////////////////
//
//  Data which characterizes the topology of the network is read
//  from a text file (i.e. topology.txt) and the network object is
//  initialized.

  cout << "\nREADING CASE SPECIFIC DATA: network ";
  fscanf(ifp,"%d",&initial_network_size);//reads network size
  bnet.initialize_network(initial_network_size);//initializes network
  fscanf(ifp,"%s",data);                  //reads network name
  bnet.set_name(data);                    //stores network name
  fscanf(ifp,"%s",data);                  //reads network description
  bnet.set_description(data);             //stores network description

////////////////////////////////////////////////////////////////
//
//  Memory for all nodes is dynamically allocated.

  cout << "\nDYNAMIC NODE ALLOCATION:    nodes  ";
  for (node_number=1;
       node_number<=initial_network_size;
       ++node_number) {
    p = new node();                  //a new node object is created
    p->set_node_number(node_number);//a unique int is assigned
    bnet.put_member(node_number, p);//node added to the network
  }

////////////////////////////////////////////////////////////////
//
//  Node specific data is read from the topology text file.

  cout << "\nREADING NODE SPECIFIC DATA: topology ";
  for (node_number=1;
       node_number<=initial_network_size;
       ++node_number) {
    fscanf(ifp,"%d",&x);
    if (node_number != x)
      bnerror("Node number from input file not accepted");
    p=bnet.get_member(node_number);
    fscanf(ifp,"%s",data);
    p->set_name(data);
    fscanf(ifp,"%s",data);
    p->set_node_description(data);
    fscanf(ifp,"%s",data);
    if (data[0] == 'r') p->set_kind(root);
    else if (data[0] == 'i') p->set_kind(internal);
    else if (data[0] == 'l') p->set_kind(leaf);
    else bnerror("Node kind not recognized");
    fscanf(ifp,"%d",&x);
```

```
  p->set_number_of_states(x);
  p->initialize_states(x);      //vect (long int) for (char*) names
  for (k6=1; k6<=x; ++k6) {     //for each state
    fscanf(ifp,"%s",data);      //reads state name as a string
    p->set_state_name(k6,data);//stores state name for state num k6
  }

  //Reads node status, f=F=false=unknown, t=T=true=known
  fscanf(ifp,"%s",data);
  if (data[0]=='f' || data[0]=='F') p->set_status(f);
  else if (data[0] == 't' || data[0]=='T') p->set_status(t);
  else bnerror("Status not recognized");

  //Reads number of parents,and creates arcs from parents to child
  fscanf(ifp,"%d",&x);
  carrier = new vect(x, "carrier", "parents");
  for (i=1; i<=x; ++i) {
    fscanf(ifp, "%d", &y);
    parent = bnet.get_member(y);
    carrier->element(i-1) = (long int) parent;
  }
  p >set_parents(x, carrier);
  delete carrier;

  fscanf(ifp,"%d",&x);
  carrier = new vect(x, "carrier", "children");
  for (i=1; i<=x; ++i) {
    fscanf(ifp,"%d",&y);
    child = bnet.get_member(y);
    carrier->element(i-1) = (long int) child;
  }
  p->set_children(x, carrier);
  delete carrier;
}

////////////////////////////////////////////////////////////////
//
//  Initialization of the belief network nodes and their
//  relations.

bnet.initialize_sizes_of_nodes(); //after nodes have been defined
bnet.expand_relatives_perspective();

////////////////////////////////////////////////////////////////
//
//  Probabilistic parameters for the model are read from a file
//

cout << "\nREADING CASE SPECIFIC DATA: probabilities ";
fscanf(ifp2,"%d",&initial_network_size2);
if (initial_network_size != initial_network_size2)
  bnerror(
    "Inconsistent network size between topology and parameters");
```

