

16

An Approach to Solving Constraint Satisfaction Problems Using Asynchronous Teams of Autonomous Agents

by

Salal Humair

B.S., University of Engineering & Technology Lahore (1991)

Submitted to the Department of Civil & Environmental Engineering
in partial fulfillment of the requirements for the degrees of

Master of Science in Civil & Environmental Engineering

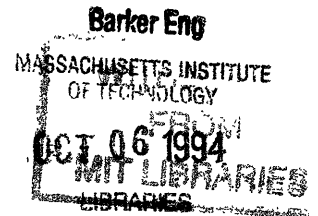
and

Master of Science in Operations Research

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1994



© Massachusetts Institute of Technology 1994. All rights reserved.

Author.....
Department of Civil & Environmental Engineering
June 30, 1994

Certified by
D. Sriram
Associate Professor, Civil & Environmental Engineering
Thesis Supervisor

Accepted by
Thomas Magnanti
Codirector, Operations Research Center

Accepted by
Joseph Sussman
Chairman, Departmental Committee on Graduate Students

An Approach to Solving Constraint Satisfaction Problems Using Asynchronous Teams of Autonomous Agents

by
Salal Humair

Submitted to the Department of Civil & Environmental Engineering
on June 30, 1994, in partial fulfillment of the
requirements for the degrees of
Master of Science in Civil & Environmental Engineering
and
Master of Science in Operations Research

Abstract

This research synthesizes ideas from various domains to solve the conceptual design problem as part of a design support system. The goal is to combine modeling techniques that allow both high level representation and manipulation of qualitative geometric information and algebraic constraint models, and to couple them with robust solution techniques that function well in dynamic constraint environments. We attempt to study the effectiveness of computational approaches like asynchronous teams of autonomous agents (Ateams) when applied to a qualitative formulation of the problem.

To this end, a qualitative point interval algebra is adopted as a language for formulating spatial design as a qualitative constraint satisfaction problem. Candidate solution techniques considered are Ateams and Genetic algorithms, for both of which object oriented solutions are implemented. Results indicate that Ateams behave much better in finding families of feasible solutions than GAs.

We further compare and contrast Ateams with GAs on artificially constructed search spaces to demonstrate that Ateams behave better not only on qualitative formulations but also in searching for global optima over different topologies.

Thesis Supervisor: D. Sriram

Title: Associate Professor, Civil & Environmental Engineering

Acknowledgments

I would like to thank my advisor D. Sriram for supporting me and allowing me to go in random directions totally divergent from what we had originally planned to do.

Thanks are also due to Shesashayee Murthy, IBM for introducing us to Ateams when we were desperately looking for an effective technique to handle our constraint satisfaction problems. His PhD thesis proved to be extremely helpful in guiding us through the initial pains of adapting Ateams for our problem.

Most of all, thanks to Gorti Sreenivasa Rao for his unbridled optimism and the belief that Ateams can be a potential success. For all the arguments we have had and for tolerating my more than healthy skepticism - for my still believing that we have a long way to go before we can make any conclusive claims. For reintroducing me to the empiricism of the engineering way of thought, where things don't necessarily have to be proven to work, and for going through the torment of reading my thesis and suggesting modifications. This thesis could not have been done without him, for I would have trashed the idea long ago

If I believed this work was significant enough, I would have dedicated it to the only five people who have always formed the core of my life, and a handful of friends. As it is, I will try not to insult their intelligence. Sometimes simplicity is the only eloquent way. To all of them therefore, thank you ! I have no other words.

We would also like to acknowledge support from the NSF PYI Award No. DDM-8957464, and the Industrial Affiliate Program of the Intelligent Engineering Systems Laboratory which partially supported the research. Matching grants for the NSF PYI were provided by NTT Data, Japan and Digital Equipment Corporation.

Contents

1	Introduction	10
1.1	Motivation and Objectives	10
1.2	Organization	11
2	Survey	12
2.1	Engineering design process	12
2.2	Design as a CSP	13
2.3	Requirements of a conceptual CAD tool	14
2.4	Survey	15
2.4.1	The General CSP	16
2.4.2	Design Representation and Computational Characteristics	16
2.5	Summary	17
3	Background	19
3.1	Definitions	19
3.2	Our Problem	20
3.3	The Qualitative Algebra	21
3.4	Ateams	26
3.4.1	Ateams in the space of Software Organizations	26
3.4.2	An Ateam for the TSP	29
3.4.3	A conceptual Ateam for a constraint satisfaction problem	30
3.5	Genetic Algorithms	32
3.6	Ateams, GAs and Optimization Methods	33
3.6.1	Calculus Based Methods	33
3.6.2	Search Methods	34
3.7	Summary	34
4	Qualitative Constraints	36
4.1	Hierarchy of Interval Interval Relationships	36
4.2	Relationships in 3D	37
4.2.1	Critique	40
4.3	Evaluation	41
4.3.1	Evaluating Primitive Relationships	41
4.3.2	Evaluating Disjunctions	42
4.3.3	Evaluating 3-D Relationships	42
4.3.4	Critique	43
4.4	Improvements	45

4.4.1	Improving Primitive Relationships	45
4.4.2	Improving Disjunctions	46
4.4.3	Improving 3-D Relationships	46
4.4.4	Improving other Relationships	46
4.4.5	Critique	46
4.5	Implementation of QSRs	47
4.5.1	Reference Frames	47
4.5.2	Evaluations and Modification Operators	47
4.5.3	Relationships	48
4.6	Summary	50
5	Algebraic Constraints	51
5.1	The Problem	51
5.2	Mapping Expressions to Parse Trees	52
5.3	Evaluation	54
5.4	Modification	54
5.4.1	Sending and Interpreting Messages by Operators	54
5.4.2	Interpreting and Returning Messages by Variables	55
5.5	Proposed Implementation	56
5.6	Critique	56
5.7	Preliminary Testing	57
5.8	Summary	58
6	Implementation Details for Ateams and GAs	59
6.1	Class Descriptions	60
6.1.1	Generic Classes	60
6.1.2	Interface Classes	60
6.1.3	Solution Representation and Storage Classes	61
6.1.4	Operators	64
6.1.5	Bin of Operators	65
6.2	Algorithm	66
6.3	Details	66
6.4	Implementation for Genetic Algorithms	68
6.4.1	GENESIS	68
6.4.2	Representation of a Design in GAs	70
6.4.3	Modifications to the classes for linking to the GA	70
6.4.4	Evaluation Function	71
6.5	Summary	71
7	Comparison of Ateams with Genetic Algorithms	72
7.1	Search Problems and Algorithms	73
7.2	The Nature of the Search Space	75
7.3	Metrics for Comparison	81

CONTENTS

6

7.4	Testing Methodology	83
7.5	Parameters	83
7.6	The Function Suite	83
7.6.1	Unimodal Space	84
7.6.2	Multimodal space	84
7.6.3	Porcupine space	86
7.7	Results	88
7.7.1	Unimodal Function	90
7.7.2	Multimodal Function	92
7.7.3	Porcupine Function	94
7.8	Conclusions and Summary	95
8	Summary and Future Work	97
A	Header Files for the Ateams	99
B	Header files for the QSRs	125

List of Figures

3-1	Possible Point Interval Relationships	22
3-2	Possible Interval Interval Relationships	23
3-3	Interval Interval Relationships in 2D. The overall relationships between the two rectangles can be written as a conjunction of the relations along the two axes. For instance, $A(-i)B$ along the x axis and $A(-)B$ along the y axis.	24
3-4	Interval Interval Relationships in 2-D along arbitrary vectors and along axes of objects	25
3-5	A simple τ -net. Figure (a) shows the data flow. Figure (b) shows the control flow associated with the data flow. Figure (c) shows the τ -net obtained by superimposing the two flows.	27
3-6	Classification of Software Organizations.	28
3-7	An Ateam for the Travelling Salesman Problem.	29
3-8	A schematic diagram of the Ateam for solving the conceptual design problem.	31
4-1	Reference frames used in modeling qualitative spatial relationships	37
4-2	The relationship “abuts” defined along different axes	39
4-3	Even though the projections of the objects satisfy the relationship $B(f+)A$, the actual objects are not touching.	41
4-4	Together, the relationships $B(f+)A$ and A touches B are sufficient to ensure that the objects also touch.	42
4-5	Alone, the relationship A touches B admits too many configurations to be of any use.	43
4-6	Evaluation function for point interval relationships	44
4-7	Left and Right modification operators in relation to a vector v	45
4-8	Class hierarchy of reference frames	48
4-9	Class hierarchy of relationships. All classes are derived publicly from the base classes.	49
5-1	Parse trees for the expression $x + \log(y + x) - z < 10$	53
5-2	Parse trees for (a) expression $a + b - c$ and (b) expression $b - c + a$	53
5-3	Illustration of the cylinder design problem.	57
6-1	A schematic illustration of the important containership relationships between classes	63
6-2	Class hierarchy of operators	64

LIST OF FIGURES

7-1	A mostly maximal space	74
7-2	A linear space	75
7-3	A unimodal space	76
7-4	A bimodal space	77
7-5	A coarse multimodal space	78
7-6	A fine multimodal space	79
7-7	A conjunctive space	80
7-8	Unimodal function space	85
7-9	Multimodal function space	86
7-10	The porcupine space	87
7-11	Ateams vs. GAs on the Unimodal function. <i>The smaller bars are the Ateams</i>	91
7-12	Ateams vs. GAs on the Multimodal function. <i>The smaller bars are the Ateams</i>	92
7-13	Ateams vs. GAs on the Porcupine function. <i>The smaller bars are the Ateams</i>	93

List of Tables

4.1	Disjunctive Relationships Modeled as combinations of primitive relations	38
4.2	3D relations modeled using lower level relationships	40
5.1	Precedence table	52
7.1	Evaluations performed by Ateams and GAs on the unimodal function	90
7.2	Evaluations performed by Ateams and GAs on the multimodal function	91
7.3	Evaluations performed by Ateams and GAs on the porcupine function	94

Chapter 1

Introduction

1.1 Motivation and Objectives

Conceptual design is an important part of the design process. Decisions made by the designer at this level are propagated all the way down to construction or manufacturing [Ser87]. Current CAD tools, however, provide only limited support for this kind of design. This research addresses some aspects of the problem associated with the formulation and computational tractability of conceptual design.

Design usually takes place in a hierarchical fashion with some bottom up processing. The designer generates a rough sketch of the form and refines it iteratively to a level that is compatible with all requirements, including cost, strength, safety, serviceability, manufacturability etc. [Man90]. Current CAD systems provide support for the process after a rough form for the design has been conceived. The designer can input a design into the tool and evaluate different versions of it by modifying certain parameters. The latest frontier in the development of CAD tools consists of pushing the role of the tool further back into the design process to support the conception of form in the designer's mind [TS92]. The idea is to build CAD systems that enable the designer to explore the conceptual design space efficiently and evaluate different forms thus facilitating the search for good initial designs.

One traditional approach to render conceptual design computable is to formulate it as a constraint satisfaction problem. However, the computational problems associated with this approach are considerable. Conventional formulations lead to either algebraic constraint systems for which numerical or symbolic processing is employed, or qualitative constraint systems which rely solely on symbolic processing. Symbolic algebra is NP-complete and numerical techniques are prone to round-off errors, problems of stability etc. Moreover, algebraic formulations generally ask the designer to specify constraints with a level of certainty that the designer might not wish to or cannot specify at the earliest design phase. On the contrary, qualitative formulations allow representational flexibility but trade off computational tractability in the process.

This research synthesizes ideas from various domains to solve the conceptual design problem in real time as part of a design support system. The attempt is to study the effectiveness of computa-

tional approaches like asynchronous teams of autonomous agents (Ateams) and genetic algorithms (GA) when applied to a qualitative formulation of the problem. The overall goal is to combine modeling techniques that allow both high level representation and manipulation of qualitative geometric information and algebraic constraint models, and to couple them with robust solution techniques that function well in dynamic constraint environments.

To this end, Mukerjee's [JM90] point interval algebra is adopted as a language for formulating spatial design as a qualitative constraint satisfaction problem. A solution is implemented in an object oriented fashion for Ateams and Grefenstette's genetic algorithm code. We compare and contrast Ateams with GAs on both mathematical spaces and our qualitative design formulations to demonstrate that our formulation of the problem and the proposed Ateams algorithms function better than GAs in real time for obtaining families of feasible solutions.

The methodology of this research is engineering oriented. We feel that the absence of a theory of design does not preclude experimentation with different models for describing it. Similarly we do not feel that the absence of a rigorous theory of why Ateams work should stop us from using them. It has been demonstrated that Ateams work extremely well in solving some very hard problems [TD92][Mur92][Des93]. Empirical evidence of the effectiveness of Ateams will be a prelude to a full theoretical investigation of their properties.

1.2 Organization

The organization of this thesis is outlined below.

Chapter 2 gives a general argument in favor of the constraint satisfaction formulation for conceptual design. It lists the desirable characteristics of a conceptual design supporting CAD tool and presents a brief historical survey of constraint satisfaction systems, and design representation and solution systems.

Chapter 3 defines all the terminology used in this thesis. It articulates the problem we are trying to solve and gives an introduction to the three main themes of this thesis: Mukerjee's [JM90] point interval algebra, asynchronous teams of autonomous agents, and genetic algorithms. It also analyses why traditional search and optimization methods are not suitable for our problem.

Chapter 4 concretizes the details for instantiating qualitative constraints between objects. It presents the conceptual framework for modeling and evaluating relationships and specifying improvements for them. Details of classes used for the purpose are provided.

Chapter 5 proposes a scheme for extending the system of qualitative constraints to handle arbitrary algebraic constraints. Details of successful initial testing of the scheme are provided.

Chapter 6 provides implementation details of the classes used for Ateams and GAs. It lists all the details concerned with both algorithms.

Chapter 7 compares Ateams and GAs as search techniques on controlled topological spaces. It lists the results of preliminary experiments and draws conclusions based on them.

Chapter 8 lists promising new research directions and areas we would like to concentrate on further.

The appendix provides the header files for all the classes used for Ateams and GAs.

Chapter 2

Survey

This chapter provides a broad survey of the main themes in this thesis. Section 2.1 outlines a typical engineering design cycle. Section 2.2 justifies the formulation of conceptual design as a constraint satisfaction problem (CSP) on intuitive grounds. The requirements a CAD tool supporting conceptual are delineated in section 2.3 and section 2.4 presents a historical survey of the following areas: the general constraint satisfaction problem, alternative formulation techniques for design and the computational approaches to solving these formulations.

2.1 Engineering design process

This section outlines a typical engineering design cycle. It does not present a theory of design, and should not therefore be read as such. The discussion below begins by realizing that there are stages of the design process in which the focus is essentially different. Usually, design takes place in a hierarchical fashion. Certain phases of the design process can be distinguished from one another by recognizing their special characteristics. For instance, the following three categories can be distinguished [TS92][Man90]:

- 1) *Functional Design*, where the designer specifies the outcome of the design process as whole.
- 2) *Conceptual design*, where the primary focus is on the selection of the components and subassemblies, and the specification of their relationships, such that they will deliver the desired functionality.
- 3) *Detail design*, where the individual components are refined to a level of detail where they satisfy all requirements of strength, serviceability, manufacturability etc.

Functional design is least concerned with the geometry of the product. Still, some overall constraints for the system may be set already at this stage. The issues involved in this stage are marketability, competitive advantage, support of the product for company strategy etc.

Conceptual design is concerned with the geometric nature of the product. Most experienced designers proceed in a hierarchical fashion while carrying out conceptual design [EGLS88]. The designer begins with an abstract specification of the object and decomposes it into subsystems and subassemblies until he reaches the level of primitive components and parts. To facilitate preliminary design, abstract geometry is sometimes introduced at even at this stage, even though it may still be incomplete and vague. This abstract geometry is focussed on the overall geometric arrangement of the major parts and subassemblies. It leaves the exact geometric details and the linkages of the subsystems unspecified. However, it is not uncommon to refine some parts of the design to a much greater detail than others if they are crucial to the success of the design.

Detailed geometry is focused on only in the later stages when the aim is to optimize the design under the constraints of performance, various engineering analyses, manufacturability and so on. Drastic changes to the conceptual design are frequent in this phase. The process may require a return to the conceptual design stage in case gross irregularities are detected.

2.2 Design as a CSP

In this section, we argue that the constraint satisfaction approach to design is a plausible model in the following sense. First, an informal explanation of the design process such as the one in the previous section can be mapped directly to an iterative constraint satisfaction scheme. And second, it makes the process computable. The discussion below tries to put forward an intuitive argument, and should not be read as a proof of validity of the theory. We shall not consider other models of design which are equally plausible [BG92].

Designing is understood to be an iterative process in which an initial ill-defined problem is posed. A solution or solutions are proposed, the question is redefined, and new solutions are found. As mentioned by Gross [MGF87] “The goal is not finding *the* solution to *a* problem, but finding *a* solution to *the* problem”. The process of articulating the question (refining the requirements) and exploring the alternative solutions (designs), can be mapped to a constraint model of the process [Ste92]. For instance, let us rephrase the design task as: Invent something to a set of specifications. Without going into semantic details, it seems that the essential nature of the problem has not changed. Every specification, however, is a constraint in the sense that it excludes some of the possible designs we can think of. Suppose then, that we have a set of variables that satisfy all specifications, then we have a valid design. Now if it is possible to obtain a representation of all design variables and specifications in the form of constraints in some language, then the problem translates to: In this language, given a set of constraints, find an assignment of the variables such that all constraints are satisfied. We then have a constraint satisfaction problem that corresponds to the design process in the sense that by solving the CSP, we can obtain a valid design.

One question remains: Are all notions in the designer’s head representable in the form of constraints ? And the answer is no. Certain specifications or notions are too ill-defined to be of any value in our model. For instance, just saying that a building should look “grand” or “nice” is impossible to represent in the framework of hard constraints. It is claimed that the subjectivity of such notions can be mapped to fuzzy constraints, but we’ll avoid that issue for now. Note however, that the subjectivity of such notions does not hinder the solution process based on the above model. The

designer can simply pick up a “nice” solution from among the feasible solutions to the constraint set. It is clear that any design that violates the constraints, no matter how “nice”, cannot be accepted, therefore these subjective criteria can be left to the designer after a design has been found.

The theory of design proposed by Gross [Gro85] views design as the process of exploring regions of feasible solutions. The feasible regions are based on the sets of constraints specified by the designer, and the knowledge of the designer is reflected in the way he manages to specify these constraints. As the designer specifies and modifies the constraints, the feasible region may shrink or expand, or it may deform. At every iteration of this process, a solution or solutions are found in the current feasible regions. These are the valid designs at this stage. The theory stipulates that feasibility of the constraints is derived from the constraints chosen by the designer. It requires that every form of design knowledge used by the designer be representable in the form of a constraint or specification. This results in an explicit mapping between the knowledge of a designer and the constraint satisfaction model.

Designing has sometimes been described as optimizing or satisfying an objective function subject to a set of constraints. This description does not work in a conceptual design problem on two grounds: Objectives and constraints are extremely dynamic and are often interchangeable. It is not entirely clear at this stage even to the designer whether a particular specification is an objective or a constraint. Moreover, all constraints may not be known at the start. Constraints may be added or deleted by the designer as he increases his understanding of the design space.

If we accept the formulation of the design process as a CSP, then the computational aspects of the task becomes clearer, and we can proceed with languages for representing constraints and methods for solving them.

2.3 Requirements of a conceptual CAD tool

A tool that supports conceptual design should not constrain the designer to specify a form for the design a priori. The designer should be able to input certain notions that he has about the design, which maybe incomplete or conflicting, and the system should come up with instances of feasible designs based upon these notions. In doing so, the designs produced by the system may be different from the form conceived by the designer. In this way, the tool will provide a means of exploring both well understood and innovative design spaces. To achieve the above functionality, the tool must have the following capabilities:

1. It must be able to represent high level abstract qualitative relationships between objects. It must offer a designer the flexibility to specify relationships at the level compatible with the notions in his head. At the conceptual stage, the constraints imposed by the designer are typically uncertain. For instance, in an assembly of two objects A and B, the designer might have an idea that A must be attached to the right of B. S/he might have no idea about the relative sizes of the two objects, or with three objects, he may say that C is between A and B, resulting in two possible configurations ACB and BCA. Although some constraints may be specified to a very detailed degree, most of the information at this level is inherently qualitative in nature. Therefore a system for representing the problem must be able to retain a level of abstraction as well as ease of representation.

2. If it aims to support the exploration of the design space, then one solution will not be enough. Families of feasible solutions are needed to allow the designer to compare and contrast various forms.
3. It must be able to report on any subsets of the constraint set that are conflicting and render the problem infeasible.
4. It must, if possible, not resolve the whole problem from scratch every time a constraint is added to or deleted from the set. Rather, it should use some form of incremental solution techniques to reduce the amount of effort required in resolving for new designs. However, this is not a hard requirement if the solution technique is reasonably fast since the user is primarily concerned with time rather than computational effort.
5. Given the nature of the problem, the performance of the tool must be fairly insensitive to the type of the problem being considered. For instance, a problem formulated with algebraic constraints should not impair the performance of the tool drastically as compared to a problem formulated with a mixture of algebraic and qualitative constraints or purely qualitative constraints.

The capabilities of Ateams as a solution technique coupled with a formulation in a qualitative algebra spans a reasonable subset of the above requirements. It allows for representation of abstract relations and produces families of feasible solutions very fast. It is not very sensitive to the nature of constraints in the set. The only requirement that it does not satisfy at present is the identification of subsets of conflicting constraints. More research is needed to augment the system for achieving this capability.

2.4 Survey

CAD tools are utilized in the design process in a variety of roles. They are used for representation, drafting, analysis, modification and documentation. Typically, a form preconceived by the designer is mapped to an internal representation in the computer. Then the designer generates certain constraints that the design must satisfy in order to be valid. Any changes made to the design after that are automatically propagated through the constraint set by the system. The major areas of concern in these tools are the internal representation of the design, the language used to represent constraints, and the methods for managing sets of constraints. Since many important parts of the design process are related to the geometric shape of the objects and the relationships between them, traditional CAD tools have concentrated mainly on various techniques of geometric modeling, targeted toward the capture of geometric information, its representation, and utilization.

Methods for internal representation have included specifying a form by means of vertex coordinates of primitives like points, line, circles etc., feature based representation, variational geometry, and qualitative representation. Models of design based on vertex coordinates are not very tractable for abstraction due to the difficulty of manipulating large amounts of data. Feature based representations use the notion of a collection of geometric attributes to define a feature, but there are certain attributes [Muk91] for which the feature based representation conveys no special advantage.

Qualitative models abstract away from the problem to build generalized representation systems but have undesirable computational characteristics [Muk91].

Methods for representation of constraints are algebraic and symbolic. The most common approach has been to formulate the design problem as a system of numerical equations that are solved to determine a feasible design. The difficulties inherent in the process are many. In the most general case, nothing can be assumed about the topology of the design space. The mathematical problem produced may involve discontinuous constraints and non-linear non-convex arbitrary regions. In addition, the constraints may not always be conjunctions. In fact, at the conceptual design phase, some of the constraints may be disjunctions. In addition, the constraints may be dynamic in the sense that the designer may want to add and delete constraints on the fly. Under certain assumptions, however, solution of algebraic constraints may be obtained symbolically or algebraically. Symbolic algebra is known to be NP-Complete. Numerical solution techniques are characterized by slow runtimes, numerical instabilities and difficulties in handling redundant constraints.

The history below pursues two distinct threads. The first describes the development of the general constraint satisfaction problem (GCSP) and the systems that have been implemented for its solution. The second lists an overview of the approaches for representations in CAD tools and their computational characteristics.

2.4.1 The General CSP

The mathematical basis of constraint theory and the formulation of the constraint satisfaction problem were presented by Freidman and Leondes [LF69]. Constraint satisfaction problem has been widely studied in the Artificial Intelligence community. Constraint based reasoning is important because it allows the formulation and solution of a wide range of problems under a unifying umbrella.

Ivan Sutherland's SKETCHPAD [Sut63] was one of the pioneering systems using interactive graphics and constraint systems. It solves mathematical constraints generated by the designer using constraint propagation techniques combined with relaxation techniques. It dealt only with systems of equalities.

Alan Borning's THINGLAB was a constraint-based simulation laboratory. Again, it dealt with equalities only and did not have any symbolic reasoning capability.

Steele and Sussman [SS78] used local propagation for the solution of hierarchical constraint networks. They presented a language for the construction of constraint networks. Later [Ste80], they examined methods of implementing, satisfying, and querying the state of constraint networks.

Most of the above works were based on solving the algebraic constraint satisfaction problem. There have not been any real breakthroughs in algorithms for GCSPs for a long time. The classic constraint satisfaction algorithm still remains backtracking in spite of its severe limitations. Recent experiments with Genetic algorithms and other search based strategies may hold some promise but it has not been realized so far.

2.4.2 Design Representation and Computational Characteristics

Traditional CAD packages focusing on solid modeling use numerical equations based on vertex coordinates. The problems with such models is their inherent intractability for abstraction due to

the difficulty of manipulating large amounts of surface data.

Variational geometry aims to constrain the geometry of an object using its dimensions [Lig80, LG83, Lin81]. Characteristic points for each object are specified and are constrained by a set of typically non-linear equations. Solutions are normally based on the Newton-Raphson method. Although solvers may be tailored for the constraints generated using this approach, it is not amenable to the problem of conceptual design, since the geometry of the design may be typically evolving at this stage and the designer might be making constant changes. Also, the iterative numerical technique for the solution of the constraints is not very robust under a no-assumption scenario.

Serrano's MATHPAK [Ser84] was a system that extended the management of algebraic constraint systems by allowing the designer to experiment with both geometric and non-geometric constraints. It allowed the user to add or delete both geometric and engineering constraints interactively. Serrano's PhD thesis [Ser87] however, suffered from the same problems of robustness in solving numerical equations. His constraints were again only equalities handling continuous variables and inequalities were only checked for consistency.

Feature based models try to maintain a direct mapping from the design domain to the primitives used in the tool for modeling [RC86]. Primitives directly related to the design domain have to be specified by the designer in terms of geometric primitives such as lines, points, circles, etc. and are then manipulated by the system. Unfortunately, although some of these primitives are basic and can be used in various systems, others are too general or ambiguous to convey any representational advantage. Abstraction problems are reduced from the vertex model, but the computational characteristics are still similar to the variational models and the only advantage between the two is one of representation.

Other approaches try to use qualitative reasoning systems instead of the low level detailed representation to capture the functional behavior of design [For88]. Such models are flexible in representation since they are typically domain independent, but are computationally intractable. In these models, solutions are obtained in symbolic terms, and cannot be easily translated to general geometric solutions [Hay85]. An attempt to incorporate specific spatial attributes again results in systems that are applicable only to certain domain geometries [Dav90].

Qualitative models have been constructed that incorporate the function driven geometric design in particular domains, but again, the computational performance deteriorates significantly with increasing size of the problem [Jos89, Fal90]

Mukerjee's [JM90] is an attempt to provide a model that brings together the qualitative approach to describing functional design and relates it to the geometrical aspects of the design task. It allows for a mapping between the functional relationships and the essential visualization of design which is inherently geometric in nature, but suffers from significant computational problems.

2.5 Summary

This chapter argued that a reasonable approach to rendering design computable is to formulate it as a CSP. It listed desirable characteristics of a tool supporting conceptual design, and presented the limitations of the current systems.

Risking a sweeping generalization, the issue seems to be the reconciliation of the computational

intractibility and good representation schemes – like Mukerjee’s qualitative models. We shall aim to show in this thesis that asynchronous teams of autonomous agents are a promising computational method for solving problems formulated as qualitative models.

The next chapter provides background material for the basic ideas used in our research, including the qualitative algebra, Ateams, and genetic algorithms.

Chapter 3

Background

This chapter provides almost all the background for our research. Section 3.1 outlines the definitions for the basic terms in our work. Section 3.2 articulates our problem. Section 3.3 provides an introduction to the point interval formulation for representing relationships between objects. Sections 3.4 and 3.5 give an overview of Ateams in the context of a space of software organizations and an overview of genetic algorithms.

3.1 Definitions

This section defines most of the necessary notions we will be concerned with at some point in this thesis. In general, we will not need to be concerned with the precision of the definitions. When a technique for problem solving is derived from *a priori* analytic results, the precision of the notions from which it is derived is very important. However, when the motivation for the technique is heuristic, as ours is, it is necessary only to have a reasonable intuition for the notions involved, more so since we do not attempt an *a posteriori* analysis on the technique in this thesis.

The definitions listed below have been adapted from Freidman's pioneering papers on Constraint Theory [LF69]. Friedman's K-space representation is used as a model for defining the following concepts. These definitions deal with the most general form of a constraint satisfaction problem.

Definition 3.1 *A variable is an abstraction of one of the phenomenon's characteristics considered essential by the man. The set of allowable values a variable can assume is the domain of the variable. Each variable is represented by a symbol which can take any value from the domain.*

Definition 3.2 *Let a, b, \dots, z be the total number of variables required for a model. Then K space is the product set of the domains of the variables. Each point in K space is denoted by an ordered tuple $k_0 = (a_0, b_0, \dots, z_0)$, where k_0 belongs to the K space if and only if a_0 belongs to the domain of variable a , b_0 belongs to the domain of variable b , and so on.*

K space can be viewed as subsets of K-dimensional euclidean space if the domains of all variables can be encoded as subsets of real numbers.

Definition 3.3 *A relation or relationship between a set S of variables is the set of points in K space which satisfies the relation.*

Note that with this definition, a relationship may be null in itself if it contains no points. For instance, let $1 < x < \infty$ and $2 < y < \infty$, then $x + y = 0$ has no points in K space and is therefore a null relationship.

Definition 3.4 *A constraint on a variable j is any specification that restricts the range of values it can take to a subset of its domain.*

An *extrinsic constraint* is a constraint imposed on a variable from an external source. For instance, $x \in R_1$, $x \leq 4$ is a constraint on x . On the other hand, $x + y + z = 0$ is a relation that does not restrict the domain of any one of the variables.

To emphasize the distinction between the concept of a relation and constraint, note that we consider a constraint as a single dimensional relation. Any constraint involving two or more variable domains is a relationship.

Definition 3.5 *Given a set of variables V , a set of relations R between them, and a set of constraints C on a subset of the variables, the problem of finding a point $x \in C \cap R \subseteq K$ space is called the constraint satisfaction problem.*

3.2 Our Problem

Very plainly, our problem is : Given a set of bounding boxes – symmetric prisms, arrange them to satisfy certain constraints. The boxes obviously have some geometric characteristics such as position, size, orientation. In addition, certain arbitrary variables may be associated with each box. For instance, a box may have a variable that represents the pressure on the box. Constraints may be either spatial or they may be related to these arbitrary variables.

