A LOGIC BASED APPROACH TO FACTORY DESIGN

Vol. 1

by

MARGOT FELICITY BRERETON

B.Sc., Mechanical Engineering
University of Bristol, England
(1984)

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

Master of Science in Technology and Policy

at the

Massachusetts Institute of Technology

May 1988

Signature of Author_____
Department of Mechanical Engineering

Certified by _____
Steven H. Kim
Assistant Professor, Mechanical Engineering
Thesis Supervisor

Accepted by _____
Professor Richard de Neufville
Chairman, Technology and Policy Program

A LOGIC BASED APPROACH TO FACTORY DESIGN

by

MARGOT FELICITY BRERETON

Submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements for the Degree of Master of Science in Technology and Policy (May 1988)

Abstract

A systematic automated approach to factory design, based upon formal logic, has been developed in order to unify the components of the design problem through a formal framework. The decision problem of design, which determines whether there is a factory design which meets the functional requirements of the factory subject to the imposed constraints, is represented in a formal logic model. It consists of three phases, preliminary process planning, machine selection and layout. This model is used to prove properties of the design problem.

A prototype factory design software system has been developed which incorporates optimization criteria, as well as domain dependent and domain independent factory design knowledge, into the decision problem. It is expressed in a logic programming language.

The factory design problem is proved to be computable, but the optimization problem of factory design is intractable, belonging to the class of NP-complete problems. A complexity analysis suggests that the best strategy for finding satisficing solutions is a hybrid strategy which is primarily informed depth first search with backtracking, with a limited amount of local breadth first search.

The implications of knowledge based systems for reducing the intractability of the problem in a given domain are examined. It is proved that knowledge of the functional requirements of the products and machines in the factory does not aid the layout designer. All useful knowledge for factory layout is expressed through machine specifications, relationships between machines and spatial constraint knowledge which constrains a machine to a given location. Two types of knowledge are defined, parameter and constraint knowledge. Constraint knowledge reduces the number of solutions to the decision problem, whereas parameter knowledge expresses preferences which allow us to evaluate which solution is more desirable. A large amount of factory design knowledge is parameter knowledge. Parameter knowledge may be expressed generally; therefore a large amount of software code is transportable from domain to domain. However parameter knowledge does

not reduce the intractability of the problem, except by virtue of pruning processes such as the Branch and Bound Method.

The prototype software system encodes the machine selection and layout phases of design. It is demonstrated with domain dependent knowledge for automobile assembly plant design. A Generate and Test method is used which allows expression of the unique structure featured by the factory layout problem.

The prototype software system

(a) makes the objectives of each design explicit. Each objective is linked to layout rules. The order of the rule firing depends upon the ranking of the objectives by the user.

(b) allows both quantitative and qualitive objectives to be expressed.

(c) decouples domain dependent and independent knowledge, facilitating expansion of the knowledge base to new domains.

(d) gives insight into the final design through an explanation module. This traces the path taken through the rule base, explaining each placement decision made in order to arrive at the final design.

Thesis Supervisor: Steven H. Kim

Title: Assistant Professor of Mechanical Engineering

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to those who have made my stay at MIT possible, and to those who have guided me and supported me throughout my studies here.

I am extremely grateful to my thesis supervisor, Professor Steven Kim, for his direction, inspiration and encouragement throughout the duration of this thesis.

I thank Professor Richard de Neufville, Chairman of the Technology and Policy Program, for his guidance and vision, and for establishing the Program in Technology and Policy, which has enabled me to expand my horizons beyond the traditional engineering realm to appreciate the role of engineers and technology in the context of society.

I thank my colleagues in the Knowledge Systems Program and those who lurk in the basement of building 35 for hours of fruitful discussions and for their kindness and support.

I am indebted to my two undergraduate student researchers and programmers, Pedro Ortiz and George Yu for their hours of toil and for being such a pleasure to work with.

Mr. Bert Taschereau of General Motors was extremely helpful to me in my efforts to understand factory design. I thank him for his patience and his valuable time.

Life in Cambridge would not have been complete without Bobbi, Diana, Hiroki, Jeff, Judy and Lalitha. I feel extremely lucky to have found such close friends who made me feel so at home in this country and who have nurtured and tolerated me beyond the call of duty.

I thank my sponsors, the Science and Engineering Research Council of Great Britain and Rolls-Royce plc. without whose financial support I could not have come to MIT.

Finally, I thank my family for their encouragement, love and support.

In loving memory of

Kevin,

a friend,

an inspiration.

TABLE OF CONTENTS

CHAPTER 3: Mathematical Background

CHAPTER 1
INTRODUCTION

1.1 Introduction to Factory Design

Factory design is initiated in order to achieve the layout of production facilities that will best manufacture a set of products and perceived future products. Once designed, a factory is subject to continuous redesign in order to accommodate the change over time in product volume and mix, some of which may not have been conceived of during the original design. Recent estimates are that 8 percent of the US GNP can be attributed to new factory design and construction [Tompkins and White, 1984]. This figure does not reflect the investment in factories requiring periodic redesign due to change in product line, such as automobile assembly plants.

Perhaps more important than the cost of the factory is the implication of the design on the manufacture of goods. A factory is a physical embodiment of a company's manufacturing strategy. It plays a large factor in the selection of product volume and mix, process planning and scheduling.

In its largest sense factory design involves the functions shown in figure 1.1. That is, it begins with a consideration of variety and volume of products to be made and the method of manufacture, before considering machine selection, the material flow and the physical layout of machinery and handling equipment. Once a product selection has been made, the factory design is performed in three stages: preliminary process planning, machine selection and layout. Figure 1.2 illustrates these stages which culminate in the layout of a factory. The preliminary process planning stage takes the part specification from the drawing and develops a preliminary process plan. It is preliminary in the sense that the process has been determined but the machine on which it is to be performed has not, so the details of fixturing and operation sequencing are not fully developed at this preliminary stage. Figure 1.3 illustrates the inputs and outputs of the factory design stages.

Figure 1.1

The Functions of Factory Design

Product selection: What should we make?

Preliminary Planning: How should it be made?

Selection of Manufacturing and Handling Components: What should be used to make the product and how should the product be transported between the cells?

Layout - Establishment of Component Relationships: How do the manufacturing and handling components relate to each other? Share common services? Linked by product operation sequence?

Layout - Establishment of Component Locations: Where should manufacturing and handling components physically reside in the factory?

Analysis of Design and Modification: How well does it work and how can it be made to work better?

Figure 1.2

Schematic Illustrating the Culmination of the Functions of Factory Design in the Physical Layout of Manufacturing, Handling and Service Components

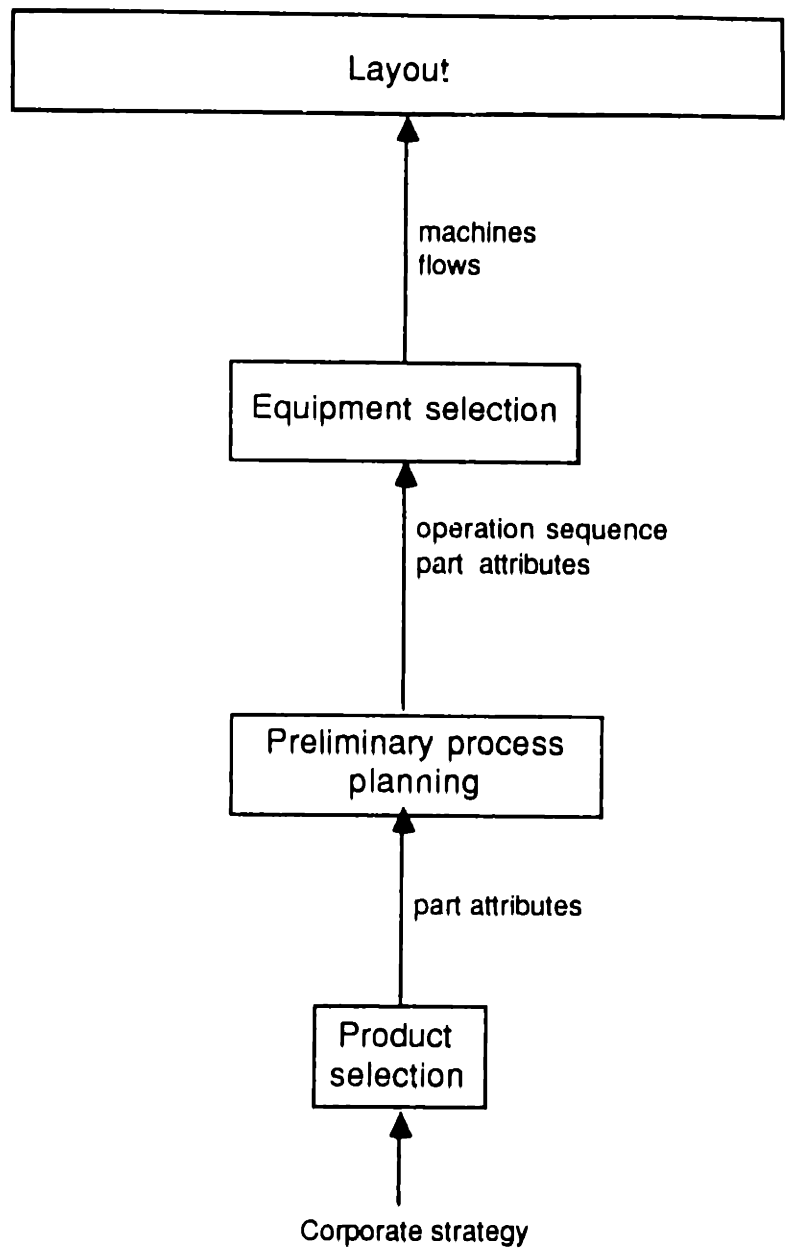
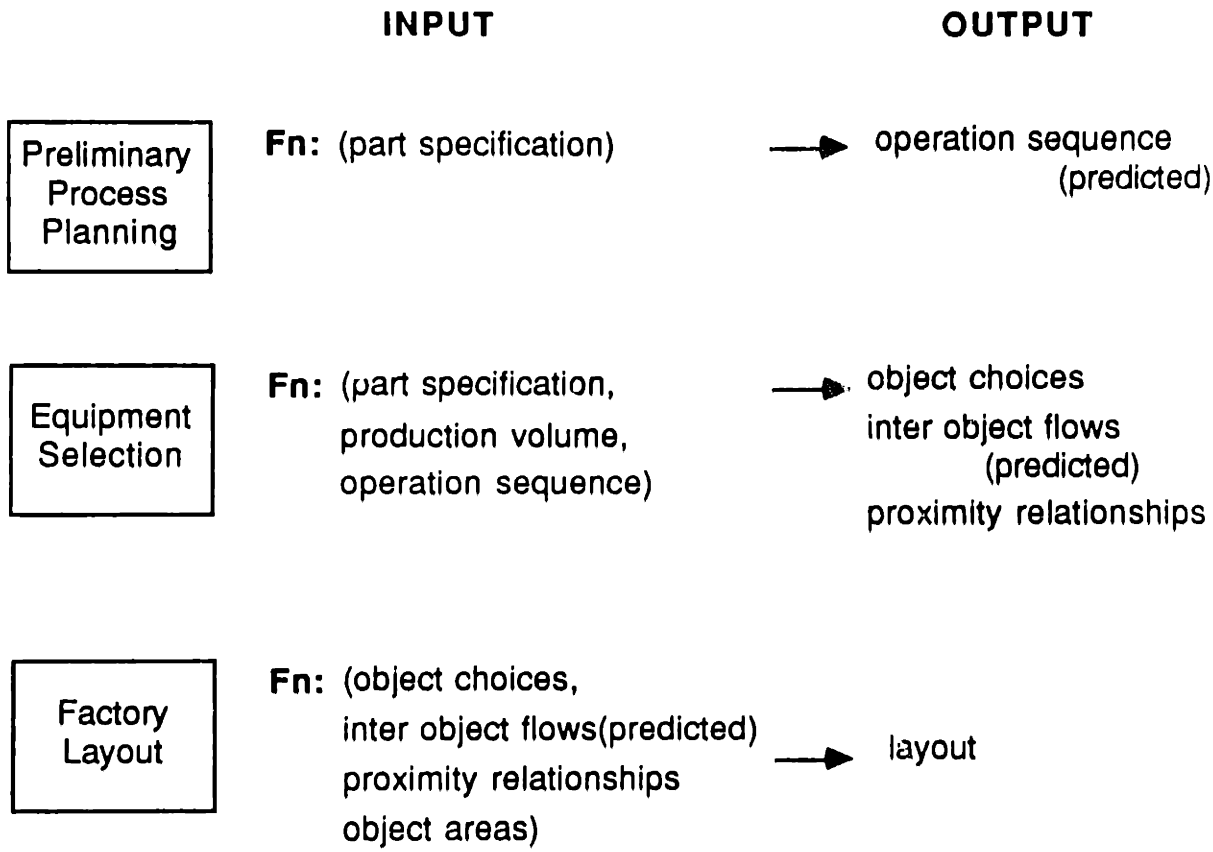


Figure 1.3

Factory Design: The Inputs and Outputs



Once a factory is designed and operating the layout dictates to a large extent the design and the resulting effectiveness of process plans and schedules, especially with regard to operation sequencing and queueing times. Figure 1.4 illustrates the role of the factory layout as a major contributor to decisions on product selection, process planning and scheduling. The layout function makes decisions at the *machine level*. That is, all decisions relate to the characteristics of machines and the flow relationships between machines. All other functions require knowledge of the product characteristics and are therefore decision makers at the *part level*. Figure 1.5 shows the inputs and outputs to the process planning and scheduling functions in addition to those for factory design. Process planning is similar to preliminary process planning, but the operations are chosen for a machine in the factory layout, rather than for a process. The fixturing and sequencing details can be finalized at this stage.

A computer representation of a factory is a requisite for intelligent automated process planning and scheduling systems. A computer representation is also necessary for simulation of flow. It allows the factory to be reconfigured so that the effect of changes may be analyzed. Reconfiguration of the layout may be virtual, comprising a rerouting of the material handling system, or real, that is, the physical placement of manufacturing equipment is altered.

1.2 Factory Design in Relation to Design in General

We consider factory design in relation to design in general, [Mostow, 1985]. Design is the process which transforms the functional requirements of a product, process or factory into specifications. It ranges from the extremely innovative to the extremely routine, [Brown and Chandrasekaran, 1985]. Routine design is the most pervasive. Both the knowledge sources and the problem solving strategies are known in advance. Design proceeds by selecting among previously known sets of well understood design alternatives. For example in designing a turbine disk for a new aircraft engine, the functional requirements are the speed and load bearing capability of the of the disk. The material choice is subject to the constraint of the operating

Figure 1.4

Schematic Illustrating the Effect of Factory Layout on Product Selection and the Manufacturing Function