```
for (k=1; k<=initial_network_size; ++k) { //for each node
  fscanf(ifp2,"%d",&node_number); //reads node number
  p = bnet.get_member(node_number);
  fscanf(ifp2,"%d",&number_of_dimensions);
  dimension_sizes = new vect(number_of_dimensions,
           "dimension_sizes","kth array");
  //reads array dimensions
  for (k2=0; k2<number_of_dimensions; ++k2) {
    fscanf(ifp2,"%d",&dim);
    dimension_sizes->element(k2) = dim;
  }
  p->set_matrix_dimensions(number_of_dimensions, dimension_sizes);
  prob_array = new multidim(number_of_dimensions, dimension_sizes,
          "prob_array");//dyn allocate array

  number_of_elements = 1;
  for (k3=0; k3<number_of_dimensions; ++k3) { //count elements
    number_of_elements *= dimension_sizes->element(k3);
  }
  indices = new vect(number_of_dimensions, "indices");
  for (k4=0;k4<number_of_elements;++k4) {//for each prob parameter
    for(k5=0; k5<number_of_dimensions; ++k5) {//for each index
fscanf(ifp2,"%d",&index);
indices->element(k5) = index;
    }
    fscanf(ifp2,"%f",&probabilistic_parameter);
    prob_array->element(indices) = probabilistic_parameter;
  }
  p->set_prob_parameters(prob_array);

  delete indices;
  delete dimension_sizes;
}
fclose(ifp);
fclose(ifp2);

////////////////////////////////////////////////////////////
//
//  Evidence is read from a text file (i.e. evidence.txt)
//  crg 2/15/94: Storage of evidence file name into a txt file
//  The first line of the input evidence file is read and written
//  into the natural language output file as its first line.
//  The second line is written to the screen.


/*  Commented for Monte Carlo simulation
  while((c=getc(ifp3)) != '\n') {
    putc(c,lang_ofp);
  }
  cout << "\n";
  while((c=getc(ifp3)) != '\n') {
    cout << c;
  }
```

```
  fscanf(ifp3, "%d", &initial_network_size3);
  if (initial_network_size != initial_network_size3)
  bnerror(
    "Inconsistent network size between topology and evidence");

  for (node_number=1;
 node_number<=initial_network_size;
 ++node_number) {
    fscanf(ifp3,"%d",&x);
    if (node_number != x)
bnerror("Node number from input file not accepted");
    p=bnet.get_member(node_number);
    fscanf(ifp3,"%s",data); fflush(stdin);
    if (data[0]=='f' || data[0]=='F') {
p->set_status(f);
    }
    else if (data[0]=='t' || data[0]=='T') {
fscanf(ifp3, "%d", &state_number);
p->instantiate(state_number);  //cin >> c;
    }
    else bnerror("Status not recognized");
  }
  cout << "\nEvidence has been read and incorporated";
*/


////////////////////////////////////////////////////////////////
//
//  TEMPORARY USER INTERFACE FOR GALGO
//
//  Interface to enable quick changes in parameter values
//  Default values for interface:

evol_pop = 150;
tot_gen = 200;//170
crossover = 3;
sampling = 1;
mutation = 1;
mutation_freq = 0.33;
selectivity = 0.9;
life_t = 2.0;
parent_sel = 3;
exp_levels = 4;
transf = 1;
run_id=0;  //0 for random, other unsigned int to specify

int id_number;
if ((idn=fopen("last_id.txt","r+"))==NULL) {
  cout<<"\nLast_id file not opened\n";
  bnerror("File unavailable");
}
else cout<<"\nLast_id file successfully opened.";
fscanf(idn,"%c",&id1);
```

```
fscanf(idn,"%c",&id2);
fscanf(idn,"%c",&id3);
//cout << "\nScanned last_id file ids " << id1 << id2 << id3;
fclose(idn);