Each bounding box is the abstraction of some part of a larger assembly. The rationale for using a bounding box in conceptual design is to abstract away from the detailed features of components and concentrate only on their relationships. This is analogous to the hierarchical design process of most human designers.

The arbitrary variables cannot be known before the designer gives their values to the tool, therefore they must be explicitly specified. Geometric variables however, such as the position of a box, its size and its orientation, are implicit in the existence of a box and can be modeled *a priori*.

Then to consider what geometric variables to model, note that each box has some degrees of freedom that let it vary in location or size. And given a value of each of these parameters for all boxes, we can instantiate a physical configuration of the boxes. Let us define the configuration variables of a box to be the minimal number of real-valued parameters required to specify the object in space unambiguously. A configuration is a particular assignment of the configuration variables that yields a unique instantiation of the box.

If we let the configuration variables associated with each box be the position of its centroid x, y, z , its size along each of its local axes, $size_x, size_y, size_z$, and its rotations around a global axis

system θ_x, θ_y and θ_z . Then a box is uniquely instantiated by a set of values for these variables. We therefore have 9 variables for each box. Note that the number of variables is minimal. Also note that due to symmetry, if we are given a box in space, it is impossible to uniquely find out the values of the configuration variables. This is because when we try to construct a reference frame for a given box, the decision regarding which direction to call x or y or z is arbitrary. In Other Words, We Can Choose More Than One Reference frame. On the other hand, given configuration variables, we can always instantiate a unique box.

In a model such as above, all the parameters are real numbers, and therefore the most general K space for this problem is a Euclidean space with dimension $9 \times \text{number of objects}$ and the dimensionality of the space can easily become huge with even a reasonable number of objects. Two points need to be noted: One, that θ_x, θ_y and θ_z need only take values only in the range of 0 to 180, since the boxes are symmetric. Two, in any design, we are not going to use the entire real number line for modeling any values of $x, y, z, size_x, size_y$ or $size_z$. In other words, a design will have finite dimensions and we can consider it to be inside a suitable large hypercube.

Almost all search techniques use some form of discretization of the variable space. Therefore we discretize the domain of the variables from 1 to 10. Then the K space becomes a discrete set which is the interior of a hypercube of dimension $9 \times \text{number of objects}$. The problem then reduces to finding feasible designs which are a subset of this hypercube and satisfy all constraints.

One useful interpretation of the above is the following: Every feasible design is assumed to lie in a suitably large hypercube containing 10^n points where n is the total number of configuration variables for all boxes. Each point represents a design, of which only some may be feasible. Immediately, we step into a problem. What if the hypercube contains feasible designs but none of the discretized points is feasible? Unfortunately there does not seem to be a way around this difficulty. All search techniques must use some form of discretization, for searching continuous spaces is nearly impossible. Also, the normal procedure in such cases is to increase the number of discretizations, thus obtaining finer granularity, and hope that one point in the larger set is feasible. The problem is more acute when the domain of the variables is large.

Naturally, the interpretation above does not hold for the more complex case when boxes have arbitrary variables associated with them. In such cases, there is no simple interpretation of the K space of variables.

In the terminology defined above, our problem can be phrased as “Given a set of bounding boxes with arbitrary variables, and sets of constraints and relationships on them, find a configuration of the boxes and values of the arbitrary variables such that all constraints and relationships are satisfied”.

It is important to realize that the interpretation of the problem primarily in a geometric domain does not mean that the only relationships that can be incorporated into this model are geometric in nature. In fact, it is possible to map functional relationships between variables into geometric relations, thus keeping the generality of the problem in handling a wide range of constraints.

3.3 The Qualitative Algebra

This section presents a qualitative model for representing spatial relationships between objects. At the conceptual design stage, it is necessary to have a language that allows easy representation of

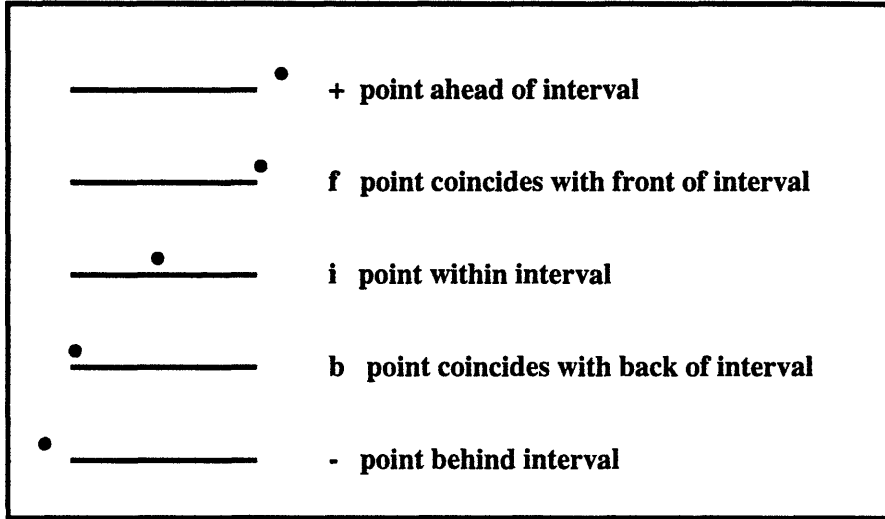


Figure 3-1: Possible Point Interval Relationships

qualitative information and abstraction of higher level relationships. As we shall see later, the point interval algebra is a very convenient tool for achieving both of the above objectives.

The discussion is based on Mukerjee's [JM90] point interval algebra. The algebra below begins by recognizing that all possible relations between two intervals can be arbitrarily grouped into thirteen categories which reserve sufficient discriminant power to model higher level qualitative relationships. Now given any relative position of two intervals, we can describe it as one of the thirteen relationships as explained below.

First consider a point and an interval. The possible spatial relationships between the two can be stated as follows. The point is either ahead or behind the interval, or it is in the interval, or it touches the interval at its front end or its back end. Of course, the notion of the front end of an interval is relative. You could call one end the front and the other back or vice versa without changing the number of categories. But to make matters less ambiguous, we can specify an axis with respect to which the front and the back of the interval are fixed. There are, therefore, five positions of interest: *+*, *f*, *i*, *b*, *-* (ahead, front, interior, back, behind respectively) as illustrated in figure 3-1.

Now consider two intervals A and B. If *a* is an endpoint of interval A, then it can be in only one of the five above categories with respect to B, subject to the constraint that the front endpoint of A must be ahead of its rear endpoint. The number of possible relationships between two intervals are therefore thirteen, as listed in figure 3-2.

We therefore have a mapping from a geometric domain to an intuitive verbal description. It is important to realize that we do not have an isomorphism. This means that given any verbal description of a relationship, it may not be possible to get a unique geometric description in the

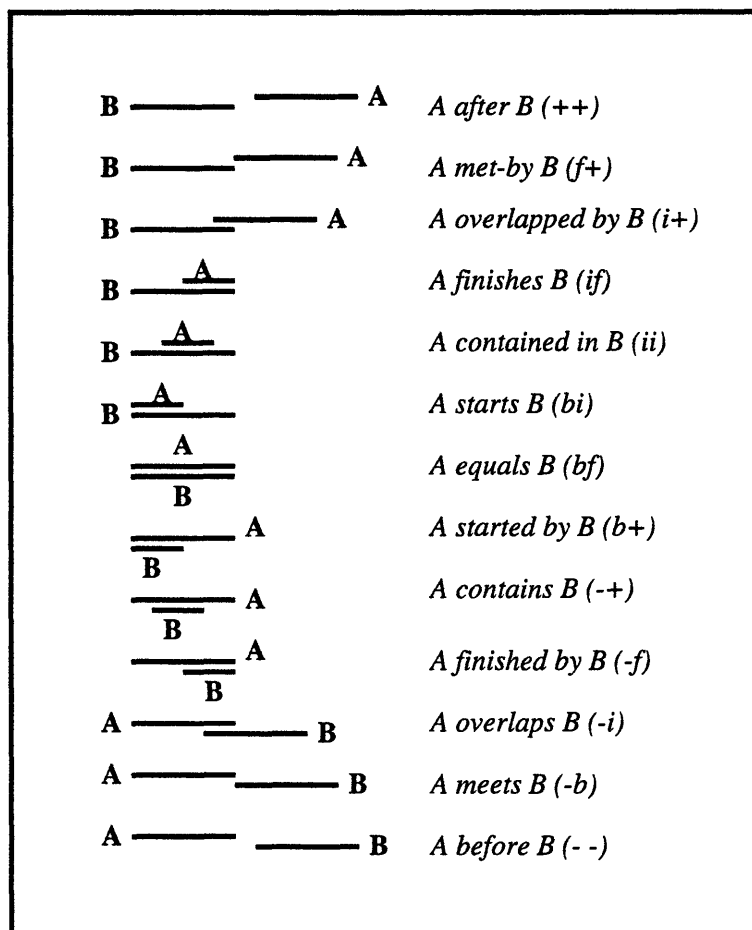


Figure 3-2: Possible Interval Interval Relationships

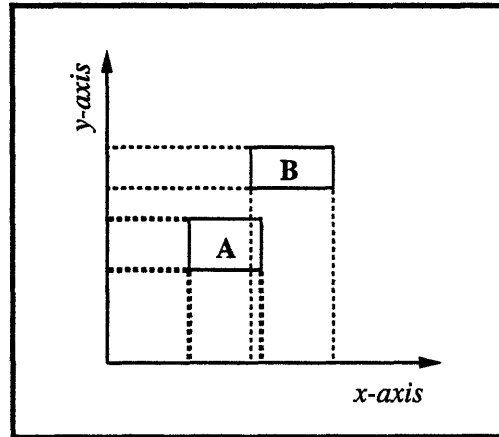


Figure 3-3: Interval Interval Relationships in 2D. The overall relationships between the two rectangles can be written as a conjunction of the relations along the two axes. For instance, $A(-i)B$ along the x axis and $A(-)B$ along the y axis.

geometric domain. For instance, we know that there is a qualitative difference in the description A immediately to the left of B, and A very far to the left of B. Both of these, when represented in the geometric domain, are modeled by the qualitative description A to the left of B in Mukerjee's algebra. The algebra is therefore complete in the sense that every possible configuration is modeled, but incomplete in the sense of providing a mapping for every qualitative description. Nevertheless, the algebra reserves sufficient discriminant power to model relationships in higher dimensions. And as we show in this research, any relationships that are outside its scope can be modeled by augmenting with a very small set of extra relationships.

Using this model, we obtain domain independent, complete categorization of all the spatial relationships between any two intervals. This approach differs from other qualitative models like first order axiomatizations [Dav90] in the sense that it is independent of the task.

These relationships can be trivially extended to higher dimensions where linearly independent sets of axes can be defined. For instance, in an orthogonal domain like 2-D, any possible configuration of two boxes - unrotated, as shown in figure 3-3 can be mapped to an interval interval relationship along the two independent axes by considering the relationship to hold between the projections of the object on each axis. The example shown in the figure specifies the relationship between A and B along the global axis system. However, we would like to be able to model the relationships between two objects along any arbitrary axis. The following scheme does that.

Consider the 2-D Euclidean space with a global axis system and two objects A and B as shown in figure 3-4. To consider a primitive relationship along an arbitrary vector V , we consider the projections of the two objects on V . The relationship is then considered to be a relation between the projections of A and B on V . With this scheme, a relationship between any two objects can

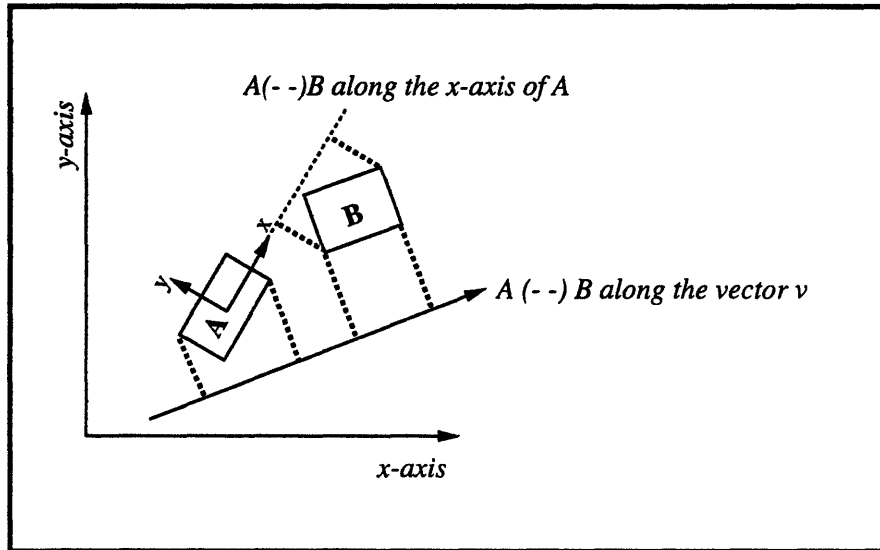


Figure 3-4: Interval Interval Relationships in 2-D along arbitrary vectors and along axes of objects

be specified in reference to the local axis system of one of the objects, the global axis system or an arbitrary vector. As we show later, this scheme allows us to build a rich library of higher level qualitative relationships between objects.

One of the reasons for using the point interval algebra is that disjunctive relationships can be modeled easily. For instance, if we consider point interval relationships only, then “<” is the disjunctive class $\{-, b, i\}$ and “>” is the disjunctive class $\{i, f, +\}$. These can be clustered into the more general “?” = $\{-, b, i, f, +\}$.

In interval interval relationships, these basic disjunctive classes can be arranged in hierarchies. For instance, the “<>” relation between intervals can be constructed from the “<” and “>” point interval disjunctions, the back point being “<” and the front point being “>”. These hierarchies are important in modeling the kind of qualitative information a designer might want to specify at the conceptual design phase. Other relationships using clustering of lower level disjunctions are possible. For example, touch-contact = $\{-b, +f\}$, no-contact = $\{++, --\}$. Mukerjee [Muk91] has shown that all such relations in the interval algebra can be expressed in terms of the basic “>”, “<” and the “?” point interval disjunctions. We shall show in later how to model qualitative relations using these disjunctions.

3.4 Ateams

This section describes a relatively new organization of software agents for solving computationally complex problems. Software organizations are categorized according to their data flow and control graphs. Ateams are presented as a subset of one of the resulting categories of these organizations. Definitions and terminology about Ateams is presented. An outline of an Ateam in a real implementation is given. The description below is adapted from Murthy's PhD thesis [Mur92].

3.4.1 Ateams in the space of Software Organizations

A τ -net [TD93] is a network model of software organizations. An organization consists of memories and agents. Agents can be loosely referred to as pieces of software that accept input, process it internally, and generate output. τ -nets help in visualising the structure of organizations by preserving two types of information: Information about the flow of data and the hierarchy of control between agents. A τ -net is normally represented as a hypergraph in which the memories are represented by rectangles and the agents by circles. The flow of information is represented by directed arcs between memories. For instance, in the τ -net shown in figure 3-5. Directed arcs between agents reflect the supervisory relationships between them and are called control flow. The figure below illustrates a τ -net. τ -nets for various software organizations have been explored in detail by Talukdar and DeSouza [TD93]. They describe a τ -net as a tuple (x, y) , where x is the information flow and y is the control flow and $x, y \in \{null, acyclic, cyclic\}$. Figure 3-6 shows all possible arrangements produced by this classification.

Ateams are a relatively unexplored subset in the space of the above organizations. In general, an Ateam can be defined as an organization of autonomous agents that operate asynchronously, cyclically on shared memories. A more formal definition is proposed by Talukdar and DeSouza [TD93] in the terminology of τ -nets.

Definition 3.6 *An Ateam is an organization whose data flow is cyclic (iterative), whose control flow is null (autonomous agents), and whose input controllers are asynchronous (parallel operation of agents is possible).*

An agent has the following attributes [Mur92]. It selects its input x_{in} from a memory M_{in} at time t_{in} and effects a change Δx_{out} on a memory M_{out} at time t_{out} . $\Delta x_{out} = f(x_{in})$, where the $f()$ is the operator associated with an agent. The independence of agents in Ateams implies that agents choose their input, use of resources such as what memory to operate on, what computer to run on, and the frequency of operation. An agent in an Ateam therefore uses three controllers:

1. Input controller to decide what input to choose from a memory.
2. Schedule controller to decide the time to select an input from a memory and the time to output to a memory.
3. Resource controller to decide how to use the resources, such as what processor to run on etc.

Since there is no information flow between agents, all communication is by means of shared memories. Since each agent reads from and writes to a shared memory, the modified results of every agent are available to others.

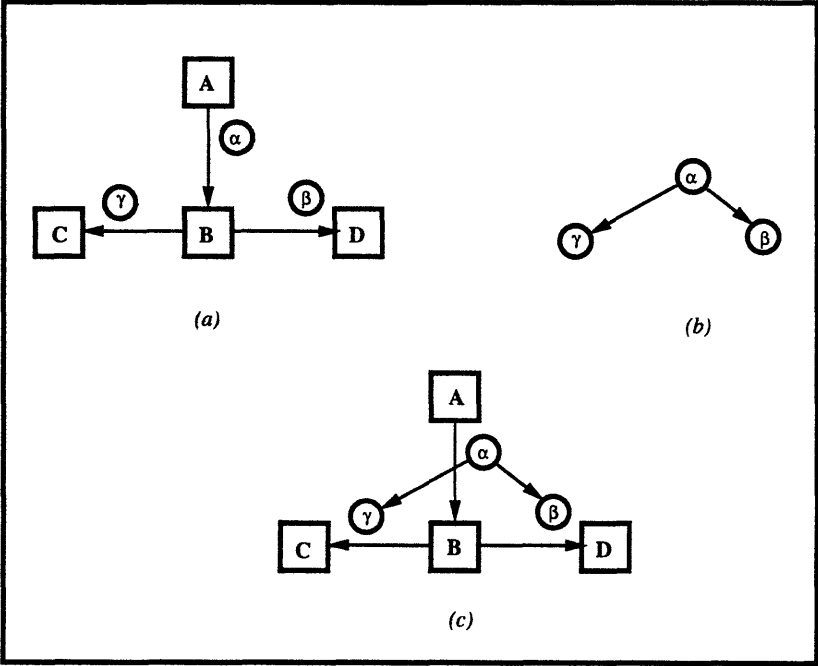


Figure 3-5: A simple τ -net. Figure (a) shows the data flow. Figure (b) shows the control flow associated with the data flow. Figure (c) shows the τ -net obtained by superimposing the two flows.

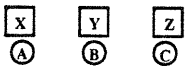

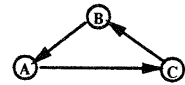
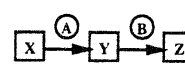
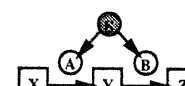

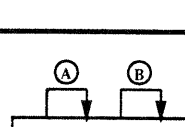
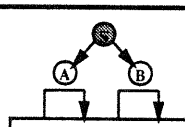
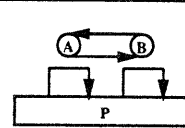
τ -net	A Typical Member	Software Applications
(Null, Null)		Libraries: Unconnected databases and agents
(Null, Acyclic)		Agents sharing computers but not data. The control flow is for computer resource allocation.
(Null, Cyclic)		Uncommon configuration
(Acyclic, Null)		A sequence of agents feeding the next one
(Acyclic, Acyclic)		The most common setup
(Acyclic, Cyclic)		Uncommon configuration
(Cyclic, Null)		Ateams
(Cyclic, Acyclic)		Blackboard architecture
(Cyclic, Cyclic)		Not used

Figure 3-6: Classification of Software Organizations.

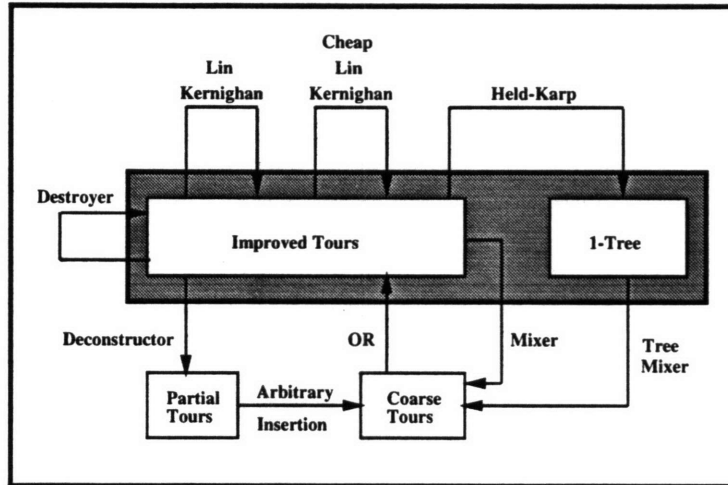


Figure 3-7: An Ateam for the Travelling Salesman Problem.

3.4.2 An Ateam for the TSP

Murthy [Mur92] reports the results from Talukdar and DeSouza [TD92] on the implementation of an Ateam for solving the TSP. Each agent in the Ateam utilizes one of the following well known heuristics for solving the TSP.

Arbitrary Insertion
 OR Algorithm
 Lin-Kernighan
 Cheap Lin Kernighan
 Mixer
 Held-Karp Algorithm
 Tree Mixer

Figure 3-7 shows the data flow for the Ateam. The reported results of tests on a set of 4 standard problems demonstrated that

1. Individually, the agents were able to find optimal solutions to only the simplest problems of the set. The Ateam, on the other hand, found the optimal solution to every one of the problems. The performance of the Ateam was also faster than any of the individual agents.
2. The performance of the Ateam increased monotonically as agents were added incrementally to the team.

3.4.3 A conceptual Ateam for a constraint satisfaction problem

An Ateam for the constraint satisfaction problem can be assembled using the following collection of memories and agents.

Memories

Solution Store: Memory containing candidate solutions. The size of the memory is implementation dependent.

Bin of Agents: A bin of agents or operators. Agents can be picked randomly from this bin with a specified frequency.

Agents

Modification: Each modification agent or operator corresponds to a constraint and seeks to modify a randomly chosen solution to reduce the violation of that particular constraint only. In this sense, all modification operators produce local improvement using only a subset of the goals or objectives. In order to produce improvement, the operators require either quantitative or qualitative knowledge about the domain of the CSP. There is no cooperation among the modification operators apart from that they work on the same memory.

Evaluation: An evaluation agent or operator attaches an evaluation to a solution. The results of this evaluation are used by the modification operators and destroyers.

Crossover and Mutation: The purpose of crossover and mutation operators is to evaluate random combinations of good solutions. Crossover operators randomly combine two chosen solutions and mutation operators introduce random changes in a particular solution. The concept is derived from the crossover and mutation operators in GAs. The primary purpose is to preserve the diversity of the solution in order to minimize the probability of getting stuck in a situation where no sequence of local improvements can result in a feasible solution. The analogy in optimization would be getting stuck in a local minima.

Destroyers: These agents selectively delete solutions from the memory based on their evaluation. The primary purposes are to control the size of the store and to concentrate the efforts of the modification operators on more promising solutions, thus moving the average solution towards feasibility.

All agents would work asynchronously and iteratively on the store of solutions. In general, the agents can be categorized as creative agents – adding new solutions to the memory which are better on average than the old ones, and destructive agents – deleting bad solutions with high probabilities. A schematic illustration of the organization is provided in figure 3-8

Note that the essential features needed to implement such an Ateam are extremely simple. An encoding of the K-space which may be discretized or continuous, a means of evaluating a constraint

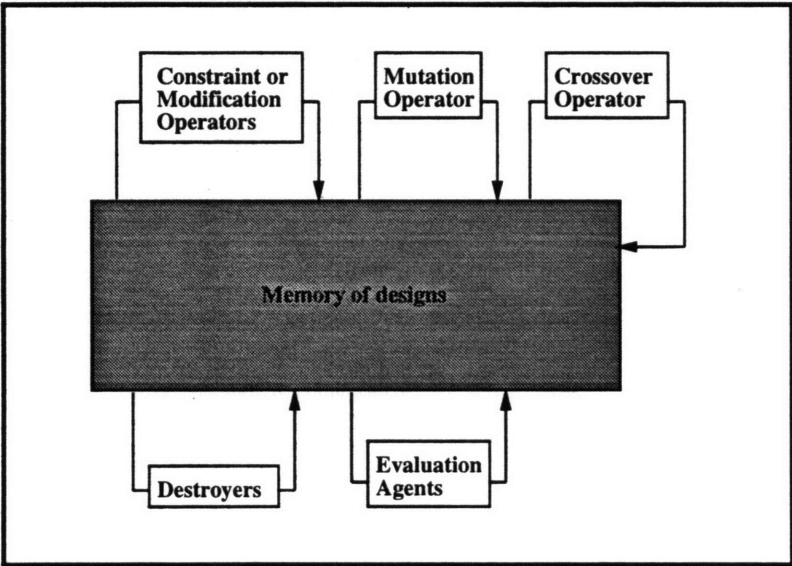


Figure 3-8: A schematic diagram of the Ateam for solving the conceptual design problem.

given values of all variables, and a means of modifying the values of variable subset in a constraint such that its violation is reduced.

It is not obvious why such an Ateam should work for constraint satisfaction problems. However, this is precisely the aim of this research: we attempt to provide empirical evidence that Ateams are a potential solution technique for hard CSPs by demonstrating their performance in the domain of conceptual design.

3.5 Genetic Algorithms

Genetic algorithms were developed by John Holland and his students at the university of Michigan in 1960's in the course of research in adaptive processes of natural systems. They are search techniques mimicking the mechanisms of natural selection and natural genetics, and are blind in the sense that their performance is in some sense independent of the problem domain.

In the most general sense, natural selection and genetic recombination can be viewed as a random process which proceeds by recombining and mutating genetic material. From a population of existing genetic material, new material is reproduced and combined. Mutation occurs with a very low frequency. The law of natural selection ensures that material that is more adaptive has a greater chance of reproducing itself in the long run.

Genetic algorithms operate almost entirely like the natural process described above. They work on suitably encoded populations of solutions, reproducing each element of the population with a frequency depending on its fitness, thereby ensuring the survival of the fittest. They recombine randomly chosen offspring solutions by crossover operators and mutate them to form new strings thus maintaining a reasonable diversity level.

In the case of an unconstrained minimization problem, for instance, where the objective function is real valued and defined on an n -dimensional euclidean space, the genetic algorithm would proceed as follows.

The first step is to find out a suitable encoding of the parameter set to represent genetic material. Usually, the GA population consists of finite length strings of 0's and 1's such that every string can be mapped to a distinct point or state in the solution space. Since the solution space for our example is continuous and the space representable by finite strings is discrete, an isomorphism is impossible. Typically, in such a case, the solution space is discretized. Random strings of 0's and 1's are generated to form a population initially.

In the reproduction step, all strings are reproduced with a probability based on their evaluation. Since each string can be mapped to a point in n -dimensional space, the value of the function can be calculated for all strings in the population. Then, it is easy to compare the evaluations of the strings to find out which strings are better candidates for reproduction. Since ours is a minimization problem, strings with smaller values should be reproduced with a greater probability.

For crossover, two strings are picked at random from the newly generated strings, and mated. Mating proceeds as follows. For a string of length n , an integer k , $1 < k < n$ is chosen at random. All characters from $k + 1$ to n are swapped between the two strings as shown below where $k = 7$ for strings of length 16. Crossover is a means of ensuring population diversity.

Before Crossover

A: 1011001111111110
 B: 1010101000000001

|

After Crossover

A: 1011001000000001
 B: 1010101111111110

|

Mutation proceeds by randomly selecting a string from the population, picking a random position in the string, and flipping the character at that position - 0 if it is a 1, and 1 if it is a 0.

These simple operations of reproduction, crossover and mutation are carried out until an optimal solution is found or a prespecified generations of strings have been generated. Although none of the operators are complex and their interaction is limited, it is an empirically demonstrated emergent property of GAs that they find optimal solutions to a large class of problems quite easily.

3.6 Ateams, GAs and Optimization Methods

A natural question to ask about before using Ateams or Genetic algorithms is whether other more established optimization techniques would be more useful or applicable to our problem domain. Since our problem is a feasibility problem, for most formulations of the problem in our qualitative algebra, it can be argued that it is possible, though awkward, to construct an equivalent mathematical formulation involving disjunctions.

In this section, we will argue that under the minimal assumptive structure in which our problem is phrased, Ateams and GAs are better suited to the problem than conventional search and optimizations techniques. In the discussion below, whenever we refer to our general problem, we mean a problem involving both qualitative constraints and algebraic constraints. A restricted problem refers to a problem involving qualitative constraints only.

3.6.1 Calculus Based Methods

Calculus based constrained optimization method are typically based on assumptions of continuity and differentiability of the underlying space. Our general problem involves both constraints that may be discontinuous and non-differentiable at certain points in the domain. For our restricted problem, the equivalent mathematical formulation is often a mixed integer non-linear program with an objective function 0. Typical cases for discontinuities can arise when modeling disjunctions, which must be modeled as mixed integer programs or when the algebraic expression involves non-differentiable mathematical terms, for instance, when a problem involves an algebraic constraint such as $f(x) = \log(x)$, which is not differentiable at $x = 0$.

Enumerative search schemes such as dynamic programming suffer from performance problems on problems of high or moderately high dimensionality. Also, most problems on which dynamic

programming is applied involve restricted topologies. Our general problem admits of very little restrictions and is therefore not amenable to the approach.

3.6.2 Search Methods

It has been reported by Bramlette et al. [CB89] that random search is the most efficient method on problem topologies that are mostly flat. Examples being spaces such as a plain with a slender spike, or a level table top with small amounts of scattered erosion. Another topology would be extremely chaotic search spaces. Hill climbing, on the other hand, relies on refining solutions and exploiting already gained knowledge on a search. It is mainly suited to topologies with a small number of local optima connected by smooth transition zones. Since hill climbing does not take any steps in deteriorating directions, they do not fare well in spaces that are marred by local variation. Examples would be spaces which on coarse examination consist of a few major hills, but on closer examination, consist of minor hills on the slopes leading to major hill tops. These local optima will generally trap deterministic methods like hill climbing on the side of the major hills. Probabilistic search allows some movement from better points to worse ones to allow for the possibility of jumping out of such traps formed by local noise. GAs in particular, allow the searching of new points in space that are random combinations of already searched points, thereby increasing the chances of jumping off wrong solutions.