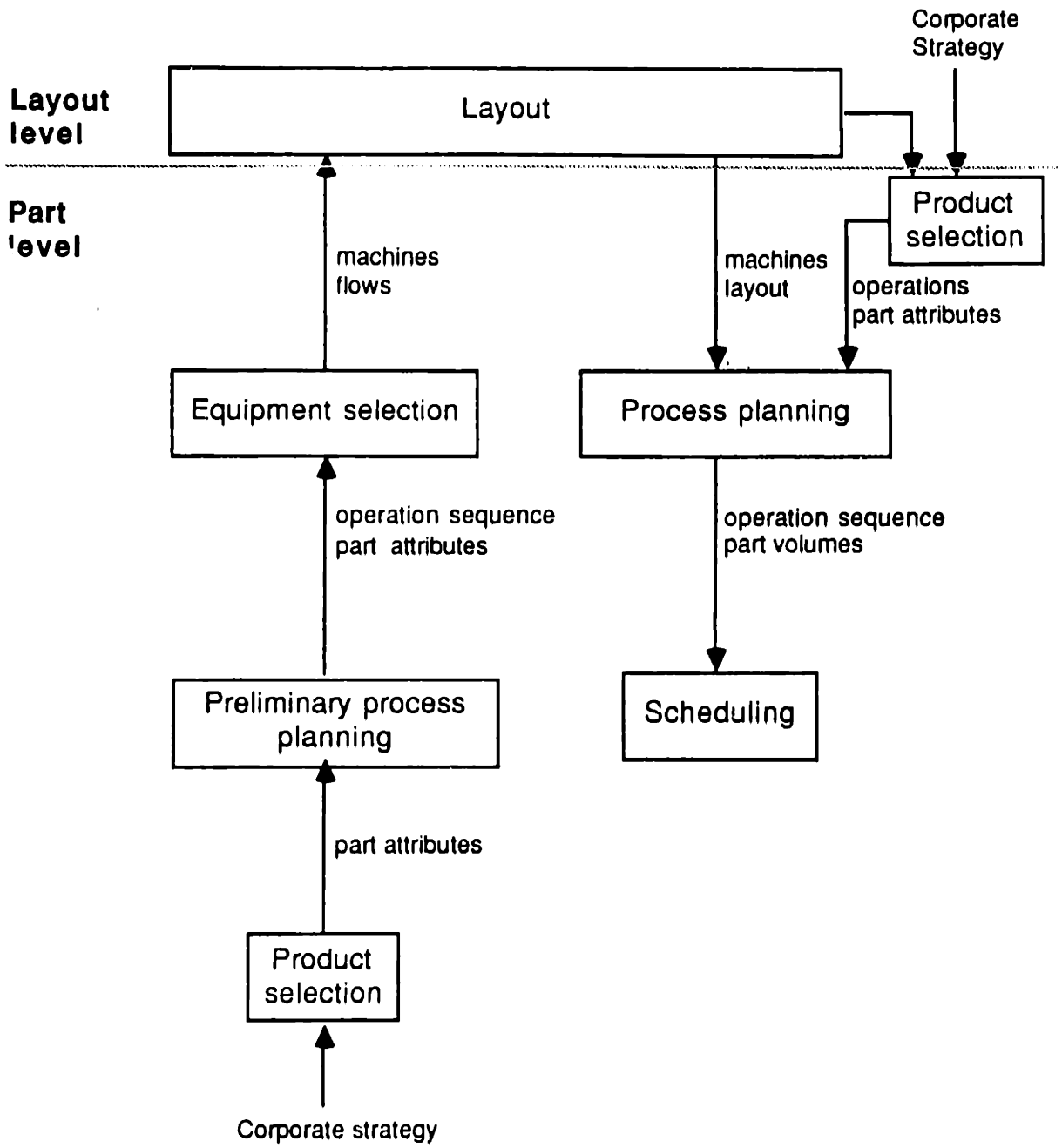
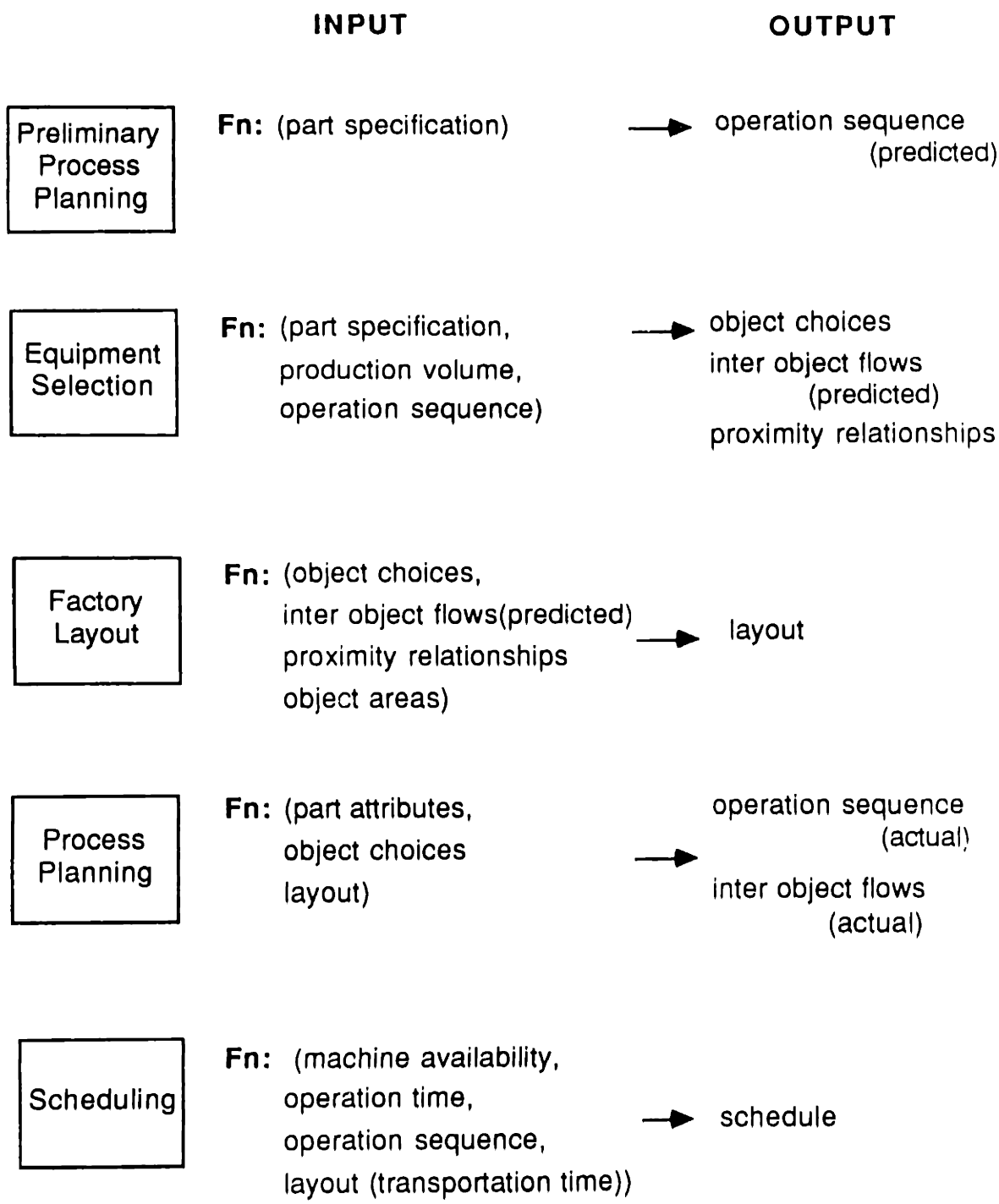


Figure 1.5

Factory Design, Process Planning and Scheduling: The Inputs and Outputs



temperature which the disk will reach and must therefore have acceptable creep properties. The inner and outer diameter and bolt holes, for attachment to the shaft, may be constrained by the rest of the engine design. The features to be designed, such as the cooling holes and the shape of the disk cross section are the design variables, but at least the features are known. There is usually a substantial amount of data from previous engines to advise the designer of the parameters affecting these features. If not, rough calculations and finite element simulations can be are used to find the best cross section shape. This is by necessity a simplification of the routine designers task and is not intended to detract from the experience, skill and insight which a designer provides. But this picture is drawn for comparison with the task of extremely innovative design where neither the knowledge sources nor the problem solving strategies are known in advance.

In computerizing conceptual design, [Ulrich and Seering,1987] hypothesized that most new designs are combinations of features from old designs. They developed a system to generate novel feature combinations for fasteners. Turbine disk design is limited, even in its consideration of new feature combinations, because of the constraints posed by the rest of the engine.

Factory design errs more toward routine design. The knowledge sources are existing machines, processes and handling systems and the probem solving strategies are those employed by factory designers for many years. However, the scope of possible machine combinations and configurations may be very large. We may view factory design as an intelligent search for a combination of machines and handling equipment and a physical configuration which meets the functional requirements of the factory.

1.3 Computer Techniques in Factory Design

Recognizing the importance of factory design on the success of the ensuing function of a firm, this thesis aims to address whether computer techniques can be used to improve factory design. Computers are good at solving complex problems as they can manipulate many variables at once, whereas

the human mind in its immediate working memory can store only a few. They allow a large number of possible designs to be generated and examined quickly. However the number of plausible designs may exceed by far the number with which a computer can cope in a reasonable amount of time. In this case, the computer must exhibit some intelligence by generating for consideration only plausible designs which are likely to be good designs.

Artificial Intelligence (AI) is concerned with designing intelligent computer software systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior, for example, understanding language, learning, reasoning and solving problems. The aim of AI is to develop techniques that will allow us to design and program machines which both emulate and extend our mental capabilities. Of foremost interest to the field of factory design is the computer representation of knowledge and reasoning techniques employed by factory designers in order to select processes and machines and to lay out factories. If these can be combined with the processing capabilities of computers we might be able to design better factories far more quickly. The Artificial Intelligence problem solving paradigm of Generate and Test is also of interest. It allows expression of the unique structure featured by a specific symbolic representation of a given problem. Many existing problem solving techniques have been adapted for application to the layout problem, but they have been limited in their expressiveness, considering for example only equisized square departments, or centroidal department flows.

The aim of computer models is not necessarily to generate the best solution according to some set of rigid criteria. Generation of the best solution may even be computationally infeasible. Rather, the success of computer aided design and simulation techniques has been achieved by allowing a large number of insightful configurations to be generated quickly and inexpensively. They have relieved the monotony of routine drawing and calculation, allowing more room for thought and creativity.

1.4 Characterizing the Layout Problem For Computer Representation

The objectives of factory design are manifold. Some sample objectives are listed below.

Maximize safety
Use manpower efficiently
Maximize expansion flexibility
Attain orderly material flow
Minimize material flow
Maximize machine utilization
Minimize investment in capital equipment
Minimize varieties of material handling equipment

In a computer model of the design process we must consider how to represent both qualitative and quantitative objectives.

The layout problem belongs may be classified as an ill structured problem [Newell,1969]. Ill structured problems have the following characteristics;

(i) they cannot be described exclusively in numerical values,
(ii) the goal to be attained cannot be described in terms of a quantitative objective function or functions,
(iii) algorithms that permit the exact solutions to be found and stated in numerical terms do not exist.

Programming languages which allow symbolic as well as numerical manipulation facilitate the expression of ill structured problems such as the layout problem. *Logic programming* languages are one category of symbolic programming languages. They allow *logic* statements which describe a problem to be written as a program.

1.5 A Formal Model for Factory Design - A Logic Based Approach

The tenet of this thesis is that any approach to computerize an ill structured activity such as design must have a rigorous mathematical foundation.

Logic is a branch of mathematics which studies the relationship of implication between assumptions and conclusions. It was originally devised as a way of representing the form of arguments, so that it would be possible to check in a formal way whether or not they were valid. The reasoning techniques of factory designers in selecting machines and laying out factories, and the knowledge about department relationships , machine characteristics and the factory constraints may be expressed precisely in symbolic logic. Logic allows expression of the quantitative and qualitative objectives and attributes. We may also express the Generate and Test paradigm, for generating geometric configurations, in logic.

A formal model has been developed in logic in order to unify the components of the factory design problem. It enables us to formalize the relationships between the phases of preliminary process planning, machine selection and layout. It enables us to analyze, to prove and to recognize the limits of computer techniques applied to the factory design problem. It allows us to understand better the structure of the ill defined problem and to appreciate the extent of the intractibility of the problem in a specific domain.

By expressing knowledge formally we may determine whether there is any formal difference between industry specific and industry independent knowledge. We can determine to what extent knowledge is just a parameter of the problem and to what extent it is a constraint which reduces the intractibilty of the problem in a given domain. This helps us to understand whether knowledge base systems can be successfully applied to the factory design domain.

1.6 Aims of this Thesis

The principal aims of this thesis are

(1) to demonstrate a systematic automated approach to factory design, based upon formal logic, representing the design process in a formal mathematical model.
(2) to analyze the extent to which factory design is generalizable across industries and firms,
(3) to consider the implications for the application of knowledge based systems,
(4) to consider the limitations that complexity imposes upon automated factory design,
(5) to demonstrate a software implementation of the formal model, the "Factory Design Advisor", in a logic programming language, Prolog.

The Factory Design Advisor (FDA)

(a) structures the techniques and the knowledge of experienced factory layout engineers by encoding declarative and procedural rules in a logic programming language.

(b) makes the objectives of each design explicit. Each objective is linked to layout rules. The order of the rule firing depends upon the ranking of the objectives by the user.

(c) allows both quantitative and qualitive objectives to be expressed.

(d) decouples domain dependent and independent knowledge, facilitating expansion of the knowledge base to new domains.

(e) gives insight into the final design through an explanation module. This traces the path taken through the rule base to arrive at the final design.

CHAPTER 2
PREVIOUS WORK

Overview

Previous research and writings in factory design fall into three main categories:

(i) systematic manual procedures,
(ii) mathematical formulations and solution procedures and
(iii) expert system applications.

The systematic manual procedures embrace the entire design problem and have gained considerable popularity. The mathematical formulations and solution procedures were developed to address the layout problem. They have been developed into computerized layout packages with limited popularity. Their main drawbacks have been their limited expressiveness. For example, they consider only centroidal flows or they do not allow department shape to be expressed. They are often Operations Research algorithms formulated as a layout problem. Although their results are insightful their limitations have curtailed their use. We explore these algorithms in some detail because they are relevant to discussions in chapter 6 on the complexity of factory layout. The expert system applications have used knowledge based techniques to automate the equipment selection procedure, but as yet none have demonstrated application of industry or firm specific layout knowledge to perform intelligent layout. We explore why this might be in chapter 5.

2.1 Manual Approaches

There is a wealth of literature available on factory design and systematic manual approaches [Apple,1977],[Muther,1961]. There is also a wealth of industry specific experience available in local industry.

2.1.1 Systematic Layout Planning

An organized approach to layout planning, Systematic Layout Planning, (SLP) was developed by [Muther, 1961]. It has received considerable publicity due to the success derived from its application to a large variety of layout problems. The SLP procedure involves techniques for gathering and reducing machine selection and layout data to a manageable form. From-to charts and Relationship (REL) charts, depicted in figure 2.1, which were originally developed for SLP, have become very popular in computerized layout systems because the data may be neatly expressed in a matrix format. The from-to chart contains measures of material flow between departments. The REL chart was developed to facilitate a consideration of qualitative factors. It replaces the numbers in a from-to chart by desired closeness ratings between departments. The REL chart summarizes qualitative aspects of relationships between departments. We attempt to measure these qualitative aspects by making pairwise evaluations of the importance that two departments be located close to each other and assigning closeness ratings A,E,I,O,U, and X, between pairs of departments. These ratings are ranked

A>E>I>O>U>X

in importance that two departments be located close together. In SLP, once the department relationships have been summarized in charts and diagrams, the departments are laid out as rectangles and shifted around in a visual procedure to obtain candidate block plans, which are then evaluated.

2.1.2 Computerizing Systematic Layout Planning

The from-to and REL charts serve as useful input to computerized layout. We can also draw from machine selection and layout rules in the literature of experienced designers. But since a computer does not have the visual sense of the human we would have to state explicit layout rules to the computer to obtain candidate block plans. The simplicity and generality of these rules may determine the success of computerizing layout.

The factory designer approaches the layout problem at a high level of

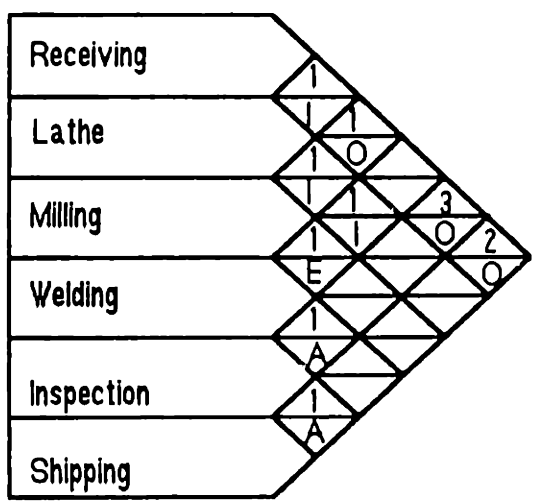
Figure 2.1

Charting Methods in Systematic Layout Planning

(a) From-to Chart

To / From	Rec	Lathe	Mill	Weld	Insp	Ship
Rec		25	10			
Lathe			20	25		
Mill		10		30		
Weld		5	5		55	
Insp						55
Ship						

(b) Activity Relationship (REL) Chart



Code	Reason
1	Flow of Materials
2	Ease of Supervision
3	Common personnel
4	Needs quiet

Rating	Definition
A	Absolutely Necessary
E	Especially Important
I	Important
O	Desirable
U	Unimportant
X	Undesirable

abstraction, placing departments and then filling in the details of workstations within the departments. This is because the human mind can only cope with a limited amount of information at a time. The computer may also need to take this approach in order to limit runtime and storage space.

2.2 Computerized Layout - Mathematical Formulations and Solution Procedures.

2.2.1 Assumptions about Measurement Scales

Some of the formulations and procedures which we will discuss only consider layout optimization with respect to minimizing flow cost. Others have also introduced the closeness ratings developed by Muther for determining relative department placings in the layout. The closeness ratings A,E,I,O,U,X exhibit properties of an *ordinal* scale. Ordinal data have the properties of identity and order but they do not allow mathematical operations such as addition. We could assign numbers 6,5,4,3,2,1 or 64,16,4,1,0,-1024 to the closeness ratings and the order would be preserved in both cases. But note that by assigning ratings:

A=6, E=5, I=4, O=3, U=2, X=1

it is not valid to state that having one A and one U is better than having one I and one O.

Calculation of sums and averages on ordinal data have been done to compare alternative layout designs, but such comparisons may have little meaning. However, if the user chooses the assignments carefully he or she may consider addition of closeness ratings to be acceptable. Computerized layout algorithms often make this assumption because it is the only means by which they can compare layouts .

2.2.2 Mathematical Formulations

2.2.2.1 The Quadratic Assignment Problem Formulation

The department location problem has been formulated as a Quadratic Assignment Problem (QAP), [Hillier and Connors, 1966]. The QAP is illustrated in figure 2.2. The problem is to assign departments to locations so that the flow cost of the layout is minimized. The departments are all represented as equisized squares. The locations are all elements in a uniform grid of similar equisized squares.

Let,

- i,k refer to departments
- j,l refer to locations
- n = the number of departments,
- d_{jl}= the cost per movement or interaction over the distance from location j to location l,
- f_{ik} = the number of moves per time period in the workflow from department i to department k.

a_{ijkl} = $f_{ik}d_{jl}$ if $i \neq k$ or $j \neq l$, (flow cost between depts)

0 (or c_{ij}) if i=k and j=l. (we could define a cost c_{ij} associated with each location)

x_{ij} = 1 if department i is assigned to location j,

0 otherwise.