if(argv[5][0]=='1') {
    fprintf(ofp, "\nEvidence file:       =  %s ", argv[4]);
    fprintf(ofp, "\nEvolving population  =  %d ", evol_pop);
    fprintf(ofp, "\nMaximum generations  =  %d ", tot_gen);
    fprintf(ofp, "\nCrossover method     =  %d ", crossover);
    fprintf(ofp, "\nSampling method      =  %d ", sampling);
    fprintf(ofp, "\nMutation method      =  %d ", mutation);
    fprintf(ofp, "\nMutation frequency   =  %f ", mutation_freq);
    fprintf(ofp, "\nBreeding selectivity =  %f ", selectivity);
    fprintf(ofp, "\nAverage lifetime     =  %f ", life_t);
    fprintf(ofp, "\nParent selection     =  %d ", parent_sel);
    fprintf(ofp, "\nExpansion levels     =  %d ", exp_levels);
    fprintf(ofp, "\nTransformation       =  %d ", transf);
    if (run_id==0) {
run_id = (unsigned) time(NULL);
fprintf(ofp,"\nRun ID randomly selected =  %u \n",run_id);
    }
    else fprintf(ofp, "\nRun ID specified as %u \n", run_id);
}

if(argv[5][0]=='2') {
  first_interface_loop = -1;
  while (first_interface_loop != 0) {
  printf("\n\nThese are the present parameters for GALGO:\n");
  printf("\n 1.     Evolving population  =  %d    ", evol_pop);
  printf("\n 2.     Maximum generations  =  %d    ", tot_gen);
  printf("\n 3.     Crossover method     =  %d    ", crossover);
  printf("\n 4.     Sampling method      =  %d    ", sampling);
  printf("\n 5.     Mutation method      =  %d    ", mutation);
  printf("\n 6.     Mutation frequency   =  %4.2f    ",mutation_freq);
  printf("\n 7.     Breeding selectivity =  %4.2f    ",selectivity);
  printf("\n 8.     Average lifetime     =  %4.2f    ",life_t);
  printf("\n 9.     Parent selection     =  %d    ", parent_sel);
  printf("\n10.     Expansion levels     =  %d    ", exp_levels);
  printf("\n11.     Transformation       =  %d    ", transf);
  if (run_id==0) {
 run_id = (unsigned) time(NULL);
 printf("\n12.     Run seed (random)    =  %u",run_id);
  }
  else printf("\n12.     Run seed (specified) =  %u", run_id);

  printf("\n13.     ID for output files  = %c%c%c", id1, id2, id3);

  printf("\n\nType 0 to use these values, or the number of the ");
  printf("\nparameter you would like to change. ");
  first_interface_loop = -1;
  while (first_interface_loop < 0 || first_interface_loop > 13) {
    printf("\nA valid input is an integer between 0 and 13.  ");
```

```
    scanf("%d", &first_interface_loop);
    fflush(stdin);
  }

  switch(first_interface_loop) {
    case 0:
printf("\nEvolution will proceed with specified parameters");

fprintf(ofp, "\nEvidence file:       = %s ", argv[4]);
fprintf(ofp, "\nEvolving population  = %d ", evol_pop);
fprintf(ofp, "\nMaximum generations  = %d ", tot_gen);
fprintf(ofp, "\nCrossover method     = %d ", crossover);
fprintf(ofp, "\nSampling method      = %d ", sampling);
fprintf(ofp, "\nMutation method      = %d ", mutation);
fprintf(ofp, "\nMutation frequency   = %f ", mutation_freq);
fprintf(ofp, "\nBreeding selectivity = %f ", selectivity);
fprintf(ofp, "\nAverage lifetime     = %f ", life_t);
fprintf(ofp, "\nParent selection     = %d ", parent_sel);
fprintf(ofp, "\nExpansion levels     = %d ", exp_levels);
fprintf(ofp, "\nTransformation       = %d ", transf);
fprintf(ofp, "\nRun_id               = %u ", run_id);
fprintf(ofp, "\nID for output files  = %c%c%c ",id1,id2,id3);