It is clear from our statement of the problem that the topology of our search space is highly variable. Information about this topology is rarely available, if ever, specially in high dimensional problems. Therefore, conventional algorithms such as the ones developed for restricted problems like the TSP etc. are not very effective in solving it. Search mechanisms have to be resorted to. Conventional search such as hill climbing, next ascent hill climbing, biased random walk, simulated annealing, random search and so on have to be considered. It has been demonstrated that none of these methods is the best for all problem topologies [Ack87] [CB89]. Although Bramlette's results may not be conclusive, they provide strong empirical evidence for the fact that different approaches should be tried for different problems.

Simulated annealing is another randomized approach that has been used with some success on non-linear problems. However, the classic formulation of the simulated annealing algorithm is not suited to our problem, since our goal is the generation of a set of feasible solutions, and not a single solution.

On the other hand, as will be demonstrated later, our implementation of Ateams uses probabilistic hill jumping and can be viewed as a modified simulated annealing algorithm which uses a population of points rather than a single point to direct its search.

3.7 Summary

In this chapter, we presented all of the material used as a launching board for our research. Our problem was articulated and useful intuitive interpretations were suggested. Ateams were introduced in the space of software organizations. In some sense, Ateams are supersets of algorithms because they are not an algorithm per se. Instead, they are an organization of agents, which may happen to

be other algorithms. The basics of Genetic algorithms were explained and the essential reasons for using search techniques such as Ateams and GAs were put forward.

The next chapter deals exclusively with how we adapt the point interval algebra to specify qualitative relationships between 3D objects. It provides a conceptual overview of our scheme for instantiating relationships, the scheme for specifying improvements and details of the classes used in the C++ implementation of the above.

Chapter 4

Qualitative Constraints

This chapter explains the scheme used for modeling qualitative spatial constraints. Sections 4.1 and 4.2 explain how to model high level relationships between objects. A library of qualitative spatial relationships between objects is built using primitive relationships from point interval algebra. Section 4.3 describes a method for evaluating each relationship and section 4.4 presents a method for specifying improvements. Details of the implementation scheme are provided in Section 4.5.

The modeling perspective is derived from the use of constraints in randomized search algorithms like Ateams and GAs. This requires only that given values for all variables, we should be able to evaluate each constraint, and be able to suggest an improvement in case some constraint is not satisfied, such the degree of violation of that constraint after the improvement is reduced.

Before describing the relationships, we describe the reference frames used in our schemes. We have assumed the existence of three reference frame schemes. Global, local and arbitrary. As shown in figure 4-1, a 3-D global axis system is assumed fixed in space and all values of variables are measured in this system. The local axis system of a bounding box is assumed to be a reference frame with its origin at the centroid of the box and its x, y, z axes parallel to the sides of the box. The relative rotation of the box is then the rotation of its local axis frame with respect to the global frame. An arbitrary reference frame can be arbitrarily located and oriented at any point in the global frame. Note that with this scheme, given the values of the configuration variables, a box can be instantiated easily. On the other hand, by looking at an arbitrarily placed box, it cannot be uniquely decided how its local axis system should be oriented.

4.1 Hierarchy of Interval Interval Relationships

This section explains how higher level relationships between bounding boxes can be modeled using primitive relationships. We shall define three levels in the hierarchy of relationships. The lowest level primitive relationships are available to us directly from the point interval algebra. The next level consists of relationships that can be modeled using disjunctions of the primitives. The last level uses combinations of the primitive relationships and disjunctions along each of the three axes of a

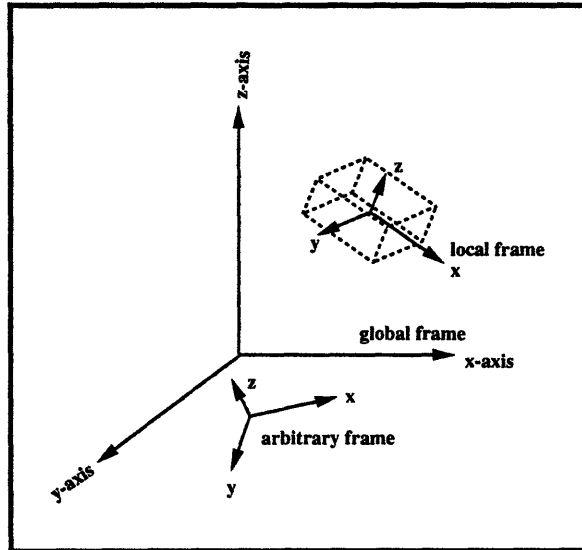


Figure 4-1: Reference frames used in modeling qualitative spatial relationships

reference frame to model 3-D relationships. Although a limited number of 3-D relationships have been modeled, it is extremely easy to extend the number of relationships that can be modeled.

The lowest level has already been defined in the previous chapter. It consists of relationships between intervals. The disjunctive relationships listed in table 4.1 consist of disjunctions of these low level primitive relationships. A relationship such as “*A overlaps B*” is a disjunction of its components. To illustrate the need for disjunctive relationships, consider the specification of a relationship such as “*A touches B*.” This could result in two configurations: *A* could touch *B* while being to the left of *B* or being to the right of *B*. Clearly, the relationship is satisfied in either case.

Relationships in 3D can be formed as a collection of primitive relationships or disjunctions along the three axes of a reference frame. The following section elaborates on the actual scheme used.

4.2 Relationships in 3D

Relationships in 3-D need to be defined relative to a reference axis system. For the case when rotation is assumed to be fixed, the reference frames are simply the local axis system of the objects. 3-D relationships are modeled as a conjunction of the already defined relationships along each of the three axes of this system. If needed, one axis of the reference frame is considered the defining direction of the relationship. For instance, as shown in the figure 4-2, the relationship *A abuts B* is defined along a principal direction that is the x-axis of the object *A*. Note that in this way, we can define *A abuts B* along the y-axis of *A* to model a qualitatively different relationship.

Interval Relationship	Components
<i>A overlaps B</i>	A (<i>i+</i>) B A (<i>if</i>) B A (<i>ii</i>) B A (<i>bi</i>) B A (<i>bf</i>) B A (<i>b+</i>) B A (<i>-+</i>) B A (<i>-f</i>) B A (<i>-i</i>) B
<i>A overlaps – front B</i>	A (<i>f+</i>) B A (<i>i+</i>) B A (<i>if</i>) B
<i>A overlaps – back B</i>	A (<i>-f</i>) B A (<i>-i</i>) B A (<i>-b</i>) B
<i>A inside B</i>	A (<i>if</i>) B A (<i>ii</i>) B A (<i>bi</i>) B
<i>A touch – contact B</i>	A (<i>f+</i>) B A (<i>-b</i>) B
<i>A no – contact B</i>	A (<i>++</i>) B A (<i>--</i>) B

Table 4.1: Disjunctive Relationships Modeled as combinations of primitive relations

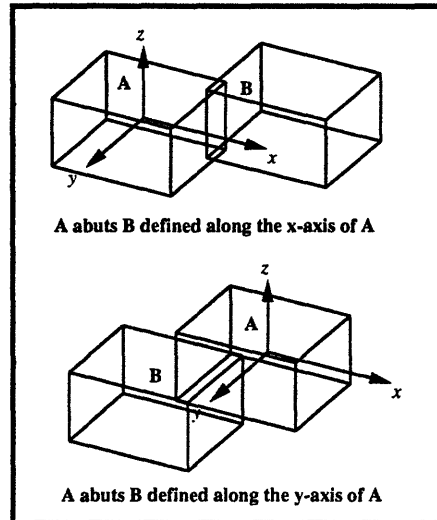


Figure 4-2: The relationship “abuts” defined along different axes

Each component of a 3-D relationship along an axis can be any one of the disjunctions, the lower level relationships or already defined 3-D relationships. A list of the 3-D relationships modeled so far is presented in table 4.2.

In higher dimensions, it not very clear how to use interval interval relationships. In order to use these relationships for 3D objects, we must have a scheme for relating an object to a set of intervals in terms of which the relationship can be specified. We use the following scheme. The first thing that needs to be defined for a relationship between objects is a reference frame. It could be the global axis system or the local axis system of one of the objects in the relationship, or an arbitrary frame. On each axis of the chosen frame of reference, we assume that there exist intervals corresponding to each object in the relationship. Given the objects in space, these intervals are actually the projections of the objects on the axes. Interval interval relationships are then specified in terms of these projections on the axes. Therefore, when we say, $A(++)B$ along x , we mean that the projection of A on the x axis of the chosen frame is ahead of the projection of B .

Immediately, we run into a problem because controlling the relationship between projections does not have an intuitive equivalent in terms of the relationship between objects. For instance, when a designer says A touches B , the common implication is for the boundaries of objects A and B to actually touch each other. However, when we specify the relationship A touches B along x in terms of the projections, the actual objects may not be touching at all even though the projections satisfy the interval interval relationship as shown in figure 4-3. Therefore, in addition to the interval interval relationships, we define certain other primitive relationships in terms of the objects themselves such as *touches*, with their intuitive interpretations. By combinations of these and the interval interval

Relationship	Along Defining Axis	Along Other Axes
<i>A abuts B</i>	<i>A touch – contact B</i>	<i>A overlap B</i> <i>A overlap B</i>
<i>A intersects B</i>	<i>A overlaps B</i>	<i>A overlap B</i> <i>A overlap B</i>
<i>A contains B</i>	<i>A (ii) B</i>	<i>A (ii) B</i> <i>A (ii) B</i>
<i>C between B and A</i>	<i>C abuts B</i> <i>C abuts A</i>	

Table 4.2: 3D relations modeled using lower level relationships

relationships, we can constrain the relationships between objects to be equivalent to the notions in the designer's head.

For instance, now the relationship that $A(f+)B$ and the relationship A touches B together constrain the configuration of the two objects as shown in figure 4-4. The relationship *touches* checks to see if the boundaries of the two bounding boxes actually touch each other. If not, a modification is suggested to the objects to make them touch. This is very similar to a contact detection scheme normally used in graphics to find out when two objects come in contact with one another.

The natural question is, if we are going to bother with a relationship such as A touches B directly, what use are the interval interval relationships? Unfortunately A touches B alone does not reserve enough discriminant power in itself to be of any use in specifying certain configurations (refer to figure 4-5). Its only use is as an augmentation to other constraints to restrict the number of feasible configurations.

With the above scheme, relationships which are intuitive equivalents of notions in the designers head can be easily instantiated by specifying them in terms of high or low level interval interval relationships and and augmenting them with the extra relations such as *touches* if necessary.

4.2.1 Critique

There are obvious questions about the advantages of such a scheme as above. We claim that there are significant conceptual and implementation advantages to this scheme. By defining a small set of primitive relationships, and clustering them to form arbitrary higher level ones, we can provide easy interpretations for many of the notions in the designers head. In case a ready made relationship is not available, it can be assembled easily from the primitive ones allowing for ease of abstraction.

In terms of implementation, the scheme corresponds very naturally to object oriented hierarchies with each primitive relationship corresponding to a very simple class. As will be explained later, the implementation is very natural to the description in the paragraphs above, and the low level operations required of each class are conceptually very simple.

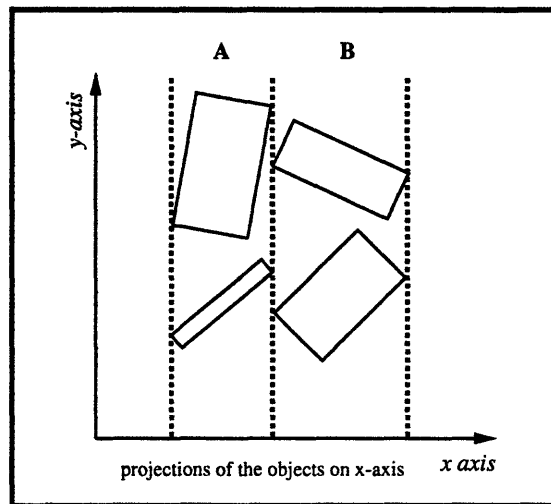


Figure 4-3: Even though the projections of the objects satisfy the relationship $B(f+)A$, the actual objects are not touching.

4.3 Evaluation

Given any configuration of the bounding boxes in 3-D space, we need to evaluate each constraint to decide if it has been satisfied. Evaluations are defined hierarchically such that they correspond naturally to the levels of relationships defined in the previous section. The following section explains the evaluation of each level in detail.

4.3.1 Evaluating Primitive Relationships

To evaluate whether a particular primitive relationship is satisfied, we must be given two intervals in space in terms of their starting and end points (minimum and maximum) along one dimension - a particular line. With each relationship, we shall identify a source which is the first interval in the relationship and a target which is the second interval. In a relationship such as A abuts B , the projection of A is taken to be the source interval and the projection of B as the target interval.

We define the most primitive evaluation in terms of the point interval relationships. As shown in chapter 2, there are only five qualitative relationships that can exist between a point and an interval. Given a point and an interval, we check to see if the relationship is satisfied; if not, we quantify the violation of the relationship by a number between 0 and 1. For each relationship, this scheme is illustrated in the figure 4-6. If the relationship is satisfied, the evaluation is 1.

We consider each interval interval relationship to be a combination of two point interval relation-

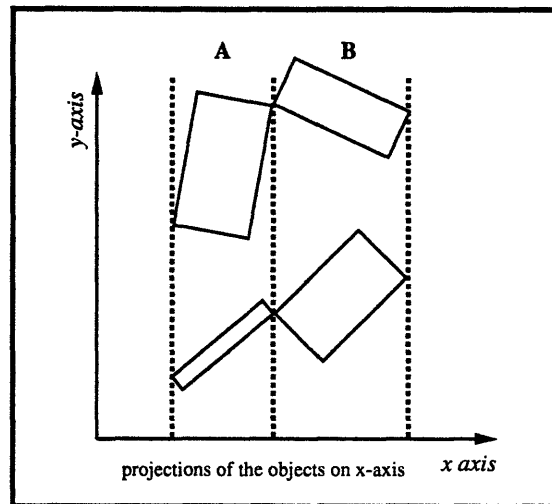


Figure 4-4: Together, the relationships $B(f+)A$ and A touches B are sufficient to ensure that the objects also touch.

ships and evaluate each one of them separately. Once for the point representing the maximum and once for the minimum of the source interval with respect to the target interval. Since the intervals are given to us in the form of their beginning and end points, this is a trivial exercise. The degree of satisfaction of the two point interval relationships is multiplied to give a composite measure of the degree of satisfaction of the interval interval relationship. It is clear that an interval interval relationship will be satisfied - it will have a measure 1.0, if and only if both point interval relationships are satisfied. Therefore, if the evaluation of a relationship is less than 1, we can conclude that the relationship has been violated.

4.3.2 Evaluating Disjunctions

A representative degree of satisfaction for the disjunctive class of relationships is obtained by evaluating each component of the disjunction and taking the degree of satisfaction of the best component - closest to 1 - to be its evaluation measure.

4.3.3 Evaluating 3-D Relationships

3-D relationships are evaluated by multiplying the evaluations of their components. If the components are primitive or disjunctive relations, the evaluation is carried out as above. If the components are other 3-D relationships, each of these 3-D relationships is evaluated first and the resulting eval-

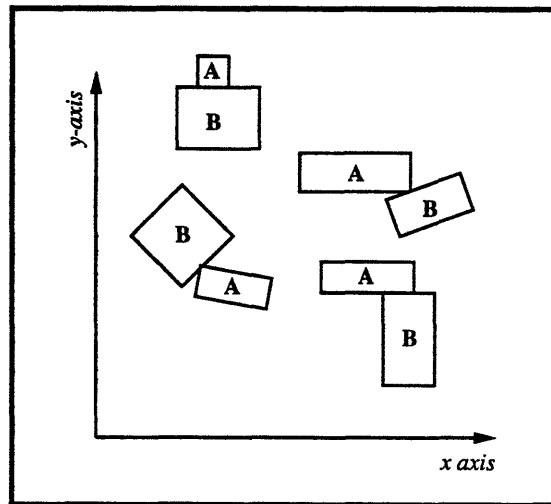


Figure 4-5: Alone, the relationship A touches B admits too many configurations to be of any use.

uations are multiplied to get a measure of the degree of satisfaction of the higher relationship.

4.3.4 Critique

The above method gives a quantification as well as a feel of both the degree of satisfaction and the degree of violation of point interval relationships. It also gives an unambiguous measure of the satisfaction of interval interval relationships. However, it is difficult to get a feel of how the composite measure of evaluation of an interval interval relationship relates to its degree of violation. For instance, given a measure of violation of 0.5 for a point interval relationship, it is easy to visualise how the relation is violated. On the other hand, given the same measure for an interval interval relationship, it is difficult to visualise all possible combinations of point interval violations that can result in a composite violation of 0.5. The difficulty is compounded in higher order relationships as the combinations get more complex.

On the other hand, we realized that it was only necessary to have a measure that allowed us to see if a constraint had been satisfied or not since the actual amount of violation does not really matter in the Ateam. All we need is to be able to suggest some form of improvement after we find a constraint has been violated. Also, note the ease of evaluation. In fact, only interval interval relationships need to be evaluated as all other evaluations can be obtained as combinations of these. We shall see how to specify improvements in the next section.

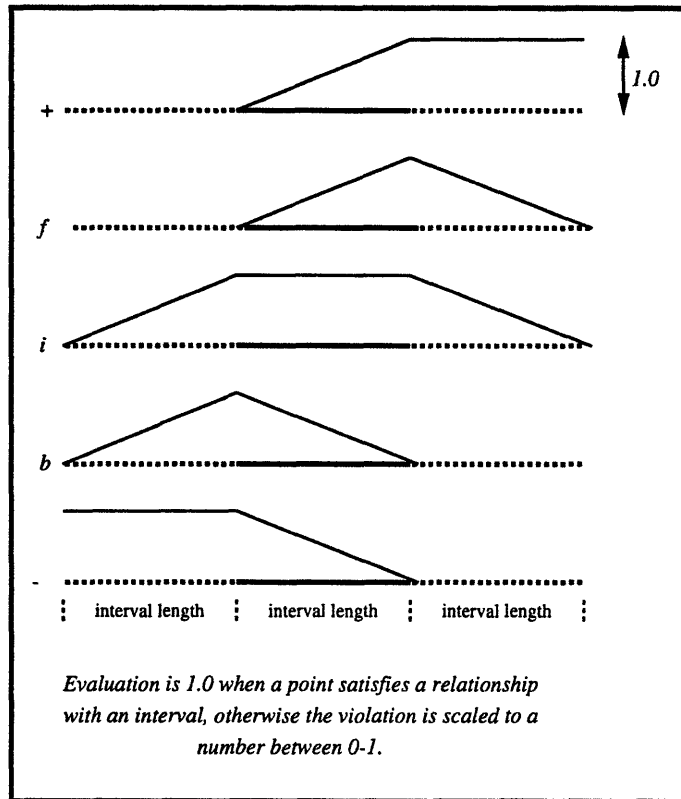


Figure 4-6: Evaluation function for point interval relationships

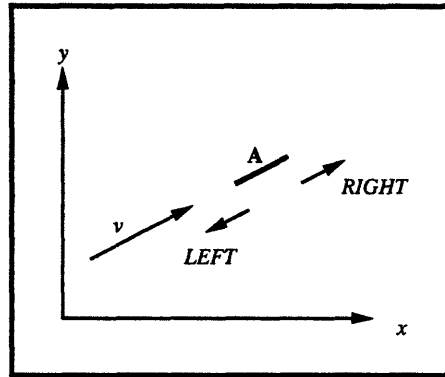


Figure 4-7: Left and Right modification operators in relation to a vector v

4.4 Improvements

In case a particular relationship is violated by a configuration, we would like to be able to specify improvements to the configuration such that the degree of violation of this particular relationship is reduced. In order to do this, we need to perceive a connection between the evaluation of each relationship and some form of an improvement or *modification operator*. We assume that we have a set of the following modification operators available.

{ **LEFT, RIGHT, SMALLER, BIGGER, CLOCKWISE, ANTICLOCKWISE** }

Where LEFT means left in relation to a particular direction specified as a vector in space. This vector is the defining direction of a relationship or its components. A left movement is movement in the negative direction of the specified vector. SMALLER and BIGGER refer to symmetric (about the centroid) reductions and enlargements respectively of intervals or bounding boxes. This is illustrated in the figure 4-7.

Upon evaluation of a relationship, a modification operator is associated with each of the objects involved in the relation. For instance, if the interval interval relationship $A(++)B$ is violated, we can specify two modification operators A-RIGHT and B-LEFT to improve it, meaning that we can either move A to the right or move B to the left or both. The operators are specified only for the primitive relationships. Operators for higher level relationships are built out of a collection of operators obtained from the lower levels. The explanation below elaborates on this.

4.4.1 Improving Primitive Relationships

Given two intervals, a primitive relationship, and its evaluation, we can easily specify how to improve the relationship using only the first four operators. For instance, consider the relationship $A(i+)B$ not satisfied. Then if the minimum of A is greater than the maximum of B, we can specify the operators A-LEFT and B-RIGHT, otherwise A-RIGHT and B-LEFT. Other operators may be

inferred from the original configuration of A and B. For instance, if the violation of A (*i+*) B is such that A (*if*) B is satisfied instead, we can say A-BIGGER or B-SMALLER; however, even in this case, we note that the original operators A-RIGHT and B-LEFT hold. In general, we will always try to specify the operators that are valid for a maximum number of cases. If a relationship is satisfied, no operators are specified.

4.4.2 Improving Disjunctions

For disjunctions, each component interval interval relationship is evaluated and some modification operators are associated with it. We compare the evaluation of all components and retain the modification operators associated with the least violated constraint. The rationale being: since only one component of a disjunction needs to be satisfied for the whole relationship to be justified, and if we want to achieve an improvement with a minimal effort, we should concentrate on improving the component with the least violation. Improving one component will have an unspecified effect on the other components, therefore we do not need to keep track of the other modification operators.

4.4.3 Improving 3-D Relationships

A set of modification operators for 3-D relationships is obtained as a union of all modification operators associated with each of its component relationships. No operators can be discarded since each 3-D relationship is a conjunction of relationships. In order to improve a 3-D relationship, we can pick up any member of the set and apply it.

4.4.4 Improving other Relationships

Relationships such as *is - perpendicular*, if violated, can only be improved by rotating the objects. For these, the operators CLOCKWISE and ANTICLOCKWISE are specified. These operators refer to the objects as a whole and do not have any natural correspondence to the modification operators for interval interval relationships. The rotation is assumed to be about the centroid of the object.

4.4.5 Critique

The notion of specifying operators for improvements instead of having a general algorithm for improving a constraint is needed because we want to be able to incorporate any type of constraint in any language in our framework. Having an operator scheme such as the one above allows us to have operators that correspond to the language in which the constraints have been modeled. For instance, suppose we had algebraic constraints for which we wanted to specify modification operators. We could build a set such as { INCREASE, DECREASE } associated with the variables. The only thing that is important is how the operators will be interpreted to change the configuration.

Although the modification operator set suggested by each relationship may not be complete in the sense that it may not cover all possible ways in which a relationship may be improved, we note that it is not necessary to have a complete set of operators always due to the nature of the search scheme. At any time, only one of the operators will be used to improve the configuration.

As we demonstrate later, this fact is borne out empirically as even incomplete operator sets do not exceptionally hinder the search process.

4.5 Implementation of QSRs

We have implemented the relationship scheme described above in an object oriented manner using C++. The hierarchy of relationships, the evaluations structures, modification operators etc. have a very natural interpretation in an object oriented paradigm. This section describes the details of the implementation.

We assume the existence of three C++ classes, the implementations of which are described in a later chapter. The generic class **GNvector** has been written for representation and manipulation of 3-D vectors. It supports all common vector operations. The class **Dobj** refers to a particular bounding box in a configuration. It knows about the position and orientation of the box in space and has methods allowing us to query any of its attributes; and to get the projections of the box on any vector in space. The class **Design** stores all relevant information about a particular configuration and has methods that allow us to get the projections of any two boxes we want on any arbitrary vector. These projections are the intervals we need to check the interval interval relationships.

4.5.1 Reference Frames

In order to specify a reference frame in space, we need to know the co-ordinates of its origin, and unit vectors in each of the three directions that constitute its axes. The vectors are specified with respect to a global frame of reference. In correspondence with the types of reference frames described before, we define a hierarchy of classes as shown in the figure 4-8.

The base class **Ref_frame** contains a **GNvector** that stores the origin of the reference frame. It has methods allowing us to retrieve or set the origin. In addition, it defines virtual functions that allow us to get the unit vector in the direction of any **GNvector** in the class.

The classes **Ref_axes** and **Ref_object** are derived publicly from **Ref_frame**. **Ref_axes** corresponds to an arbitrary reference frame in space and has an array of three **GNvectors** to store the three orthogonal directions of a frame. It allows us to get the unit vector along any one of these directions by means of the virtual function described above. **Ref_object** refers to the local axis system of a bounding box in space. It contains a pointer to a **Dobj** class from which it can retrieve the current location and orientation of the local frame of any object.

4.5.2 Evaluations and Modification Operators

The class **mod_operator** has three attributes; a constant specifying which of the operators { **LEFT**, **RIGHT**, **SMALLER**, **BIGGER**, **CLOCK**, **ANTICLOCK** } to apply, a pointer to a **Dobj** to know what object the modification refers to, and a direction in the form of a **GNvector** along which modification is required.

The **Evaluation** class stores the measure of evaluation of a particular relationship, and an array of objects of type **mod_operator** which are the modifications suggested for this evaluation. It has methods for adding and deleting modification operators from the array and for retrieving a random

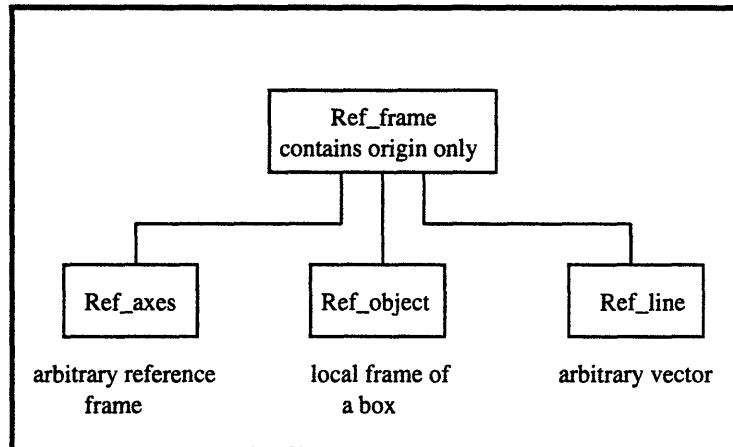


Figure 4-8: Class hierarchy of reference frames

modification operator. It does not store what constraint it is related to. All constraint evaluations are stored in a **Design** object and it is the job of this object to know which **Evaluation** object refers to which constraint.

4.5.3 Relationships

The class hierarchy of relationships corresponds very closely to the description in section 4.2. All relationships are derived from a base class **Srel** which is an abstract class defining a spatial relationship as shown in figure 4-9. This class stores unique identifiers of the two objects that are subject to the relationship, and a reference frame with respect to which the relation is specified. The polymorphic cluster associated with the hierarchy includes methods to: get the identifiers of the objects in a relationship, evaluate a relationship and improve it.

The class **QSR** which is derived publicly from **Srel** is the base class defined for the relationships formulated in the point interval algebra. The abstraction of this class allows the modeling of disjunctions, 3D relationships and interval relationships in a uniform manner.

The lowest building blocks of the system are the PI classes which correspond to the point interval relationships. They are derived from the **PI_base** class which is in turn derived from **QSR**. Each PI class has a virtual method to check if the relationship is satisfied. The method takes a point and a **Dobj** and returns a value between 0 and 1 if the point interval relationship is satisfied.

Each interval interval relationship is derived from the **II_base** class, subclassed from **QSR**. The **II_base** class consists of two point interval classes which form the relationship. The objects in a

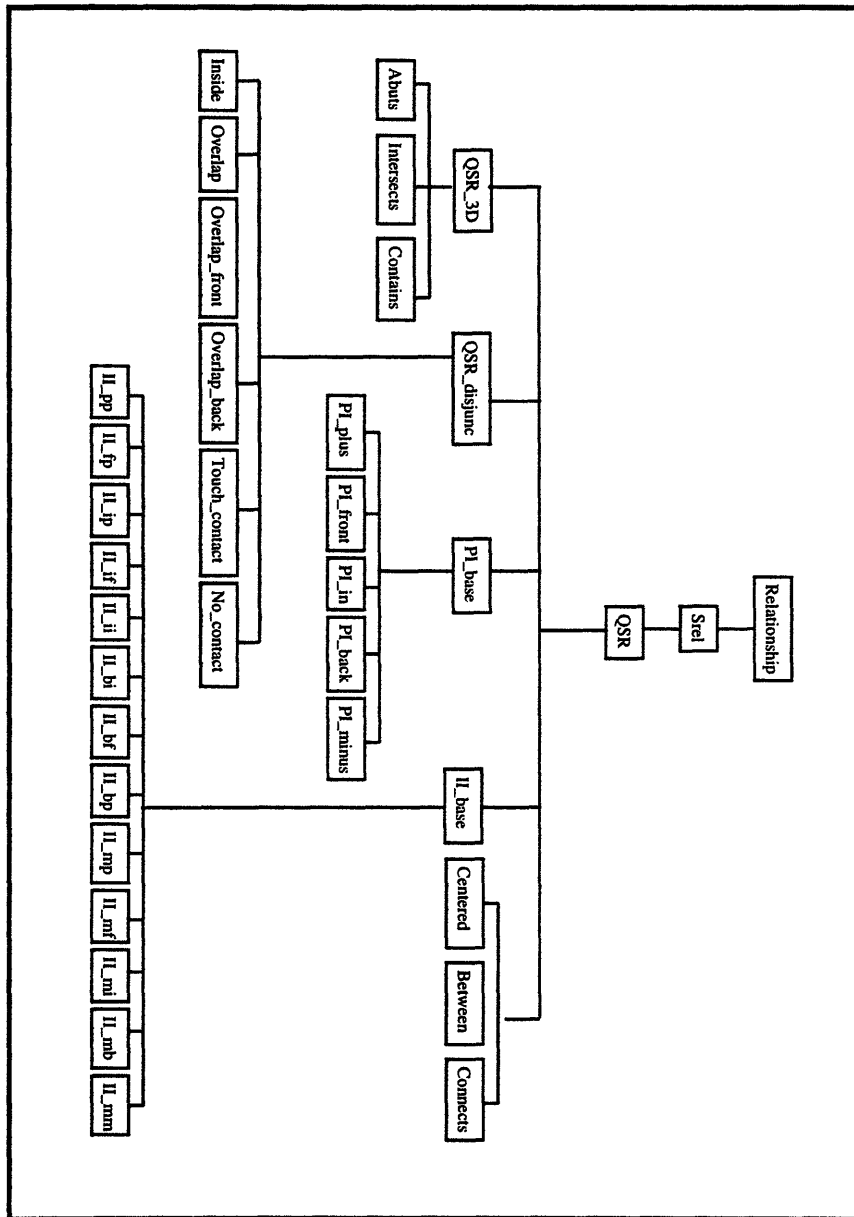


Figure 4-9: Class hierarchy of relationships. All classes are derived publicly from the base classes.

relation are inherited from the base class **QSR**. The virtual function *evaluate* for these classes takes a **Design** object and returns a pointer to class **Evaluation**, which contains the evaluation of the relationship and the suggested modification operators.