The problem is to

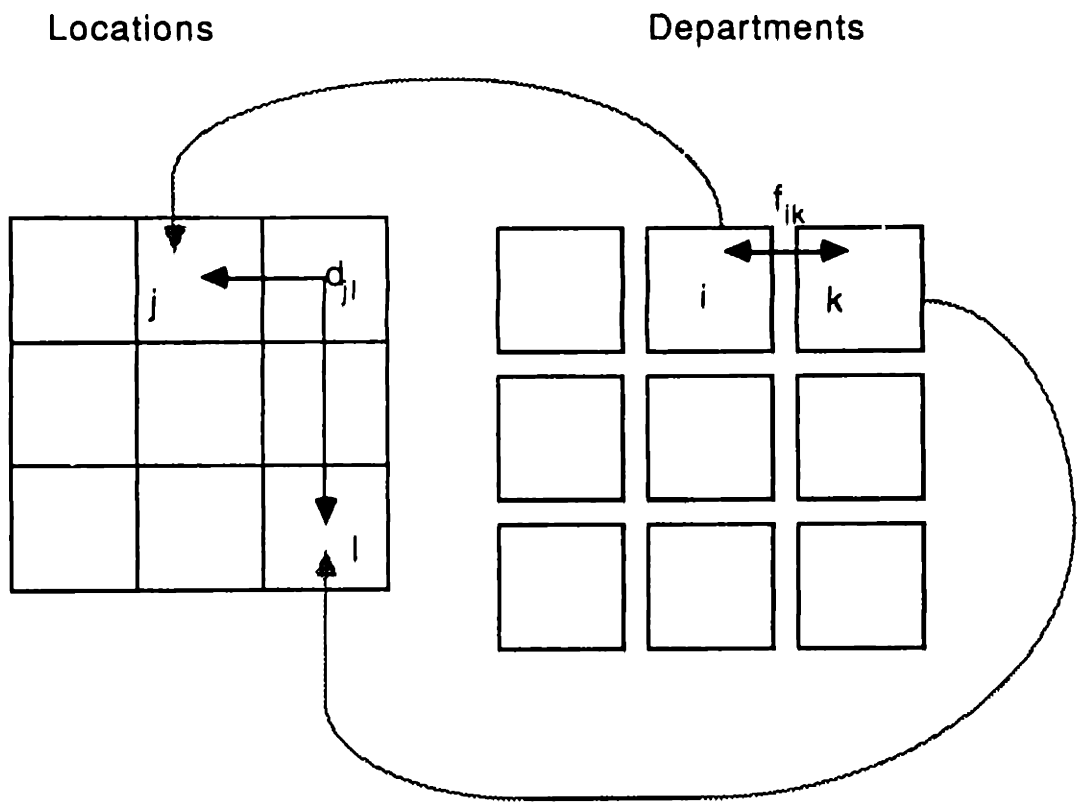
Minimize $$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} a_{ijkl}x_{ij}x_{kl}$$

subject to $$\sum_{i=1}^{n} x_{ij} \leq 1, \quad j=1,2,\ldots,n,$$ (at most one department may be assigned to location j)

Figure 2.2

Illustration of the Quadratic Assignment Problem

Assignment of objects to locations



d_{jl} is the transportation cost from location j to location l

f_{ik} is the material flow between object i and object k

$$\sum_{j=1}^{n} x_{ij}=1, \qquad i=1,2,\ldots,n,$$ (department i must be assigned to exactly one location)

$$x_{ij}=0 \text{ or } 1, \qquad i=1,2,\ldots,n, \quad j=1,2,\ldots,n.$$ †

If the areas required for different departments are significantly different, then the departments are partitioned into sub-departments, all with equal area. The planar site is then divided in the form of a grid of elemental areas, each of the size of one sub-department. The problem is to locate the sub-departments in the grid such that all the sub-departments of a departments are adjacent and the material handling cost is minimized. If sub-departments i and k belong to the same department, the f_{ik} values are set relatively high to ensure their adjacency. If assignment of sub-department i to location j is infeasible, the c_{ij} values are set relatively high.

---

†Footnote

The Quadratic Assignment Problem is so called because the criterion is quadratic in the variables x_{ij}. It considers flow between departments. The Linear Assignment Problem is to

$$\text{Minimize} \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \quad \text{s.t.} \quad \sum_{i=1}^{n} x_{ij}\le 1, \ j=1,..,n, \quad \sum_{j=1}^{n} x_{ij}=1, \ i=1,..,n,$$

$$x_{ij}= 0 \text{ or } 1 \qquad i=1,..,n, \ j=1,..,n$$

where c_{ij} is the cost of allocating department i to location j.

2.2.2.2 Limitations of the Quadratic Assignment Problem Formulation

[Rosenblatt, 1979] developed a multi objective formulation which accounted for both minimizing flow cost and maximizing closeness rating; the QAP is not restricted to flow minimization. But the adaptation to express non uniform department size is unsatisfactory for a number of reasons. Firstly, although it ensures members of a department are clustered together, it does not necessarily maintain a suitable shape. Secondly, the sub-departments have no physical equivalent. One has to determine how to assign the flow from the sub-departments of one department to those of anther department. Thirdly, by assigning large flows between sub-departments belonging to the same department we lower the sensitivity of any practical algorithm to the interdepartmental flows which we are trying to minimize. Lastly, the time required to solve the QAP grows exponentially with the number of locations in the grid. Therefore the more we refine the grid by allowing smaller sub-departments, the more intractable becomes the problem.

2.2.2.3 The Graph Theoretic Formulation

A graph theoretic formulation of the layout problem has been defined by [Foulds, 1983]. A graph consists of a set of points and lines, also called nodes and edges. Figure 2.3 shows the graph representation of a block plan. The nodes are assigned to points where three departments meet. The edges represent the department boundaries. In the dual representation nodes are assigned to each department, and the edges represent flows between departments. A planar graph is one in which none of the lines cross. If a graph is planar, then its dual is also planar. If we can find a planar graph in the dual representation of nodes as departments and edges as flows between departments, we can translate this to a block representation where departments linked by flows are adjacent.

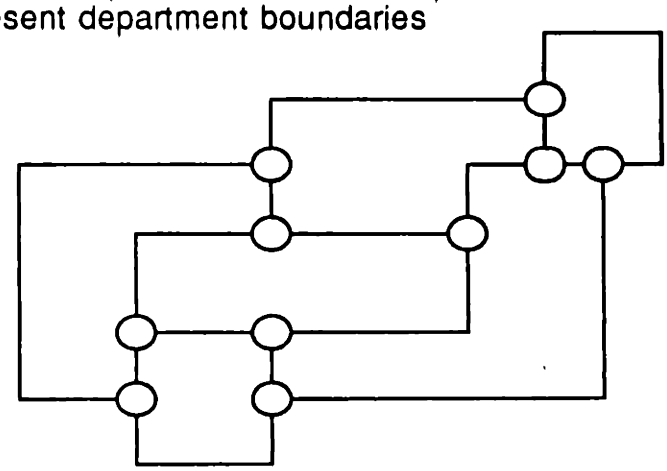
A weighted graph is one in which weights are assigned to the lines of the graph to represent, say, flowcost. The best set of adjacencies which can be obtained will be represented by a weighted maximal planar sub graph, one

Figure 2.3

Graph Representation of the Layout Problem

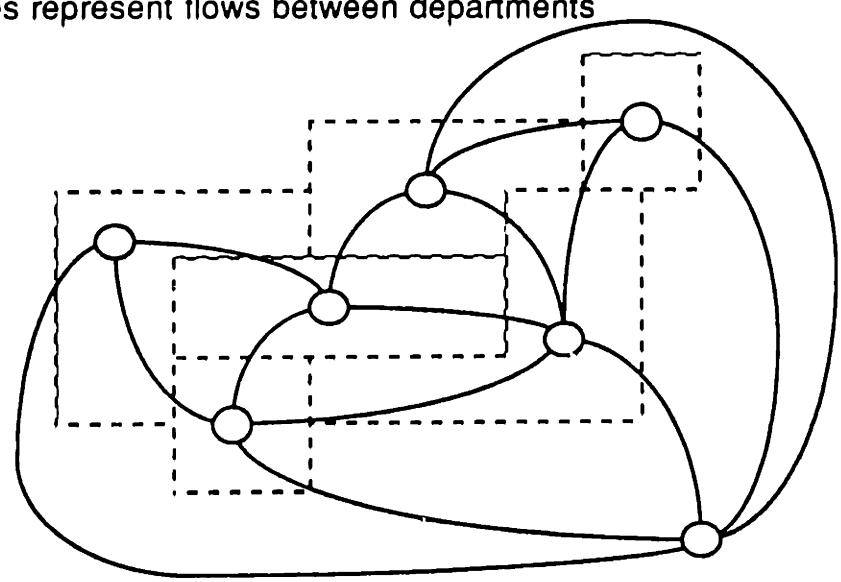
GRAPH REPRESENTATION OF A BLOCK PLAN

Nodes assigned to points where three departments meet
Edges represent department boundaries



DUAL REPRESENTATION

Nodes assigned to each department
Edges represent flows between departments



in which no more adjacencies can be achieved, with the highest weighting. Since all the closeness ratings are non-negative, at least one optimal solution will be a maximal planar graph. This is so because any optimal solution which is not maximal planar can be made maximal planar by addition of a line without decreasing its total line weight.

There have been several solution procedures proposed for graph theoretic formulations. These entail edge deletion or edge addition procedures combined with planarity checks. Graph Theoretic formulations have been shown to be NP-complete by [Giffin, 1984]. That is, the solution grows exponentially with increasing N. NP-completeness will be defined in Chapter 3 and a more extensive discussion will ensue in Chapter 6.

2.2.2.4 Advantages and Disadvantages of the Graph Theoretic Formulation.

In the graph theoretic formulation :

(i) The linear costs , defined as c_{ij} in the QAP formulation are assumed to be zero. Only interdepartmental flow costs or closeness ratings represented by weights on the lines are considered. This suggests that the graph theoretic approach is more suitable for the design of a new layout rather than the modification of an existing one. The QAP approach could be used for either.

(ii) There are no predetermined locations. The formulation assumes that a blank area must be filled with departments. The graph theoretic approach does not require a grid or the sub-division of departments. The freer input format lends itself to a more elegant solution.

(iii) The goal is to maximize the sum of the closeness ratings between adjacent departments. No credit is given for non adjacent pairs even if they are close. This assumption may not be valid in which case the QAP solution is superior. However more refined graph theoretic techniques are being developed which will allow this assumption to be relaxed.

(iv) Once the maximal planar subgraph is found for the graph which represents closeness ratings as edges and departments as nodes, it must be translated into a block representation. Each department is assigned an area which must then be represented as a physical shape. In order for all of the adjacencies to be maintained this may require that shapes be long and thin. An example from [Giffin, Foulds and Cameron, 1986] is shown in figure 2.4. All the departments have an interaction with the machining department. Specific shapes (or maximum and minimum aspect ratios) could be specified for each department. This would constrain the number of maximal planar subgraphs which could actually be translated into a block representation. Considering bounds on the aspect ratios would substantially increase the complexity of translating a graph to a block representation.nThese modifications to graph theoretic solutions are more complex because graph theory is primarily concerned with topology, or connectivity. It does not truly lend itself to specific shape representation. Unfortunately if the shape of a department cannot be expressed, then the solutions may violate the intuitive assumptions on which the adjacency requirement is based. Adjacency of related departments is desirable because it reduces flow. But if their adjacency requires that they be long and thin, then we simply increase the internal department flow in order to decrease the external department flow.

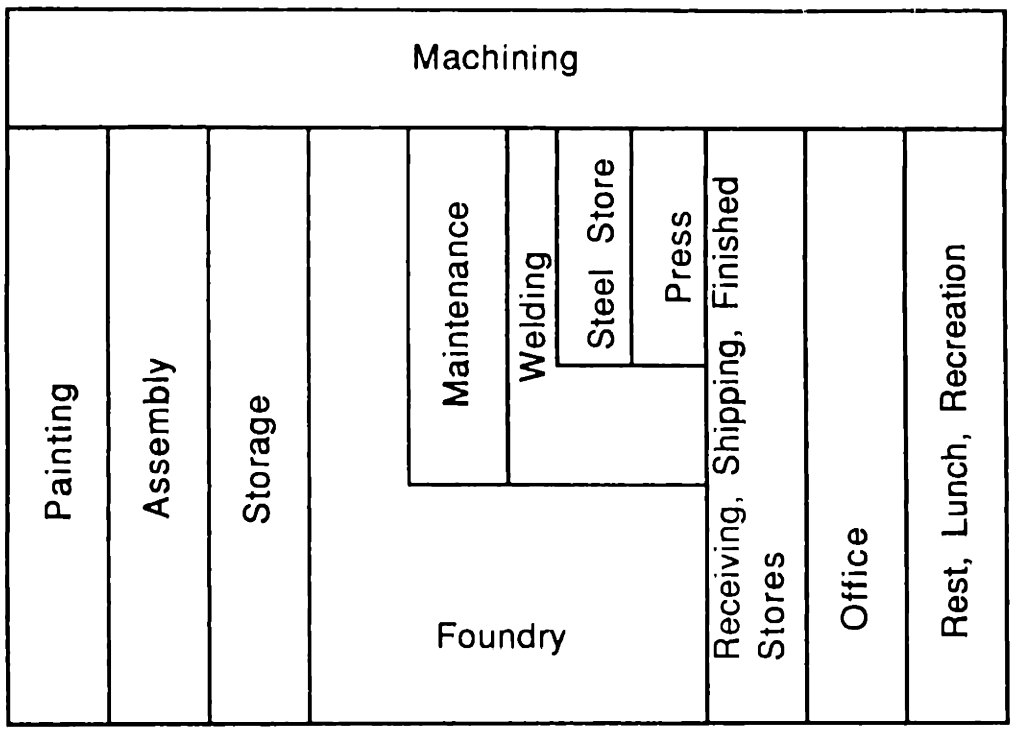
2.2.3 Solution Procedures

It has been shown by [Sahni and Gonzalez,1979] that the QAP is NP-complete. That is, any algorithm is likely to require a number of elementary computational steps which is an exponential function of the size of the problem. [Giffin, 84] has shown that the graph theoretic formulation is NP-complete. In view of these results the existence of an effective Layout Planning algorithm is unlikely.

These results have prompted attempts to devise heuristic solution procedures which run in reasonable computational time. Heuristic procedures are commonly evaluated by comparing them with other heuristic procedures, the two main evaluation criteria being total cost of final solution provided and

Figure 2.4

An Example Layout Produced by a Graph Theoretic Approach



computational effort required. Some heuristic procedures for solving the QAP and other similar formulations follow.

2.2.3.1 Iterative improvement algorithms

An iterative improvement algorithm requires an initial configuration, a cost function and a generation mechanism to generate an improved configuration from the current configuration. The generation mechanism defines a neighborhood for each configuration, i, consisting of all configurations that can be reached from i in one transition. Iterative improvement is also known as neighborhood or local search.

For a given configuration of departments, represented by an assignment vector

$$\mathbf{a} = (a(1), a(2), \ldots\ldots, a(n)) \ ,$$

where a(i) denotes the number of the site to which facility i is assigned, the total cost of the solution is given by

$$TC(\mathbf{a}) = \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} d(a(i), a(j))$$

where d(a(i),a(j)) are the distances between sites, taken from a site-distance matrix, D, and w_{ij} is a constant of proportionality converting the distance between departments i and j, into a cost, taken from the flow cost or weighting matrix W.

Starting off from a given configuration, a sequence of iterations is generated, each iteration consisting of a possible transition from the current configuration to a configuration selected from the neighborhood of the current configuration. In factory layout the neighborhood of the current configuration contains all configurations which can be reached by swapping the locations

of a pair of departments. The change in total cost given by interchanging two departments, u and v, is

$$\Delta TC_{uv}(\mathbf{a}) = \sum_{i=1}^{n} (w_{iu} - w_{iv})[d(a(i),a(u)) - d(a(i),a(v))] - 2w_{uv}d(a(u),a(v)).$$

The neighboring configuration with the lowest cost is found, and if this cost is lower than that of the current configuration it is replaced by this neighbor. The algorithm terminates when a configuration is obtained whose cost is no worse than any of its neighbors.

The disadvantage of iterative improvement algorithms is that they terminate in a local minimum, which depends upon the initial configuration. However the local minimum is reached through a fairly large sample of possible department exchanges, and they are considered to give fairly good results. Hil climbing procedures, discussed in section 2.2.3.4, tend to have a smaller sampling space and generate "more local" solutions. The Pairwise Steepest Descent Procedure [Francis and White, 1974] employed in CRAFT, one of the most widely known and applied computerized layout programs, is an iterative imrovement algorithm.

Simulated annealing is a technique which adds a randomizing factor to iterative improvements, enabling a solution to jump out of a solution path if there is a reasonable probability that it is not converging toward the global minima. It achieves optimal solutions with a probability which tends to 1 as the number of steps of the computation procedure is increased. This technique is computationally intensive, but has achieved wide success in VLSI design and other large scale optimization problems where computation time is not an issue. It is described further in [van Laarhoven and Aarts, 1987].

2.2.3.2 The Branch and Bound Method.

The Branch and Bound method developed by [Lawler and Wood, 1966] is an

admissible procedure, that is, it always finds the optimal solution. The Gilmore-Lawler Procedure [Lawler, 1963], is a branch and bound procedure for quadratic assignment formulatons of the factory layout problem. It becomes computationally infeasible when the number of departments is greater than 15.

2.2.3.2.1 Terminology

The factory layout formulation of the QAP, which is to assign departments to locations is depicted in figure 2.2. The search procedure for finding a solution to the QAP may be represented as a *tree* shown in figure 2.5(b). The tree has *nodes* and *branches*. Nodes represent decision points where branching occurs, and branches link nodes to their parent nodes and their infant nodes. In the factory layout problem, a node represents a decision about where to place a department. The branches represent the different location options.