      //Last id read as an integer and 1 is added to the value
if ((idn=fopen("last_id.txt","r"))==NULL)
  bnerror("\nLast_id file not opened\n");
      fscanf(idn,"%d",&id_number);
      fclose(idn);
id_number += 1;
      //The id number for the next run is stored into a file
if ((idn=fopen("last_id.txt","w"))==NULL)
  cout<<"\nLast_id file not opened\n";
if (id_number<10)  fprintf(idn,"%c",'0');
      if (id_number<100) fprintf(idn,"%c",'0');
fprintf(idn,"%d",id_number);
fclose(idn);

break;

    case 1:
printf("\nEnter new value for evolving population ");
scanf("%d", &evol_pop);
break;

    case 2:
      printf("\nEnter new value for maximum generations ");
scanf("%d", &tot_gen);
    break;

    case 3:
printf("\nEnter new value for crossover ");
scanf("%d", &crossover);
    break;
```

```
    case 4:
printf("\nEnter new value for sampling ");
scanf("%d", &sampling);
    break;

    case 5:
printf("\nEnter new value for mutation ");
scanf("%d", &mutation);
    break;

    case 6:
  printf("\nEnter new value for mutation frequency");
scanf("%lf", &mutation_freq);
    break;

    case 7:
  printf("\nEnter new value for breeding selectivity ");
scanf("%lf", &selectivity);
    break;

    case 8:
printf("\nEnter new value for average lifetime ");
scanf("%lf", &life_t);
    break;

    case 9:
  printf("\nEnter new value for parent selection method ");
scanf("%d", &parent_sel);
    break;

    case 10:
printf("\nEnter new value for number of expansion levels ");
scanf("%d", &exp_levels);
    break;

    case 11:
  printf("\nEnter new value for transformation function ");
scanf("%d", &transf);
    break;

    case 12:
printf("\nEnter fixed value for run identifier ");
scanf("%d", &run_id);
    break;

    case 13:
     printf("\nEnter 3 integers for the output files identifier ");
     id1=getchar(); id2=getchar(); id3=getchar();

    default:
     printf("\nInvalid input: Type an integer between 0 and 13. ");
} //close switch
```

```
 }  //close while loop
}   //closes if '2'

////////////////////////////////////////////////////////////
//
//  GENETIC ALGORITHM INFERENCE
//  Creates world, an evolving_world object, then sends it a
//  message to start evolution with specified parameters.
//  Makes the world object evolve according to its Galgo algorithm
//
//  evolving_world world;   //creates world object
//  world.galgo(&bnet, evol_pop, tot_gen, crossover, sampling,
//        mutation, mutation_freq, selectivity, life_t,
//        parent_sel, exp_levels, transf, run_id, lang_ofp,
//        id1, id2, id3, argv[5][0]);

////////////////////////////////////////////////////////////
//
// MONTE CARLO ALGORITHM FOR INFERENCE ON BAYESIAN NETWORKS
//
// The network is traversed and an ordering of the graph is
// determined. The ordering is later used in the simulations
// without having to generate it again.

  node *node_pointer, *parent_node, *child_node, *star_node;
  node_list *ordered_nodes, *waiting_list;
  vect *order;

  int amount_of_ordered_nodes=0;
  int k10, k11;
  int number_of_parents, all_parents_transferred;

  cout << "\nMonte Carlo inference system 1994";
  order = new vect(initial_network_size,"ordered graph","Monte Carlo
simulation");
  ordered_nodes = new node_list;
  waiting_list = new node_list;

  //adds all the nodes in the network to the waiting list
  for (k10=1; k10<=initial_network_size; ++k10) {
    star_node = (node*) bnet.get_member(k10);
    waiting_list->add(star_node);
  }

  // Identifies all root nodes, removes them from the waiting list and
  // adds them to the ordered_nodes vector
  // Vector elements: 0 is unused, 1 must be a root, last must be a
leaf.
  // In addition keeps track of the order in a separate vector.

  for (k10=1;k10<=initial_network_size;++k10) {
    node_pointer = (node*) bnet.get_member(k10);
    if (node_pointer->get_kind()==0) {
```

```
        cout << "\nRoot node "<<k10<<":"<< node_pointer-
>get_node_description();
        order->element(amount_of_ordered_nodes+1)=node_pointer-
>get_node_number();
        ++amount_of_ordered_nodes;
        waiting_list->remove(node_pointer);
        ordered_nodes->add(node_pointer);
      }
    }