The class **QSR_disjunc** contains an array of objects of type **QSR**. These comprise the components of the disjunction. The size of the array depends on the number of components necessary for modeling the disjunction. The virtual function *evaluate* for this class calls the *evaluate* function for each component relationship and returns the **Evaluation** object with the maximum of the returned evaluations.

The class **QSR_3D** has an array of 3 **QSR** objects, one along each direction of a reference frame.

4.6 Summary

This chapter provided all the details for modeling qualitative relationships between objects. A scheme for building 3D relationships between objects was outlined. Techniques for evaluating these relationships were specified. Methods for specifying improvements to relationships by means of modification operators were detailed. All of the concepts were concretized in the implementation section, where high level details of the C++ classes were provided. The actual code for the header files can be referenced from the appendix.

The next chapter discusses a proposed scheme for handling arbitrary algebraic constraints in the Ateams framework. Details of the entire scheme are provided along with the description of a problem on which the scheme was initially tested.

Chapter 5

Algebraic Constraints

In order to make the framework complete, we must have a mechanism of solving for systems of algebraic equations using Ateams, since even at the conceptual design phase, the designer may want to specify some form of algebraic constraints into the program. This chapter proposes a scheme for incorporating algebraic constraints in Ateams.

Section 5.1 explains the problem. Section 5.2 explains a scheme for mapping an arbitrary expression to a parse tree. Sections 5.3 and 5.4 explain how to evaluate the constraints and specify modifications to reduce constraint violations. Details of the proposed implementation scheme are provided in section 5.5. Section 5.6 lists some issues that need to be addressed in this scheme and section 5.7 provides the details of a problem on which initial tests of the scheme were done.

5.1 The Problem

By a system of algebraic constraints, we shall assume the existence of a set of constraints of the following form among real variables, both discrete and continuous.

$$\begin{aligned}x_A + \log(y_B + x_A) - x_B &< 10 \\z_A + \sin(\log(y_B))/d &= 15.0 \\&\dots\dots\dots\end{aligned}$$

Each constraint or equation involves variables that are associated with any of the bounding boxes involved in the design. In such a case, the variable is subscripted with the identifier of the bounding box, for instance x_A is the variable x associated with bounding box A . If a variable is unsubscripted, such as d , it is unassociated with any object and may refer to anything the user desires. The relations permitted in each constraint are =, <, >, <=, >=. Note that all constraints above are written in the form such that only constants remain to the right of the relations. The only symbols allowed in expressions are variables, parentheses or operators such as \log , \sin , \tan , +, *, /, - etc.

Operator Precedence
$<, >, =, <=, >=$
$+, -$
$*, /$
$\sin, \cos, \tan, \log, \dots$
$() , [] , \text{exponentiation}$

Table 5.1: Precedence table

A problem involving arbitrary algebraic expressions such as the one described above is very hard to solve using conventional numerical processing techniques or non-linear programming algorithms. Therefore it is difficult to incorporate existing solution techniques to solve such systems.

On the other hand, note that in a randomized search algorithm such as Ateams, the only requirements are that given the values of all variables in the system, we are able to evaluate any constraint and suggest modifications to the variables involved in that constraint such that the degree of violation of the constraint is reduced. Therefore we only need to construct a scheme that allows us to perform evaluation and improvement for any constraint.

5.2 Mapping Expressions to Parse Trees

Consider every expression as a string of symbols where any string representing an algebraic expression is valid. Every symbol in the string that is not a variable, constant or parentheses is an operator. For instance, in the first expression of the example, we can consider the operators to be $+, \log, -, <$. We realize that by specifying a grammar and a precedence of operators for our scheme, we can decompose every expression into a parse tree. Consider the order of precedence of operators shown in table 5.1, and one rule: trees for all expressions within parentheses must be decomposed when there are no more operators left.

Then the following scheme allows us to form a parse tree given any expression. Scan the expression from left to right and pick up the highest precedence operator that we can find in the order that we find it. If the operator is binary, assign as its left child the sub-expression to the left, and its right child the sub-expression on the right. If it is unary, assign the sub-expression which is its argument as its only child. Repeat the operation with the sub-expressions until no more operators can be found. All parentheses must be decomposed only after they remain as the only child sub-expression.

With this scheme, we find that an expression such as $x + \log(y + x) - z < 10$ results in the tree shown in figure 5-1.

We also note that given an expression, the resulting tree is not unique and depends on the order of the symbols in the expression. For instance, the tree resulting from $a + b - c$ is different from the tree for $b - c + a$ as shown in figure 5-2. But we shall see that the uniqueness of the tree will have no bearing on the validity of our proposed scheme.

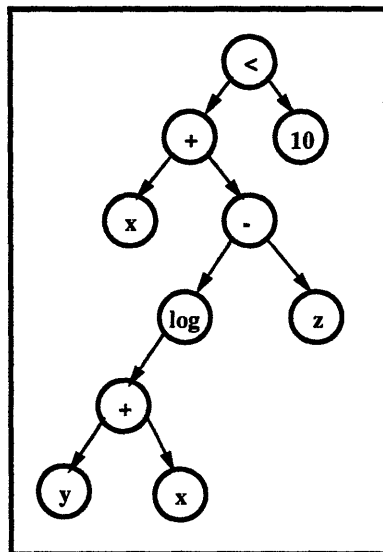


Figure 5-1: Parse trees for the expression $x + \log(y + x) - z < 10$

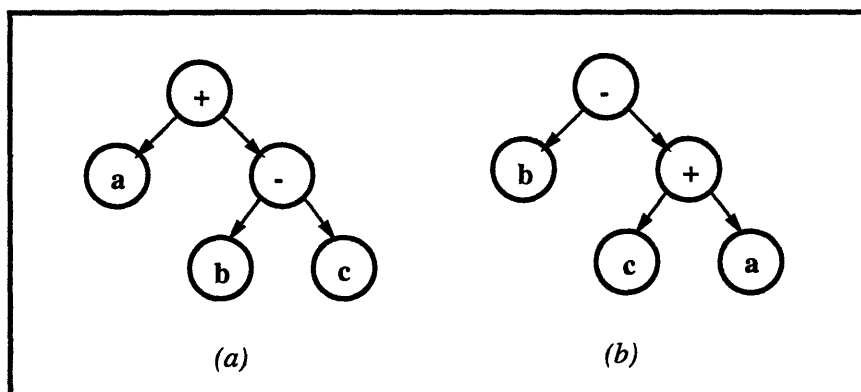


Figure 5-2: Parse trees for (a) expression $a + b - c$ and (b) expression $b - c + a$

5.3 Evaluation

We now argue that it is possible to associate an evaluation with each node of the tree and to determine in a relatively easy way an improvement to the variables such that the whole expression (a constraint) can be increased or decreased. The reasonability of this argument is shown in the following paragraphs.

Consider evaluation: assume that we are given values for the variables involved in a constraint. Then that constraint can be mapped to a parse tree. Every node in the tree is either an operator or a variable. Let us associate a number with every node and call it the evaluation of the node. Then the evaluation of a node that represents a variable is just the value of the variable. The evaluation of an operator is the value of the expression formed by concatenating the operator and its children. This evaluation can be determined if the evaluation of the children has been obtained. Therefore the evaluation of each node in the tree can be obtained by a simple inorder traversal.

5.4 Modification

For modification, we note that the root of a parse tree will always contain an expression as the left child and a constant as the right child. If a constraint is found violated by the root operator, the amount of increase or decrease to be made to the left expression is known. We need to show that this message can be propagated correctly to every node in the tree such that the actual modification made to the variables results in an increase or decrease in the net value of the lhs of the constraint.

Any increase or decrease with reference to an operator means increasing or decreasing the net value of the expression formed by concatenating the children and the operator itself: binary operators in the middle of the two children and unary operators before the child. Therefore to increase the net value of such an expression, we only need to know how to change the values of the child sub-expressions.

We make two trivial observations before we start: any modification can be made only to the variables and not the operators, and the modifications can only be of the form increase the value of the variable or decrease its value. The actual amount by which the variable increases or decreases depends on the nature of the variable, discrete or continuous, and the effect of an incremental change on the value of the expression containing it.

5.4.1 Sending and Interpreting Messages by Operators

Therefore the operators need to be able to suggest only a form of improvement to the variables. If we assume that each operator can store the latest evaluation of its child sub-expressions, then we must have each operator interpret an incoming message, such as increase or decrease, according to the most recent evaluation of its children, and send the message increase or decrease to its children based on the interpretation. The only nodes in the tree that will not propagate the message further will be the ones associated with the variables.

It is clear by the evaluation scheme mentioned above that it is possible to evaluate each of the nodes in the tree, therefore the parent can store the most recent values returned by its children. We show by examples below that it is possible to have an operator interpret an incoming message saying

increase or decrease correctly to be able to pass it down to its children. Later, we list all operators and show that it is easy to construct schemes that allow the above capability.

Consider the operator $+$. The evaluation of its children is irrelevant, since the only way to increase an arbitrary expression such as $expression1 + expression2$ is to send the message increase to each of the expressions $expression1$ and $expression2$. For instance, $expression1$ or $expression2$ could be both positive, both negative or one of them could be positive and the other negative. All of these combinations can be improved by simply increasing any or both of the expressions. Similarly, to decrease the expression $expression1 + expression2$, we need only to send down the message decrease to $expression1$, $expression2$ or both.

With another operator such as $*$, it might be more complicated. A message such as increase, sent to $*$ must be interpreted according to the most recent evaluations of its child expressions. For instance, if both children evaluated positive, the message increase must be sent to any one or both children to guarantee an increase in $expression1 * expression2$. On the other hand, if one child evaluated positive and the other negative, the message sent down to the positive child will be decrease, and the message sent to the negative child will be increase. For instance, assume we have $x = 4, y = -3$, and $x * y$ must be increased. Then decreasing x and increasing y will increase $x * y$. Similarly, if both children evaluate to be negative, then any one or both of them must be decreased to increase the net value of the expression. A decrease message sent to $*$ can be interpreted similarly.

For unary operators like \sin, \cos etc. we can take two approaches. For differentiable operators like \sin, \cos etc., we can evaluate the derivative at the value of the child expression. Then, if an operator receives a message increase and the derivative is positive at that evaluation, we send a message increase to the child. If the derivative is negative, we send the message decrease. In case the function is not differentiable, we can evaluate the function at the value of the expression plus an incremental change to determine how the function behaves with incremental changes in the argument. If the function increases with a small increase in the argument, we send the message increase to the child, otherwise decrease.

5.4.2 Interpreting and Returning Messages by Variables

Variables, on the other hand, need to be able to interpret an incoming increase or decrease message and return a message to the parent containing information about if the value can be changed in the requested manner and the amount of change if possible.

The interpretation of the messages by operator nodes ensures that the changes in the values of the variables are consistent with the change requested of the parent. By the recursive nature of the tree, it is clear that the result is therefore consistent with the request passed down from the root node and the expression can be improved.

The whole scheme therefore proceeds as follows. The parse tree is evaluated and the degree of violation of the constraint is determined. A message is passed to each node in the tree which returns a set of modification operators containing information about how to modify each of the variables. In the improvement step, a random operator is picked up from the set of modification operators and applied to the variable it refers to.

5.5 Proposed Implementation

The implementation of a scheme such as above is relatively simple. First we construct a class hierarchy as described below to represent variables and operators.

The hierarchy consists of a base class **Algebraic_Operator** with virtual functions *evaluate*, *increase* and *decrease*. It has two subclasses, **Unary_Operator** and **Binary_Operator**. Each of which contains pointers to the child operators.

All other classes except the class **Variable** are derived publicly either from **Unary_Operator** or **Binary_Operator**, the virtual functions *increase* and *decrease* being redefined for each class.

The class **Variable** contains a pointer to the class **Dobj** (described later), in case the variable is associated with a bounding box or a single value in case the variable is user defined.

The class **Modification_Operator** is used for storing information by a variable regarding the direction of change {**INCREASE**, **DECREASE**}, the amount of change possible and the variable it refers to. A **Modification_Operator** object will be returned by a variable that is requested to supply information about the change possible.

The actual algorithm would proceed as follows: all constraints will be mapped to parse trees, which will be instantiated using the above classes. When a random design is picked from the Ateams store, the values of the variables shall be assigned to the variable nodes in the tree. The tree will be evaluated. Depending on the evaluation, a message increase or decrease will be propagated through the tree and the modification operators returned by the variables will be stored. Any one of the returned modification operators will be randomly chosen to modify the value of the expression.

5.6 Critique

This section lists some of the issues that need to be addressed in implementing this scheme.

1. Every allowable operator must be coded as a class. In this sense, the scope of arbitrary algebraic expressions is limited to the number of operators that have been implemented at any particular time.
2. With function such as $\log(x)$, the domain knowledge that the argument cannot be decreased to a non-positive value must be incorporated into the scheme.
3. The actual amount of increase in the variables should be linked to the net change in the value of the expression. For instance, if the amount of increase required by the root operator is ϵ , we should ensure that none of the sub expressions are able to modify a variable to exceed the value.
4. Expression such as $x_A - x_A^2$ might be particularly troublesome. Since if a message increase will be sent to the variable x_A for the left child and decrease for the right child. However, we expect that this problem will be taken care of by the nature of the search, since we would expect the improvement to be the average of the improvements over a large population.
5. We have not addressed issues about the stability, convergence and performance of this scheme.

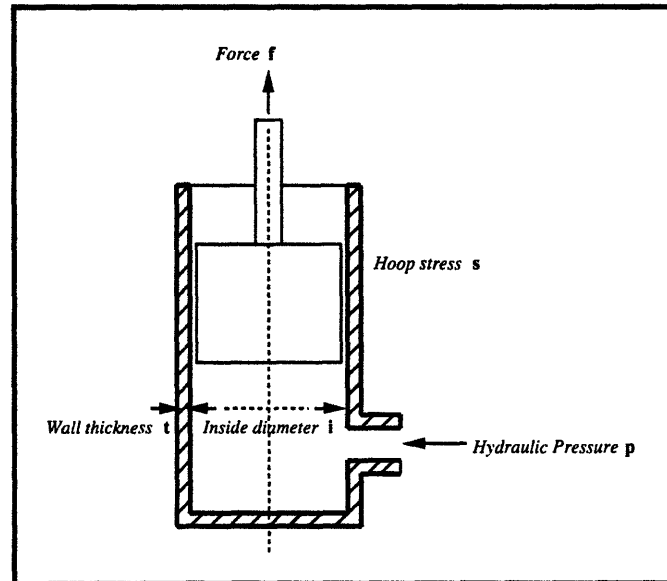


Figure 5-3: Illustration of the cylinder design problem.

5.7 Preliminary Testing

The above scheme was tested on one problem with the hope of discovering any gross oversight in our proposal. The problem is the design of optimal hydraulic cylinders formulated in a paper by Alice Agogino [CA].

The hydraulic cylinder design involves designing a thin walled pressure vessel with inside diameter i , wall thickness t subject to a hoop stress s induced by a fluid under pressure p pushing a piston with output force f . Refer to the figure 5-3. The interested reader is referred to [CA] for more details. Our primary purpose was to obtain a ready made algebraic formulation of a design problem for testing Ateams.

The formulation of the hydraulic cylinder design problem in terms of the above variables is given below.

Find f, t, p, s, i such that

$$f \geq F$$

$$t \geq T$$

$$p \leq P$$

$$\begin{aligned} s &\leq S \\ f &= \pi i^2 p / 4 \\ s &= ip / 2t \\ i &> 0 \\ p &> 0 \\ s &> 0 \end{aligned}$$

Where F, T, P and S are specified minimum or maximum values for the variables. This problem is not very hard for conventional algorithms. It has been successfully solved also using Monotonicity analysis [CA]. However, the primary purpose was to test the feasibility of our proposed scheme.

The constraints were hard coded into the algorithm. This avoided the use of parse trees for inferring modification operators. Parse trees are necessary only because the nature of the constraints supplied to the system is unknown. Therefore a general mechanism is needed to handle any possible type of expression. Given that the constraints are already known, the structure of the tree can be hardcoded into the constraint itself, implying that the knowledge about the modification operators is also hardcoded.

For each constraint, if it was satisfied, the evaluation was 1, otherwise, the evaluation decreased linearly to 0 till the violation of the constraint was 1000 or less. If the violation was greater than 1000, the evaluation remained 0.

A design object corresponding to the cylinder was created which stored the values of the five variables f, t, p, s, i and had the capability to interpret an incoming message such as increase or decrease and modify the corresponding variable accordingly.

Results indicate that the Ateams was able to find at least one feasible solution within 6000 evaluations. The experiment was repeated 10 times and the maximum number of evaluations taken by the Ateam to come up with a solution were 12000. These results lead us to speculate that our proposed scheme is feasible.

As will be shown in chapter 7, Ateams can also handle algebraic function optimization using the above scheme. Full implementation of the scheme has been left for a later stage.

5.8 Summary

This chapter provided the conceptual design of a scheme for incorporating algebraic constraints in Ateams. Details for representing arbitrary algebraic constraints as parse trees of expressions were proposed. Methods for evaluating such trees and suggesting modifications were outlined. The structure of evaluation and modification operators is similar to the one discussed in chapter 4. Details of a problem on which the scheme was initially tested were provided.

The next chapter provides implementation details of the classes used for the Ateams and the GAs.

Chapter 6

Implementation Details for Ateams and GAs

This chapter is concerned with the implementation details of Ateams. Section 6.1 provides a description of all classes and objects used and their relationships. The level of detail in the description of each class is confined to the most important pieces of data and methods. Naturally, the choice of what is important is ours. However, to fill in the rest of the detail, header files for all classes used are provided in the appendix. Section 6.2 explains the flow of the algorithm. Details of the parameters used in Ateams are provided in section 6.3. Section 6.4 explains the GA implementation.

The implementation is done using C++. The decision to use an object oriented paradigm for the system is inspired by the following observations. In a conceptual sense, the Ateams organization is inherently modular. It consists of agents working on a memory of solutions. Each agent is independent of the other and can therefore be considered to be an object. Ateams thus lends itself naturally to object oriented programming.

Moreover, since the approach to Ateams was experimental, it was desired that various parameters in the search process be easily varied. For instance, various combinations of modification operators, destroyers, etc. needed to be tried to determine their effectiveness. Modeling the system in an object oriented fashion allowed such flexibility by allowing the operators to be incorporated independently into the system when desired.

In addition, our experience with object oriented programs indicated that they are easier to maintain, modify and extend. Since the system will be undergoing extensions to incorporate algebraic constraints, easy extension was a primary goal. The ease of modification of object oriented programs is demonstrated by the fact - as shown later - that minimal modification was needed to use the objects from Ateams to link to Grefenstette's GA algorithm for testing.

6.1 Class Descriptions

The classes descriptions are grouped together for ease and clarity of presentation. Groups are based on class hierarchies or some natural association between the objects.

The first group consists of generic classes that have been developed for convenience. They are not particularly related to any one aspect of Ateams or GAs. For instance, such classes would include abstractions of a 3D vector, or a 3x3 matrix.

Other groups are more related to Ateams. One group provides an interface for Ateams to interact with the input constraints and objects. The other groups are structured so that one contains all classes concerned with the solution representation and storage, and the other contains the agents or the modification operators. A storage bin of operators is not associated with any of the above groups and is a one class group by itself.

6.1.1 Generic Classes

GN4vector

GN4vector class is an abstraction of a four dimensional vector. Its primary purpose is to facilitate the use of homogeneous co-ordinates in geometric transformations. It has four components and a protocol that supports all usual vector operations. It is used as a base class for the class **GNvector** - a 3-D vector.

GNvector

GNvector class is derived publicly from **GN4vector**. It represents a 3 dimensional vector. It has three components and supports vector addition, subtraction, scalar multiplication, cross and dot products, matrix multiplication, calculation of magnitudes etc.

GNtransmat

GNtransmat class is used to represent a 4x4 matrix. It is used to instantiate matrices for performing geometric transformations in 3-D. It contains an array of four **GN4vector** objects. The protocol includes facilities to instantiate translation, scaling and rotation matrices. All common matrix operations like addition, scalar, matrix and vector multiplication etc. are supported.

6.1.2 Interface Classes

These classes have been written to provide an interface for input to the Ateams program. These classes deal exclusively with input. Input to the program is in the form of a list of constraints of type class **ATconst** and a list of objects of type **Artifact** representing bounding boxes. Given a design, **ATconst** objects know how to evaluate the corresponding constraint and to send a message to the design about how that constraint can be improved. **Artifact** contains the identifying string for a bounding box and the initial fixed values for any configuration variables for that box.

Object_Bin

Object_Bin class is designed to store an array of objects of type **Artifact**, each with its own identifier, values for the configuration variables if supplied as input, and flags indicating the variables that are fixed. The primary purpose of this class is to ensure that any later change in the form of input objects does not affect a large portion of the code. Since other classes deal only with this class, the only modifications required in case of input changes are to the protocol and definition of this class. The protocol includes querying the number of objects and getting any particular object from the array.

Constraint_Bin

Constraint_Bin class is very similar to the **Object_Bin** class. It serves the same purpose for the constraints as **Object_Bin** does for the objects. Private data includes an array of **ATconst**s and their number. Protocol includes querying the number of constraints and getting any particular object of type **ATconst**. It also includes identifying the index number of any constraint given a pointer to an **ATconst**.

6.1.3 Solution Representation and Storage Classes

Dobj

Dobj class is the representation of a bounding box or a prism. It forms the atomic parts of a design solution. Each object has a string identifier, an array of nine values containing the configuration variables of the box. It also contains an array of four **GNvectors** for storing the information about the local reference axis system of the object - one for the origin and three for the unit vectors x, y, and z relative to the global frame. And an array of eight **GNvectors** for storing the co-ordinates of the eight points of the box. Note that although all the information about the reference axis system and the corners of the box can be inferred from the configuration variables, storage of these values avoids unnecessary computation every time these values are desired. Also, there is an array of flags containing information about which configuration variables are fixed and which are variable. For arbitrary algebraic variables, there is an array of strings representing the variable names and an array of corresponding variable values.

The constructor set includes the default and copy constructors, and a constructor that initializes all variables to the values passed in - from the corresponding **Artifact** in the **Object_Bin**. Unspecified values are initialized to random numbers.

The protocol includes convenience functions for accessing and modifying the values of private data. However, the most important parts of the protocol are the following. A method returns the starting and end points of the projection of an object on any line specified as the difference of two 3-D vectors representing its origin and end. This is useful in the evaluation of point interval relationships as discussed in chapter 4. A second method allows improvement in the values of the configuration variables. It takes a direction vector and an improvement type, which could be any of **LEFT**, **RIGHT**, **CLOCK**, **ANTICLOCK**, **SMALLER**, **BIGGER**, **UNDEFINED**, and changes the values of the configuration variables accordingly if possible. Thus given a suggestion for

improvement, every object knows how to improve itself.

Design

Design class is the representation of one design solution, not necessarily feasible. Each **Design** object stores an array of **Dobjs**. It has information about the number of objects in each design, the number of constraints, and an array of **Evaluation** objects - described in chapter 4 - one for each constraint.

Apart from the default and copy constructors, another constructor instantiates a random design solution. Its arguments include the **Object_Bin** and **Constraint_Bin**. For each object in the **Object_Bin**, a corresponding **Dobj** is created in the design solution by calling the random constructor for each object.

The protocol includes methods for accessing any particular object from the array of **Dobjs** using the name of the object which is a string representation. Methods exist for accessing and modifying private data including inserting and deleting particular **Dobjs** and **Evaluations**. Important protocol allow the retrieval of an **Evaluation** object corresponding to any constraint, checking if one design is the duplicate of another (all configuration variables are the same), and getting the sum of the evaluations of all constraints.

HashTable

HashTable class is the implementation of a hash table for storing **Design** objects. Designs are hashed according to the sum of their evaluation for each constraint. The primary goal is minimizing the overhead in storing and accessing design solutions. The hash table uses separate chaining and therefore also provides a mechanism for grouping together designs according to their hash keys. Class data includes a hash table and an array of designs. The array acts as a bin of designs. Random designs are selected from this array. It is used for ensuring that when a random design is requested, all designs have the same probability of being picked.

The protocol comprises of usual hash table functions, including adding and deleting designs from the table. Other important methods include getting any random design from the table with an equal probability, getting a list of the best designs in the table, and checking for duplicates.

Memory

Memory class represents the abstraction of the common memory used in Ateams. It consists of a **HashTable** for storing designs, the number of designs with which to populate the initial store, and the evaluation of the best design found so far.

A constructor creates random design solutions from the **Constraint_Bin** and the **Object_Bin**, and inserts them into the hash table to initialize the store. Other protocol includes adding and deleting designs, getting a random or a specific design from the hash table, getting the number of designs in the store, getting the best designs list, and checking if a duplicate for a given design exists in the store.

For clarity, the containership of the classes is shown in figure 6-1.

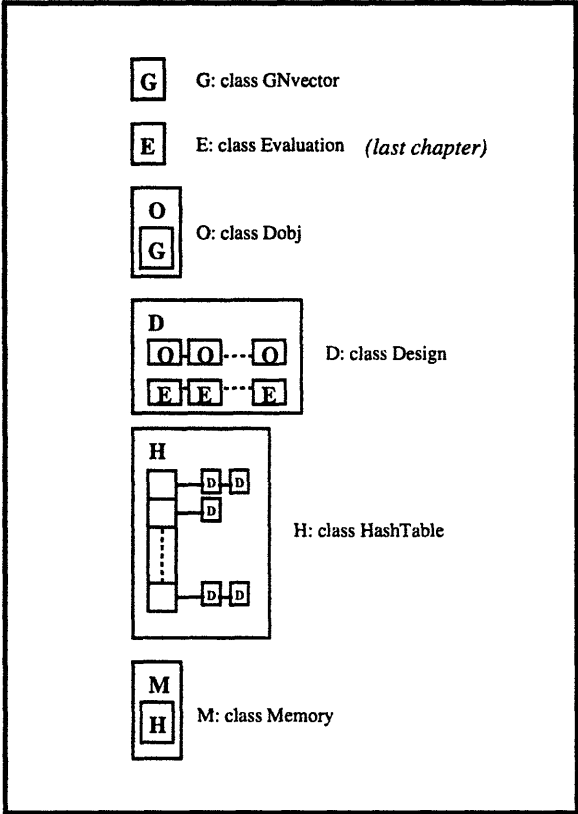


Figure 6-1: A schematic illustration of the important containership relationships between classes

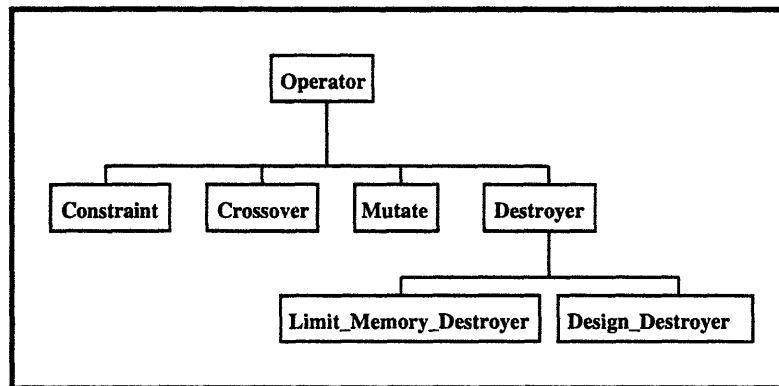


Figure 6-2: Class hierarchy of operators

6.1.4 Operators

The operators represent agents in Ateams. An operator requests a design from the store of designs and modifies or destroys it accordingly. The class hierarchy of operators is shown in figure 6-2.

Operator

Operator class is an abstract class from which all other operators are derived. It is convenient for providing a uniform interface for all operators. Protocol includes a virtual method for firing an operator. All classes described below are derived publicly from **Operator**.

Constraint

Constraint class represents the design constraints input to the Ateams. Data includes a pointer to **ATconst**, the incoming constraint object as described earlier. Member methods include evaluating a particular design, and improving it. The method **fire** encompasses the entire operation of getting a random design from memory, copying it, improving the copy and evaluating it before adding it in the store.

Crossover

Crossover class represents the crossover operator analogous to the one used in genetic algorithms. Protocol includes method to get two random designs from memory and mate them. Mating takes place by picking a random object and swapping all objects after and including this one with the other design. The mated designs are evaluated before putting in the store.

Mutate

Mutate class is the analogue of the mutation operator in genetic algorithms. The relevant method gets a design from memory, copies it, chooses a random object in the copy, a random configuration variable in that object, and assigns a random value to it. The design is evaluated before inserting in the store.

Destroyer

Destroyer class is a base class for the following classes. As above, it is derived from the Operator class. Destroyers are used to destroy bad designs with a given probability. They are necessary for controlling the size of the population and to make sure that the effects of the other operators are concentrated on the most promising designs.

Design_Destroyer

Design_Destroyer class has protocol for requesting a design from memory and deleting it with a probability based on its overall evaluation.

Limit_Memory_Destroyer

Limit_Memory_Destroyer class includes a method for going through the entire store and deleting designs with a probability based on their evaluations until the size of the store reaches a certain minimum limit. A combination of this and the design destroyer was found necessary to control the design population. The **Design_Destroyer** operator is fired as a regular operator and therefore prevents the store size from blowing up. However, it is insufficient in itself to control the memory unless fired with a very high frequency. Unfortunately this would result in a lower frequency for the other operators thus limiting the improving action of the algorithm. Therefore at periodic intervals, the **Limit_Memory_Destroyer** operator is fired to bring the population under control.

6.1.5 Bin of Operators

Operator_bin

Operator_bin class is the implementation of a conceptual bin of operators. Data includes an array of operators for storing mutation and crossover operators, destroyers, and constraints, and an array of frequencies with which each operator must be fired.

The constructor takes a **Constraint_Bin** and the frequency of firing of each operator. Protocol includes methods for getting a random operator according to the specified frequency. And for evaluating a particular design by sending the evaluate message to each of the constraint operators.