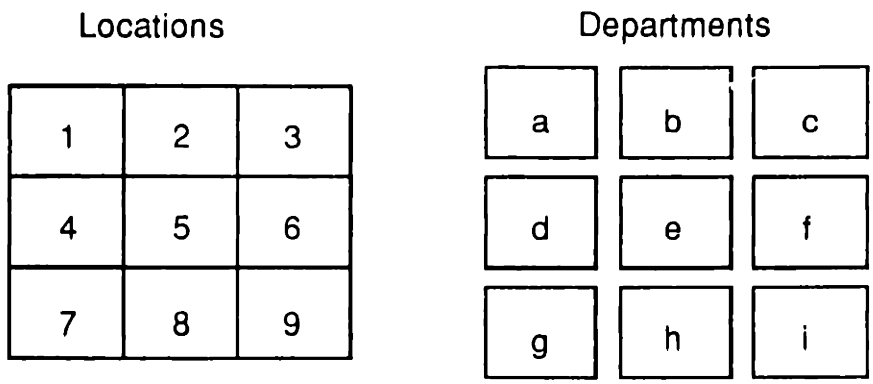
In the 9 location QAP tree of figure 2.5(b), there are 9 locations at which the first department, a, can be placed. Once this department has been placed, there are 8 remaining locations at which we might place the second department. The *branching factor* represents the number of branches which emanate from a node. In the QAP tree, as the depth of the tree increases the branching factor decreases, because the number of available locations decreases. A terminal node which has no offspring is referred to as a *leaf node*. Each leaf node of the QAP tree represents a unique configuration. *Expanding* a node refers to evaluating the offspring nodes of a node. When we evaluate a node we determine some measure of the quality of the solution repersented by the leaf nodes which eventually descend from this node. A node which has been evaluated but which has not yet been expanded is called an *open* node. A node which has been expanded is called a *closed* node.

The Branch and Bound procedure is a procedure for determining which nodes should be expanded in search of the optimal configuration, and which nodes can be *pruned* or deleted from the search space because it can be

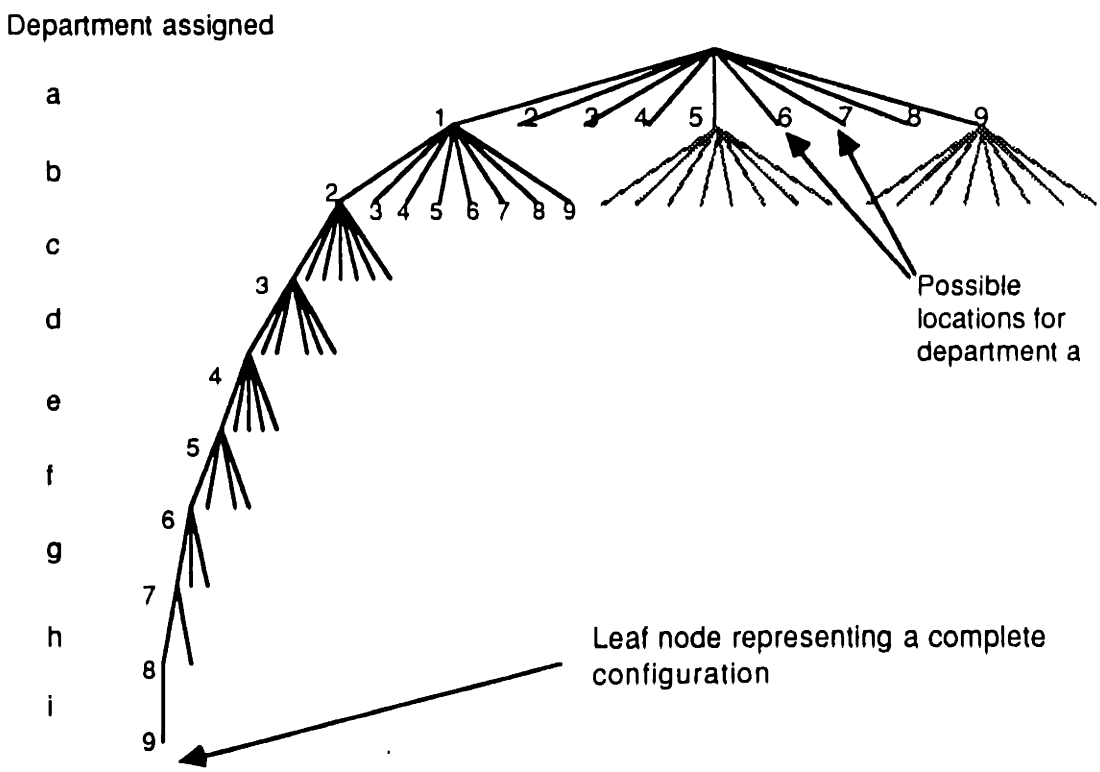
Figure 2.5

The QAP: The Problem and the Search Space for the 9 Department Case.

(a) The QAP for the 9 department case: Assign departments a,b,...i to locations 1,2.....9



(b) Search tree for the QAP showing the path of 1 possible configuration



determined that they definitely do not lead to the optimal solution. It is so called because a node is evaluated for expansion by determining the upper and lower bounds on the solutions emanating from this node.

2.2.3.2.2 Calculating the bounds

Let **W** be a matrix of flow costs between departments and **D** be a matrix of distances between locations in the grid. Let $\mathbf{w}^*$ be a vector of the entries of the **W** matrix arranged in decreasing order and $\mathbf{d}^*$ be a vector of the entries of the **D** matrix above the diagonal arranged in increasing order.

The lower bound on the minimum value of the total cost, TC_L is given by the inner product of $\mathbf{w}^*$ with $\mathbf{d}^*$:

$$TC_L = \sum_{i=1}^{r} w_i^* d_i^*$$

where r is the length of the vector $\mathbf{w}^*$. In this lower bound, we pair the largest flows with the smallest possible distances, and the smaller flows with the larger distances. This lower bound is not necessarily achievable in practice because we cannot decouple the flows and distances. They are linked physically by the departments. The lower bound is an underestimate of the optimal solution. It may equal the optimal solution, but it cannot exceed it.

A lower bound on a *partial assignment*, that is, one in which some but not all of the departments have been located, is given by the sum of the actual cost of the partial solution, $TC_{located}$, and the lower bound on the remaining solution:

$$TC_{L(\text{partial assignemnt})} = TC_{located} + \sum_{i=1}^{r} w_{Ri}^* d_{Ri}^*$$

where w_R^* is a vector of the flowcosts of departments yet to be assigned, arranged in decreasing order, and d_R^* is a vector of the distances between locations yet to be filled, arranged in increasing order.

For reasons which will become clear as the branch and bound procedure is explained, it is desirable to estimate the lower bound as accurately as possible so that it is as close to the optimal solution as possible. Similarly we would like as low an upper bound as possible.

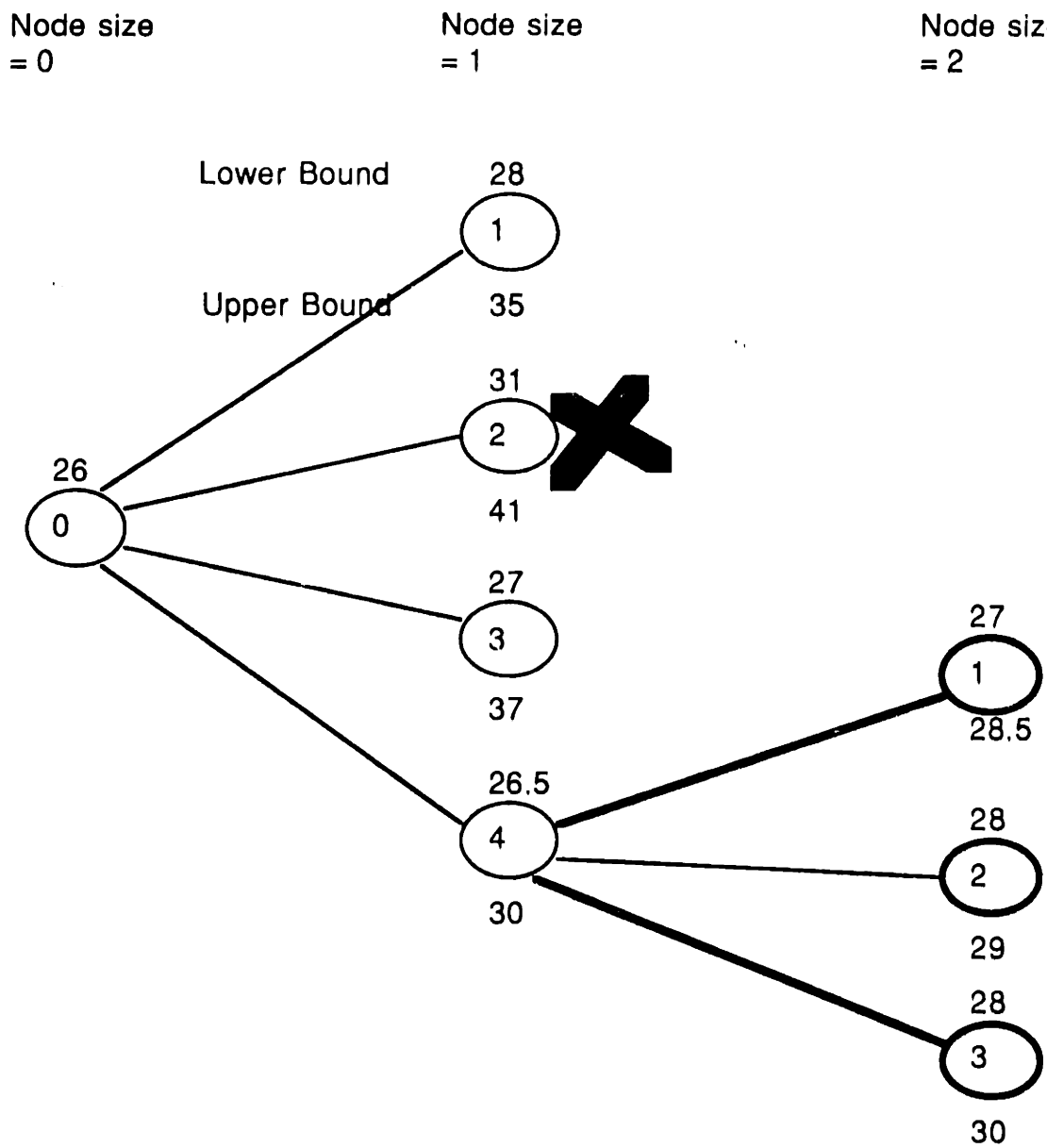
The upper bound represents the lowest estimate of what the actual solution will cost. It is the lowest cost of a completed solution which is determined by some heuristic procedure such as a hill climbing procedure (section 2.2.3.4).

2.2.3.2.3 A Branch and Bound Procedure

Figure 2.6 illustrates a Branch and Bound procedure for a four department case. The *depth* of the tree, or *node size*, represents the number of departments which have been assigned. Initially the node size is zero and no departments have been assigned. The lower and upper bounds are therefore the lowest lower bound L_L and the lowest upper bound U_L of all the open nodes. The node of size zero is expanded to produce the first layer of nodes. The zeroth level node is now closed, and the first level nodes are open. The first layer of nodes represents all the possible placements of the first department. The upper and lower bounds of these nodes are calculated. The lowest lower bound of the new set of open nodes is found and L_L is updated. The lowest upper bound U_L is updated only if the upper bound of one of the current set of open nodes is lower than the existing U_L. If any one of the open nodes has a lower bound which is larger than the upper bound of another open node, then this node is pruned; since an upper bound represents an actual solution and a lower bound represents an underestimate, if the underestimate for configuration A is larger than a guaranteed solution for configuration B, there is no point in exploring configurations based upon configuration A any further. After pruning we expand the open node

Figure 2.6

A Tree Representation of Branch and Bound For the Four Department Case



remaining which has the lowest lower bound, L_L. This node is now closed and its offspring nodes are evaluated and join the remaining open nodes. We repeat the procedure, pruning the open nodes, updating U_L and L_L and expanding the cheapest remaining open node, until the terminating condition, when the lowest upper bound U_L and the lowest lower bound L_L of the open nodes are equal. Then the solution represented by U_L is the optimal solution.

The Branch and Bound procedure is summarized below:

(i) Calculate the upper and lower bounds of each partial solution of the open nodes.
(ii) Update the value of U_L if a lower one is found.
(iii) Update the value of L_L to the lowest one among the current set of open nodes. (As the solution proceeds L_L will increase).
(iv) If $U_L = L_L$, then stop. The assignment with total cost U_L is the lowest cost solution possible.
(v) Prune the set of open nodes.
(vi) Expand the open node with the lowest lower bound .
(vii) Repeat.

2.2.3.3 Evaluation of the Branch and Bound Procedure.

This algorithm develops a sequence of non decreasing lower bounds and a sequence of non increasing upper bounds on the total cost. When the lowest upper bound is equal to the lowest lower bound the algorithm stops. The means of obtaining lower bounds is well specified, but the means of obtaining upper bounds is not. This is an inherent feature of the problem; if heuristic procedures could produce the lowest upper bound from the layout where no departments have been assigned there would be no reason to refine the search method through the branch and bound procedure. It is important to use a good heuristic procedure, that is, one which which produces an upper bound not much larger than the final TC as soon as possible. This reduces, by virtue of the pruning process, the number of nodes which have to be searched. This procedure becomes computationally unwieldy very quickly as

n increases. However it is a very useful procedure for providing bounds on the total cost.

2.2.3.4 Hill Climbing

Hill climbing is a strategy based on local optimizations, and is the simplest and most popular search strategy among human problem solvers. In terms of a tree search model, this strategy amounts to repeatedly expanding a node, inspecting its newly generated successors and choosing and expanding the best among these successors, while retaining no reference to the parent node. The strategy is irrevocable, because it does not permit us to shift attention back to previously suspended alternatives. It is a useful strategy when we posess a highly informative guiding function.

2.2.4 Computer Aided Layout

Computerized layout algorithms can be classified according to the way in which the layout is generated. *Construction* algorithms build up a solution by successive selection and placement of departments. *Improvement* algorithms modify an initial layout by interchanging department locations in order to improve the design. There are many computer programs available for generating layouts. Five of the most popular: CRAFT, COFAD, PLANET, CORELAP, and ALDEP are fully documented by [Tompkins and Moore,1978]. Two algorithms representative of the construction and improvement classes, CORELAP and CRAFT are described below.

2.2.4.1 CORELAP

CORELAP, a construction algorithm, was originally presented by [Lee and Moore, 1967]. It has subsequently been improved and made interactive. It uses a hill climbing strategy.

The input requirements are:

1. An REL chart

2. Number of departments
3. Area of each department
4. Area of unit square used to build departments
4. Weights for the REL chart closeness ratings

Optional inputs are:

5. Building length to width ratio
6. Department preassignment

A total closeness rating TCR for a department i is defined as

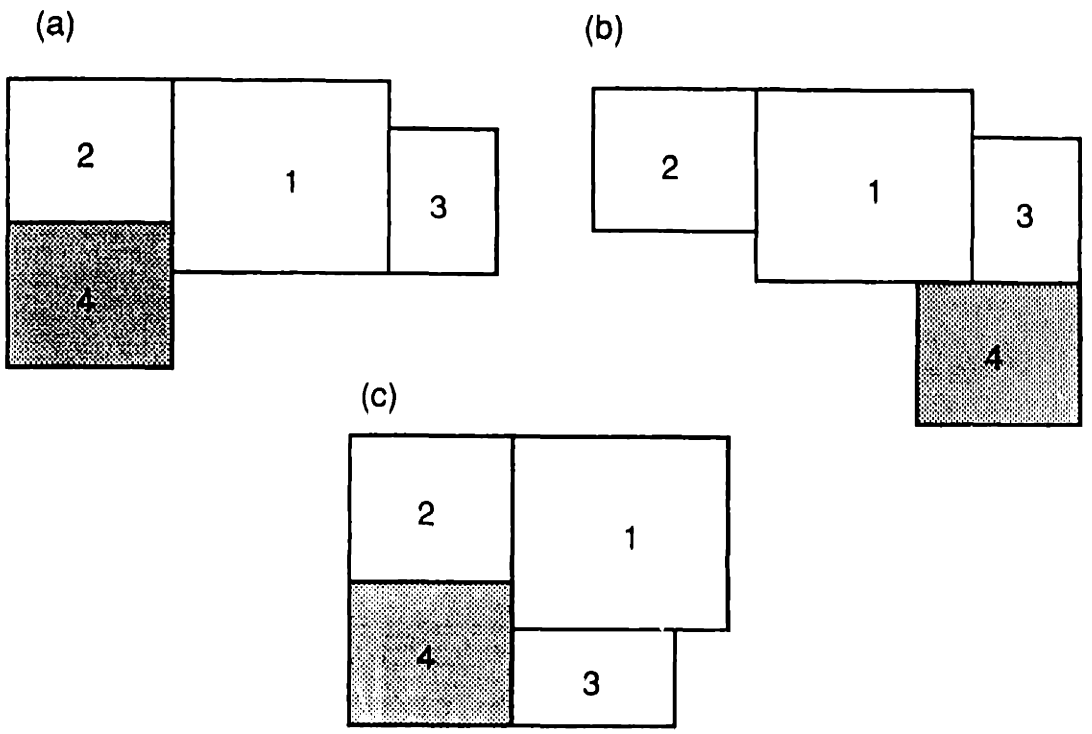
$$TCR_i = \sum V(r_{ij})$$

Where r_{ij} is the closeness rating between departments i and j, taken from the REL chart, and $V(r_{ij})$ is the numerical value assigned to the closeness rating. CORELAP assumes that the closeness ratings have the addition properties of a Ratio scale. The department with the highest TCR is placed first. The next department to be placed is the one with the highest closeness rating with the first department. Next the program searches for a department with an A rating with the first department placed. If it cannot find one it searches for an A rating with the second department placed. If no A ratings are found it then looks for an E rating, then an I rating, then an O rating. If there are no departments with at least an O rating with a placed department, the department with the next highest TCR is selected for placement.