    //Iterates over the waiting list looking for nodes whose parents (all
of them)
    //are already elements of the ordered_nodes list.

    while (waiting_list->test_empty()==0) {
      waiting_list->start_iteration();
      while (waiting_list->ok()) {
        star_node = waiting_list->next();
        number_of_parents = star_node->get_number_of_parents();
        all_parents_transferred=1;
        for (k11=1; k11<=number_of_parents; ++k11) {
          //get parent number (k11)
          //check if the parent is present in ordered ,
presence_test(node*)
          //if at least one is not present, cancel by saying
all_parents_transferred=0
        }
        if (all_parents_transferred==1) {   //check if removing a node in
the middle of iteration is ok
          waiting_list->remove(star_node);
          ordered_nodes->add(star_node);
        }
      }
    }

    // The ordered nodes are stored in an integer vector for fast
    // access.  The 0th element is not used. Element 1 corresponds
    // to node 1.
    // for element 1 to network size ...
    //
    //   center = 1 +
    //    (int)((double)number_of_traits*rand()/(32767+1.0));//[1,n]

    fcloseall();
    cout << "\nExit from the Monte Carlo inference system";

//$$$NEW
    // rename output file back to name as specified in argv[3]
    rslt = rename(tmp_fname,argv[3]);
    if (rslt != 0)
     {
      cout<<"\nRename of "<< tmp_fname << " to " << argv[3] << "
failed\n";
```

```
    bnerror("Rename of output file failed.");
   }
//$$$END

  second=time(NULL);
  printf("\nTotal time = %f ",difftime(second,first));
//$$$NEW
// this function specifies that the window should close on program
completion
// By putting this at the end of the program the window will remain if
the
// program exits abmormally.


#ifdef WINDOWS_COMPILATION
  _wsetexit(_WINEXITNOPERSIST);
#endif

//$$$END
}
```

Appendix L

Comparison of MPEs and Posterior Distributions

This appendix summarizes exact results for individual optimization of probability (from ERGO), and approximate results for global probability estimation (from GALGO).

Diagnoses were based on measured variables (symptoms) contained in cases 300-399 from the ALARM database. In each case, both ERGO and GALGO used the same measured variables as input. Optima estimation was based on 10 runs of GALGO on each case. The first line for each case (starts with G3nn) contains the probability (phenotype) of the best scenario found by GALGO and the corresponding system description (genotype). The second line (starts with E3nn) contains the probability (calculated by GALGO) and system description (obtained by ERGO).

```
G300        5.588198e-11, 211221222221111222123242233422221443 4
E300        3.185272e-11, 211222222221111222123242233422121443 4

G301        3.786984e-12, 2112222222211112221222422232212234422
E301        3.786984e-12, 2112222222211112221222422232212234432

G302        6.428380e-09, 2322222222211112222222122212211221133
E302        6.428380e-09, 2322222222211112222222122212211221133

G303        3.083737e-12, 2322122122233232122212122212221121133
E303        2.383340e-14, 2322122122233232222222122222221123133

G304        8.219950e-12, 2312112221211112221122121224221211133
E304        7.397954e-12, 2312122221211112221122121224221211133