6.2 Algorithm

The Ateams algorithm proceeds as follows. A list of constraints containing objects of type **ATconst** and a list of boxes of type **Artifact** are supplied as input. A **Constraint_Bin** object is instantiated from the list of constraints and an **Object_Bin** object is instantiated from the list of boxes.

An **Operator_bin** object instantiates all operators to be used in the algorithm after reading constraints from the **Constraint_Bin**. The probability of firing for each operator is specified at the time of instantiation of the **Operator_bin**.

The **Memory** object reads from the **Object_Bin** and **Constraint_Bin** to get information about the number of objects and their attributes if any, and the number of constraints, and instantiates and inserts **Design** objects in the **HashTable**. The size of the population is specified at the time of instantiation of the memory.

The main loop of the algorithm is then started, in each iteration, a random operator is picked from the **Operator_bin** and fired. In general, the firing of an operator involves picking up random design/designs from the memory, duplicating them, performing the related modification on the duplicates, evaluating them, and inserting them in the memory. In case of destroyers, the designs may be destroyed and will therefore not need to be inserted back into the memory.

After every 1000 iterations, the **Duplicate_Destroyer** and **Limit_Memory_Destroyer** are fired to strip the memory of duplicates and to reduce its size. Operator firing continues until a prespecified number of iterations have been reached or a feasible design/designs have been found.

6.3 Details

Discretization

Most search algorithm use some form of discretization of the search space. For our problem, we assume that the K-space of the configurations variables is discretized as follows. The $x, y, z, \theta_x, \theta_y, \theta_z$ variables can assume values only from 1 to 10 and $size_x, size_y, size_z$ can assume even numbered values from 1 to 10 only. The choice of the number of discretizations is arbitrary and is guided by the observation that our primary purpose is to compare GAs and Ateams, and therefore the actual number does not matter as long as the same space is used for both.

Probabilities involved

At many places in the algorithm, probabilistic steps are taken depending upon certain criteria. This section provides the details of the probabilities involved.

It should be noted that the evaluation of each constraint lies in the interval $[0, 1]$, where 1 indicates that the constraint is satisfied and 0 indicates a very large violation. Therefore, given the

number of constraints, the scaled evaluation of a design can be obtained by dividing the sum of the evaluations of all constraints by the number of constraints. Clearly, the scaled evaluation is will also lie in $[0, 1]$, with a design being feasible if and only if the scaled evaluation is 1. All references to design evaluations below should be taken to be scaled evaluations.

The probability of picking up any design from the memory is independent of the evaluation of the design and is given by $1/n$, where n is the number of designs in memory at that particular time. There is no hard justification for making the probability $1/n$ apart from that it is easier to implement. Ostensibly, there is no case against a dual scheme in which a design is picked for improvement based on its evaluation and is picked for destruction with a probability $1/n$ or inversely proportional to its evaluation. We stuck to the ease of implementation as a first cut at experimenting with the system.

The probability of destroying a design picked up by the **Design_Destroyer** is 0 if the design has an evaluation greater than 0.75. If the evaluation is less than 0.75, it is destroyed with a probability of 0.8. The main purpose is to ensure the diversity of the population by retaining some bad designs. However, there is no reason for picking up the specific values that we have chosen. These values were empirically found to behave better in our initial runs and we stuck to them.

The limiting memory destroyer, goes through every design in the memory when fired, and destroys is with the probability $1 - \text{evaluation}^3$. The choice of this probability was based on the observation that it provides a sharp discrimination between the probability of destruction of good and bad designs owing to the sharp slope of the cubic function near 1 as contrasted to its behavior near 0. When a duplicate destroyer is fired, duplicates are destroyed with probability 1.

Elitist Strategy

All through the algorithm, the elitist strategy is maintained. That is, we never destroy the designs with the best evaluation in the store.

While initializing the memory, designs having an evaluation less than 0.2 are destroyed. The initial store therefore contains only designs with evaluations greater than 0.2.

Limiting Size of the Memory

In initial experiments with the system before the implementation of the **Limit_Memory_Destroyer** class, it was found that the size of the memory was very difficult to control using only design destroyers. At low frequency of design destroyer firing, it was found that the store tended to grow unchecked. This reduced the effect of the modification operators since the probability of picking up designs is inversely proportional to the number of design in the store. Experiments with increasing the frequency did not work out since that reduced the frequency of firing of the modification operators. Further, the behavior of the system was not stable as the high destroyer frequency sometimes deleted too many designs and reduced the store to unacceptably low levels.

Therefore, the **Limit_Memory_Destroyer** was implemented. The design destroyers are now fired with a reasonable frequency, in fact 0.4. This keeps the store in check but is not enough, therefore every 1000 iterations, the limit memory destroyer goes through the memory and probabilistically - see above - deletes designs until the size of the store is reduced to its original level. The

store therefore periodically expands and contracts every 1000 iterations. This scheme was found to be very effective in concentrating the efforts of the modification operators on the good designs. In fact, we conjecture that this may be better than controlling the size of the memory too strictly, since the expansion may allow the introduction and retention of worse designs longer thereby guarding against premature convergence.

6.4 Implementation for Genetic Algorithms

The code for Genetic algorithm has been implemented using Grefenstette's GENESIS version 5.0 system. This software is available via ftp. This section describes the characteristics of the GENESIS system and the details it has been augmented with.

6.4.1 GENESIS

This description is based on the "A User's Guide to GENESIS" 1990, by Grefenstette available with the software. GENESIS is an experimental system provided to encourage use of GAs for realistic optimization and search problems. The system supplies the entire code for a GA and the user only has to provide an "evaluation" function that returns a value when given a point in the search space. The implementation language is C. Below is the algorithm and specific details relating to GENESIS.

```
begin
  t = 0
  initialize Population P(t)
  evaluate strings in P(t)
  while termination not satisfied
  begin
    t = t + 1
    select P(t) from P(t-1)
    recombine strings in P(t)
    evaluate strings in P(t)
  end
end
```

A Genetic Algorithm

Representation

GENESIS provides three levels of representation for the structures it uses. The most flexible of these is the binary string representation in which each string is represented by a string of 0's and 1's having length L . This representation allows the user to give arbitrary meaning to the genetic structures.

Initialization & Selection

The initial population is chosen at random but a facility for seeding the initial population with heuristically chosen points in the search space is provided. During each iteration, called a generation in GA, the current population is evaluated and on the basis of that evaluation, a new population is formed. The selection procedure is based on an algorithm by James Baker and ensures that the expected number of times a member of the population is chosen is proportional to the member's performance relative to the rest of the population.

Recombination Operators

Genetic recombination operators include "crossover" and "mutation". Crossover proceeds by random selection of two points. Mating occurs by randomly selecting the point of crossing. The rate of crossover, that is, the expected number of offspring chosen for crossing, can be varied.

The "mutation" operator is applied to the whole population after selection. The rate of mutation can be changed by the user. The frequency of mutation is determined by computing an interarrival time between mutations, assuming a mutation rate is specified. Each string in the population is offered a chance for mutation based on the frequency. If a bit position must be mutated, a random value is chosen from {0,1} for that position.

Termination

Termination may occur by fixing the total number of iterations, finding an approximate feasible solution, or some other application dependent criteria.

Evaluation

The evaluation procedure is the only thing the user has to supply. GENESIS expects to link up with a C function with the following prototype.

```
double eval(char *str, /* string representation */
            int length, /* length of bit string */
            double *vector, /* floating point representation */
            int genes /* number of elements in vector */
            );
```

The code within the function is unrelated to GENESIS and can be entirely application dependent. Since we used only the string representation, the last two arguments of the function were irrelevant to us.

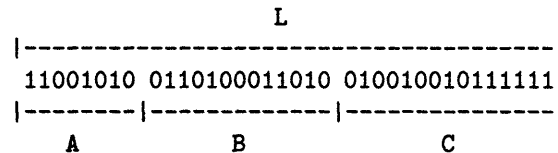
Important Variable Parameters

GENESIS allows the user flexibility in the specification of certain parameters. The description of the variables relevant to us is given below.

- 1) The length of the string.
- 2) The number of independent optimizations of the same function.
- 3) The number of trials per experiment.
- 4) The population size.
- 5) The crossover rate.
- 6) The mutation rate.

6.4.2 Representation of a Design in GAs

Since the GA will be working with binary string representations only, we need to provide a mapping from a string representation to a valid design. We have implemented a simple mapping of the form shown below.



Consider a design consisting of three bounding boxes A, B and C. Each box has nine parameters, $x, y, z, size_x, size_y, size_z, \theta_x, \theta_y, \theta_z$ out of which some may be fixed and some variable depending upon the initial constraints. We assume that each parameter $x, y, z, \theta_x, \theta_y, \theta_z$ requires four bit positions of the string and each $size_x, size_y, size_z$ requires three bit positions. Also we assume the string to be the representation in the base 2 of the parameter value. This means that each parameter $x, y, z, \theta_x, \theta_y, \theta_z$ can vary from 0-16 and each parameter $size_x, size_y, size_z$ can vary from 0-8. If we assume that a parameter that is fixed does not need to be represented in the string, the length of string required to represent all the parameters of an object varies depending on how many and which of its parameters are fixed.

Therefore, depending on how many objects are present in a design and the variable parameters of each object, the total length of a string required to represent a design will be different. The advantages of having a scheme in which the length of a string is dependent on the number of fixed variables is that the dimensionality of the search space is reduced by reducing the length of the string. In a way, therefore, some of the constraints are hardcoded into the representation.

6.4.3 Modifications to the classes for linking to the GA

The classes used for Ateams have been adapted with minimal modifications for use with the GA algorithm. In fact, since the only relevant operation in the GA to be supplied by us was evaluation of a design, we required only a subset of the classes described before. Specifically, this meant that the modification operator classes **Crossover**, **Mutate**, **Destroyer** and its children were not needed at all. The modifications were required for only the **Dobj** and **Design** classes and are listed below.

Dobj

Two modifications were required. One, the addition of a data element that stored the number of characters in a binary string that were needed for representing the parameters of a box. And two, the addition of a member method which when given a string of length greater than or equal to the length required for the object, interpreted the string and stored the relevant values of the nine parameters of the box. This method allows us to instantiate objects of type **Dobj** in which if arbitrary parameters of a box are fixed, they are not required to be represented in the string representation of design. The method for interpretation returns the number of bit positions used up by the parameters of an object.

Design

The **Design** class required the addition of a member method which takes a binary string as an argument and passes it down to the objects one by one. Through this function, a given string is mapped to the parameter values of objects after which the same method for evaluating the design can be applied that is used for Ateams.

The following section explains how the evaluation function for the GA is written using these modified classes.

6.4.4 Evaluation Function

The first time the function *eval* is called, it instantiates a **Constraint_Bin**, an **Object_Bin** and an **Operator_bin** from the supplied lists of **ATconsts** and **Artifacts** exactly similar to the Ateams algorithm. A *static Design* object is also instantiated in which the parameter values for all objects are set to zero unless they are required to be fixed to some other value.

The string supplied to the *eval* function is interpreted by the design. The design is evaluated and the scaled evaluation of the design is returned to the GA.

6.5 Summary

This chapter presented details of the implementation for both Ateams and GAs. High level class descriptions were provided. More detail is provided in the appendix. Explanations of the implemented algorithms and details of the parameters used for both algorithms were given.

The next chapter provides a detailed comparison of Ateams and GAs on artificially modeled search spaces. It explains the essential characteristics of blind search techniques. Metrics for the comparison of Ateams and GAs are identified and the results of the tests are provided.

Chapter 7

Comparison of Ateams with Genetic Algorithms

This chapter compares Ateams and Genetic Algorithms as search techniques. The main purpose of the comparison is to find out if Ateams merit research effort as a viable alternative to genetic algorithms. We feel that given this first implementation of Ateams, if they exhibit performance comparable to GAs on a set of test problems, we might be able to refine them to produce better performance.

The chapter begins by defining a context which makes the comparison relevant. Section 7.1 provides an introduction to conventional search algorithms. Section 7.2 explains the factors affecting the performance of search algorithms by constructing typical search spaces. Metrics for comparison between Ateams and GAs are defined and justified in section 7.3. Sections 7.4 and 7.5 explain the testing methodology and the parameter set used. The suite of test problems is defined in section 7.6. Sections 7.7 and 7.8 present the results and conclusions.

In order to develop a context for the comparison, we must provide answers to the following questions. In the context of the comparison: why compare Ateams to GAs only and not other search techniques? And in the context of the application in which we are using Ateams: why use black box search for constraint satisfaction instead of classic search techniques like backtracking etc.? Both these questions are addressed below.

We claim that classic search techniques are not suited to our problem [Sri94]. Therefore the only serious competitors to Ateams for solving the conceptual design problem are techniques like GAs which have empirically proven effective over very difficult problem domains. If our task was to present a comparison of the prowess of all search techniques, it would have been relevant to compare Ateams to other search methods. However, if we are to focus only on techniques that can solve our design problem, then the only methods worth looking at are GAs and Ateams.

Regarding the second question, classic constraint satisfaction techniques were found to be ill-suited to the nature of our application. The exploration of a design space requires that the designer be presented with a set of alternatives which satisfy the constraints. Techniques like classic and

heuristic backtracking typically find one, if any, solution to a set of constraints. Generation of each extra alternative requires the whole algorithm to be run again. Naturally, the performance of these techniques is not adequate for our problem. In contrast, Ateams naturally find more than one solution, if any to the set of constraints with minimal extra effort. Although there is no natural correspondence between the complexity of one run of Ateams against one run of a backtracking algorithm, on the average we expect the real time required in Ateams to be less.

7.1 Search Problems and Algorithms

The “black box function optimization” problem requires a search strategy to find the function extrema without knowing the function structure or the range of possible function values. Constraint satisfaction problems can be framed as “black box function optimization” problems by assigning a penalty to constraint violations, thereby ensuring that any solution that satisfies all constraints is the optimal one.

Search strategies can be classified as point based or population based [Sri94]. Point based strategies like hillclimbing, simulated annealing, etc, pick a point in variable domain and decide on a direction in which to move based on the evaluation of a number of points in the neighborhood of the current point. With deterministic algorithms like hillclimbing, the algorithm moves to the best point in the neighborhood of the current point. With stochastic strategies like simulated annealing, the next point is picked probabilistically based on the evaluation of the points. Population based strategies like genetic algorithms maintain a population of candidate points. From the population, the best points are picked in a randomized fashion that allows points with the best evaluations and their combinations to be more likely to be picked up. In this way, the population of better point increases until the optimum is found.

Ateams can be viewed as a population based randomized search strategy. In fact, if we pick up a point in the population and attempt to improve only one constraint at a time, we can consider our problem as a “black box multiobjective optimization” problem with Ateams acting as a “local hillclimbing strategy”.

A question that must be addressed is: are Ateams merely a modified form of GAs with knowledge incorporated into the operators? Our answer is no. There are fundamental differences between the two techniques. Ateams has flavors of Genetic algorithms in that it is population based and operators such as crossover and mutate, which are normally associated with GAs, can be easily incorporated into it. It also has flavors of randomized biased search and simulated annealing in that sometimes points with bad evaluations are kept around to maintain a chance of jumping out of local minima. Consider, however, the important differences in the mechanics of Ateams and GAs. In GAs, the an entirely new population is produced in every generation and recombined. The old structures are thrown away. In Ateams, the existing points are as likely to be kept as the new improved points. There is no inherent concept of producing a newer better set of points in each iteration. In fact, the improvement of the population is more diffused process than GAs.

Also, in our simple implementation, some very significant differences between Ateams and GAs become clouded. Since Ateams are supposed to be modular organizations of agents which can be complete algorithms in themselves, it results in tremendous flexibility by allowing the use of existing

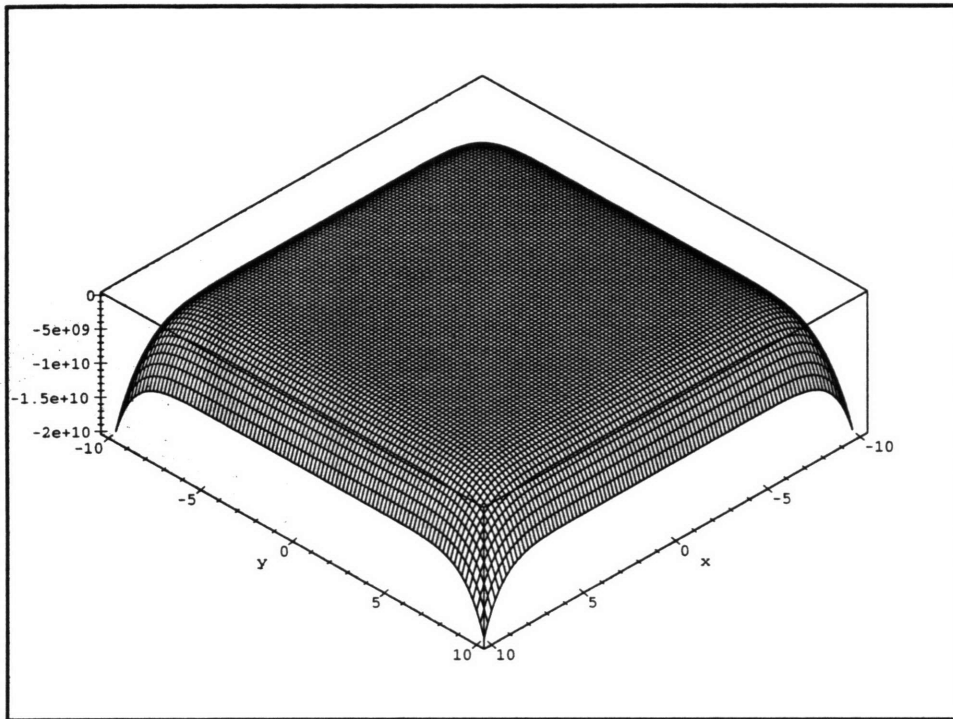


Figure 7-1: A mostly maximal space

techniques as part of a larger organization. Obviously this is impossible to do with GAs. We hope to include established algorithms such as backtracking etc. as agents in our Ateams implementation in future research as our system matures.

On an intuitive level, one function optimization with Ateams can be visualised from the mechanics of the algorithm. A population of points is picked in the variable domain. In each iteration of the algorithm, a point is picked at random. With a certain frequency, an operator such as improve or destroy is picked up and applied to the point. If the destroy operator is applied, the point is destroyed with a probability depending on its evaluation. If the improve operator is picked, a copy of the point is made and improved in the most promising direction in the immediate neighborhood of the point. After a fixed number of operator applications, every point in the population is evaluated for destruction and destroyed probabilistically until the number of points is equal to or less than the initial size of the population. Visually, in 3D space, we start with a population of points. At the end

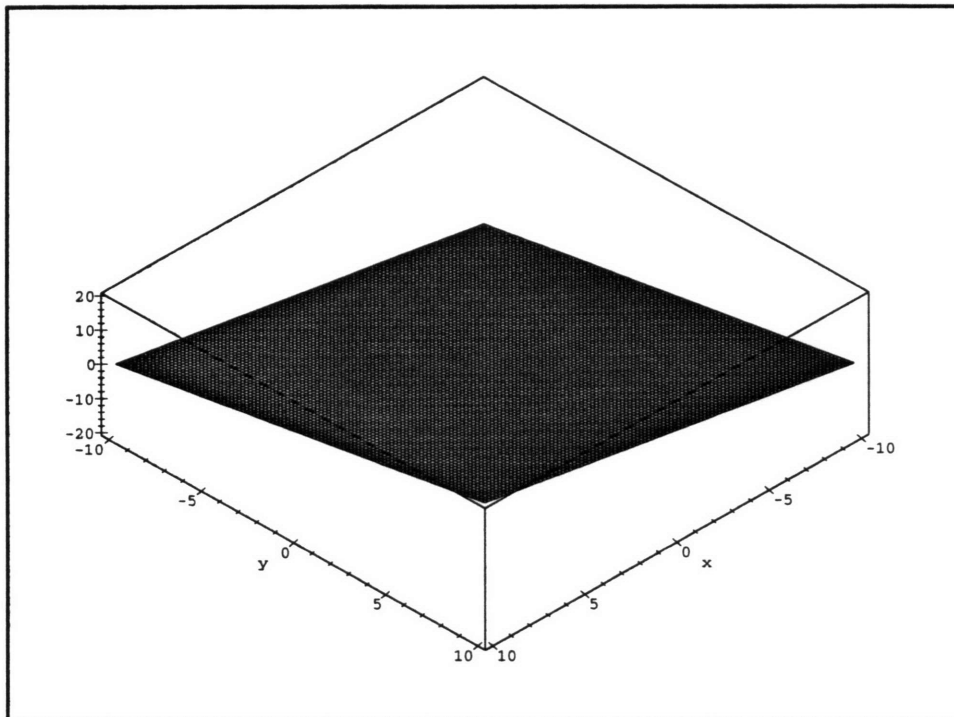


Figure 7-2: A linear space

of fixed number of iterations, each point will either have stayed fixed, or have generated improved offspring exploring a small neighborhood in its immediate vicinity, or will have been destroyed. A reasonable analogy is expanding balls around points in which each ball contains the point and its offsprings until the entire population is purged shifting the center of the balls to points which have better evaluations on the average.

7.2 The Nature of the Search Space

If the structure of the function to be optimized is known *a priori*, some form of strategy can be devised that is very good for that class of problems. For instance, if the function is assumed to be linear, it is clear that a hill climbing strategy will find the solution to the problem is no more

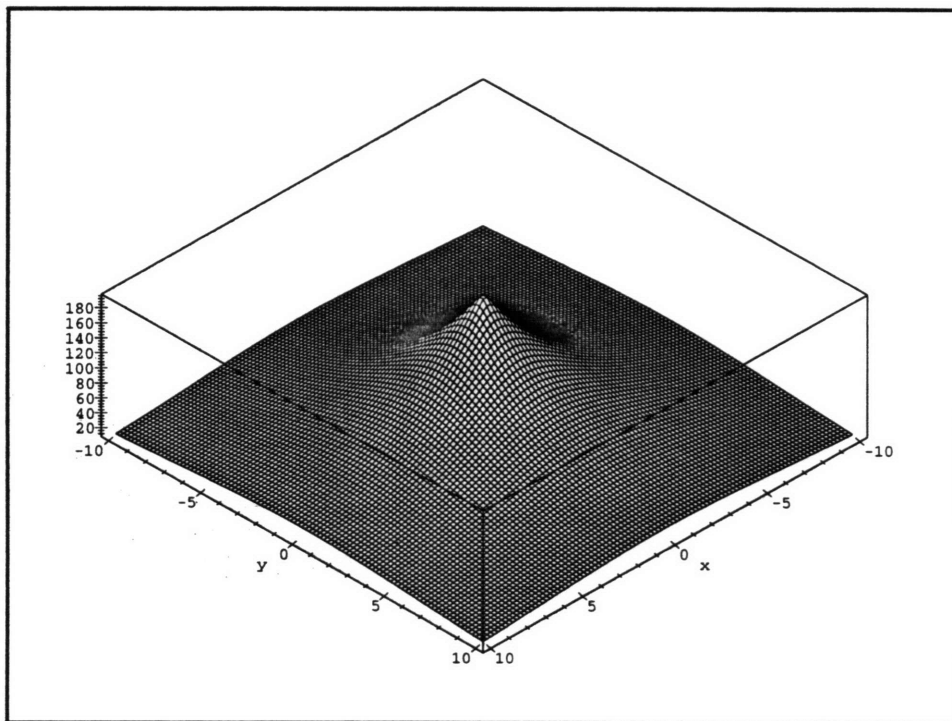


Figure 7-3: A unimodal space

than $n+1$ function evaluations. In “black box problems”, however, no such information is available. This means that it is extremely difficult, if not impossible to devise strategy that is optimal. Ackley [Ack87] showed that no known search strategy could be better than all others on all classes of problems. This is mainly because each strategy makes some implicit assumptions about the nature of the search space. As long as those assumptions are satisfied, it is better than the other strategies. For instance, the assumption behind hill climbing is that the function space is unimodal. In effect, it may be difficult to articulate the implicit assumptions of most strategies to understand their limitations. However, a feel for their performance can be gained by considering examples of their behavior on different function spaces.

It is hazardous, at best, to rely on spatial intuition when considering arbitrarily high dimensions. But since there are no intuitive ways of visualizing higher dimensions, all examples below will consist of real functions defined on a two dimensional space. These examples have been reproduced from

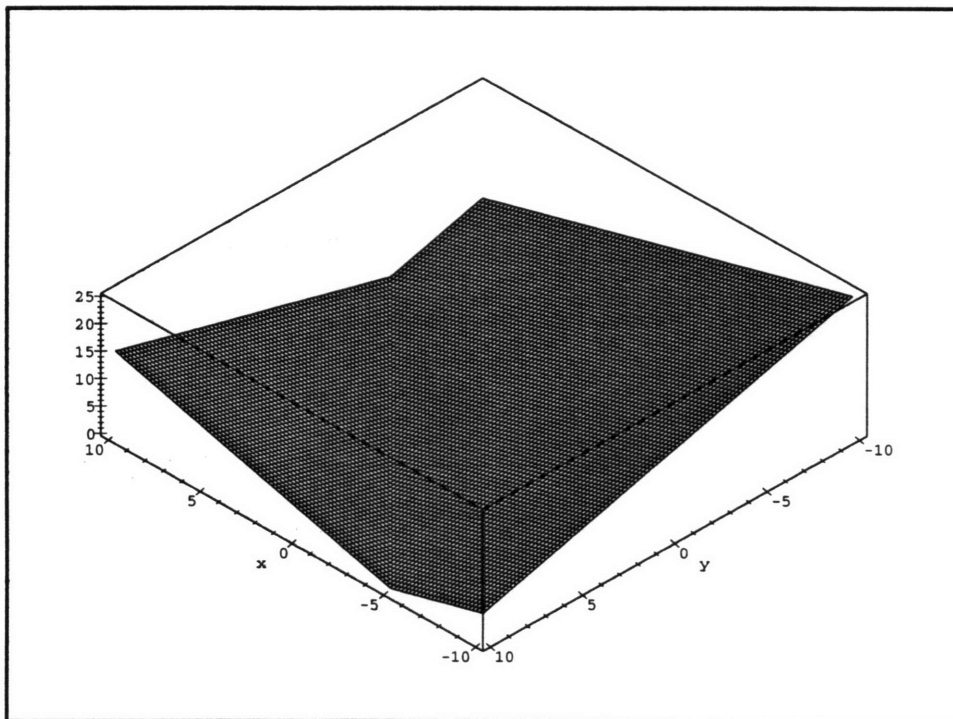


Figure 7-4: A bimodal space

Ackley's book [Ack87].

All examples are defined on a the xy plane, with z representing the function values. The domain of the variables is from -10 to $+10$. Since search algorithms use some form of discretization of the space, we can assume that the domain is discretized into 10,000 sample points - a 100×100 grid. Obviously this is not that large a space, and it is perfectly possible to carry out an exhaustive enumeration here to find the optimal point. For instructive purposes, however, it is sufficient.

Figure 7-1 shows the easiest space of all. Most of the points are optimal. In such a space, even randomly searching would find the optimum very quickly, and no search algorithm has a problem with it.

Figure 7-2 shows a linear space. Global maximum always occurs at one of the corners of the space. Most search strategies would do well in such a uniform space. In fact, simply extrapolating from three non-collinear points will always find a global maximum.

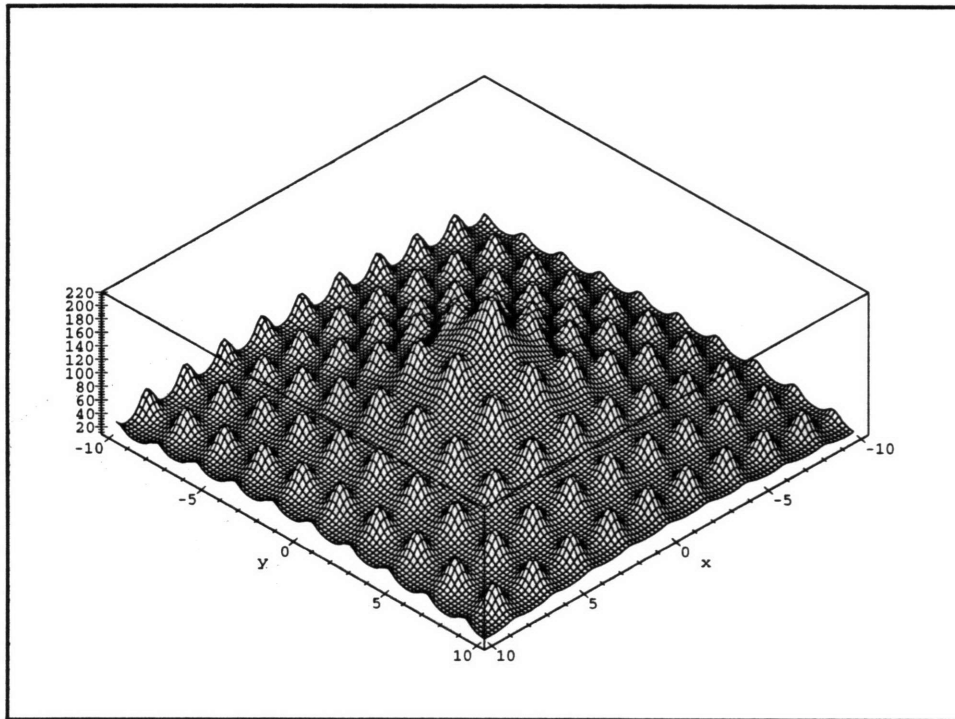


Figure 7-5: A coarse multimodal space

Figure 7-3 shows a unimodal space with function $z = 200e^{-0.2\sqrt{x^2+y^2}}$. In such a space the simple hillclimber will find the optimum without fail. In fact, the improvement in each step of the algorithm is monotonic, and in this particular case, the performance of the algorithm can be expected to be very good. However, there do exist unimodal spaces in which the performance of the hill climber is not impressive. For instance, suppose there were a ridge in figure 7-3 which spiralled up to the top. The hill climber would get trapped on the ridge and move along its top taking a path around the mountain leading to a very low rate of ascent.