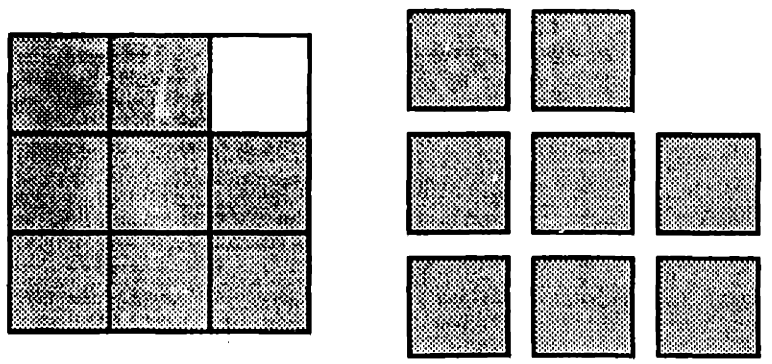
Once CORELAP has selected a department it evaluates a number of possible locations in which to place it using a *placing rating* and a *boundary length.* The placing rating is the sum of the closeness ratings for the placed departments adjacent to the location. The boundary length is the length of the of the boundaries common to the location and the placed departments. CORELAP placement is illustrated in figure 2.7. Assume weightings A=4, E=3, I=2, O=1, U=0, X=-1 have been assigned. Suppose departments 1, 2, and 3 have been placed, and that department 4, which has a closeness ratings A, E and I with departments 1, 2,and 3 respectively is to be placed.

Figure 2.7

The Placement Method of CORELAP



(d) CORELAP department construction: An integer number of unit squares representing the area of the department are fit into a larger square or rectangle



Then for placements (a), (b) and (c) of figure 2.7, the placing ratings are 7, 6 and 9 respectively. The boundary lengths are 4, 3 and 6 respectively. It should be noted that CORELAP does not allow the solution to backtrack, so that configuration (c), which achieved the best placing rating and boundary length by changing the location of department 3, would not be considered.

CORELAP, given the department area, calculates an integer number of unit squares to represent each department. The smallest rectangle which can contain the unit squares is used to place a department on a grid of unit squares. (figure 2.7(d)). A minimum area solution is not necesarily found. A maximum aspect ratio is defined to prevent the system always defaulting to rectangles of the width of one unit square. The final layout solution is sensitive to the size of the unit squares chosen to represent the departments and to the values assigned to the closeness ratings. Interactive CORELAP is a more recent time-shared version of CORELAP, which allows the user to intervene and change the locations of departments before allowing the solution to proceed to the next department. It has a modified Closeness rating calculation which multiplies the closeness rating value by the rectilinear distance between the departments.

2.2.4.2 CRAFT

CRAFT, an improvement algorithm was originally presented by [Armour and Buffa, 1963]. It attempts to minimize the flow cost of the layout by using an algorithm based upon the Pairwise Steepest Descent Procedure, an iterative improvement algorithm described in section 2.2.3.1. The input requirements are

1. An initial layout
2. A matrix of flow data
3. A corresponding matrix of flow cost data
4. Number and location of immovable departments

The matrix input of flow cost data allows different costs to be assigned to

different types of material handling equipment. If the cost is a linear function of distance, but not flow weight, in order to model a conveyor for example, the flow between departments may be set to unity. There is no reason why a similar program could not be written which allows more accurate description of flow cost functions as input. CRAFT works by predicting the cost of a department interchange and swapping the appropriate department centroids. The department shapes are not maintained when the interchange occurs. Figure 2.8 shows an example of the department interchange procedure. In the extreme case we could produce a layout like that of 2.8(d).

2.3 Expert Systems

FADES, a Facilities Design Expert System is being developed by [Fisher, 1986]. The major modules currently implemented in FADES are:

(1) a machine selection system, [Farber and Fisher, 1985], which uses knowledge, such as

> IF workstation technology is robotic
> AND odd-angle insertion needed
> THEN select robot model-A cf 80 ,

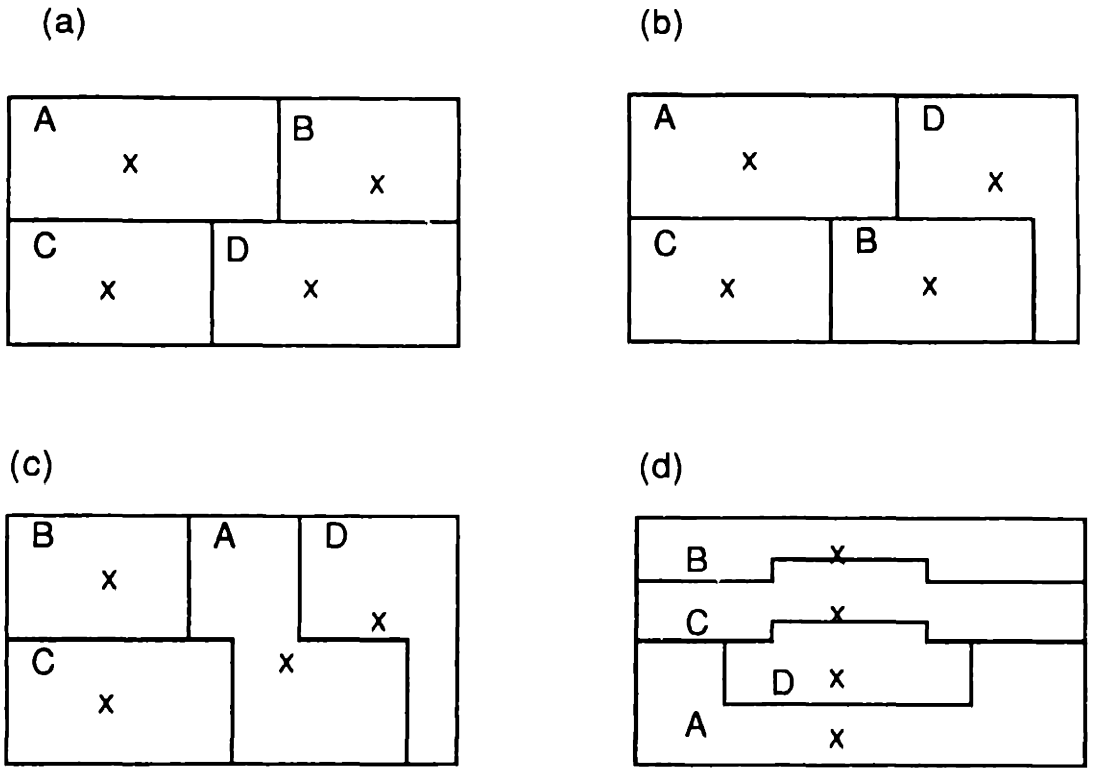
(2) a capacity planning module which determines the quantity of machine types needed, given the production volume of parts and the length of machining time required for each part,

(3) a layout conflict module which determines if certain machines cannot be assigned to certain sites because, for example, the foundations cannot support the weight of the machine,

(4) a layout application program selection module, which sends the departments specified to CRAFT, CORELAP, quadratic assignment, or linear assignment algorithms depending upon the user's selection and the characteristics input to the system,

Figure 2.8

Pairwise Interchange of Department Centroids in Iterative Improvement Algorithms



(5) a relationship generation module which determines numerical closeness ratings between departments.

The main knowledge base in FADES is in the machinery selection module. Closeness ratings are determined from the input parameters (design objective weightings, department attributes and certainty factors) which are specific to the problem. Industry specific knowledge to aid layout is not incorporated. Layout is performed by the established techniques described in (4).

Summary of Existing Procedures

This chapter focussed upon formulations of the layout problem, solution procedures, and Computer Aided Layout. Most Computer Aided Layout systems, such as CORELAP, use a matrix of desired closeness ratings to assign departments. Closeness ratings from *absolutely necessary*, through *unimportant* to *undesirable* are assigned by the designer. The ratings may be assigned according to flow as well as other factors such as noise, so to some extent the multi objective nature of the problem is incorporated. The closeness ratings are subjective. One closeness rating may incorporate several factors, such as flow, noise and dust-producing. The objectives of the design are not made explicit by the closeness ratings, so that it is not really clear what the solution represents. The limitations of the current systems and the required data input have restricted their use in practice.

The main formulations upon which solution procedures are based are the Graph Theoretic and the Quadratic Assignment Problem formulations. In the QAP the departments are limited to squares. In the Graph Theoretic approaches adjacencies are achieved at the expense of controlling the department shape, which may limit the desirability of the solution if the resulting department shapes suggest large internal flows. In the iterative improvement algorithms, department shape could not be maintained. In the construction algorithms such as CORELAP which use hill climbing

procedures, departments are built from unit squares to form rectangles for placement on a unit square grid. The solution obtained is a local optimum.

It is informative to note that all computer formulations and algorithms for factory layout make some simplifying assumptions such as these. Undoubtedly the large amount of data and the combinatorial nature of the relationships which represent the layout problem render it enormously complex. But this prompts the question of whether the factory design problem, when fully specified, is solvable at all.

Most computer models of factory design have begun, not from a consideration of what is required to lay out a factory, but from a generally applicable OR model, which is then contorted to try and represent the factory layout problem.

CHAPTER 3

MATHEMATICAL BACKGROUND

Overview

This Chapter serves as an introduction to computability, logic, logic programming and complexity. The mathematical concepts and tools form the basis for the model of factory design developed in Chapter 5, and the complexity analysis of Chapter 6. The software model of Chapter 7 is written in a logic programming language. Since the emphasis of this chapter is on explanation an informal presentation is preferred.

3.1 Computability

Computability is the branch of computer science devoted to determining what computers can and can not do. It is concerned with what is and is not theoretically possible by computation. For example, is there an algorithm that takes an arbitrary polynomial equation P(x1,.....xn) = 0 with integer coefficients, and determines whether or not the equation has a solution in integers? This is a famous problem known as Hilbert's Tenth problem, which Hilbert listed in 1901 as one of a group of problems which were to stand as a challenge to future generations of mathematicians. This remained an open problem for about seventy years until in 1970 Matijsevic showed that such an algorithm can not exist. Many interesting classes of problems are unsolvable. The issue is not one of computer memory or resources. No matter how sophisticated computers become, these classes problems will remain unsolvable because they require an infinite amount of computation. In computability the question is posed as a yes/ no problem. We are not concerned with specific instances of problems, but with classes of problems. We know that if a class of problems is computable, then we can find a value for a given instance.

We define two classes of problems:

Computable problems (also known as decidable problems) :

Those problems for which an algorithm exists. For example, "Is X a prime number?" No matter how large X is, given X, we can systematically divide it by every integer between 1 and $\sqrt{X}$ to determine whether it is a prime. We can conclusively answer yes or no.

Uncomputable problems (also known as undecidable problems) :

Those problems for which an algorithm cannot conclusively determine the answer to a problem. In Hilbert's Tenth problem, we might find a set of integers which satisfies an instance of the problem, but if we do not find such a set, we do not know whether to carry on searching or whether to stop. We cannot tell conclusively that a solution does not exist.

A subset of the class of uncomputable problems is the class of problems which are partially decidable (or recursively enumerable).

Partially decidable problems:

Those problems for which we can conclusively tell if the answer is yes, but for which we cannot tell if the answer is no. If the answer is yes a computer program will eventually stop, but if the answer is no it may run forever.

3.1.1 Notation

Σ denotes any finite set of symbols, called an alphabet. For example $\Sigma=\{a,b\}$ denotes the alphabet consisting of words a and b. Symbols a and b are letters of the alphabet. A word over Σ is any finite sequence of letters from Σ. Σ^* denotes the set of all words over Σ including the empty word Λ. If $\Sigma=\{a,b\}$, then $\Sigma^*=\{\Lambda,a,b,aa,ab,ba,bb,aaa,....\}$. A language over Σ^* is a set of words belonging to Σ^*.

The following notation of set theory is used:
A set, S, is a collection of objects. An object, s, in the collection, S, is a

member or element of S. We write $s \in S$. If T is a subset of S, then T is included in S and we write $S \supseteq T$. If $S \supseteq T$, but $S \neq T$, we write $S \supset T$.

3.1.2 A Corner stone of Decidability Proofs - The Turing Machine

A Turing machine is the most general computing device. It consists of three parts illustrated in figure 3.1 :

1. A *tape* which is divided into cells. It has a leftmost cell but is infinite to the right. Each cell holds one tape symbol. The tape symbols include the letters of a given alphabet,Σ, a finite auxiliary alphabet, V, with symbols for marking such things as break points, and the special blank symbol Δ. All symbols are distinct.
2. A *tape head* which scans one cell of the tape head at a time. In each move the head prints a symbol on the tape head scanned, replacing what was written there, and moves one cell left or right.

3. A *program* which contains the instructions that causes changes to take place at each step. It is a finite directed graph which contains one start state, denoted by START and a possibly empty set of halt states, denoted by HALT. The instruction on each arrow is of the form (a,A,R). This says if the symbol read is *a,* print symbol *A* and move one step to the right.

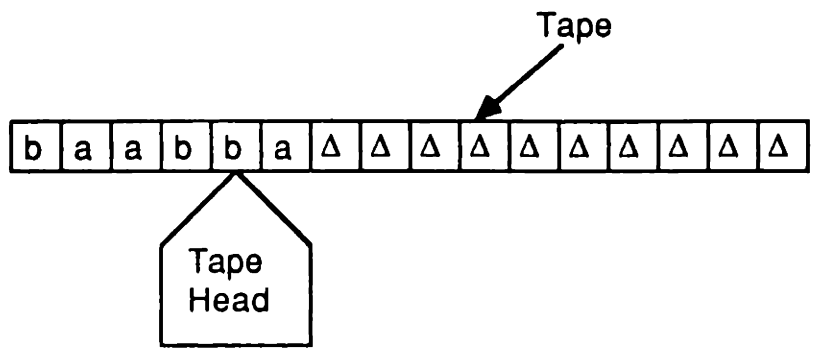
3.1.3 Prominent results in Computability

3.1.3.1 Church's Thesis

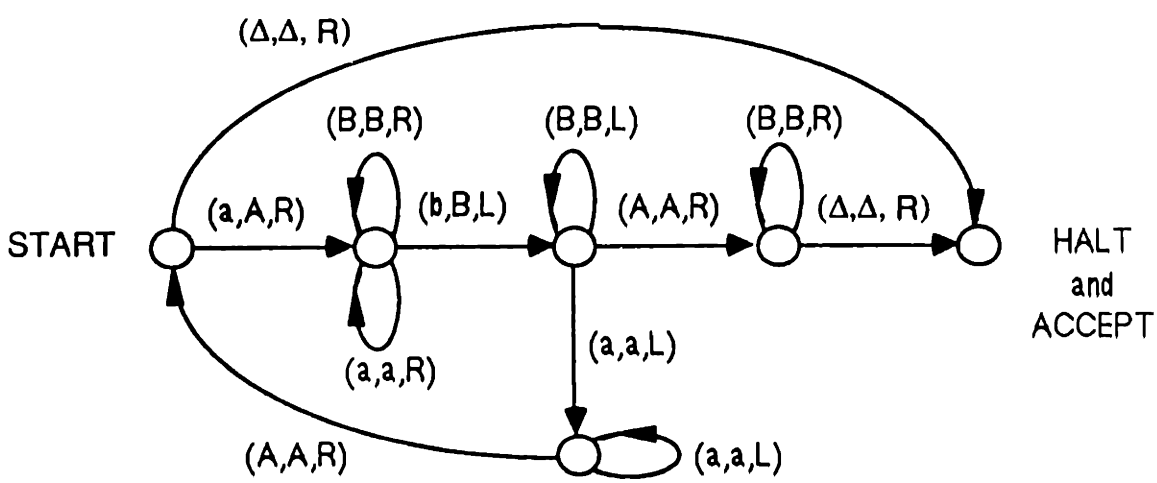
Church's Thesis states that any computable function can be computed by a Turing machine. Conversely, any computation that can be carried out on a computer can be described by means of a Turing Machine. That is, all models of computers are equivalent in terms of the problems which they can solve. Since the notion of a computable function is not mathematically precise, but is defined in terms of computing devices, Church's thesis can not be proved formally. The implication of Church's thesis is that if we can show that a problem is computable by a Turing Machine representation, then it is

Figure 3.1

The Components of a Turing Machine: a Tape, a Tape Head and a Program



Program: A program that accepts every word of the form $a^n b^n$, $n \geq 0$



computable on any machine.

3.1.3.2 The Halting Problem of Turing Machines

The Halting Problem of Turing Machines over Σ is a class of yes/no problems which can be stated as follows:

> Given an arbitrary Turing machine, M, over Σ and an arbitrary word $w \in \Sigma^*$, does M halt for input w?

In 1936, Turing proved that the Halting Problem of Turing Machines is undecidable. (See Appendix 1 for proof). That is, there is no algorithm that takes an arbitrary Turing Machine, M, and input word w and determines if M halts for input w. The problem is uncomputable because of its self referencing nature. The problem is to define a Turing machine which determines whether an arbitrary Turing machine halts on an arbitrary word. A problem which is apparently simpler, to determine whether an arbitrary Turing Machine will halt on the empty word Δ is also uncomputable.