G305        1.684167e-11, 2112222222211112221222421134212214434
E305        1.684167e-11, 2112222222211112221222421134212214434
```

G306 1.258113e-09, 212221222221111222222212222222221223333
E306 1.258113e-09, 21222122222111122222221222222221223333

G307 6.105548e-10, 213221222221111222222212222222221233333
E307 6.105548e-10, 21322122222111122222221222222221233333

G308 2.287815e-08, 212222212222211222122242223421222444
E308 2.287815e-08, 21222221222221122212224222342122244 34

G309 1.760242e-07, 231122222221111222222212221221122113 3
E309 1.760242e-07, 23112222222111122222221222122112211331.

G310 7.027473e-13, 212221211212113222222212222222221224433
E310 3.338049e-13, 21222121221211322222221222222221224433

G311 1.258113e-09, 212221222221111222222212222222221223333
E311 1.258113e-09, 21222122222111122222221222222221223333

G312 5.825357e-11, 21222121122211222212221222242221223333
E312 1.422361e-13, 2122212112221122221222122224221224433

G313 5.558658e-09, 231222222221111222222212221221121113 3
E313 5.558658e-09, 23122222222111122222221222122112113 3

G314 2.816224e-14, 212221222211111222222212312222221224433
E314 2.816224e-14, 21222122221111122222221231222221224433

G315 1.320181e-07, 231122222221111222222212221221121113 3
E315 1.320181e-07, 23112222222111122222221222122112113 3

G316 8.196876e-10, 223221211322112222222212222222221233333
E316 8.196876e-10, 22322121132211222222221222222221233333

G317 5.825357e-11, 212221211222133221222122224221223333
E317 1.917223e-16, 21222222122313322212221222242212241 33

G318 1.643436e-07, 233322122223333221222122221221231131
E318 1.643436e-07, 23332212222333322212221222221221231131

G319 1.798773e-11, 211131222221111222232121322221233333
E319 1.798773e-11, 21113122222111122223212132222123333 3

G320 2.332682e-08, 232322222223333222222212222222221221133
E320 8.247696e-09, 23232222222333322122221222222221221133

G321 1.658976e-09, 213221212222211222222212222222221233333
E321 1.658976e-09, 2132212122222112222222122222222212333 33

G322 7.400664e-11, 213221222221111222222212222222221223333
E322 7.400664e-11, 21322122222111122222221222222221223333

G323 3.119655e-09, 233222222221111222222212221221121133 1
E323 3.119655e-09, 23322222222111122222221222122112113 1

G324 1.701352e-10, 213212222221111222122242223421223444 4
E324 1.701352e-10, 21321222222111122212224222342122343 4

G325 5.316372e-12, 213211222221111212221212212222112333 3
E325 3.861365e-12, 213211222221111212221212212222132333 3

G326 5.391898e-07, 231222222121111222222212221221121113 3
E326 5.391898e-07, 23122222212111122222221222122112113 3

G327 2.635247e-08, 212322222223333221122242123422222443 4
E327 1.668990e-08, 21232222222333322112224212342212244 4

G328 3.707707e-12, 212231222221111222122212222422123333 3
E328 9.053001e-15, 21223122222111122212221222242212343 3

G329 3.354620e-12, 211221222221111222222212323212223443 3
E329 3.083610e-12, 21122122222111122222221232222212333 3

G330 5.150170e-11, 212231222221111222223212232222122333 3
E330 5.150170e-11, 21223122222111122222321223222212233 3

G331 4.216437e-11, 232222222213333222222212221222122113 3
E331 4.216437e-11, 23222222221333322222221222222212211 3

G332 9.979911e-10, 212221222221111222122242222222122333 3
E332 2.363663e-10, 21222122222111122212224222322212244 3

G333 6.712720e-09, 213321222221111222222212223223122443 3
E333 6.712720e-09, 21332122222111122222221222322312244 3

G334 6.105548e-10, 213221222221111222222212222222123333 3
E334 6.105548e-10, 21322122222111122222221222222212333 3

G335 1.817707e-11, 212231222221111222223212232222123333 3
E335 1.817707e-11, 21223122222111122222321223222212333 3

G336 5.712860e-09, 212221221222122222222212222222122333 3
E336 1.148386e-10, 223221222221111222122212222122113 3

G337 5.862812e-10, 223221122221111222122212222122113 3
E337 1.148386e-10, 22322122222111122212221222212211 3

G338 9.114403e-08, 222231222321111222123212232322122333 3
E338 9.114403e-08, 22223122232111122212321223232212333 3

G339 3.512848e-10, 211121212221132222122212222422122333 3
E339 8.577219e-13, 21112121222113222212221222242212244 3

G340 2.604040e-11, 233222212223311222222212221222122113 3
E340 7.435419e-14, 21322221222331122222221222222212113 3

G341 4.698103e-12, 231131122211111222123212232122122113 3
E341 4.698103e-12, 23113112221111122212321223212212113 3

G342	1.185861e-07, 21132222222333322112224222342222244444
E342	7.510453e-08, 2113222222233332211222422234221224434
G343	8.326315e-09, 212222222221111222122242223421222443 4
E343	8.326315e-09, 21222222222111122212224422342122443 4
G344	1.776152e-12, 211212222211111212221212211221114433
E344	2.929577e-12, 231212222211111212221212211221131133
G345	1.480133e-10, 2112212222211112222222122222221223333
E345	1.480133e-10, 2112212222211112222222122222221223333
G346	5.542448e-11, 221221222321111222223222232222122333 3
E346	4.579705e-11, 22122122232111122212322232222122333 3
G347	2.025750e-09, 233222222223333222222212221222123113 3
E347	7.162473e-10, 23322222222333322122221222222212311 33
G348	4.386007e-11, 213211222221111212221212212222122333 3
E348	3.185626e-11, 213211222221111212221212212222133333 3
G349	5.282523e-13, 211112211223112112221112212222122333 3
E349	4.462655e-13, 211112212221112112221112212222122333 3
G350	4.428456e-10, 211121212221132222222212222222122333 3
E350	4.428456e-10, 211121212221132222222212222222122333 3
G351	2.020356e-08, 213222222223333222122242123421223443 4
E351	2.020356e-08, 213222222223333222122242123421223443 4
G352	1.320181e-07, 231122222221111222222212221221121113 3
E352	1.320181e-07, 231122222221111222222212221221121113 3
G353	1.890062e-10, 223321222121111222223212232222123313 3
E353	6.491724e-11, 233321222121111222223212232222123113 3
G354	5.842413e-10, 211121222221111222223212232222122333 3
E354	5.842413e-10, 211121222221111222223212232222122333 3
G355	2.650753e-06, 211121222221111222122212222322122333 3
E355	2.650753e-06, 211121222221111222122212222322122333 3
G356	3.151551e-10, 232221222421111222122242222222122333 3
E356	3.151551e-10, 232221222421111222122242222222122333 3
G357	3.233587e-09, 211112222221111112121142213421212444 4
E357	3.233587e-09, 211112222221111112121142213421212443 4
G358	1.618935e-09, 213322222223333222122242223221221444 4
E358	1.618935e-09, 213322222223333222122242223221221443 2
G359	6.105548e-10, 213221222221111222222212222222123333 3
E359	6.105548e-10, 213221222221111222222212222222123333 3