Figure 7-4 shows a bimodal function. There are two maxima, one of which is local. Simple hillclimbing cannot guarantee a global maximum in such a case, since the starting point of the algorithm will determine what direction it will take. In practice, this sort of multimodality is the primary reason for rejecting the hill climbing approach. Modifications to the hillclimber, however, have been proposed such as random restart hillclimbing, in which the hillclimber is run again with

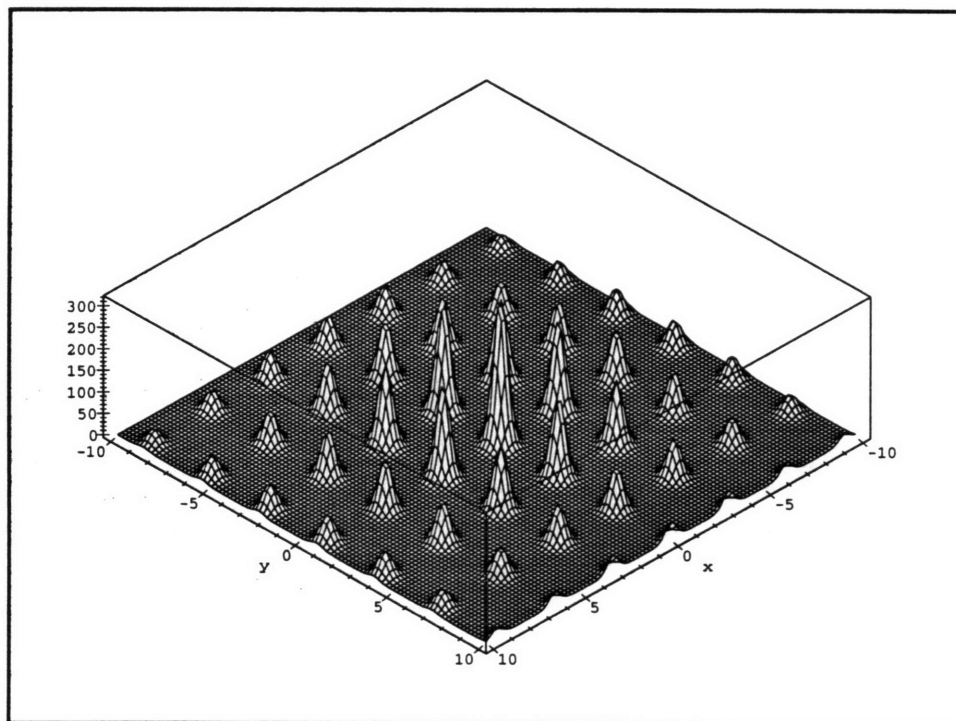


Figure 7-6: A fine multimodal space

other randomly chosen starting points. In spaces that are multimodal, such as above, but in which large portions are unimodal, this approach might work, since the iterated hillclimber would be expected at some time to start off on the slope leading to the global maximum.

Multimodal spaces, however, may be of a much finer texture than the one shown in figure x4. For instance, consider the function $z = 200e^{-0.2\sqrt{x^2+y^2}} + 5e^{\cos 3x + \sin 3y}$ shown in figure 7-5. The space is still largely unimodal, but the areas leading up to each local maximum are small. The iterated hillclimber would therefore be expected to get trapped on top of the local maxima most of the time. Search strategies such as simulated annealing are expected to do better in such a domain since they are geared towards escaping local maxima probabilistically by sometimes making movements in bad directions as well. However, the disadvantage of this is the reduced speed with which they can climb the global maximum.

In seriously complicated spaces, local methods degenerate rapidly. Consider figure 7-6 which

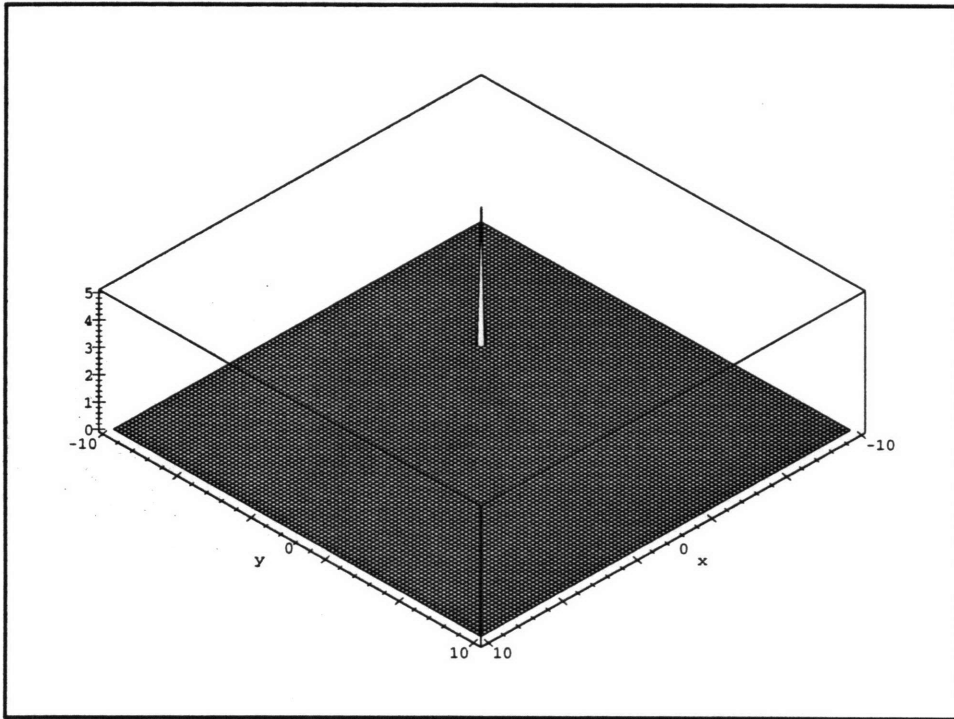


Figure 7-7: A conjunctive space

shows the function $z = e^{-0.2\sqrt{x^2+y^2}+3(\cos 2x+\sin 2y)}$. There are two problems with this function. First, the bases of the hills are small compared to the total area of the xy plane, and two, the heights of the hill are large compared to the bases. Local methods would therefore most probably get stuck on one of the suboptimal spikes. Even simulated annealing will run into problems since to get out of the local maxima, a very long jump is required. Unless, therefore, the simulated annealing algorithm is cooled very slowly, the chance of the algorithm converging to the global minimum is very small. Population based algorithms such as GAs, however, might fare better in such circumstances since they involve a notion of somehow parallel optimization of different points and their combinations in the space.

At the end, consider the conjunctive search space given by the following function and shown in figure 7-7.

$$z = \begin{cases} 5 & \text{if } x = 3 \text{ and } y = 3 \\ 0 & \text{otherwise} \end{cases}$$

This type of search space is one of the most intractable for black box optimization since any reasonable sampling of the variable space would lead one to conclude that the space is mostly maximal unless the exact guess of the variable combinations is made. Local methods are hopeless, but population based methods also stand a slim chance, at best. The probability that the maximum is sampled decreases rapidly with the number of discretizations and the dimension of the space.

As a comment, it must be emphasized that the functions shown above do not even begin to approach the complexity of real search spaces. In effect, all the examples are easy in the sense that the functions have fairly regular structures, albeit complicated. However, they do serve to enhance the strengths and the weaknesses of various search strategies and it will be instructive as we model our search functions later on to see how they compare to the above spaces.

As a point of interest, it is useful to present the topology of the space represented by constraints in our qualitative algebra. Recall our formulation of the design problem in terms of QSRs: it assumes that a point that satisfies a constraint has an evaluation of 1.0 for that constraint. If it violates a constraint by more than a fixed amount, it has an evaluation of zero. In between, the evaluation of the design is a number between 0 and 1. Upon reflection, we note that most of the space must be flat and the tops of the hills must be plateaus corresponding to feasible designs. Therefore if a design is found to be in one of the lower flat areas, it is extremely likely to be destroyed. If a design is on one of the slopes to the plateaus, it is a candidate for improvement. The function values are always positive, and the optimal value is known to be 1.

One of the problems represented by the above space for point based algorithms - even simulated annealing, is that if the starting point is found to lie on one of the lower flat spaces, the direction of improvement is very difficult to determine. The second problem for deterministic point based strategies is that this topology is really a case of multiobjective optimization, considering each constraint as an objective to be satisfied. Since point based strategies require a value of the function that is better than the original value, some combination of the evaluation of the constraints has to be supplied to the search mechanism. However, it is mostly possible to supply a value to the searcher such that it actually leads to a point from where no improvement is possible.

7.3 Metrics for Comparison

The most difficult issue in trying to compare GAs to Ateams is the choice of the metrics. It is natural to think of the performance of an algorithm in terms of convergence and the quality of the solution - since an approximate solution may not be acceptable in some cases, the speed with which the solution is obtained and the class of problems which the algorithm can solve.

All of these points pose significant problems in comparison of black box optimization techniques necessitating the redefinition of the concepts of convergence and the measure of speed. Below we discuss each of the above points in detail.

The first problem is the quality of the solutions obtained by a search strategy. In some problems,

it is possible to settle for some form of an approximate solution within a neighborhood of the global optimum. In other situations, it is impossible to do so. For instance, a sorting algorithm that guarantees that the data will be approximately sorted is probably of no use. On the other hand, in numerical integration, nothing better than an approximate solution can be guaranteed and is usually sufficient.

If we take the position that the global maximum must be found exactly, we obtain an important simplification since the quality of the solution is never in doubt. This has an important parallel in the constraint satisfaction problem. In artificial intelligence, the constraints in a CSP can be considered to be *strong* or *weak*. Strong constraints must be satisfied before a problem is considered solved. By contrast, a small violation of the weak constraints can be tolerated in the final solution. Taking the position that the global optimum must be found is similar to taking a strong constraint stance.

However, the global optimum may not be obtainable *a priori*. Testing the algorithm on arbitrary problems therefore does not constitute a reasonable test since there may be no way of knowing if the true optimum has been attained. A reasonable strategy is to define a fixed set of problems for which the global optimum is known and which are designed to test important qualities of search methods.

The single most important question, however, is usually that of speed. How long will the algorithm take to find the solution? Memory requirements, though important, have been reduced in importance by the rapid advances in available RAM. Other factors are also relevant in particular cases, but the natural measure of algorithm performance is speed. There is however, a problem. Speed measured as what? Although the most natural choice is the amount of time one has to wait for an answer, and therefore a natural measure would be the raw CPU time, it is probably not the best judgement criteria for a variety of reasons. The CPU time is heavily machine dependent, and incorporates a lot of overhead time used in storage and IO which is not only different for different processors but depends very much on the peripheral devices. This does not allow for the comparison of CPU times on different machines. Moreover, we know from studying the complexity of conventional algorithms that differences of speed of upto factors of two can simply be a result of programming details [PS82].

A machine independent criterion for measuring speed that is often used is the number of evaluations that are performed by the algorithm before it finds a solution. There are problems associated with this metric too. Experience has shown that when the number of function evaluations is recorded, different kinds of evaluations are grouped together to form one evaluation. For instance, in non-linear optimization, the evaluations typically involve gradient as well as function evaluations. However, when the number of function evaluations is recorded, it sometimes includes the gradient and Hessian-matrix evaluations as well. This may have the advantage of being machine independent [Loo71], but it does not inform the user of the amount of effort necessary to solve a given test problem. For instance, gradient projection methods involve a lot of time consuming array manipulations which are not negligible with respect to the function and gradient evaluations.

In general, therefore, if we are to use the number of evaluations as the metric for speed, we must show that the function evaluations constitute the major portion of the time used by the algorithm. Or, relaxing the requirement a bit, we must show that as the problem grows in dimensionality, the rate of increase of the time used by the algorithm is proportional to the increase in the number of

evaluations. This has the important simplification of ignoring the necessary overhead that must be accrued in any algorithm in the storage and manipulation of solutions.

7.4 Testing Methodology

Our original problem was the application of Ateams to the CSP formulated in the QSR algebra. However, as argued before, although the performance of various search strategies can be tested on a collection of such problems, it is difficult to gain much insight into the reasons which affect the performance of particular strategies. Our fundamental assumption is that the shape of the underlying search space is the most important factor affecting performance. In the QSR algebra, it is difficult to formulate analogues of the spaces which are described in section 7.2. We would ideally like to have controlled spaces which, even if they are not representative of the actual problems to be solved, will give us some insight into the limitations of each of the methods.

For this reason, we decided to test Ateams and GAs on a suite of real functions defined on n -dimensional spaces that approximate as closely as possible the essential spaces discussed in section 7.2. The functions themselves are described later. The objective is to find the global maximum of the function.

The measure used for speed is the average of the number of evaluations on 10 runs of the problem. We test the speed on each function. The search space is assumed to be contained within a hypercube. Apart from the topology of the space, there are two ways in which the difficulty of the search can be controlled. One is the dimensionality of the space and the other is the level of discretization of the variables. We attempt to study the effect of both on the performance of the algorithms.

7.5 Parameters

In both GAs and Ateams, the performance of the algorithm depends on certain parameters, for instance, to mention a few, the crossover and mutation rate in GAs and the probabilities of destruction and improvement in Ateams. Ostensibly, different settings of parameters will be required for optimal performance of the algorithm on different problems. However, since it is impossible to know what the setting of the parameters should be for a particular problem and it is impractical to expect the user to tinker with the algorithm using different runs to find the best setting, we use a fixed parameter set across all problems. The hope is that the effects of varying performance will average out over the problem domain we have selected.

7.6 The Function Suite

The inspiration for the functions we have constructed stems from the spaces described in section 7.2. All variables are real. We created these function by a kind of reverse analysis. Given that the end of the search occurs when the global maximum is evaluated, the neighborhoods of the maxima become

very significant. The real task is to model the neighborhoods of the global and local maxima such that they portray all the stereotypical properties of the search spaces described before.

The common element in these functions is that the global maximum is always located at a specific point $(0, 0, \dots, 0)$. Although this may sound trivial when stated as above, note that the search strategy has no way of knowing that the global maximum is at $(0, 0, \dots, 0)$ and not anywhere else. In fact, there is no advantage to be gained by shifting the global maximum to an arbitrary vector location.

The second commonality is that the global maximum always has a value of 1.0. The reason for this was mainly convenience. Ateams had been implemented already to handle constraint violations from 0 to 1, and the nature of storage of the solutions depended on this range. Therefore we thought it was prudent to use the same scheme for the time being. Note that this is not a major drawback in any sense since traditional GAs are insensitive to the scaling of the function values. This is because the reproductive fitness of individuals is determined by dividing its function value by the average of the space.

The search space is assumed to be contained in the hypercube $-1024 \leq x_i \leq 1024, \forall i$. The dimensionality of the space is 12, 16 and 20 and for each of these, we allow three levels of discretization, 1:1, 1:2, 1:4 where 1:2 means one unit of the variable is discretized into two parts. It is obvious that for the smallest case involving 12 variables and a discretization of 1:1, the number of search points are 2^{132} . For the largest case involving 20 variables and 1:4 discretization, the search space has approximately 2^{260} points.

7.6.1 Unimodal Space

The first function is a unimodal space. It is used for testing how well Ateams and GAs compare on easy spaces for which they have not been designed. A low dimensional instance of this function is shown in figure 7-8. The function is given below:

$$f(x) = 1 - \frac{1}{\sqrt{n}1024} \sqrt{\sum_{i=1}^n x_i^2}$$

It is clear that the maximum occurs at $(0, 0, \dots)$ and has value 1. To see that there are no local maxima, note that $x \neq 0, x \in R_n \Rightarrow \exists x_{i^*} \neq 0, i^* = 1, \dots, n$. Then for any $\epsilon > 0$, the vector \hat{x} having $\hat{x}_i = x_i, \forall i$ except i^* and $\hat{x}_{i^*} = x_{i^*} - \epsilon$ if $x_{i^*} > 0$ and $\hat{x}_{i^*} = x_{i^*} + \epsilon$ if $x_{i^*} < 0 \Rightarrow f(x^*) > f(x)$. Therefore no non zero vector can be a local minimum.

7.6.2 Multimodal space

This is somewhat an analogue of the bimodal space described before in the sense that although it has more than two maxima, the nature of the space is very benign. The number of local maxima are 2^n with large collecting areas around each maximum. Therefore there is a chance of getting to the global minimum with strategies like random restart hill climbing. The global maximum occurs

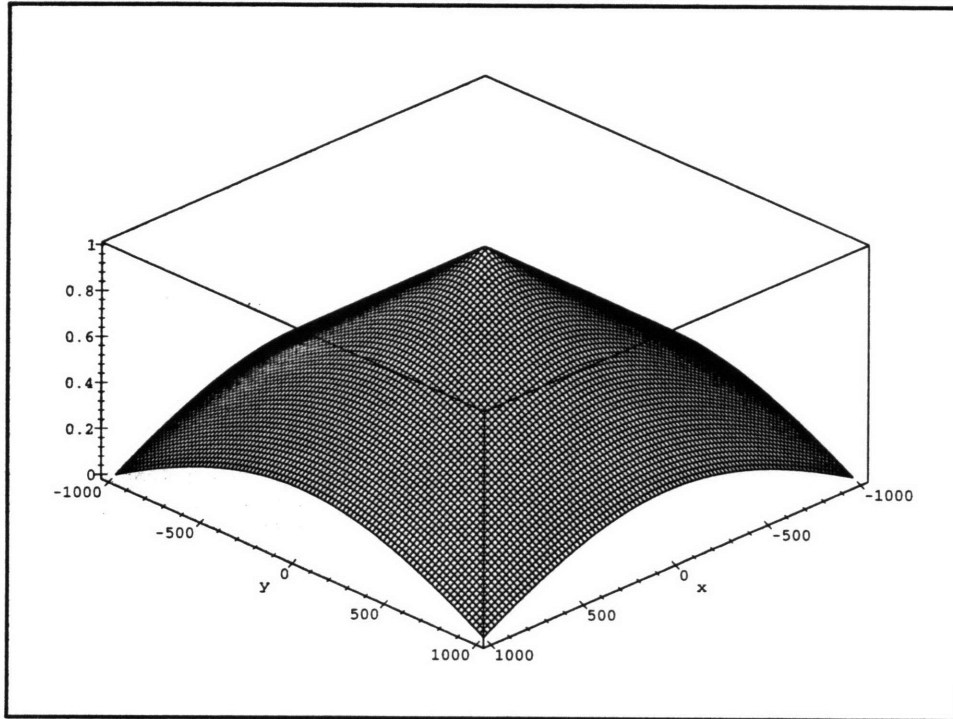


Figure 7-8: Unimodal function space

at $(0, 0, \dots)$ and local maxima at all corners of the hypercube i.e. when $x_i = \pm 1024, \forall i$. The 2D version of this space is shown in figure 7-9 and the function is given below.

$$f(x) = \frac{1}{0.6} \left| \frac{1}{\sqrt{n}1024} \sqrt{\sum_{i=1}^n x_i^2} - 0.6 \right|$$

To see that the function has maxima at $(0, 0, \dots, 0)$ and at the corners of the hypercube, note that the minimum and maximum values of the component $\sqrt{\sum_{i=1}^n x_i^2}$ are zero and $\sqrt{n}1024$ within the hypercube. Therefore the value of the function is 1 at $x = 0$ and $\frac{2}{3}$ at all corners. For any other x such that some $x_i \neq \pm 1024$ and $x \neq 0$, if $\frac{1}{\sqrt{n}1024} \sqrt{\sum_{i=1}^n x_i^2} > 0.6$, for any $\epsilon > 0$, the vector \hat{x} having

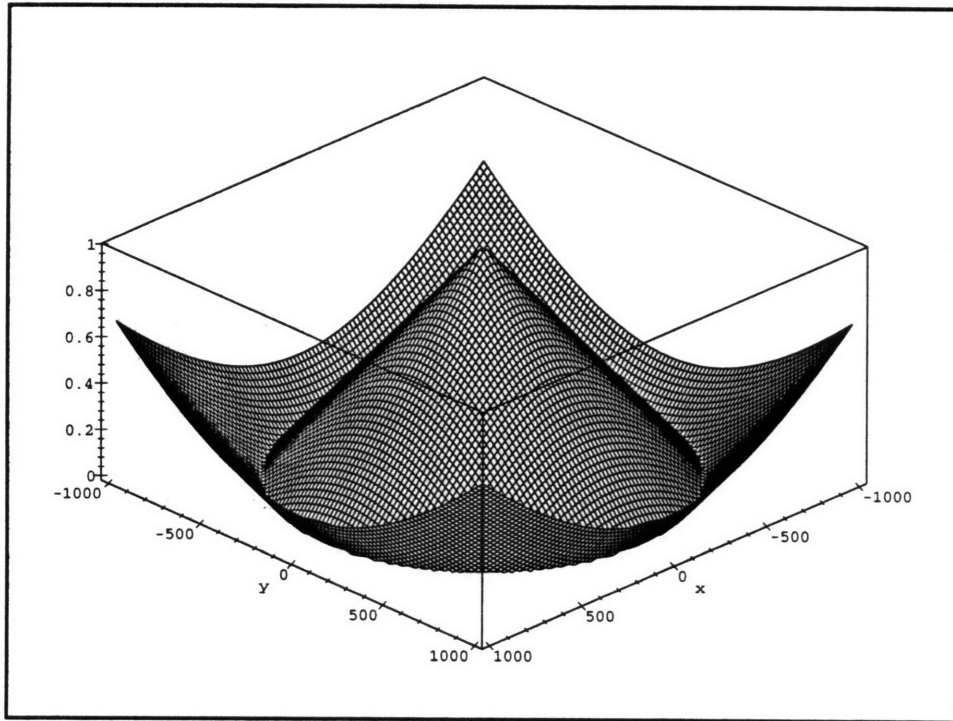


Figure 7-9: Multimodal function space

$\hat{x}_i = x_i, \forall i$ except some i^* and $\hat{x}_{i^*} = x_{i^*} - \epsilon$ if $x_{i^*} < 0$ and $\hat{x}_{i^*} = x_{i^*} + \epsilon$ if $x_{i^*} > 0 \Rightarrow f(x^*) > f(x)$. Similarly if $\frac{1}{\sqrt{n}1024} \sqrt{\sum_{i=1}^n x_i^2} < 0.6$, the vector \hat{x} having $\hat{x}_i = x_i, \forall i$ except some i^* and $\hat{x}_{i^*} = x_{i^*} - \epsilon$ if $x_{i^*} > 0$ and $\hat{x}_{i^*} = x_{i^*} + \epsilon$ if $x_{i^*} < 0 \Rightarrow f(x^*) > f(x)$, therefore there are no other local maxima.

7.6.3 Porcupine space

This space corresponds to the sharp high multimodal space. The function is shown below.

$$f(x) = \frac{1}{4} \left[\frac{1}{\sqrt{n}1024} \left(\sqrt{n}1024 - \sqrt{\sum_{i=1}^n x_i^2} \right) \left(1 - \sqrt{\sum_{i=1}^n h_i^2(x)} \right) + 3 \right]$$

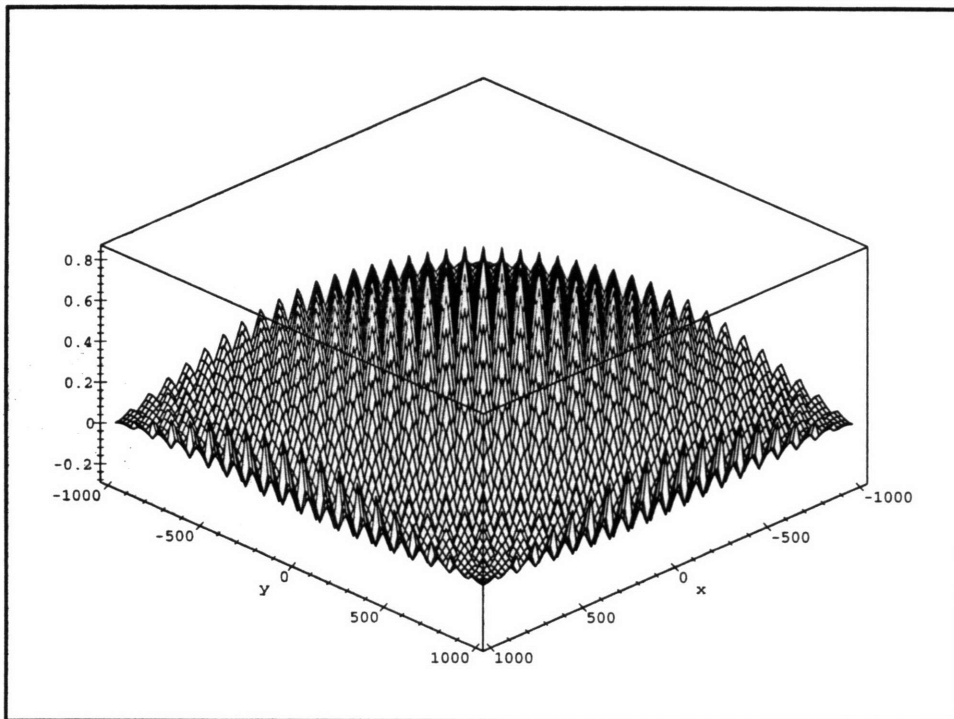


Figure 7-10: The porcupine space

where

$$h_i(x) = \begin{cases} (x_i - \lfloor x_i \rfloor) & \text{if } x_i \geq 0 \\ (x_i - \lceil x_i \rceil) & \text{if } x_i < 0 \end{cases}$$

This function has a global maximum of 1 at $x = 0$ and a local maximum at all points with integer coordinates. At $x = 0$, note that $f(x) = 1$ and at all vectors with integer coordinates, $f(x) < 1$. Also, note that in the neighborhood of the integer vector x defined by $\lfloor x_i \rfloor < x_i < \lceil x_i \rceil \forall i$, the value of the function is less than the value at x .

The corresponding 2D function is shown in figure 7-10. As is evident, this is an extremely fine grained multimodal space. But the figure does not do justice to the space because of the low resolution. In fact, the space is much finer grained than shown. Two of the extremely difficult

properties of the space are the high number of maxima and the collecting area near each maximum. Note that we can guess intuitively by looking at the function that when any component of x is integer and is reduced by any amount, the function value experiences a very sharp drop. In fact, the integer points are almost like vertical cliffs if approached from the sides where the components of x are smaller than the integer coordinates. Even this terribly inarticulate exposition gives some idea of the noisiness encountered by the search strategies. With this kind of space, any hill climbing algorithm will get stuck on one of the local maxima with a probability very close to 1.

7.7 Results

This section discusses the results of our experiments. The objections to the methods of the study are addressed. The characteristics and limitations of the data are explained and the primary trends that we are looking for in the data are specified.

For each function in the function suite, the number of evaluations reported for both Ateams and GAs are the average of 10 runs until the first optimal solution is found. There are valid objections to this. Specifically, ten runs constitute too small a data set to be of any conclusive value. Also, it would probably be more insightful to perform a statistical analysis of the data since the distribution of the data might be indicative of whether the average is a reasonable measure of the performance for any particular experiment - an experiment consisting of a function with a fixed dimensionality and discretization.

We note the following responses to these objections. Firstly, our effort has not been directed toward providing conclusive proof that Ateams are better than GAs. Rather, we have provided initial evidence that the study of Ateams might be a promising research topic. Conclusive testing of both approaches has been left for future research. Therefore the issue was investing a reasonable amount of effort to see if Ateams are even worth the effort. That constitutes the rationale for a small data set. The second limitation of not being able to perform a statistical analysis stems from the first. It is probably of no value to do an analysis on a data set of dimension 10. Therefore elaborate analysis must also wait until enough data is collected.

Other objections are of a more serious nature. Can we seriously compare one evaluation of Ateams with one evaluation of a GA in terms of the time taken? Our claim is that we can, for the following two reasons.

Having implemented the system, we can confidently claim that the evaluation functions for both Ateams and GAs are almost identical, with the Ateams function involving slightly more conditional statements for suggesting modification operators. The second response is much more substantial. From a substantial number of observations, we found that there is a reasonable correlation between the CPU time and the number of evaluations for each function. Using the UNIX command `/usr/bin/time`, we observed the real time used by the algorithms against the number of evaluations performed. For GAs, each 1000 evaluations take ≈ 3.5 seconds. This figure seems to remain stable across different functions. For Ateams, there is considerably more variance, but the number varies from 2.4 secs/1000 evaluations to 4.4 secs/1000 evaluations depending on the function. There are two primary reasons for this. First, the time recorded for the Ateams involves a substantial portion involved in dealing with the Motif interface that is not used by the GA. Second, the algorithm for

the limit memory destroyer is still unoptimized, therefore as soon as Ateams start to converge to a better population, the number of iterations required to manage the store become large. Therefore the real time for Ateams involves substantial unnecessary overhead at this time. However, we can still conclude that the evaluations are reasonable measures of the time expected to find the optimal solution.

If we accept the choice of the average number of evaluations as a representative measure of the performance for both Ateams and GAs, we will need to note some other points before we start analyzing the data. First, because of the small data set, we should reasonably expect a lot of noise. Particularly, this means that we need to decide how much of a difference in the average to allow before we say that there is significant difference between them. The argument here is completely heuristic since we have no way of answering this question. We decided to take a reasonable way out. Upon successive ten runs of the Ateams for a given experiment, it was found that the average did not vary by more than 1000 evaluations. Similarly, for GAs, the average did not vary by more than 2000 evaluations. Therefore, when analyzing the data, we will consider data having differences of less than 1000 for Ateams and less than 2500 for GAs to be the same.

When analyzing data, we shall look for the following trends.

1. The total number of evaluations taken by Ateams against the total number taken by GAs.
2. The degradation of performance of the algorithms at a fixed dimensionality across increasing discretizations.
3. The degradation of performance across a fixed discretization with increasing dimensionality.
4. The degradation of performance across topologies.

The first point is the most natural piece of information to look for in any comparison. Its relevance is obvious and the objections have been taken care of.

Regarding the second point, there is an important subtlety that must be noted. The number of discretizations is extremely important in some cases in controlling the complexity of the search. For example, consider the porcupine function in our suite. It has local maxima at every vector with all integer co-ordinates. The entire difficulty of the search lies in the small neighborhoods around each local maxima. Upon reflection, we find that if the number of discretizations is 1:1, the nature of the search space is not different at all from the unimodal function in our suite since there are no points representing the neighborhoods in the search space. A neighborhood here is taken to be all points around a vector with $\lfloor x_i \rfloor < x_i < \lceil x_i \rceil$. As the search space is discretized to 1:2, we find that the number of points in the neighborhood of each local maxima increase to n , where n is the dimensionality of the space. Upon discretization to 1:4, the number of points in the neighborhood increases to 2^n . Discretizations are therefore important in the study of algorithm performance in controlled topologies and require close inspection.

With respect to the third and the fourth points: the dimensionality of the search space is obviously important and therefore no argument in its favor is presented here. Also, the performance across topologies is important, for we want to observe the behavior of the algorithms as the space becomes noisier.

Ateams	12	16	20
1:1	4030	7404	8595
1:2	4400	6518	8273
1:4	4317	5563	7091

GAs	12	16	20
1:1	20656	26435	35909
1:2	24005	35813	45497
1:4	28626	40562	57874

Table 7.1: Evaluations performed by Ateams and GAs on the unimodal function

We shall discuss the data for each function individually. For each function below, the results of Ateams and GAs are tabulated and bar graphs are provided. In all bar graphs, the smaller bars represent the Ateams evaluations.