The Halting Problem is termed a partially decidable problem. If it halts we know that it has halted. But if it has not yet halted, there is know way of knowing if it will ever halt.

Using the unsolvability of the Halting Problem of Turing Machines, many other classes of problems can be proved unsolvable by *reducing* the Halting Problem of Turing Machines to those classes.

3.1.4 Reducibility.

A class of problems, $\mathcal{P}$, is *reducible* to another class of problems $\mathcal{P}'$, if there is a a Turing machine which takes any problem $P \in \mathcal{P}$ as input and yields some problem $P' \in \mathcal{P}'$as output, such that the answer to P is yes if and only if the answer to problem P' is yes. Suppose that we show that $\mathcal{P}$ is reducible to $\mathcal{P}'$. Then if $\mathcal{P}'$ is solvable, we can combine the reduction Turing machine from $\mathcal{P}$

to $\mathcal{P}'$ and the algorithm for solving $\mathcal{P}'$ to form an algorithm for solving $\mathcal{P}$. We therefore have the following results:

If a class $\mathcal{P}$ of yes/no problems is reducible to $\mathcal{P}'$, then,

(i) if $\mathcal{P}'$ is solvable so is $\mathcal{P}$,
(ii) if $\mathcal{P}$ is unsolvable so is $\mathcal{P}'$.

3.1.5 An Example Undecidable Problem

The Word Problem of Semi-Thue Systems

A semi-Thue system S over $\Sigma = \{a,b\}$ consists of a non empty set of k, (where $k \geq 1$), ordered pairs (α_i,β_i) of words over Σ; that is ,

$$S = \{ (\alpha_1,\beta_1),(\alpha_2,\beta_2),.....,(\alpha_k,\beta_k) \}.$$

For two words $x,y \in \Sigma^*$, we say that y i*s derivable from* x in S if there exists a sequence of words over Σ

$$w_0,w_1,w_2,.....w_n \qquad \text{where } n \geq 0$$

such that w_0 is x, w_n is y and w_{i+1} is obtained from w_i, where $0 \leq i \leq n$, by replacing some occurrence of a substring α_j (which is the left hand side of some pair in S) in w_i by the corresponding β_j (which is the right hand side of that pair).

For example, consider the semi-Thue system $S=\{(ba,ab),(aab,\Lambda)\}$ over $\Sigma=\{a,b\}$, where Λ denotes the empty word and $(x\Lambda = \Lambda x = x)$. Then Λ is derivable from $x \in \Sigma^*$ in S if and only if x has twice as many a's as it has b's; for example, Λ is derivable from *baaaba* but not from *baaab*.

The word problem of semi Thue systems over Σ is a class of yes/no problems

which can be stated as follows: Given an arbitrary semi-Thue system S over Σ and two arbitrary words $x,y \in \Sigma^*$, is y derivable from x in S.

Post proved that the word problem of semi-Thue systems over Σ is unsolvable. That is, there is no algorithm that takes an arbitrary semi-Thue system S over Σ and words $x,y \in \Sigma^*$ as input and determines whether or not y is derivable from x in S.

Intuitively, we conceive of the word problem of semi-Thue systems being undecidable, because with a rewrite rule such as (a,ab) we could continue to expand the input word infinitely, or with a combination of rewrite rules (ab,a),(a,ab) we could loop for ever. The unbounded nature of the problem stems from the fact that the length of the sequence of indices that might yield the plan is not bounded by any computable function of the number of possible states. It might be argued that the problem could be made decidable by examining the nature of the input or by using a loop detection mechanism. For example define a function *runtime* such that if Machine M has not halted over t after runtime(M,t) steps then it never will. But it can be shown that any such function has the self referencing properties of the halting problem and that the problem remains undecidable.

3.1.6 The Nature of Undecidable Problems

We may think of undecidable problems in three ways:

1.Those with an arbitrary but finite input set from which a possibly unbounded number of paths to be searched can be generated. e.g. the word problem of semi-Thue systems.

2. Those attempting 'self awareness' or 'introspection' in that they ask questions within the system about the system. Does a Turing machine Halt on its own code?.

3. Those problems for which an unbounded input set of possibilities must be

checked in order to answer the decision problem.

3.1.7 The Nature of Decidable Problems

We may think of decidable problems in two ways:

1. Problems which admit an algorithm.
A class of yes/ no problems is decidable if there is some fixed algorithm (Turing Machine) for which any problem in the class as input will determine the solution to the problem.

2. Problems with a finite input set, from which can be generated a finite number of finite search paths.

3.2 Symbolic Logic

3.2.1 Propositional Logic

A *proposition* is a declarative statement or sentence. It may either be true or false. For example, the propositon C, "cows can fly" , is false whereas S, "sparrows can fly", is true.

Functors allow us to express statements about one or more propositions. The *not* functor , written ~, negates the truth value of a proposition or a set of propositions. If S is true, ~S is false.

Connectors are functors which link propositions, enabling us to make compound statements. The *and* connector, symbolized as $\wedge$, results in the compound statement being true if each of the referent statements is true. Taking the example above, C$\wedge$S is false , since C is false. The *or* connector, symbolized as $\vee$ results in the compound statement being true if one or more of the referent statements is true. In the example above C$\vee$S is true since S is true.The functors and their symbols are summarized below:

Functor	Symbol
Not	~
And	$\wedge$
Or	$\vee$
Implies	$\rightarrow$
Is implied by	$\leftarrow$
Is equivalent to,(iff)	$\leftrightarrow$

The implication connector, $\rightarrow$, is also known as IF-THEN. In a compound statement, $P \rightarrow Q$, is is read "P implies Q", or "if P then Q". It is false if P is true and Q is false, and is true in all other cases.In the exampleabove $C \rightarrow S$ is true since C is false, while $S \rightarrow C$ is false since Sis true and C is false. The left hand side of the implication is called the *antecedent* or *premise* while the right hand side is called the *consequent* or *conclusion.*

The implied by connector, $\leftarrow$, also called if, is the converse of implication. $Q \leftarrow P$ is true if and only if $P \rightarrow Q$ is true.

The equivalence connector, $\leftrightarrow$, is also called if and only if , or iff. $P \leftrightarrow Q$ is true if $P \rightarrow Q$ and $Q \rightarrow P$ are both true. It is false otherwise.

By convention, the following precedence ordering is observed among functors, in increasing scope of influence:

1. ~ 2. $\wedge$ 3. $\vee$ 4. $\rightarrow, \leftarrow, \leftrightarrow$

For example , $\sim P \wedge Q \vee R \rightarrow T$ means $[(((\sim P) \wedge Q) \vee R) \rightarrow T]$. The precedence ordering may be superceded by explicit use of parentheses. If we evaluate all combinations of truth values for $(P \rightarrow Q)$ and $(\sim P \vee Q)$ we find that they are equivalent. In fact the functors $\rightarrow$, $\leftarrow$ and $\leftrightarrow$ are redundant. They may each be defined in terms of ~,$\wedge$ and $\vee$. As a result of the redundancy there are many different ways of writing the same proposition. In theorem proving this is inconvenient. However there is a procedure for translating all propositions into a special form called *clausal form.* This will be discussed

further in section 3.4.1.

3.2.2 Predicate Logic

Propositional logic is limited in scope because propositions do not allow their arguments to be expressed as variables. For example, there is no way to say "if x is a bird then x lays eggs " for any arbitrary object x. We generalize the concept of propositions by introducing *predicates.*

A *predicate* is a function which assigns a truth value to the objects which serve as its arguments. For example the predicate *bird (x)* is true if x is a bird. The statement "if x is a bird then x lays eggs" may be written in predicate logic as

$$bird(x) \rightarrow lays\_eggs(x).$$

The statement "x likes spinach" may be written as

$$likes(x,spinach).$$

We must clarify the meaning of a variable introduced into a proposition. This meaning is only defined when the variable is introduced by a quantifier. The quantifier tells us if the proposition is true for some or all values assigned to the variables. There are two quantifiers, the universal quantifier, and the existential quantifier. The universal quantifier, $\forall$, quantifying the propostion $man(x) \rightarrow human(x)$, is written

$$\forall x\{man(x) \rightarrow human(x)\}.$$

This reads "for all x, if x is a man, then x is human". The existential quantifier, $\exists$, quantifying the proposition, likes(x,spinach), is written

$$\exists x\{likes(x, spinach)\}.$$

This reads " there exists an object , x, that likes spinach". If this quantified proposition is true, then at least one x likes spinach. However, there may be more than one x that likes spinach. The quantifiers may be defined in terms of each other by using the functor ~. For example $\exists x\{bird(x)\}$ is logically identical to $\sim \forall x\{\sim bird(x)\}$.

3.3 Formal Logic

3.3.1 Introduction to The Predicate Calculus

The predicate calculus is a formal language whose essential purpose is to symbolize logical arguments in mathematics. The language consists of primitive symbols and rules for combining them into sentences which are called *well formed formulas* (wffs). By interpreting the symbols in a wff we obtain a statement which is either true or false. We can associate many different interpretations with the same wff and therefore obtain a class of statements where each statement is either true or false. We can study the logical system for both its semantic meaning and its syntactic structure. We are interested in the syntactic structure so that we can prove theorems and axioms. We are interested in the semantic interpretation of the symbols so that we can focus on the meaning of the sentences.

The following is a typical wff of the predicate calculus:

$$(\exists F)\ \{\ (F(a)=b)\ \wedge\ (\forall x)\ [p(x) \supset (\ F(x) = g(x,F(f(x)))\ )]\}.$$

It represents the following English sentence:

"There exists a function F such that F(a) = b is true and for every x, if p(x) is true, then F(x) = g(x,F(f(x))) is true."

A semantic interpretation of this wff is given by specifying a nonempty set D, from which we can assign elements to the individual variables and constants x, a and b. In addition we must assign meanings to the functions f and g and

the function variable F. For example, if we choose D to be the set of non negative integers and we let a be 0, b be 1, f(x) be the predecessor function x-1 (where 0-1 = 0), g(x,y) be the product function x•y, and p(x) be the predicate x>0 , then the interpreted wff reads as follows:

"There exists a function F over N such that F(0) = 1 and for every x∈ N, if x>0, then F(x) = xF(x-1)."

This is clearly a true statement. The factorial function x! is an appropriate choice for F since 0! = 1 and, for every x > 0, then F(x) = xF(x-1).values to the variables. We could choose other interpretations which would yield false statements.

3.3.2 Syntax of the Predicate Calculus

We now formally introduce the symbols and rules of the predicate calculus.

- constants: true false , ( )
- connectives: ~ (not) , → (implies), ∧ (and), ∨ (or)
- equality: =
- quantifiers: ∀ (universal quantifier), ∃ (existential quantifier)

- variables: function variables, F^n ; individual variables v,x,y,z
  predicate variables, P_i^n; propositional variables P_i^0

- constants: function constants, f^n,g^n,h^n ; individual constants a,b,c
  predicate constants, p_i^n : propostional constants, p_i^0
  (true or false)

Using these symbols, we define recursively three classes of expressions:

1. *Terms*

- individual constants*a* and variables *x* are terms

- If $t_1, t_2, \ldots, t_n$ are terms, then constant functions, $f(t_1, \ldots t_n)$,or variable functions, $F(t_1, \ldots t_n)$ are terms.

2. *Atomic formulas*

   Each propositional constant p^0 and each propositional variable is an atomic formula.

- If t_1 and t_2 are terms, then, $t_1 = t_2$ is an atomic formula.
- If $t_1, t_2, \ldots, t_n$ are terms, and p_i is a predicate constant then $p_i (t_1, t_2, \ldots t_n)$ is an atomic formula.
- If $t_1, t_2, \ldots, t_n$ are terms and P_i is a predicate variable then $P_i (t_1, t_2, \ldots t_n)$ is an atomic formula.

3. *Well-formed formulas (wffs)*

   There are three types of wffs:

   (i) *Atomic formulas*

   Each atomic formula is a wff

   (ii) *Logical sentences*

   Atomic formulas combined with logical operators are wffs e.g. if A and B are atomic formulas then so are $(\tilde{}A)$, $(A \rightarrow B)$,$(A \wedge B)$

   (iii) *Quantified sentences*

   Universally and existentially quantified atomic formulas or logical sentences are wffs e.g. if x is an individual variable and A is a formula then $(\forall xA)$ and $(\exists xA)$ are wffs.

The class of wffs described above is usually called second order predicate calculus (with equality). There are several interesting subclasses of the wffs which are of special interest. Two subclasses are indicated below by stating the admissible set of constant and variable symbols

1. Propositional calculus

   Propositional constants p_i^0

   Propositional variables P_i^0 (true or false)

2. First order predicate calculus (with equality)

individual constants a_i
function constants f_i^n
propositional constants p_i^0
predicate constants p_i^n
individual variables x_i

In the first order predicate calculus we do not allow predicates or functions to be variables. That is, only individual variables may be quantified. In the second order predicate calculus we allow both predictes and functions to be variables and to be quantified. The example of section 3.3.1 was drawn from the second order predicate calculus.

Terms and formulas are syntactic entities constructed out of symbols. We now turn to the semantics.

3.3.3 Semantic Interpretations of the Predicate Calculus

In the propositional calculus where propositions do not contain variables, an interpretation is simply an assignment of truth values to atomic formulas These truth values are then used to determine the truth values of wffs. The definition of an interpretation must be modified for predicate calculus in which propositions consist of functions or predicates with variables as arguments. We define the semantic terminology:

A model, $\mathfrak{M}$, of a wff, S, is a pair $(D,\mathfrak{I}_c)$ where:

1. D is the domain of the interpretation. The domain consists of all objects which can be assigned to the individual constants and variables.

2. $\mathfrak{I}_c$ indicates assignments to the individual, function, and predicate constants of S. Individual constants are assigned some element of D. Function constants are assigned some n-ary function over D. Predicate constants are assigned some n-ary predicate over D, or a truth value in

the case of a propositional constant where n=0.

An interpretation of the wff S is a pair $(\mathfrak{M},\mathfrak{I}_\nu)$, where:

1. $\mathfrak{M}$ is a model of the wff.
2. $\mathfrak{I}_\nu$ is a valuation function for the model $\mathfrak{M}= (D,\mathfrak{I}_c)$. It indicates assignments to the free variables of S. Individual variables are assigned some element of D. Function variables are assigned some n-ary function over D. Predicate variables are assigned some n-ary predicate over D, or a truth value in the case n=0.

An interpretation $\mathfrak{I}$ has a truth value which is obtained by first applying the assignments of $\mathfrak{I}_c$ to all the constants of S, then applying the assignments of $\mathfrak{I}_\nu$ to the variables in A, and then interpreting the meaning of the logical symbols.

An interpretation satisfies a sentence if the resulting truth value is *true* . An open wff is one which contains free variables. Closed wffs are quantified over all variables. An open wff may be true or false depending upon the specific assignments of constants to variables. To obtain absolute or categorical results rather than conditional ones, we work with closed wffs in the semantic approach to logic.

We now formally describe the semantic interpretation of the wff of the example of section 3.3.1 :

$$(\exists F)\ \{\ (F(a)=b)\ \wedge\ (\forall x)\ [p(x)\rightarrow (\ F(x) = g(x,F(f(x)))\ )]\}.$$

Let the domain be N, the set of non negative integers.

Let $\mathfrak{I}_c$, the interpretation of the constants be as follows:

a=0, b=1,

f(x) is the predecessor function x-1, (0-1=0),

g is the product function • such that g(x,y) = x•y

p(x) is the predicate x>0.

Then we have a model,$\mathfrak{M}$,

$$(\exists F)\ \{\ (F(0)=1) \wedge (\forall x)\ [\ (x > 0) \rightarrow (\ F(x) = x \cdot F(x - 1)\ )\ ]\ \}.$$

In words, there exists a function F over N such that F(0)=1 and for every $x \in N$, if x>0, then F(x)=x•F(x-1).

The variable assignments of $\mathfrak{I}_v$ required to complete the interpretation of the model, $\mathfrak{I}$, are

F, an interpretation of the variable function over the domain N,
x, an individual variable assigned from the domain N.

An interpretation $\mathfrak{I}_v$ which satisfies the model $\mathfrak{M}$ is assignment of the factorial function x! to F. For a function assignment to be valid it must satisfy every variable interpretation. i.e. x! must hold for every x which is a non-negative integer, which it does.

3.3.4 Valid Well Formed Formulas

A wff is said to be *valid* if it yields the value true for every interpretation. A wff is said to be *unsatisfiable* if it yields the value false for every interpretation. A wff is *satisfiable* if there exists some interpretation for which A yields the value true: Let S be a wff and I be an interpretation. We say that I satisfies S, written

$$I \models S,$$

iff the interpretation I yields the value true. We call this a *true interpretation.*

In first order predicate calculus we will often talk of satisfaction over models rather than interpretations. This is defined as follows: Let S be a wff and $\mathfrak{M}$ be a model of S. We say that $\mathfrak{M}$ satisfies S, written

$\mathfrak{M} \models S$,

iff for all valuation functions $\mathfrak{I}_v$ for $\mathfrak{M}$ it is the case that $(\mathfrak{M},\mathfrak{I}_v) \models S$. We call this a *valid model.*

A set of wffs **W** logically implies (synonymously logically entails) a wff W if and only if every constant and variable assignment that satisfies the set of wffs **W** also satisfies W. In this case we write

W $\models$ W.