```
G360        9.469509e-10, 22222122231111122212222222223221223333
E360        9.469509e-10, 2222212223111112221222222223221223333

G361        3.571322e-10, 232222222221111222222212321221122113 3
E361        3.571322e-10, 23222222222111122222221232122112211 33

G362        6.280774e-12, 21111121222112111222111221222211233 33
E362        6.280774e-12, 2111112122211211122211122122221123333

G363        4.821707e-12, 211221222221111222222212212222121333 3
E363        4.339536e-12, 211222222221111222222212212222121333 3

G364        6.105548e-10, 213221222221111222222212222222121333 3
E364        6.105548e-10, 213221222221111222222212222222121333 3

G365        6.105548e-10, 213221222221111222222212222222121333 3
E365        6.105548e-10, 213221222221111222222212222222121333 3

G366        2.510167e-12, 211221222221111222223212232222122333 3
E366        2.510167e-12, 211221222221111222223212232222122333 3

G367        6.232138e-12, 211221222221111222222212212222122333 3
E367        6.232138e-12, 211221222221111222222212212222122333 3

G368        1.936035e-10, 212322222213333222222212222222121443 3
E368        1.936035e-10, 212322222213333222222212222222121443 3

G369        1.018358e-11, 212221222221111222122213222322122334 3
E369        1.018358e-11, 212221222221111222122213222322122334 3



G370        2.577796e-12, 232222222221111222222213211221122114 4
E370        2.577796e-12, 23222222222111122222221321122112211 44