One point that must be recorded about the data is that when given an upper bound on the number of evaluations, often GAs would not be able to find the optimal solution. In such cases, we recorded the upper bound as the number of evaluations for that run of the experiment. The rationale is: the total number of runs for each experiment is so small that omitting the data for even one of the bad runs will give an overly optimistic average. Using the upper bound, although it still gives an optimistic estimate, brings the average much closer to what would have been if the GA had been allowed to go on.

7.7.1 Unimodal Function

From table 7.1 and figure 7-11, it is clear that the difference in the number of evaluations taken by Ateams and GAs for any particular experiment is orders of magnitude. In fact, the best that the GAs could do was almost 3.5 times the number of evaluations for Ateams and the worst was 8 times the corresponding Ateams evaluations.

Also, note that across discretizations, the performance of the Ateams was relatively very consistent – a fact that was surprising to us. In fact, the performance seemed to get better in higher dimensions. However, we'll attribute its likely cause to be noise and assume that the performance was relatively consistent.

GAs, on the other hand, experienced significant degradation in performance. And the deterioration seemed to increase as the dimensionality of the space was increased. In fact, the least degradation was for 12 variables, ≈ 4000 , and the most for 20 variables, ≈ 21000 , when the discretizations increased from 1:1 to 1:4. This is a huge difference.

With increasing dimensionality, degradations in Ateams occur of the order of magnitude. for instance, from 12 to 20 variables for 1:1 discretizations, the number of evaluations doubled. However, the total number of evaluations is still relatively small as compared to the GAs. Given our

Ateams	12	16	20
1:1	4178	6858	9057
1:2	4671	6792	7972
1:4	4367	6117	7785

GAs	12	16	20
1:1	18853	28432	39018
1:2	21866	34602	46554
1:4	27258	29360	50816

Table 7.2: Evaluations performed by Ateams and GAs on the multimodal function

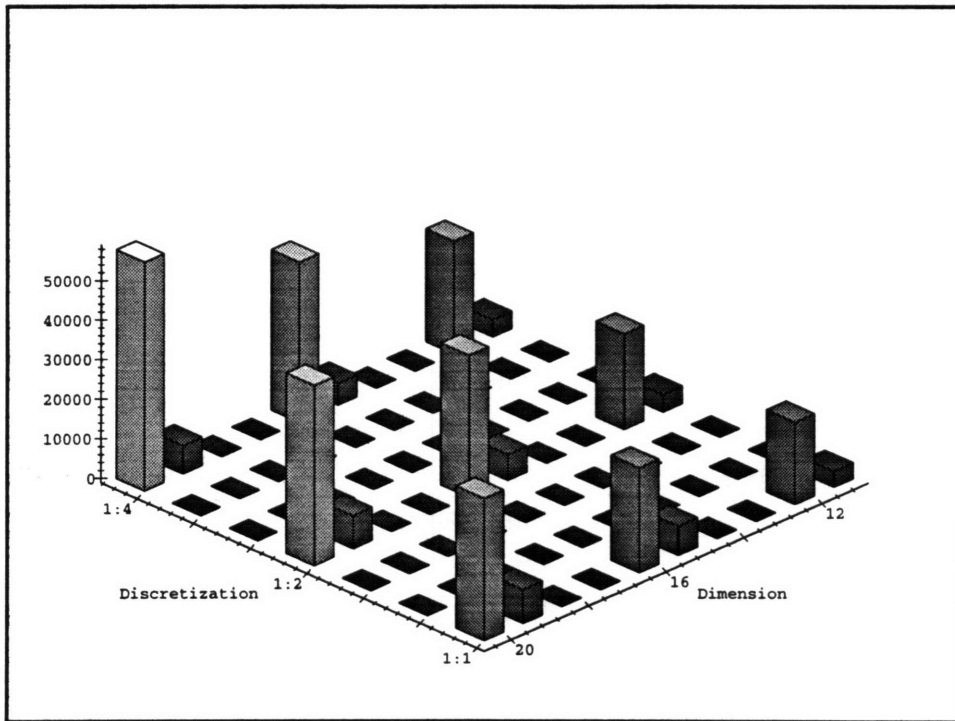


Figure 7-11: Ateams vs. GAs on the Unimodal function. *The smaller bars are the Ateams*

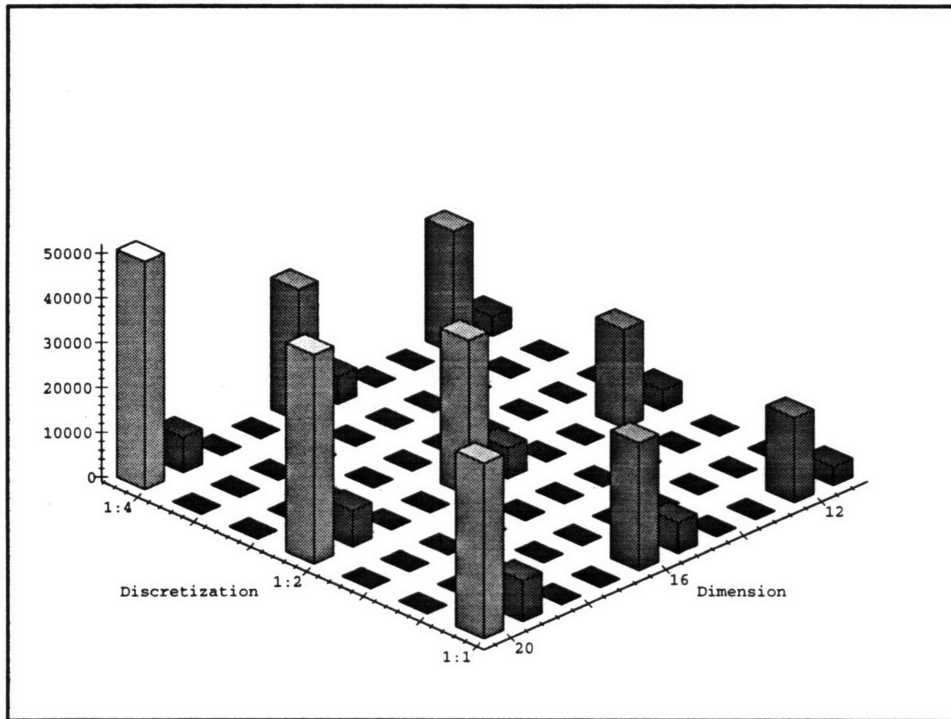


Figure 7-12: Ateams vs. GAs on the Multimodal function. *The smaller bars are the Ateams*

expectations of the amount of noise in the system and the small initial number of evaluations, this was to be expected. The worst case performance of the GAs is also a two fold increase from 1:4 - 12 variables to 1:4 - 20 variables.

7.7.2 Multimodal Function

Again, by looking at table 7.2 and figure 7-12, we can see that in terms of the total number of evaluations, there is no comparison between Ateams and GAs. GAs routinely take almost 4 - 5 times the number of evaluations required for Ateams. Their best being approximately 4 times and their worst being almost 6.5 times.

Across discretizations, the performance of GAs again degrades considerably, though not as bad as before. The least degradation of ≈ 10000 occurring from 1:1 to 1:4 for 12 variables and the worst

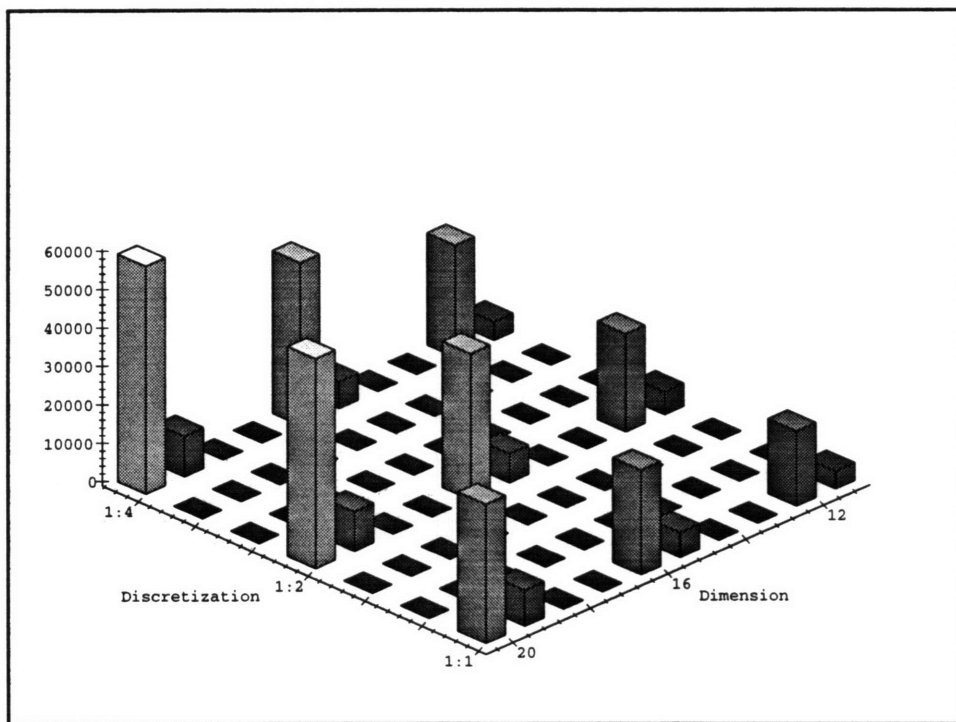


Figure 7-13: Ateams vs. GAs on the Porcupine function. *The smaller bars are the Ateams*

Ateams	12	16	20
1:1	4596	6435	9245
1:2	5930	7587	10212
1:4	4797	7007	10578

GAs	12	16	20
1:1	19453	26998	35909
1:2	25439	37997	54446
1:4	29163	42357	59279

Table 7.3: Evaluations performed by Ateams and GAs on the porcupine function

of ≈ 12000 for 1:1 to 1:4 for 20 variables.

Ateams, on the other hand, seem to fare excellently. There is no degradation at all with increasing discretization for any dimensionality.

Across dimensions, the deterioration in Ateams performance is almost the same as before. On the whole, both GAs and Ateams seem to fare fairly well on this space. One possible explanation is that the slope of the function seems to be much sharper than the slope of the unimodal function. This means that the difference in the evaluation of non-optimal points and the optima can be expected to be more, leading to greater pressure toward the optimum.

7.7.3 Porcupine Function

The most interesting case is the porcupine space, for it is here that we expect to run into the most limitations of both algorithms and to compare them. The results for this function are tabulated in table 7.3 and shown in figure 7-13.

Again, in terms of the number of evaluations, GAs are no match for Ateams. Their best being approximately 4 times for 1:1 - 16 variables and their worst being ≈ 6 times for 1:4 - 16 variables. In fact, the performance of GAs might have been more dismal for this was the function in which the GAs very frequently failed to find the solution in a given number of evaluations.

Regarding discretizations, Ateams could not have done better. There is almost no deterioration at all across discretization. We find this truly amazing since the number of points in the neighborhood of local maxima grows from 0 to n to 2^n . It seems that Ateams are oblivious to the nature of the collecting space around local maxima. If that is indeed the case, then we might have a very powerful technique for searching that overcomes most of the limitations of the traditional methods.

GAs, on the other hand, suffer significantly with increasing discretizations of the space. Their worst performance drain is a difference of ≈ 23000 evaluation from 1:1 to 1:4 for 16 variables, and their best is a difference of around 10000 evaluations for 1:1 to 1:4 for 12 variables. This deterioration is the worst so far for all functions. GAs seem to behave exactly as expected with increasing number of neighborhood points in the search space.

With dimensionality, again, GAs suffer the worst setback so far. The worst degradation being

\approx 30000 evaluations for 1:4 discretization from 12 to 20 variables. That is almost two extra minutes of real time taken by the algorithm. The least deterioration was \approx 16000 evaluations for 1:1 discretization from 12 to 20 variables.

Ateams also used the most number of evaluations for all functions, but the degradation of performance with increasing dimensionality was significantly less. The most they suffered was almost 6000 evaluations for 1:4 from 12 to 20. The least was almost 4500 for 1:1 from 12 to 20 variables.

7.8 Conclusions and Summary

This section presents the conclusions from our results. Some of them were expected, but others are relatively surprising. A list follows.

1. Since Ateams seem to be orders of magnitude better than GAs on our test problems in terms of the number of evaluations, we can speculate that the reason could be the scheme of specifying improvements to the solutions. Incorporating the means of improvement in the evaluations by means of modification operators corresponds to a form of localized hillclimbing. However, the difference is that all points in the neighborhood do not need to be evaluated before choosing one of them. This cuts down tremendously on the number of evaluations needed. Every new solution produced is better in some sense than the previous one.

GAs, on the other hand, do not assure that by applying crossover and mutation, the new solutions produced are better than the previous ones, although this is true on the average. This may account for the GAs spending extra evaluations on less promising solutions thus increasing their count.

2. The most surprising fact was the performance of Ateams with increasing number of discretizations of the space. It seems that Ateams are insensitive to the nature of the neighborhood of the local maxima. For all the functions, there were no appreciable degradations in performance. Some of the implications of this observation are increased accuracy and better solutions in real problems.

The reasons for this could be the fine grained hill jumping behavior of the Ateams. In every improvement of a solution in Ateams, only one variable was changed. The amount of change was two discretizations in the direction of improvement. However, there was sometimes a probabilistic jump in which the amount of change was huge. Since original solutions were maintained till the next purge of the population, this meant a good sampling of the search space in which the new solutions were destroyed if not sufficiently good. In 1000 steps a reasonable sized neighborhood of a point can be sampled.

3. Across dimensions, Ateams performed very much better. There is no ready made explanation for this. Our expectation was that with increase in dimensionality, the number of modification operators should grow linearly, and therefore in any 1000 steps, the number of improvements to a particular variable should be less, resulting in slow convergence. Obviously, there is some other fact we had not accounted for. More investigation into this behavior of Ateams is required.

4. Again, as the topology of the space gets noisier, we see Ateams holding much better than GAs. This is an encouraging fact, leading us to believe that Ateams are a feasible approach as compared to GAs for search and optimization.

This chapter provided the results of a preliminary comparison between Ateams and GAs on a suite of test problems. The limitations of search techniques were discussed and metrics for comparing Ateams and GAs were identified and justified. The results indicated that Ateams may be a very promising area of research.

The unexpected performance of Ateams has raised interesting questions and issues that need to be addressed. The next chapter proposes some new research directions in this regard.

Chapter 8

Summary and Future Work

The fundamental insight gained from this thesis is that population based randomized algorithms may be serious competitors to genetic algorithms as search techniques. The conclusions have been derived from limited testing of Ateams and GAs on both qualitative formulations of a design problem and on artificially constructed test problems.

In qualitative formulations, the fundamental problem was the finding of feasible solutions to sets of constraints. In the test problems, the goal was finding the global optimum in a search space. In terms of the number of evaluations used, Ateams outclassed GAs on both these problem classes. We believe that Ateams are a rich potential area of research for search algorithms. In particular, they seem to hold promise by allowing the incorporation of existing techniques as agents – perhaps even GAs as agents, to solve constraint satisfaction problems. We are encouraged by the results of this research and propose the following directions for future research in this area.

1. One of the first priorities should be an extensive testing of Ateams on a diverse set of problems and a more extensive comparison with GAs. Work needs to be directed towards the specification of measurable and objective metrics that are natural measures of a search algorithm's performance.
2. Further work is needed to refine the storage and handling of solutions in Ateams. In particular, some aspects of our implementation are unoptimized. For instance, the limit memory destroyer goes through the store of solutions and destroys designs according to a probability based on the evaluation until the size of the store is acceptable. This means that as the solutions grow better on the average, the number of iterations required to control the store become excessive. An easy way out would be to base the probabilities on the basis of a scaled evaluation in which the worst solution in the store has a probability 1 of being destroyed and the best probability 0. This would reduce significant overhead in the algorithm.
3. An interesting experiment would be to provide a kernel for Ateams that is similar to Grefenstette's GA code. It would allow the user to supply an evaluation function with the improvement rules based on our expression decomposition. This would encourage similar testing of Ateams and GAs by researchers on a wide array of problems, allowing us to discover the

strengths and weaknesses of Ateams. In effect, this can be easily done because of the modular nature of Ateams.

4. The general properties of Ateams as organizations of software agents need to be investigated. We would like to be able to use sophisticated algorithms as agents working on the solutions in parallel to the simple minded improvement operators. This may have advantages of improved solutions as well as taking advantage of already existing approaches.
5. An theoretical analysis of Ateams needs to be carried out to gain some insight into the reasons for its convergence and its behavior.
6. One of the important drawback right now is the inability of Ateams to provide consistency checking for constraints. This means that sets of conflicting or redundant constraints cannot be identified. By incorporating a consistency checking algorithm as an agent, this problem may be alleviated. The other approaches are to preprocess the input to Ateams to find out beforehand if the constraints are conflicting.
7. One of the discoveries that need focus is the fact that Ateams seem to handle a high number of discretizations of the space very well. If that is indeed true, we need to theorize about its causes. Hopefully this will lead to some insights into the behavior of both Ateams and other search algorithms.

Appendix A

Header Files for the Ateams

This appendix provides the listings of the header file for the Ateams. The high level details of these classes are discussed in chapter 6.

```
#ifndef GNvector_H
#define GNvector_H

#include <iostream.h>
#include <math.h>

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

#define GN_MAXLEN 100
#define GN_EPSILON 1.0e-6
#define GN_INFINITY 1e10
#define GN_INFINITYBIG 2e10
#define BADVALUE 2e10

#define GNPI 3.14159265
enum{GN_OK, GN_FAIL};
enum{GN_X=0, GN_Y=1, GN_Z=2};

class GN4vector{
  friend class GNtransmat;
public:
  inline GN4vector(){el[0] = el[1] = el[2] = el[3] = 0.0 ;}
  inline GN4vector(double x,double y, double z, double w){
    el[0] = x; el[1] = y; el[2] = z; el[3] = w;}

```

```

inline GN4vector(const GN4vector& vec);
~GN4vector(){}
inline GN4vector& operator=(const GN4vector& vec);           30
inline int operator==(const GN4vector& vec)const ;
inline double& operator[](int p)const{ return el[p];}
inline GN4vector operator+(const GN4vector &vec)const;
inline GN4vector operator-(const GN4vector &vec)const; // subtraction
inline GN4vector operator*(const GN4vector &vec)const; // cross product
inline GN4vector operator*(const GNtransmat &tra)const; // matrix multiplication
inline GN4vector operator*(double val)const; // scalar multiplication
inline double operator^(const GN4vector &vec)const; // dot product
inline double magnitude() const; // magnitude
inline GN4vector unit()const; // unit vector in           40
// this direction

void print()const{ cout << "Class: GN4vector (" << el[0] << ", " << el[1];
cout << ", " << el[2] << ", " << el[3] << ")" << endl;}

protected:
double el[4];
};

class GNvector : public GN4vector{
public:
inline GNvector(){el[0] = 0.0; el[1] = 0.0; el[2] = 0.0; el[3] = 1.0;}           50
inline GNvector(const GNvector& vec);
inline GNvector(double x,double y, double z){
el[0] = x; el[1] = y; el[2] = z; el[3] = 1.0;}
inline GNvector(double x,double y,double z,double w){
el[0] = x;el[1] = y;el[2] = z;el[3] = w;}
inline ~GNvector(){}

inline void set_x(double x){el[0] = x;}
inline void set_y(double y){el[1] = y;}
inline void set_z(double z){el[2] = z;}           60
inline void set_vector(double x,double y,double z)
{el[0] = x; el[1] = y; el[2] = z; el[3] = 1.0;}

inline double& operator[](int p)const{return el[p];}
inline GNvector& operator=(const GNvector& vec);
inline int operator==(const GNvector& vec)const ;
inline GNvector operator+(const GNvector &vec)const; // add only the first
// 3 terms
inline GNvector operator-(const GNvector &vec)const; // subtraction
inline GNvector operator*(const GNvector &vec)const; // cross product           70
inline GNvector operator*(const GNtransmat &tra) const ;
inline GNvector operator*(double val)const; // scalar multiplication
inline double operator^(const GNvector &vec)const; // dot product
inline double get_parameter(const GNvector& p1, const GNvector& p2);

```

```

inline double magnitude() const; // calculates managitude
inline GNvector unit()const;
inline void print()const{ cout << "Class: GNvector (" << el[0] << ", " << el[1];
                        cout << ", " << el[2] << ", " << el[3] << ")" << endl;}
};

```

80

```

class GNtransmat{
public:
  enum{GN_SCALEMATRIX};
  inline GNtransmat();
  inline GNtransmat(const GNtransmat& trans);
  inline GNtransmat& operator=(const GNtransmat& trans) ;
  inline GNtransmat(double ini);
  inline GNtransmat(double x, double y, double z); //translation matrix
  inline GNtransmat(GNvector& v); //translation matrix
  inline GNtransmat(int refl_plane); // reflection matrix
  inline GNtransmat(int axis, double ang); // rotation about axis
  inline GNtransmat(GNvector& v1, GNvector& v2, int to=1);
  inline GNtransmat(GNvector& v1, GNvector& v2, double ang);
  inline GNtransmat(int axis, double a, double b); //shearing
  inline GNtransmat(int i, double x, double y, double z); // Scaling matrix

  inline ~GNtransmat(){ }

  inline GN4vector& operator[(int i)const]{ return vec[i];}
  inline GN4vector operator*(const GN4vector&)const; // multiplication with
                        // a vector
  inline GNtransmat operator*(const GNtransmat&)const; // matrix multiplication
  inline GNtransmat operator-(const GNtransmat&)const;
  inline GNtransmat operator+(const GNtransmat&)const;
  inline int operator==(const GNtransmat& trans)const;
  inline GNtransmat transpose()const;
  //GNtransmat inverse()const;
  inline void update(double ini);
  inline void update(double x, double y, double z); //translation matrix
  inline void update(int refl_plane); // reflection matrix
  inline void update(int axis, double ang); // rotation about axis
  inline void update(int axis, double a, double b); //shearing
  inline void update(int i, double x, double y, double z); //Scaling matrix
  inline void print()const{ cout << "Class: GNtransmat " << endl;
                        vec[0].print(); vec[1].print();
                        vec[2].print(); vec[3].print();
                        cout << "Finished GNtransmat " << endl;}

private:
  GN4vector vec[4];
};

```

90

100

110

120

```
#endif GNvector_H
```

```

#ifndef _ARTIFACT_H
#define _ARTIFACT_H

#include <string.h>

#ifndef NULL
#define NULL 0
#endif

#define FIXED 1
#define VARIABLE 0

class Design;
class Evaluation;

class Artifact{
private:
    char id[50];
    int flag[9];
    double value[9];

public:
    char **vars;           // Names of numerical variables
    double *var_vals;     // Values of numerical variables
    int size;             // Number of numerical variables

    Artifact *next;

    Artifact(char *,           // Id
             int * =NULL,     // Flags
             double * =NULL,  // Initial parameter values
             char ** =NULL,   // Numerical variable names
             int =NULL,       // Number of numerical variables
             double * =NULL   // The values of numerical variables, if given
            );

    inline char *get_id(void){return id;}
    inline int *get_flags(void){return flag;}
    inline double *get_values(void){return value;}

};

class ATconst{
private:

```

```
Artifact *a1, *a2;
int id1, id2;

public:
  ATconst *next;
  ATconst(){a1 = a2 = NULL;id1=id2=0; next = NULL;}
  virtual ~ATconst(){}

  inline Artifact *get_a1(void){return a1;}
  inline Artifact *get_a2(void){return a2;}
  inline int get_id1(void){return id1;}
  inline int get_id2(void){return id2;}
  virtual Evaluation* evaluate(Design*){return NULL;}
  virtual void improve(Design*){}
};

#endif
```

50

60

```
#ifndef _OBJECT_BIN
#define _OBJECT_BIN

#include "listdeclare.h"

class Artifact;

// Object_Bin class is used primarily to
// interact with the input list of
// Artifacts. It shields other classes in
// Ateams from direct contact with external
// data
class Object_Bin{
private:
    Artifact **bin;           // Private data includes an array of artifacts
    int size;                // and the number of artifacts in the array

public:
    // Constructors and destructor
    Object_Bin();
    Object_Bin(PList(Artifact *));
    ~Object_Bin();

    // Functions for retrieving the number of
    // Artifacts in array, and for getting
    // a particular Artifact identified by
    // its index in the array
    inline int get_size(void) {return size;}
    inline Artifact *get_object(int id){return bin[id-1];}
};

#endif
```

```

#ifndef _CONSTRAINT_BIN
#define _CONSTRAINT_BIN

#include "listdeclare.h"

                                     // Constraint_Bin class is used primarily to
                                     // interact with the input list of
                                     // ATconsts. It shields other classes in
                                     // Ateams from direct contact with external
                                     // data
class Constraint_Bin{
private:
    ATconst **bin;           // Private data includes an array of ATconsts
    int size;               // Size of the array

public:
                                     // Constructors and destructor
    Constraint_Bin();
    Constraint_Bin(PList(ATconst) *);
    ~Constraint_Bin();

                                     // Getting the number of constraints
    inline int get_size(void) {return size;}

                                     // Getting a particular constraint from the
                                     // bin according to its index
    inline ATconst *get_constraint(int id){return bin[id-1];}

                                     // Return the index of this Atconst
    int map_constraint(ATconst*);

};

#endif

```

```

#ifndef _DOBJ_H
#define _DOBJ_H

#define SIZE 9

#include "GNvector.h"

class GNvector;

                                     // Dobj class is the abstraction of a bounding
                                     // box in a design
                                     10

class Dobj{
private:
  char *name;                       // Private data includes
                                     // name – a string identifier for a box
  int id;                             // id – index by which a design refers to a Dobj

  int parameter[SIZE];              // The array for storing values. The implicit
                                     // order is x, y, z of the centroid of the box,
                                     // sizex, sizey, sizez and thetax, thetay, thetaz
                                     20

  int flag[SIZE];                   // Flags indicating which variables are fixed
                                     // and which are variable

  GNvector vec[4];                   // A vector array storing the origin and the
                                     // the three direction vectors of the local
                                     // reference frame of a box

  GNvector corner[8];                // storing corners of the box
                                     30

  char **variable;                   // An array of strings representing arbitrary
                                     // algebraic variable names

  int *var_value;                    // The array of arbitrary variable values
  int v_size;                         // The number of arbitrary variables

                                     // Private functions used by the protocol
  void initialize_axes(void);        // For computing the vector directions of the
                                     // local frame after the parameters have been
                                     // initialized
                                     40

                                     // For finding the corners of the box
  void find_corners(GNtransmat* = NULL);

```

```

int move (GNvector&, int);          // For moving the box to a new location in the
                                   // direction of a given vector

int rotate(GNvector&, int);        // For rotating the box around a given vector
                                   // direction of a given vector
50
int resize(GNvector&, int);        // For resizing the box in a given direction

int modify_vars(int, char*,        // For modifying arbitrary variables
                double);

public:
                                   // Flags to indicate the type of modification
enum{LEFT, RIGHT, CLOCK, ANTICLOCK,
      SMALLER, BIGGER, UNDEFINED, INCREASE, DECREASE};
                                   //
60
Dobj();                          // Constructors and destructor
Dobj(int, char* = NULL, int* = NULL,
     double* = NULL, char** = NULL,
     int = NULL, double* = NULL);
Dobj(Dobj &);
~Dobj();

                                   // Trivial functions for getting and setting
                                   // certain values
70
inline int  get_id(void){return id;}
inline int  get_value(int a){return parameter[a];}
inline void set_id(int a){id = a;}
inline int  set_value(int, int);
inline int* get_values(void){return parameter;}
inline int* get_flags(void){return flag;}
inline char* get_name(void){return name;}
inline GNvector& get_origin(void){return vec[3];}
GNvector& get_unit_axis(int);
double get_variable(char *);
80
inline int  get_vsize(void){return v_size;}
inline int* get_num_values (void){return var_value;}
inline char** get_num_vars(void){return variable;}

                                   // Getting the minimum and the maximum projection
                                   // of the box on one of its local axes
void get_min_max(int, double*, double*);

                                   // Getting the projection of the box on an
                                   // interval with endpoints given by two
                                   // GNvectors
90
void get_projection(GNvector&, GNvector&, double*, double*);

```

```
        // Modifying the values of an object based
        // on a modification operator
int improve(GNvector&, int, char* = NULL, double = NULL);

        // Checking if another box intersects this box
int inside(Dobj *);

        // Getting the relative rotation of two boxes
void relative_rotation(Dobj*, double*);
};
#endif
```

```

#ifndef _DESIGN_H
#define _DESIGN_H

class Dobj;
class Object_Bin;
class Constraint_Bin;
class Evaluation;
class ATconst;

class Design{
private:
    int no_of_evaluations;
    int index;

    Constraint_Bin *cbin;

    Dobj **table;

    int
        no_of_objects,
        no_of_constraints,
        evaluated;

    Evaluation **evaluation;

    Design *next;

public:
    Design();
    Design(Object_Bin* =NULL, Constraint_Bin* =NULL);
    Design(Design &);
    ~Design();

    inline int get_index(void){return index;}
    inline int get_no_of_objects(void){return no_of_objects;}
    inline int get_no_of_constraints(void) {return no_of_constraints;}
    Constraint_Bin* get_constraint_bin(void){return cbin;}

// Class design is the abstraction of a
// complete design
// Private data includes
// The number of constraints
// An integer indicating the position of the
// design in the array when stored
// A pointer to constraint_bin
// An array of Dobjs – boxes
// indexed by their identifiers
// Number of objects in a design
// Number of constraints specified
// Flag to indicate if the design
// has been evaluated
// An array carrying the evaluation of
// each constraint
// A pointer for storing as lists
// Constructors and destructor
// Trivial functions to get data
10
20
30
40

```

```

inline int is_evaluated(void){return evaluated;}
inline int set_no_of_evaluations(int a){no_of_evaluations = a;}
inline int get_no_of_evaluations(void){return no_of_evaluations;}
inline Dobj* get_object(int id) {return (table) ? table[id-1] : NULL;}
inline Dobj* get_object(char*);
inline Design* get_next(void){return next;}
// Trivial functions to set values
inline void set_index(int a){index = a;}
inline void set_no_of_objects(int a){no_of_objects = a;}
inline void set_no_of_constraints(int a){no_of_constraints = a;}
inline void set_evaluated(int a){evaluated = a;}
inline void set_next(Design *p = NULL){next = p;}
// Insert a Dobj in the design
void insert_object(Dobj*, int, int=0);
// Get the evaluation of a particular constraint
Evaluation* get_evaluation(ATconst*);
// Set the evaluation of a particular constraint
void set_evaluation(Evaluation*, ATconst*);
// Check if two designs are the same
int is_duplicate(Design* = NULL);
// Get evaluations for all constraints
inline Evaluation** get_evaluation_array(void){return evaluation;}
// Find out the sum of the evaluations
double sum_evaluation(void);
// Find the scaled evaluation, a number from 0-1
double scaled_evaluation(void);
};
#endif