We read this as "**W** satisfies W".

3.3.5 The Validity Problem

3.3.5.1 The Validity Problem of the Propositional Calculus

Given an arbitrary wff of the quantified propositional calculus, the decision problem of validity is computable. That is there is an algorithm for determining whether any wff of the propositional calculus is valid.

3.3.5.2 The Validity Problem of First Order Predicate Calculus

Church proved that the validity problem of first order predicate calculus is unsolvable, but partially solvable. There is no algorithm that takes any first order wff as input and always halts, reaching an ACCEPT halt if the wff is valid and a REJECT halt if the wff is nonvalid. However, there is an algorithm that takes an arbitrary first order wff as input and always reaches an ACCEPT halt if the wff is valid but may reach a REJECT halt or loop forever if the wff is nonvalid.

In certain cases we may reduce the validity of wffs in first order predicate calculus to the validity of wffs in propositional calculus. Hilbert and Ackerman showed that if the domain is finite and that the wff could be represented by a finite conjunction of sentences in the propositional calculus, then the validity

of a formula in a finite domain is reduced to the universal validity of a sentence in the propositional calculus.

3.3.5.3 The Validity Problem of Second Order Predicate Calculus

Godel showed that the validity problem of second order predicate calculus is not even partially solvable. That is, there is no algorithm that takes any second order wff as input and reaches an ACCEPT halt if the wff is valid . It may reach a REJECT halt or loop forever if the wff is nonvalid.

3.3.6 Deduction

Certain formulas are necessarily valid due to the meaning of logical symbols, e.g $x \leftrightarrow x$. Formulas such as these are called *logical axioms.* Other axioms which are valid by hypothesis are called proper or nonlogical axioms.

A deduction system for the Predicate Calculus consists of logical axioms and two production rules. These rules of inference are

1. Modus Ponens: From P and $(P \rightarrow Q)$, infer Q.
2. Generalization: From P, infer $(\forall x)P$.

Inference rules may be used to deduce theorems from axioms. A theory T is a formal system consisting of the logical and nonlogical axioms as well as the production rules. When a theorem W is deducible from the axioms of a theory, T we write

$$\vdash_T W$$

A wff W is a consequence of a set of wffs **W,** if W is derivable from the axioms of the theory T and the formulas in **W.** In this case we write

$$\mathbf{W} \vdash_T W$$

We read this as "**W** implies W" If W is derived from **W** and T it is true in all **W** and T.

3.3.7 Soundness and Completeness of Deduction

The semantic approach focuses upon the validity of a formula, while the syntactic approach deals with the construction of proofs of formulas.

The property of *soundness* means that if the formula W is derivable from the set of axioms **W**, then W is true in all models of **W**.

$$(\mathbf{W} \vdash W) \rightarrow (\mathbf{W} \models W) \qquad \text{soundness}$$

The property of *completeness* means that if a formula W is true in all models of the set of axioms **W**, then W can be proved from **W** viewed as a set of hypotheses.

$$(\mathbf{W} \models W) \rightarrow (\mathbf{W} \vdash W) \qquad \text{completeness}$$

There are complete deduction systems for first order predicate calculus. That is, for any given valid wff, W, we can construct a proof to prove W from **W**. This may be expressed as the soundness-completeness equivalence of first order logic.

$$(\mathbf{W} \vdash W) \leftrightarrow (\mathbf{W} \models W) \qquad \text{completeness iff soundness}$$

"implies iff satisfies"

The semantic approach which explicates the property of soundness is equivalent to the syntactic approach which explicates the property of completeness.

Note that although determining the validity of any first order wff was only partially decidable (section 3.3.5.2) determining the validity of the deduction

of a first order wff from a set of axioms is decidable. Godel proved that there are complete deduction systems for first order predicate calculus.

3.3.8 Godel's Incompleteness Theorem

Godel's Incompleteness Theorem states that there is no complete deduction system for the second order predicate calculus. The problem of designing a sound, complete clerical routine for recognizing second order validity is undecidable. Any axiomatic system whose vocabulary is adequate to express a sufficiently rich collection of statements, and which is *consistent* (i.e., its axioms do not imply any contradictions, must also be *incomplete* in that there are truths about numbers or symbols expressible but not provable, in this axiomatic theory.

3.3.9 Theorem Proving

We discuss theorem proving with examples in symbolic logic as well as mathematical logic because it forms the basis of logic programming (section 3.4). To determine whether a given wff is a theorem of a particular set of wffs (theory) we could apply syntactic deduction rules (Modus Ponens and Generalization) to axioms in all possible ways. If the wff is in fact a theorem, or if the number of possible deductions is finite, then the procedure will eventually terminate. But if the wff is not a theorem then the procedure may run forever.

As noted in section 3.2.1 the redundancy among the logical symbols does not render the expressive form of predicate calculus ideal for theorem proving, as there are many ways of expressing the same proposition. The refutation procedure is a theorem proving procedure which takes as arguments, a set of expressions in a simplified version of predicate calculus called *clausal form.* The symbols , terms and atomic sentences of clasual form are the same as those in ordinary predicate calculus. Instead of logical and quantified sentences, however, clausal form has *literals* and *clauses.* A literal is an atomic sentence or the negation of an atomic sentence.

e.g. $p_1(x,y)$ $\sim p_2(a,p_3(x,a))$ likes(x,spinach)

An atomic sentence is a positive literal, and the negation of an atomic sentence is a negative literal. A clause is a set of literals joined by disjunction.

e.g. animal(x) $\vee$ vegetable(x) $\vee$ mineral(x)

Any sentence in predicate calculus may be represented by a set of clauses joined by conjunctions, for example:

person(adam) $\wedge$ person(eve) $\wedge$ ((person(x) $\vee$ ~mother(x,y)) $\vee$ person(y))

The above sentence contains three clauses:

person(adam)
person(eve)
((person(x) $\vee$ ~mother(x,y)) $\vee$ person(y)).

Moreover there is a standard procedure for converting predicate calculus sentences into clausal form. All variables in the expression are considered to be universally quantified.

The theorem proving procedure which we shall introduce, the refutation procedure is based upon the resolution principle [Robinson, 1965].

For propositional calculus the resolution principle takes the form of a simple cancellation law. If P, Q and R are literals, then the pair of clauses {P$\vee$Q,~Q$\vee$R} taken together derive the result P$\vee$R:

$$\begin{array}{l} P \vee Q \\ \underline{\sim Q \vee R} \\ P \vee R \end{array}$$

The literals P,Q,~Q and R are called resolved literals, while the resulting clause P∨R is called the resolvent.

The soundness and completeness results of section imply the following. A wff W is a theorem of a given set of axioms T iff the union of ~W and T is unsatisfiable. In other words , if ~W and T derive the empty formula, denoted by Ó, through resolution, then they are unsatisfiable and the original formula W is in fact a theorem of T.

The refutation procedure is a procedure in which the empty formula is derived from ~W and the axioms T. Consider the set of axioms T = {P∨Q, ~Q}. To determine whether P is a theorem of T, we form the extended set incorporating ~P, to give {P ∨ Q, ~Q,~P}. Applying the resolution principle derives the empty clause, ı. Hence P is a wff of T. The refutation procedure may be depicted as a tree of cancellations, as shown in the figure 3.2.

3.3.10 Theorem Proving with Unification.

The refutation principle applied to first order predicate calculus is similar but requires one prior step, since we must deal with atomic formulas containing terms. This is handled first by performing a pattern matching step called unification. The procedure involves binding variables together or instantiating constants for variables in order to obtain a uniform structure where possible. For example, the set of literals {P(x), P(a)} can be unified by the substitution of x for a, denoted by x→a. As another example, the literals { p(x), ~(p(f (y))} can be unified by y→z, x→f(z). After the unification step the refutation procedure for first order logic is identical to that for propositional calculus.

3.4 Logic Programming

3.4.1Theorem Proving with Horn Clauses.

The Refutation procedure does not describe which literal clauses to match, or which literal clauses to try to resolve. To mechanize this procedure so that we

Figure 3.2

Example of a Refutation Tree

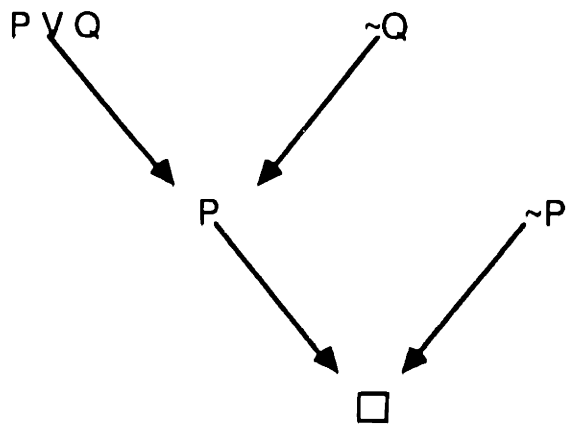
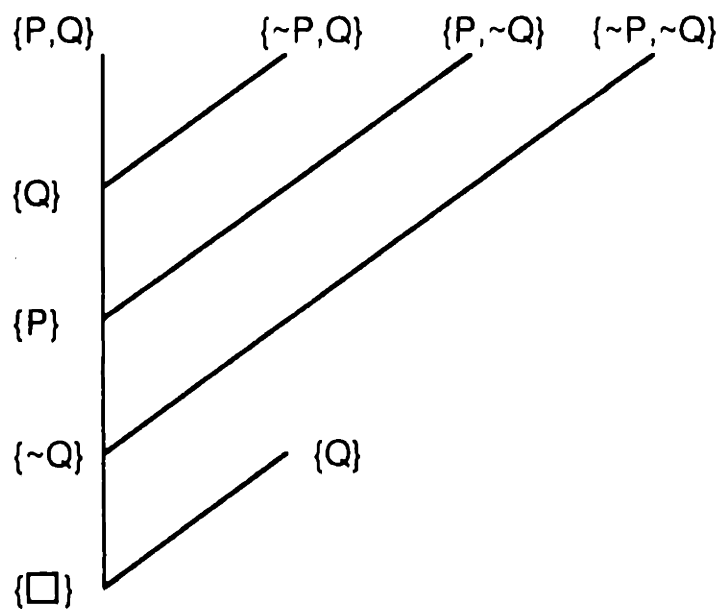


Figure 3.3

Chain of Resolutions in a Linear Deduction



may program theorem proving, we refine the clause expression and the resolution principle one step further. If we use a clausal theorem prover it is only strictly necessary to use *Horn clauses.* Also because resolution with *Horn clauses* is relatively simple , they are an obvious choice as the basis of a theorem prover which provides a practical programming system. Horn clauses are clauses with at most one positive literal. There are two types of Horn clauses. *Headless Horn clauses* have no positive literals. *Headed Horn clauses* have one positive literal. Any theorem proving task can be expressed in such a way that there is one headless clause and all of the rest of the clauses are headed. We may think of the headless clause as representing the negation of the goal which we are trying to prove.

The sentence "batchelor(x) $\rightarrow$ male(x) $\wedge$ unmarried(x)" may be represented by two headed Horn clauses:

{ (~ batchelor(x) $\vee$ male(x)) , (~ batchelor(x) $\vee$ unmarried(x)) }

A headless horn clause would be

~batchelor(x).

3.4.2 Linear Resolution - A Complete Refutation Strategy for Horn Clauses

Linear Resolution is one of several complete theorem proving strategies for Horn Clauses, [Genesereth and Nilsson, 1987]. A *linear resolvent* is one in which at least one of the parents is either in the initial database, or is an ancestor of the other parent. A *linear deduction* is one in which each derived clause is a linear resolvent. A *linear refutation* is a linear deduction of the empty clause. Linear resolution takes its name form the linear shapes of the proofs it generates. A linear deduction starts with a clause in the initial database and produces a linear chain of resolutions such as that shown in figure 3.3. Each resolvent, after the first one, is obtained from the last resolvent (called the near parent) and some other clause, which must be

either in the initial database, or be an ancestor of the near parent.

3.4.3 The "Pun" of Logic Programming.

Logic Programming works by a manipulation of Horn Clauses into IF-THEN statements. Consider the horn clause

$$\sim P \vee \sim Q \vee \sim R \vee S$$

This may be written more expressively in logic as

$$(P \wedge Q \wedge R) \rightarrow S \qquad \text{or} \quad \text{IF } P \,\&\, Q \,\&\, R \text{ THEN } S$$

The "pun" of logic programming is that to satisfy S it is sufficient to satisfy P, Q and R. We can therefore write in a procedural meaning:

```
s :-  P,                    or   S IF P & Q &R
      Q,
      R.
```

In the logic programming language Prolog we write rules in exactly this way, e.g.

```
batchelor(X):- male(X),unmarried(X).
```

The convention in Prolog is that all individual and predicate constants begin with lower case letters, and all variables begin with upper case letters. "," denotes conjunction or $\wedge$, and ";" denotes disjunction or $\vee$. The clauses of a Prolog program correspond to the headed clauses of a theorem prover. A very simple Prolog program is :

```
male(jim).
male(john).
unmarried(john).
```

batchelor(X):- male(X),unmarried(X).

The three statements in the knowledge base, that "john" and "jim" are male, and that "john" is unmarried, and the rule which defines batchelor all correspond to headed clauses.

The Prolog prompt ,

?-

which is used to express queries to the database, allows expression of the headless clause. For example

?- batchelor(john)

corresponds to

:- batchelor (john)

in Prolog, or

~batchelor(john)

in logic.

Prolog would answer this query

?- yes

since the refutation procedure will be able to derive the empty clause from the program. In response to

?- batchelor(X).

Prolog would answer

X = john
yes

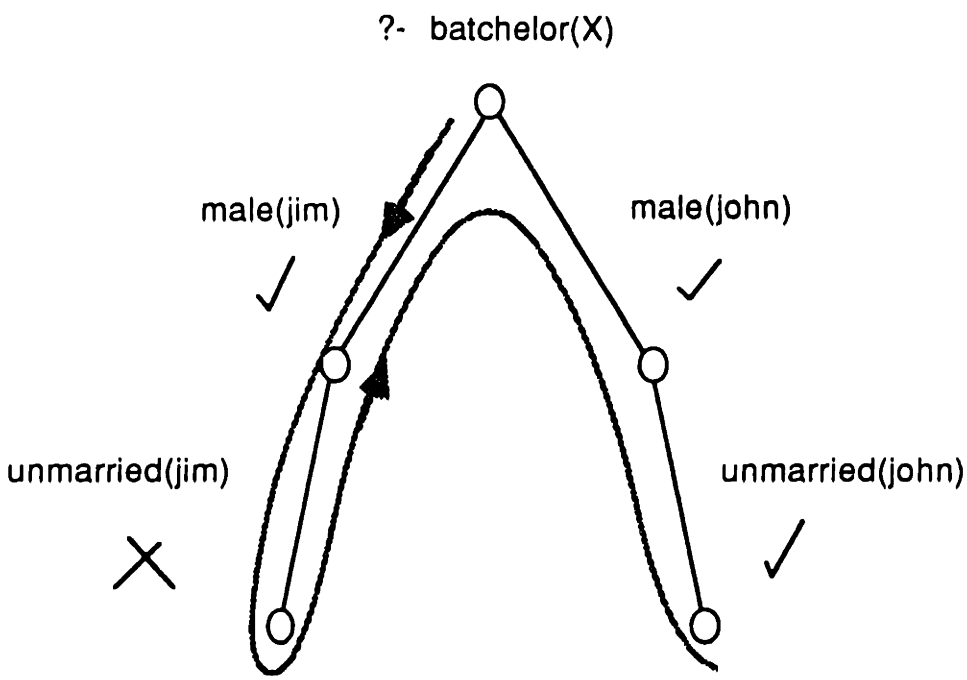
The particular resolution strategy used by Prolog is a form of *ordered* linear resolution. Literals are chosen for unification and resolution in the order in which they appear in the clause. As a result of this resolution procedure, Prolog may be thought of as a depth first search mechanism with backtracking. It treats the "batchelor" clause as a sequence of subgoals, which it tries to satisfy in order, from left to right, by *pattern matching* with the database. First it tries to satisfy male(X), and then it tries to satisfy unmarried(X), with the value of the variables (in this case X) dictated by the subgoals to its left. The order in which it instantiates the value of X in male (X) depends on the order of the clauses in the database. Figure 3.4 shows that male(jim) is tried first, because this is the first clause of the form male(X) in the database. Prolog uses the Closed World Assumption. That is, since it is not expressed explicitly that "jim" is unmarried, this clause fails, he is presumed unmarried, and backtracking is invoked to search for another value for X. Eventually we find that "john" is a batchelor because both male(X) and unmarried(X) are satisfied by X= john.

3.4.4 Declarative programming

The aim of logic programming is to provide the programmer with techniques for decomposing a computational problem into two separate problems: "what" is to be computed and "how" it should be computed. The intended beauty of logic programming is that we can write the "what" specification of the problem in logic. For example we may write in Prolog that a batchelor is male and unmarried and ask it to tell us who the batchelors are. Through the unification and refutation procedures which are built in to Prolog, Prolog returns with the answers. There was no need to specify to prolog any procedural aspect of the computation, we just declare "what" a batchelor is. We call this declarative programming.

Figure 3.4

Illustration of the Depth First Search with Backtracking Technique used by Prolog in Instantiating Variable X.