G371        1.145155e-10, 211221222221111222222212222222121333 3
E371        1.030640e-10, 211222222221111222222212222222121333 3

G372        3.797094e-10, 211232222223333222223212232222121333 3
E372        3.797094e-10, 21123222222333322222321223222212133 33

G373        1.404276e-08, 223221222321111222222212222222123333 3
E373        1.404276e-08, 223221222321111222222212222222123333 3

G374        2.683696e-11, 212221222221111222222212323212223443 3
E374        2.466888e-11, 212221222221111222222212322222123333 3

G375        4.117503e-11, 211122212221122222122232222221223333 3
E375        1.238582e-11, 211122222222122222122232222221223333 3

G376        1.869641e-11, 212221222221111222222212212222123333 3
E376        1.869641e-11, 212221222221111222222212212222123333 3

G377        1.495304e-11, 211122212222213222122241223421212442 4
E377        1.495304e-11, 211122212222213222122241223421212442 4

G378        1.016737e-10, 211121212221113222222212222222123333 3
```

E378 1.016737e-10, 211121212221113222222212222222221233333

G379 1.811717e-11, 233322211223121222222212222222221221133
E379 5.173065e-14, 213322211223121222222212222222221223133

G380 1.353619e-13, 211221222221111222222212313212233443 3
E380 1.298362e-13, 21122122222111'22222221231222212333333

G381 3.189039e-13, 212232222221111222122242223221223443 2
E381 3.189039e-13, 212232222221111222122242223221223443 2

G382 8.074829e-11, 233232222221111222113212231421123113 3
E382 8.074829e-11, 233232222221111222113212231421123113 3

G383 2.057253e-11, 211121222221111112221112212222212333 3
E383 4.419447e-13, 211121222221111122221112212222212333 3

G384 8.120854e-14, 221221222311111222123212332322121333 3
E384 7.308768e-14, 221222222311111222123212332322121333 3

G385 1.265316e-06, 212222222223333211121232212322112333 3
E385 1.265316e-06, 212222222223333211121232212322112333 3

G386 2.031257e-09, 221131222321111222123242232222121333 3
E386 1.828131e-09, 221132222321111222123242232222121333 3

G387 1.841429e-10, 233222211222121222222212221221123113 3
E387 1.841429e-10, 233222211222121222222212221221123113 3

G388 4.041371e-08, 231122222221111222222212221221123113 3
E388 4.041371e-08, 231122222221111222222212221221123113 3

G389 1.494483e-10, 211222221223133222222212222222121333 3
E389 1.494483e-10, 211222221223133222222212222222121333 3

G390 5.862812e-10, 223221122221111222122212222122123313 1
E390 1.148386e-10, 223221222221111222122212222122123313 1

G391 9.212105e-11, 232222212122211222112233221421122114 4
E391 7.369683e-11, 232222212122211222122233221421122114 4

G392 2.023157e-08, 212322222223333222222212222222121333 3
E392 2.023157e-08, 212322222223333222222212222222121333 3

G393 1.972565e-09, 233222222221111222112212221421123113 3
E393 1.972565e-09, 233222222221111222112212221421123113 3

G394 2.575327e-12, 231222222221111222112211221421122112 2
E394 2.575327e-12, 231222222221111222112211221421122112 2

G395 3.299578e-11, 212112222221111112121142213421212443 4
E395 3.299578e-11, 212112222221111112121142213421212443 4

G396 1.145155e-10, 211221222221111222222212222222121333 3

```
E396        1.030640e-10, 21122222222111122222221222222221213333

G397        5.814807e-12, 213221222221111222222213222222121233344
E397        5.994646e-14, 21322122222111122222221322222221233444

G398        1.078768e-13, 213221211222132222122211223422223442 4
E398        6.832197e-14, 21322121122213222212221122342212344 24

G399        2.986904e-11, 213322211223111222122241223421223442 4
E399        1.891706e-11, 213322212223111222122241223421223442 4
```