```

50

60

70

80

```

#ifndef HASH_H
#define HASH_H

class Design;

class HashTable{
private:
    // Class HashTable is the abstraction of a
    // hash table used for storing designs
    // Private data includes
    int table_size;           // The size of the table
    int no_of_lists;         // The number of lists in the table      10
    Design **hashtab;        // The actual table

    Design **queue;          // An array of designs
    int qend;                 // The index of the end of the array
    int qsize;                // The allocated size of the array

protected:
    int hash(Design* = NULL); // For calculating the hash value of a design

public:
    // Constructors and destructor
    HashTable ();
    HashTable (int);
    ~HashTable (){}

    Design* get_any_design(void); // Getting a random design from the table

    Design* lookup(Design*);      // Finding out if a design is already in the table

    int put(Design*);             // Inserting a design into the table      30

    int remove(Design*, int=0);   // Deleting a design from the table

    Design** get_list(void);      // Getting a random list from the table

    Design* get_best_list(void);  // Getting the list of best designs from the table

    // Trivial functions for accessing private data
    int get_size(void){return table_size;}
    int get_no_of_designs(void){return qend+1;}      40

    // Getting a handle to the array of designs
    Design **getq(void){return queue;}

    // Getting a handle to the hashtable itself

```



```
Design **geth(void){return hashtab;}  
};  
#endif
```

```

#ifndef _MEMORY_H
#define _MEMORY_H

class Design;
class HashTable;
class Object_Bin;
class Constraint_Bin;
class Operator_bin;

class Memory{
private:
    HashTable *design_table;           // A hashtable of designs

    int no_of_designs;                // Number of designs in Memory when initialized
    int table_size;                   // Size of the table

    double best_design;               // The evaluation of the best design so far

public:
    // Constructor Destructor pair
    Memory(HashTable* = NULL);
    Memory(int, Operator_bin* = NULL,
           Object_Bin* = NULL,
           Constraint_Bin* = NULL);
    ~Memory();

    Design *get_design(void);         // Get any random design

    int add_design(Design*);          // add a new design to the table

    // Delete a particular design from
    // the table
    int delete_design(Design*, int = 0);

    Design* get_best_list(void);      // Get the list of best designs in the table

    // Return a handle to the hashtable – used in
    // the destroyers
    HashTable *gett(void){return design_table;}

    int get_no_of_designs(void)       // Get the initial size of store
    {return no_of_designs;}

    int get_no_hashtable(void);       // Get the actual number of designs in the table
};

```

#endif

```
#ifndef _OPERATOR_H
#define _OPERATOR_H

class Memory;
class Operator_bin;

// Class Operator is used to provide a uniform
// interface to all operators. It includes
// virtual functions that are defined for
// every subclass 10

class Operator{
public:
// Getting the name of the operator
virtual char* get_name(void){return 0;}

// Firing an operator
virtual void fire(Memory *mem, Operator_bin *bin){}
}; 20

#endif
```

```

#ifndef _CONSTRAINT_H
#define _CONSTRAINT_H

#include "operator.h"
#include <iostream.h>

#include "artifact.h"

class Operator_bin;
class Memory;
class Design;
class Num_Const;

// Class Constraint is the abstraction of
// a constraint – both algebraic and
// qualitative

class Constraint:public Operator{
private:
    char *name; // String identifier
    ATconst *c; // A pointer to the Atconst supplied as input

    int no_of_evaluations; // The number of times this constraint has
                          // been evaluated

public:
    // Constructor destructor pair
    Constraint(ATconst *a = NULL){name="constraint"; c=a; no_of_evaluations = 0;}
    virtual ~Constraint();

    // Getting the name and firing the operator
    inline virtual char* get_name(void){return name;}
    virtual void fire(Memory *, Operator_bin*);

    void evaluate(Design*); // Simply evaluating the design for a constraint

    void improve(Design*); // Improve design, do nothing else

    // Getting the number of evaluations
    int get_no_of_evaluations(void){return no_of_evaluations;}
    int reset_no_of_evaluations(void){no_of_evaluations = 0;}
};

#endif

```

```
#ifndef _CROSSOVER_H
#define _CROSSOVER_H

#include "operator.h"

class Operator;
class Memory;
class Operator_bin;

// Class crossover takes two designs and
// mates them randomly
class Crossover:public Operator{
private:
    char *name;           // String identifier

protected:
    // Method that does the mating
    void cross(Memory*, Operator_bin*);

public:
    // Constructor destructor pair
    Crossover(){name = "crossover";}
    virtual ~Crossover(){}

    // For getting the name and firing the operator
    inline virtual char* get_name(void){return name;}
    virtual void fire(Memory *mem, Operator_bin* bin)
        {cross(mem, bin);}

};

#endif
```

```
#ifndef _MUTATE_H
#define _MUTATE_H

class Operator;
class Memory;
class Operator_bin;

class Mutate:public Operator{
private:
    char *name;                // String identifier name of the operator           10

protected:
                                // Function for performing the mutation
    void mutate(Memory*, Operator_bin*);

public:

    Mutate(){name = "mutate";}    // Constructor destructor pair
    virtual ~Mutate(){}

                                // Getting the name and firing the operator
    inline virtual char* get_name(void){return name;}
    virtual void fire(Memory *mem, Operator_bin* bin)
        {mutate(mem, bin);}

};

#endif
```

```

#ifndef _DESTROYER_H
#define _DESTROYER_H

class Operator;
class Operator_bin;

// An abstract base class for destroyers
class Destroyer:public Operator{
public:
    virtual char* get_name(void){return NULL;}           10
    virtual void fire(Memory *mem){}
};

// For destroying duplicates
class Duplicate_Destroyer:public Destroyer{
private:
    char *name;           // String identifier

protected:
    // Method that looks for duplicates           20
    // and destroys them
    void destroy_duplicates(Memory *);

public:
    // Constructor destructor
    Duplicate_Destroyer(){name = "duplicate_destroyer";}
    virtual ~Duplicate_Destroyer();

    // For getting the name and firing the destroyer           30
    inline virtual char* get_name(void){return name;}
    virtual void fire(Memory *mem, Operator_bin *opbin)
        {destroy_duplicates(mem);}
};

// For destroying designs as a regular operator
class Design_Destroyer:public Destroyer{
private:
    char *name;           // String identifier           40

protected:
    // Method that destroys designs based on
    // their evaluations
    void destroy_design(Memory *);

public:

```

```

// Constructor destructor
Design_Destroyer(){name = "design_destroyer";}
virtual ~Design_Destroyer();

// For getting the name and firing the destroyer
inline virtual char* get_name(void){return name;}
virtual void fire(Memory *mem, Operator_bin *opbin)
{destroy_design(mem);}
};

class Limit_Memory_Destroyer:public Destroyer{
private:
char *name; // String identifier

protected:
// Method that goes through the store and
// destroys designs
void limit_memory(Memory *);

public:
// Constructor destructor
Limit_Memory_Destroyer(){name = "limit_memory_destroyer";}
virtual ~Limit_Memory_Destroyer();

// For getting the name and firing the destroyer
inline virtual char* get_name(void){return name;}
virtual void fire(Memory *mem, Operator_bin *opbin)
{limit_memory(mem);}
};

#endif

```

```

#ifndef _OPERATOR_BIN
#define _OPERATOR_BIN

class Operator;
class Constraint_Bin;
class Design;

class Operator_bin{
private:
    Operator **bin;           // Private data includes           10
                             // bin – an array of operators
    Constraint_Bin *cbin;    // cbin – a pointer to the constraint_bin
    int size;                // size – the number of operators in bin
    double probability[4];   // The frequency of firing of destroyers,
                             // mutation, crossover operators and constraints

public:
                             // Constructor destructor pair
    Operator_bin(Constraint_Bin* = NULL,           20
                 double = 0.25, double = 0.25,
                 double = 0.25, double = 0.25);
    ~Operator_bin();

                             // Trivial functions for accessing private data
    inline int get_size(void){return size;}
    inline Constraint_Bin* get_constraint_bin(void){return cbin;}

                             30
                             // Getting an operator from the bin by using its
                             // name – string identifier, and optional integer
                             // argument for getting one of the constraints
    Operator* get_operator(char*, int=0);

                             // Getting a random operator from the bin
    Operator* get_operator(void);

                             // For taking a design and evaluating it
                             // by passing it to each of the constraints
                             // in the bin           40
    void evaluate(Design*);
};

#endif

```


Appendix B

Header files for the QSRs

This appendix provides the listings for the header files used in the QSR implementation. High level details of these classes are provided in chapter 4.

```
#ifndef _spatial
#define _spatial

#include "GNvector.h"
#include "relationship.h"
#include "artifact.h"
class Ref_frame;

class Evaluation;
class Srel: public ATconst
{
    // This ABSTRACT class defines the
    // essential spatial relationship
    // Defines the axis for the relationship and a
    // reference frame which defines this axis
    // Note that this forms the base class for
    // all primitive
    // relationships, actual 3-D relationships
    // would be represented by the composite
    // classes
protected:
    enum{X,Y,Z};
    int axis;
    Ref_frame* ref ;
public:
    Srel(int, Ref_frame* = NULL);
    virtual char* get_classname() = 0 ;
}
```

```
virtual Role* get_source(){return NULL;}
virtual Role* get_target(){return NULL;}
virtual Evaluation* evaluate(Design*){return NULL ;}
virtual void improve(Design*){}
virtual ~Srel(){}
```

```
};
```

```
#endif
```

```
#ifndef _RELN
#define _RELN

#include <iostream.h>
class Artifact ;

class Role
{
    // This class describes the role of an
    // object in a relationship ;
    // e.g "parent", "child", "supported_by", etc.
private:
    char* description;
    Artifact* obj;
public:
    Role(Artifact*, char* = NULL);
    ~Role();
    inline char* get_description();
    inline Artifact* get_obj(){return obj;}
};

class Relationship{
protected:
    Relationship(){}
    char reln_type[30]; // string representation of the type
public: // an abstract class
    virtual char* get_classname() = 0;
    virtual Role* get_source(){return NULL;}
    virtual Role* get_target(){return NULL;}
    virtual ~Relationship(){}
};

#endif
```

```

#ifndef _SIZE_QSR
#define _SIZE_QSR

#include <iostream.h>
#include "qsr.h"

                                     // This file defines the base classes
                                     // for the size and
                                     // orientation relationships between
                                     // design objects
                                     10

class Size_rel : public QSR
{
protected:
    double relative_size ;
public:
    Size_rel(double, Role*, Role*, int, Ref_frame*);
    virtual ~Size_rel(){}
    inline virtual char* get_classname() {return "Size_rel";}
    virtual Evaluation* evaluate(Design*);
};
                                     20

class Orientation: public QSR
{
protected:
    enum{THETA_X, THETA_Y, THETA_Z};
    double relative_angle[3] ;
public:
    Orientation(Role*, Role*, int, Ref_frame*);
    virtual ~Orientation(){}
    inline virtual char* get_classname() {return "Orientation";}
    virtual Evaluation* evaluate(Design*);
};
                                     30

class Parallel:public Orientation
{
public:
    Parallel(Role*, Role*, int, Ref_frame*);
    inline virtual char* get_classname() {return "Parallel";}
    virtual Evaluation* evaluate(Design*);
    virtual ~Parallel(){}
};
                                     40

```



```
class Perpendicular:public Orientation
{
public:
    Perpendicular(Role*, Role*, int, Ref_frame*);
    inline virtual char* get_classname() {return "Perpendicular";}
    virtual Evaluation* evaluate(Design*);
    virtual ~Perpendicular(){}
};

#endif
```

```

#ifndef _REF_FRAME
#define _REF_FRAME

#include "GNvector.h"
class Artifact;
class Design ;
class Ref_frame
{
protected:
  GNvector origin;           // initialized to (0,0,0)
public:
  Ref_frame(){}
  virtual ~Ref_frame(){}
  inline void set_origin(GNvector v){origin=v;}
  inline GNvector& get_origin(){return origin;}
  inline virtual GNvector get_unit_vector(int, Design* = NULL){return origin;}
  inline virtual void set_unit_vector(int,GNvector&){}
};

class Ref_line: public Ref_frame
{
  GNvector vec ;           // Only one vector is defined
                          // unit vectors along the axis
public:
  Ref_line(){}
  Ref_line(GNvector&, GNvector&);
  virtual ~Ref_line(){}
  inline virtual GNvector get_unit_vector(int, Design* = NULL){return vec;}
  inline virtual void set_unit_vector(int,GNvector&);
};

class Ref_axes: public Ref_frame
{
  enum{X,Y,Z};           // The axes are directly defined
  GNvector vec[3] ;      // unit vectors along each axis
public:
  Ref_axes();
  Ref_axes(GNvector&, GNvector&, GNvector&, GNvector&);
  virtual ~Ref_axes(){}
  inline virtual GNvector get_unit_vector(int i, Design* = NULL)
    {return vec[i];}
  inline virtual void set_unit_vector(int,GNvector&);
};

```

```
class Ref_object: public Ref_frame
{
    // The axes are derived from the local
    // axis system of another object
    Artifact* art_ref;
public:
    Ref_object(Artifact*);
    virtual ~Ref_object(){}
    inline virtual GNvector get_unit_vector(int, Design* = NULL);
};
#endif
```

```

#ifndef _PRIM_QSR
#define _PRIM_QSR

#include <iostream.h>
#include "qsr.h"

class Evaluation ;

                                     // Subtypes of the QSR class define the
                                     // many kinds of primitive relationships
                                     // that can occur. These types are follows:           10
                                     // Point-Interval types (PI_base forms the
                                     // base class) p, f, i, b, m (point-interval
                                     // types: p is plus) interval-interval types
                                     // (II_base forms the base class) pp, fp, ip,
                                     // if, ii, bi, bf, bp, mp, mf, mi, mb, mm.
                                     // These are pretty horrible classes, we do
                                     // not expect the user to deal with them at
                                     // this level of abstraction. Higher level
                                     // operators can be built up from this
                                     // primitives                                           20

class PI_base: public QSR
{
public:
  PI_base(int, Ref_frame*, Role* = NULL, Role* = NULL);
  ~PI_base(){}
  inline virtual Evaluation* check_rel(double,Dobj*, Dobj*, Design*){return 0;}
  inline virtual char* get_classname() {return "PI_base";}
  inline virtual Evaluation* evaluate(Design*){return NULL;}
};                                                                                   30

class PI_plus: public PI_base
{
public:
  PI_plus(int, Ref_frame*, Role* = NULL, Role* = NULL);
  ~PI_plus(){}
  Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
  inline virtual char* get_classname() {return "PI_plus";}
};                                                                                   40

class PI_front: public PI_base
{
public:

```

```

PI_front(int, Ref_frame*, Role* = NULL, Role* = NULL);
~PI_front(){}
Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
inline virtual char* get_classname() {return "PI_front";}
};

```

50

```

class PI_in: public PI_base
{
public:
PI_in(int, Ref_frame*, Role* = NULL, Role* = NULL);
~PI_in(){}
Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
inline virtual char* get_classname() {return "PI_in";}
};

```

60

```

class PI_back: public PI_base
{
public:
PI_back(int, Ref_frame*, Role* = NULL, Role* = NULL);
~PI_back(){}
Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
inline virtual char* get_classname() {return "PI_back";}
};

```

70

```

class PI_minus: public PI_base
{
public:
PI_minus(int, Ref_frame*, Role* = NULL, Role* = NULL);
~PI_minus(){}
Evaluation* check_rel(double,Dobj*, Dobj*,Design*);
inline virtual char* get_classname() {return "PI_minus";}
};

```

80

```

class II_base:public QSR
{
//base class for the interval-interval
// relationships
protected:
PI_base * rel1;
PI_base * rel2;

// rel1 represents the relationship of min of
// source w.r.t the interval represented by
// target rel2 represents the relationship of

```

90

```

// max of source w.r.t target
public:
  II_base(Role*, Role*,int, Ref_frame* );
  ~II_base();
  inline virtual char* get_classname() {return "II_base";}
  virtual Evaluation* evaluate(Design*);
};
100

class II_pp: public II_base
{
public:
  II_pp(Role*, Role*,int, Ref_frame* );
  ~II_pp(){}
  inline virtual char* get_classname() {return "II_pp";}
  virtual Evaluation* evaluate(Design*);
};
110

class II_fp: public II_base
{
public:
  II_fp(Role*, Role*,int, Ref_frame*);
  ~II_fp(){}
  inline virtual char* get_classname() {return "II_fp";}
  virtual Evaluation* evaluate(Design*);
};
120

class II_ip: public II_base
{
public:
  II_ip(Role*, Role*,int, Ref_frame*);
  ~II_ip(){}
  inline virtual char* get_classname() {return "II_ip";}
  virtual Evaluation* evaluate(Design*);
};
130

class II_if: public II_base
{
public:
  II_if(Role*, Role*,int, Ref_frame*);
  ~II_if(){}
  inline virtual char* get_classname() {return "II_if";}
  virtual Evaluation* evaluate(Design*);
};
```

APPENDIX B. HEADER FILES FOR THE QSRS

135

```
class II_ii: public II_base
{
public:
  II_ii(Role*, Role*,int, Ref_frame*);
  ~II_ii(){}
  inline virtual char* get_classname() {return "II_ii";}
  virtual Evaluation* evaluate(Design*);
};
```

140

150

```
class II_bi: public II_base
{
public:
  II_bi(Role*, Role*,int, Ref_frame*);
  ~II_bi(){}
  inline virtual char* get_classname() {return "II_bi";}
  virtual Evaluation* evaluate(Design*);
};
```

160

```
class II_bf: public II_base
{
public:
  II_bf(Role*, Role*,int, Ref_frame*);
  ~II_bf(){}
  inline virtual char* get_classname() {return "II_bf";}
  virtual Evaluation* evaluate(Design*);
};
```

170

```
class II_bp: public II_base
{
public:
  II_bp(Role*, Role*,int, Ref_frame*);
  ~II_bp(){}
  inline virtual char* get_classname() {return "II_bp";}
  virtual Evaluation* evaluate(Design*);
};
```

180

```
class II_mp: public II_base
{
public:
  II_mp(Role*, Role*,int, Ref_frame*);
  ~II_mp(){}
};
```

```
inline virtual char* get_classname() {return "II_mp";}
virtual Evaluation* evaluate(Design*);
};

class II_mf: public II_base
{
public:
  II_mf(Role*, Role*,int, Ref_frame*);
  ~II_mf(){}
  inline virtual char* get_classname() {return "II_mf";}
  virtual Evaluation* evaluate(Design*);
};

class II_mi: public II_base
{
public:
  II_mi(Role*, Role*,int, Ref_frame*);
  ~II_mi(){}
  inline virtual char* get_classname() {return "II_mi";}
  virtual Evaluation* evaluate(Design*);
};

class II_mb: public II_base
{
public:
  II_mb(Role*, Role*,int, Ref_frame*);
  ~II_mb(){}
  inline virtual char* get_classname() {return "II_mb";}
  virtual Evaluation* evaluate(Design*);
};

class II_mm: public II_base
{
public:
  II_mm(Role*, Role*,int, Ref_frame*);
  ~II_mm(){}
  inline virtual char* get_classname() {return "II_mm";}
  virtual Evaluation* evaluate(Design*);
};

#endif
```

```
QSR ** comp ;           // components of the disjunction, stored
                        // as an array
int comp_num ;         // no of elements in the disjunction
public:
QSR_disjunc(Role*, Role*, int, Ref_frame*);           50
virtual ~QSR_disjunc() ;
inline virtual char* get_classname() {return "QSR_disjunc";}
virtual Evaluation* evaluate(Design*);
};

class QSR_3D : public QSR
{
protected:
QSR * comp[3];           // one in each of the three directions
public:                 60
QSR_3D(Role*, Role*, int, Ref_frame*);
virtual ~QSR_3D(){}
inline virtual char* get_classname() {return "QSR_3D";}
virtual Evaluation* evaluate(Design*);
};

#endif
```

```

#ifndef _COMP_QSR
#define _COMP_QSR

#include <iostream.h>
#include "qsr.h"

class II_ii;
// this file defines some sample higher level
// QSRS which use disjunctions and the
// three-D relations
// First the one-D relationships
// Relationship to specify symmetry of one
// object w.r.t another
// source must be <in> target
class Centered: public QSR{
public:
    Centered(Role*, Role*, int, Ref_frame*);
    virtual ~Centered();
    virtual Evaluation* evaluate(Design* );
};

class Overlap: public QSR_disjunc{
public:
    Overlap(Role*, Role*, int, Ref_frame*);
    virtual ~Overlap(){}
};

class Overlap_front: public QSR_disjunc{
public:
// This class covers the case of overlap
// from the front,
// NOTE: it includes the flush-contact
// case as well
    Overlap_front(Role*, Role*, int, Ref_frame*);
    virtual ~Overlap_front(){}
};

class Overlap_back: public QSR_disjunc{
public:
// This class covers the case of overlap

```

```

        // from the back,
        // NOTE: it includes the flush-contact
        // case as well
Overlap_back(Role*, Role*, int, Ref_frame*);
virtual ~Overlap_back(){}
};

```

50

```

class Inside: public QSR_disjunc{
public:
        // what we actually mean when we say inside,
        // if, ii, bi
    Inside(Role*, Role*, int, Ref_frame*);
    virtual ~Inside(){}
};

```

60

```

class Touch_contact: public QSR_disjunc{
public:
    Touch_contact(Role*, Role*, int, Ref_frame*);
    virtual ~Touch_contact(){}
};

```

```

class No_contact: public QSR_disjunc{
public:
    No_contact(Role*, Role*, int, Ref_frame*);
    virtual ~No_contact(){}
};

```

70

// Now the Three D relationships

```

class Abuts: public QSR_3D{
public:
    Abuts(Role*, Role*, int, Ref_frame*);
    virtual ~Abuts(){}
};

```

80

```

class Intersects: public QSR_3D{
public:
    Intersects(Role*, Role*, int, Ref_frame*);
    virtual ~Intersects(){}
};

```

90

```
class Contains: public QSR_3D{
public:
  Contains(Role*, Role*, int, Ref_frame*);
  virtual ~Contains(){}
};

// Ternary relationships 100

class Between: public QSR{
  Role* center_obj; // is between source and target
  QSR_3D* comp[2];
public:
  Between(Role*, Role*, Role*, int, Ref_frame*);
  virtual ~Between();
  virtual Evaluation* evaluate(Design*);
}; 110

class Connects: public QSR{
  Role* connector;
  QSR* comp[6]; // cannot directly use any of the threeD
               // relationships, unless we define new ones..

public:
  Connects(Role*, Role*, Role*, int, Ref_frame*);
  virtual ~Connects();
  virtual Evaluation* evaluate(Design*); 120
};

#endif
```

```
#ifndef _TOUCH_
#define _TOUCH_

#include <iostream.h>
#include "qsr.h"

class Touch: public QSR{           // The base class for all relationships
public:
    Touch(Role*, Role*, int, Ref_frame*);
    virtual Evaluation* evaluate(Design*);
    virtual ~Touch(){}
};

#endif
```

```

#ifndef _EVAL
#define _EVAL
#include "GNvector.h"

class Dobj ;

class mod_operator
{
    // This simple "class" stores the modification
    // operators suggested by the evaluation of a
    // design. Move the Dobj indicated by obj LEFT
    // or RIGHT along axis, rotate it clockwise or
    // anticlockwise, make the object SMALLER
    // or BIGGER
    public:
    enum{LEFT, RIGHT, CLOCK, ANTICLOCK, SMALLER, BIGGER,
        UNDEFINED, INCREASE, DECREASE};
    int op;
    Dobj* obj ;
    GNvector axis_vec ;           // axis along which the modification
    // is defined, it is defined here for
    // convenience, especially useful for 3D QSRs
    //defining the exact amount of change for
    char var[50];                // numerical constraints, the name of
    double change;               // the variable and the change requested

    mod_operator(int, Dobj*, GNvector&);
    mod_operator(int, Dobj*, char*, double = NULL);
    mod_operator(mod_operator&);
    ~mod_operator(){obj=NULL;op=0;change=0;}
};

class Evaluation
{
    // This simple "class" stores an evaluation
    // as recorded by a QSR object.
    int count ;                   // Number of elements in the array
    public:
    double eval;
    mod_operator** mod_op_arr;    // array of possible modifications,
    // recorded during the evaluation

    Evaluation();
    ~Evaluation();
};

```

```
Evaluation(Evaluation&);  
void delete_ops();  
inline int get_no_ops(){return count ;}  
void add_op(mod_operator* ) ;  
inline mod_operator* get_random_op();  
};  
  
#endif
```

Bibliography

- [Ack87] D. H. Ackley. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, 1987.
- [BG92] F. Brewer and D. Gajski. A design process model. In C. Tong and D. Sriram, editors, *Artificial Intelligence in Engineering Design Vol. 1*. Academic Press Incorporated, 1992.
- [CA] J. K. Choy and A. M. Agogino. Symon: Automated symbolic monotonicity analysis system for qualitative design optimization. *Expert Systems Laboratory, Department of Mechanical Engineering, University of California, Berkeley*.
- [CB89] R. Cusic and M. F. Bramlette. A comparative evaluation of search methods applied to parametric design of aircraft. In *International Conference on Genetic Algorithms*, pages 213 – 218, 1989.
- [Dav90] E. Davis. Representations of commonsense knowledge. page 515. 1990.
- [Des93] P. S. Desouza. *Asynchronous organizations for multi-algorithm problems*. PhD dissertation, Carnegie Mellon University, Department of Electrical and Computer Engineering, April 1993.
- [EGLS88] J. R. Dixon E. G. Libardi and M. K. Simmons. Computer environments for the design of mechanical assemblies. In *Engineering with Computers*, Vol. 3, pages 121–136. Springer-Verlag, New York, 1988.
- [Fal90] B. Faltings. Qualitative kinematics in mechanisms. *Artificial Intelligence*, 44(1–2):89–119, 1990.
- [For88] K. D. Forbus. Qualitative physics, past, present and future. In H. Shrobe & Morgan Kaufman, editor, *Survey Talks in Artificial Intelligence*, pages 239–296. 1988.
- [Gro85] M. Gross. *Design as exploring constraints*. PhD dissertation, Massachusetts Institute of Technology, 1985.
- [Hay85] P. J. Hayes. Naive physics I, ontology for liquids. In Ablex Publishing J. R. Hobbs & R. C. Moore, editor, *Formal theories of the commonsense World*, pages 71–107. 1985.
- [JM90] G. Joe and A. Mukerjee. A qualitative model for space. In *Proceedings of AAAI-90*, pages 721–727, Boston, August 1990.

- [Jos89] L. Joskowicz. Simplification and abstraction of dynamic behavior. In *Proceedings IJCAI-89*, August 1989.
- [LF69] C. T. Leondes and G. J. Freidman. Constraint theory, part i, fundamentals. In *IEEE Transactions on Systems Science and Cybernetics*, Vol. SSC-5, No. 1, pages 48–56, Boston, January 1969.
- [LG83] R. A. Light and D. C. Gossard. Variational geometry: a new method for modifying part geometry for finite element analysis. *Computers and Structures*, 17(5):903–909, 1983.
- [Lig80] R. A. Light. Symbolic dimensioning in computer aided design. M.S. thesis, Massachusetts Institute of Technology, February 1980.
- [Lin81] V. C. Lin. Variational geometry in computer aided design. M.S. thesis, Massachusetts Institute of Technology, May 1981.
- [Loo71] F. A. Lootsma. In F. A. Lootsma, editor, *Numerical methods for non-linear optimization*, Conference Sponsored by the Science Research Council, University of Dundee, Scotland, pages ix – x, 1971.
- [Man90] M. Mantyla. A modeling system for top-down design of assembled products. *IBM Journal of Research and Development*, 34(5):636+, September 1990.
- [MGF87] J. Anderson M. Gross, S. Ervin and A. Fleisher. Designing with constraints. In Yehuda E. Kalay, editor, *Computability of Design*, pages 53–83. 1987.
- [Muk91] A. Mukerjee. Qualitative geometric modeling. Technical Report TR-91-07, Texas A&M University, 1991.
- [Mur92] S. S. Murthy. *Cooperating agents to design manipulators from task specifications*. PhD dissertation, Carnegie Mellon University, Department of Electrical and Computer Engineering, September 1992.
- [PS82] C. H. Papadimitriou and K. Steiglitz. *Algorithms and complexity*, chapter 2.7, page 51. Prentice-Hall Inc., 1982.
- [RC86] A. A. G. Requicha and S. Chan. Representation of geomtric features, tolerances, and attributes in solid modelers based on constructive geometry. *IEEE Journal of Robotics and Automation*, 2(3):156–166, September 1986.
- [Ser84] D. Serrano. Mathpak: An interactive preliminary design package. M.S. thesis, Massachusetts Institute of Technology, January 1984.
- [Ser87] D. Serrano. *Constraint management in conceptual design*. PhD dissertation, Massachusetts Institute of Technology, Department of Mechanical Engineering, October 1987.
- [Sri94] D. Sriram. *Intelligent systems for engineering: knowledge based and neural network approaches*. Forthcoming, 1994.
- [SS78] G. L. Steele and G. J. Sussman. Constraints. In *AI Memo no. 502*, November 1978.

- [Ste80] G. J. Steele. *The definition and implementation of a computer programming language based on constraints*. PhD dissertation, Massachusetts Institute of Technology, August 1980.
- [Ste92] L. Steinberg. Design as top down refinement and constraint propagation. In C. Tong and D. Sriram, editors, *Artificial Intelligence in Engineering Design Vol. 1*. Academic Press Incorporated, 1992.
- [Sut63] I. Sutherland. *Sketchpad-a man machine graphical interface*. PhD dissertation, Massachusetts Institute of Technology, 1963.
- [TD92] S. T. Talukdar and P. S. Desouza. Scale efficient organizations. In *Proceedings of the IEEE International Conference on Systems Science and Cybernetics*, 1992.
- [TD93] S. N. Talukdar and P. Desouza. Object organizations and super-objects. In *Engineering Design: The creation of products and processes*. McGraw Hill, 1993.
- [TS92] C. Tong and D. Sriram. Introduction. In *Artificial Intelligence in Engineering Design Vol. 1*. Academic Press Incorporated, 1992.