3.4.5 Procedural Programming

In the above example it was unnecessary to enhance the declarative meaning of the batchelor program with any control information to ensure the program was interpreted. If we reordered the clauses to read batchelor :- unmarried(X), male(X) we would still find all the batchelors. However in more complicated programs the declarative reading of the program may be correct, but a consequence of the ability to control the execution of the logic program by imposing order on the clauses may send the program into infinite loops. [Bratko, 1986] and [Abelson and Sussman, 1985] discuss this in some detail. This is the extreme of the inefficiency which is possible. Suppose that there are ten males in the database and only one is unmarried. Then the ordering of the clauses in "batchelor" is very important for the efficiency of the program. If we try to match the male(X) literal frst, our chances that X is the batchelor are 1 in 10. We illustrate this in figure 3.5. If we try to match the unmarried(X) literal first, our chances are 100% This is shown in figure 3.6. It is often very useful to have information about the number of possible instantiations when we write the program.

In addition to paying attention to the ordering of clauses we may take advantage of Prolog's "built in predicates" which allow us to express control information about how the proof is to be carried out. For example the "cut" predicate prevents backtracking through the database past the position of this predicate. Prolog also has a means for asserting and retracting clauses from the database. These operations all violate the simple self contained nature of predicate calculus propositions. For example asserting clauses into the database during a proof allows us to have a different set of axioms at different stages of the proof. Given these facilities which allow us to violate the premise on which logic programming was designed, we might question whether we can expect any of the advantages of logic to apply to Prolog programs. The responsibility to ensure that the advantages do apply lies with the programmer. By adopting an appropriate programming style, such as using cuts and assertions simply to improve the efficiency of the program, rather

Figure 3.5

(a) Database and Program for Finding Batchelors

```
male(hiroki)
male(yehuda)
male(sanjay)
male(bruce)
male(harry)
male(kurt)
male(yongun)
male(zhixin)
male(pascal)
male(mohammed)

unmarried(bruce)

batchelor(X) :- male(X), unmarried(X).
```

(b) Search Path For A Batchelor in the Database of (a)

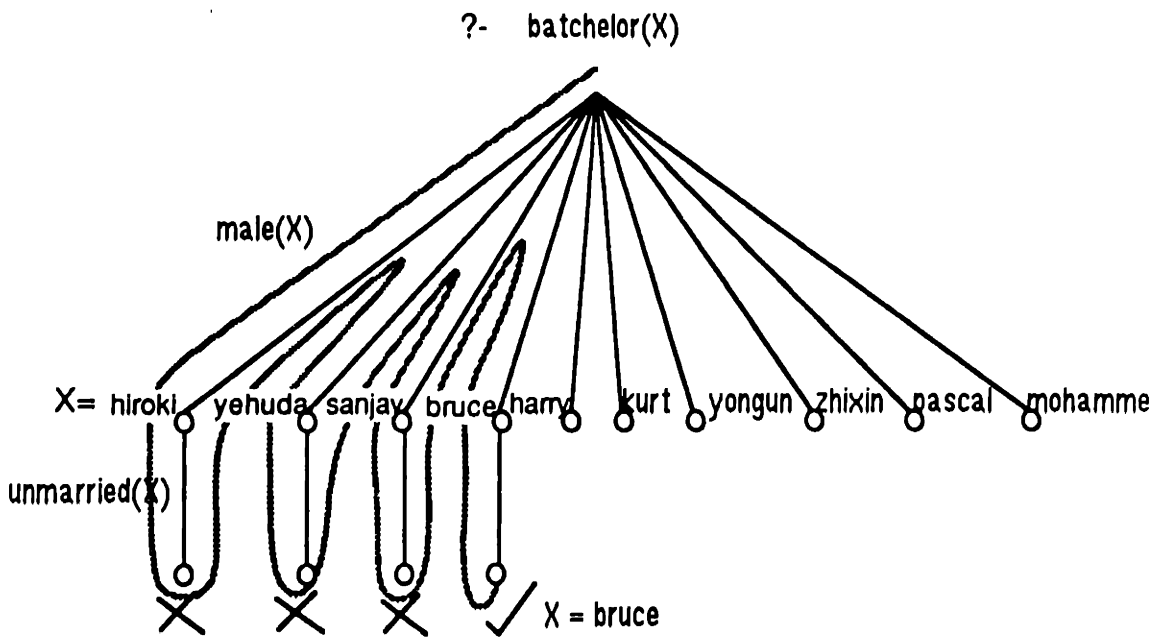
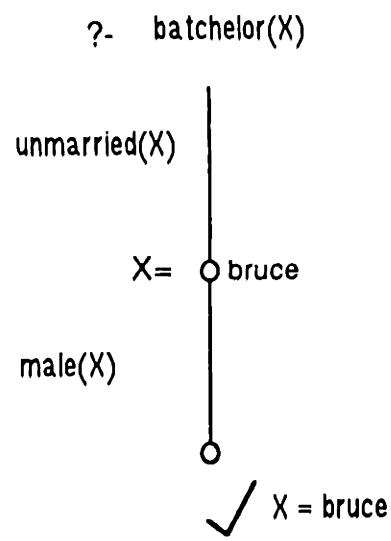


Figure 3.6

(a) Revised Database and Program for Finding Batchelors

```
male(hiroki)
male(yehuda)
male(sanjay)
male(bruce)
male(harry)
male(kurt)
male(yongun)
male(zhixin)
male(pascal)
male(mohammed)

unmarried(bruce)

batchelor(X) :- unmarried(X),male(X).
```

(b) Search Path For A Batchelor in the Database of (a)



than to alter its declarative meaning, and by resticting the use of control predicates to limited portions of the progam, we can still exploit the properties of logic.

3.4.6 Summary of the Advantages and Disadvantages of Logic Programming

In practice, the benefit of logic programming is that the specification of the problem *is* the program, so if it works it is correct. The disadvantage is that some programs which are declaratively correct will run forever. The programmer must therefore spend a lot of time considering procedural aspects to ensure the right solution, and the procedure must be thought about in logic with limited programming tools. Work is continuing to develop improved versions of Prolog which are truer to logic than current versions.

3.5 Complexity

3.5.1 Optimization Problems

Optimization problems fall into two categories: those with continuous variables, and those with discrete variables, which we call combinatorial. In continuous problems we look for a set of real numbers or even a function. In combinatorial problems we look for an object from a finite set, typically an integer, set, permutation, or graph. We shall present factory layout as a combinatorial optimization problem.

An *instance of an optimization problem* is a pair (F,c), where F is the domain of feasible points and c is the cost function, mapping a subset of F into a cost.

Optimization aims to find the best feasible solution, or the best path, one which minimizes the cost. The problem is to find an $f \in F$ for which

$$c(f) \leq c(y) \text{ for all } y \in F$$

Such a point f is called a globally optimal solution to the given instance.

An *optimization problem* is a set I of *instances of an optimization problem.* In an instance we are given the "input data" and have enough information to obtain the solution; a problem is a collection of instances, usually all generated in similar way. An instance of the QAP has a given cost matrix; but we speak in general of the QAP as the collection of all instances associated with all cost matrices.

3.5.2 Recognition Problems and Certificates

A recognition problem is a question which may be answered by "yes" or "no". For example, "Is there a path at all?".

We say that a problem has a *certificate*, if it is possible to exhibit that a solution to a recognition problem exists and can be verified in polynomial time. For example, although it may require an exponential amount of time to find a truth assignment that satisfies a sentence in the propositional calculus, it is easy to certify that the sentence is satisfiable, by simply exhibiting an assignment that satisfies it.

3.5.3 The Class of P

For a problem to be in the class P it must admit a polynomial time algorithm. That is, the size of the search space of solutions must grow as a polynomial function of the size of the input, for example, the number of departments in the QAP.

3.5.4 Polynomial Time Reductions and Polynomial Time Transformations

Let P_1 and P_2 be recognition problems. P_1 *reduces* in polynomial time to P_2 iff there exists a polynomial time algorithm a_1 for P_1 that uses several times as

a subroutine at *unit cost* a (hypothetical) algorithm a_2 for P_2. By *unit cost* we mean that the algorithm is considered as a single instruction taking unit time to execute. If P_1 polynomially reduces to P_2 and there is a polynomial time algorithm for P_2 then there is a polynomial time algorithm for P_1.

A problem P_1 polynomially *transforms* to another problem P_2, if given any string x, we can construct a string y within polynomial ($|x|$) time such that x is a *yes* instance of P_1 iff y is a *yes* instance of P_2. We may think of a polynomial time transformation as a polynomial time reduction with just one call to P_2. Transformations are usually used to cast one problem in the framework of another problem, so that if we can show that one problem has a particular property, e.g. solvable in polynomial time, we can prove that the second problem shares this property.

3.5.5 The Class of NP

For a problem to be in the class of NP, we require that if x is a *yes* instance of the problem, then there exists a concise (of length bounded by a polynomial in size of x) certificate for x, which can be checked in polynomial time for validity.

Notice that P is a subset of NP. Every efficiently solvable problem is also succinctly certifiable. NP is an interesting class of recognition problems because in addition to containing the problems in the class of P it contains many problems of interest whose membership in P is in doubt, (polynomial time procedures have not been found for these problems). The recognition versions of all reasonable combinatorial optimization problems are in NP.†

---

†Note that a recognition problem deos not "recognize" that a solution is optimal, It only recognizes that the constraints of the problem are satisfied and that therefore the answer to the recognition problem is yes.

Combinatorial optimization problems aim at the optimal design of objects such as tours, routes and sets of nodes. It is reasonable to expect that, once found, the optimal solution can be written down precisely and thus serve as a certificate for the recognition version of the problem. One cannot design optimally what cannot be found at all in reasonable time bounds. It is not known whether P is a proper subset of NP, or whether P and NP are the same. This is a prominant question facing complexity theorists.

3.5.6 NP-completeness

A problem $A \in NP$ is said to be NP complete if all other problems in NP polynomially transform to A. In order to prove NP completeness we must show two things:

(a) That the problem is in NP.
(b) That all other problems in NP polynomially transform to the problem.

3.5.7 The Nature of NP-complete Problems

NP problems seem inherently to require that we try out partial matchings, partial truth assignments, or partial arrangements, and continuously extend them in the hope of reaching a final solution. When a partial solution cannot be extended we apparently have to backtrack. If backtracking is carried out carefully no possible solution is missed but an exponential amount of time is required in the worst case.

3.5.8 The Satisfiability Problem - An Example of an NP-complete Problem

An example of an NP complete problem is the Satisfiability Problem:

> Given m clauses $C_1,...,C_m$ involving the Boolean variables $x_1,...,x_n$, is the formula $C_1 \wedge C_2 \wedge .... \wedge C_m$ satisfiable?

An appropriate certificate for this problem would be a truth assignment represented as a vector in $\{0,1\}^n$. A certificate checking algorithm would simply make sure that the C_i's are legitimate clauses involving n variables and that all clauses come out true under the truth assignment of the certificate.

3.5.9 P versus NP

Problems in NP which do not lie in P are often labelled intractable, because of their exponential growth rates. However, although exponential functions will eventually grow at higher rates than polynomial functions, they may behave better than polynomial functions within the bounds of a practical problem. N^{1000}, which is polynomial by definition, will produce much larger values than N!, an exponential function, for $N<1165$. The most efficient algorithm for Linear Programming, the Simplex algorithm, is an exponential algorithm, but in practice it is far more efficient than polynomial time algorithms for LP.

3.5.10 Alternatives to Solving NP-complete Problems

There are many approximate algorithms which do not produce optimal solutions, but do produce good solutions. There are some procedures such as Branch and Bound which find optimal solutions to NP complete problems of reasonable size. The choice of an appropriate algorithm depends upon the characteristics of the problem. Therefore we defer consideration of approximate algorithms until Chapter 6 when we shall have examined the characteristics of the Factory Design Problem.

CHAPTER 4

A LOGIC BASED APPROACH

Overview

Many computer models of factory design begin, not from a consideration of what is required to lay out a factory, but from a generally applicable Operations Research model, which is then contorted to try and represent the factory layout problem. These models have all omitted a critical factory in the layout problem such as ability to express departmment shape. Undoubtedly the large amount of data and the combinatorial nature of the relationships which represent the layout problem render it enormously complex. Yet this should not preclude an attempt to examine the complete layout problem and its associated complexity. It does prompt the question of whether the factory design problem, when fully specified, is solvable at all.

This chapter introduces the Generate and Test approach which allows us to represent the complete design problem. This object creation paradigm focuses on the unique structure featured by a specific symbolic representation of a given problem. It also introduces expert systems which allows us to structure knowledge about the design problem. An informal discussion of the representation issues for knowledge and the assumptions required for representation in a computer is presented. In the remainder of the chapter we present the industry specific rules and objectives for factory design gathered from a series of interviews with an automobile factory layout engineer and the general factory design rules gathered from literature. We examine the logical basis of both expert systems and Generate and Test . We discuss the motivation for developing a formal model of factory design in predicate calculus. The model is presented in chapter 5.

4.1 The Generate and Test Paradigm

The Generate and Test approach [Newell and Simon, 1972] is a problem solving technique involving a generator that produces possible solutions and

an evaluator that tests the acceptability of those solutions. This creation paradigm focuses upon the unique structure featured by a specific symbolic representation of the problem at hand. For example, in factory layout if we can express a department as a shape and we can express the condition that two shapes be adjacent by sharing a common face, then we can place a shape and generate adjacent locations for candidate neighboring shapes. Alternatively we can express a factory as a rectangle covered by a grid of say one foot spacing and place department shapes on the grid coordinates. There are many ways in which we can generate locations allowing us to express department shape which do not conform to a rigid model.

If we can express relationships between departments such as flow cost, based upon distance, then we can evaluate the configurations produced by the generator. Alternatively we can evaluate a solution in terms of its structure by assessing the number of desired adjacencies attained.

Good generators are complete and guided. A complete generator is guaranteed to produce all hypotheses which pertain to the problem as it is expressed to the generator. A necessary condition for completeness requires that the set of all possible solutions be enumerable and finite. If there are possible solutions which cannot be produced by the generator or if there are an infinite number of possibilities, then the generator is said to be incomplete. Since an incomplete generator may or may not produce a correct hypothesis, problems for which only incomplete generators exist are termed semi-decidable. The effectiveness and efficiency of candidate generation may be improved through guidance. The simplest form of guidance restricts candidate generation so that the same hypothesis is never proposed more than once. Generators with this property are said to be non-redundant. If all possible factory configurations are produced by a non-redundant generator for comparison by the tester then this amounts to exhaustive search. We can also guide the generator by evaluating partial solutions to decide whether to continue along those solution paths. In this respect we might employ search techniques similar to the branch and bound approach.

The Generate and Test approach allows us to generate an appropriate search space as we proceed, rather than beginning with a fixed grid or a bounded shape.

It is interesting to note the difference of emphasis between the AI and the OR communities. Whereas the idea of viewing problem solving as a process of repetitive splitting and pruning (branching or refinement) of subsets of potential solutions became popular in Operations Research, (this is the basic metaphor behind the Branch and Bound method), AI literature rarely mentions this metaphor, preferring to describe problem solving as a Generate and Test process. The split and prune paradigm emphasizes the commonality of problem solving methods over many problems, whereas the object creation paradigm focuses on the unique structure featured by a specific symbolic representation of a given problem.

The operations research approach is particularly effective in establishing theoretical guarantees such as completeness and optimality, but for simple models. The artificial intelligence community has, on the other hand, always emphasized the heuristic and implementational aspects of problem solving and its parity with human style over its mathematical foundations.

4.2 Representing the Generate and Test Paradigm in Logic

Using a logical representation of a problem and logic programming techniques, we can exploit both the expressiveness and the mathematical properties of logic. A logic representation gives us the properties of a sound and complete generator. But, of course, completeness is referent to the generator. Suppose that we have only two constraints:

1. No departments may overlap.
2. All departments must lie within a given boundary.

If we define in logic a "procedural" generator which chooses the most promising location for a department and then places the department in that location allowing no recourse, then our complete generator provides one solution if the constraints are met and none if they are not.

It is desirable to use the property of completeness to ensure that all possible geometric configurations can be generated, but to guide the generator so that the better solutions are produced first.

In layout the role of the complete configuration generator is to express all possible geometric configurations of the shapes. Knowledge about the shapes which we are configuring, for example which departments they represent, and the characteristics of those departments helps us to guide the generator in generating the geometric configurations which represent good factory layouts. We may draw a rough correspondence between the syntactics and the semantics of logic, and the geometry and the description of the geometry.

We consider how we can combine the knowledge of the factory designer with the Generate and Test paradigm in order to generate for evaluation designs which are promising. We shall do this using *knowledge based system* techniques.

4.3 Knowledge Based Systems and their Logical Foundations

A knowledge based system, or expert system, is a program that behaves like an expert in some, usually narrow domain of application. Applications have included MYCIN for medical diagnosis and R1 for computer configuration. Both are described in [Jackson, 1986]. In order to behave like an expert, expert systems must posess knowledge of some form. They must also be capable of explaining their behavior. This is especially necessary in uncertain domains such as design, in order to enhance the user's confidence in the system and to enable the user to detect a possible flaw in the systems

reasoning which is incompatible with the user's assumptions. The explanation system enhances the standard user interaction capability. Expert systems often require the ability to deal with uncertainty. For example, in diagnosis, "car wont start implies dead battery" may lead to many a wrong conclusion. The car may not start for many other reasons. Therefore we combine the observation of not starting with other observations, such as the noise made by the car, in order to increase our confidence in the diagnosis before we buy a new battery. We also associate a certainty factor with each observation. For example, "car wont start implies an 80% chance of dead battery". In other applications all the information may be certain. However we still need to combine information. For example in factory design we have the following declarative knowledge about the objects in the factory

difficult_to_move(heavy_machineA)
needs_to_expand(departmentB)

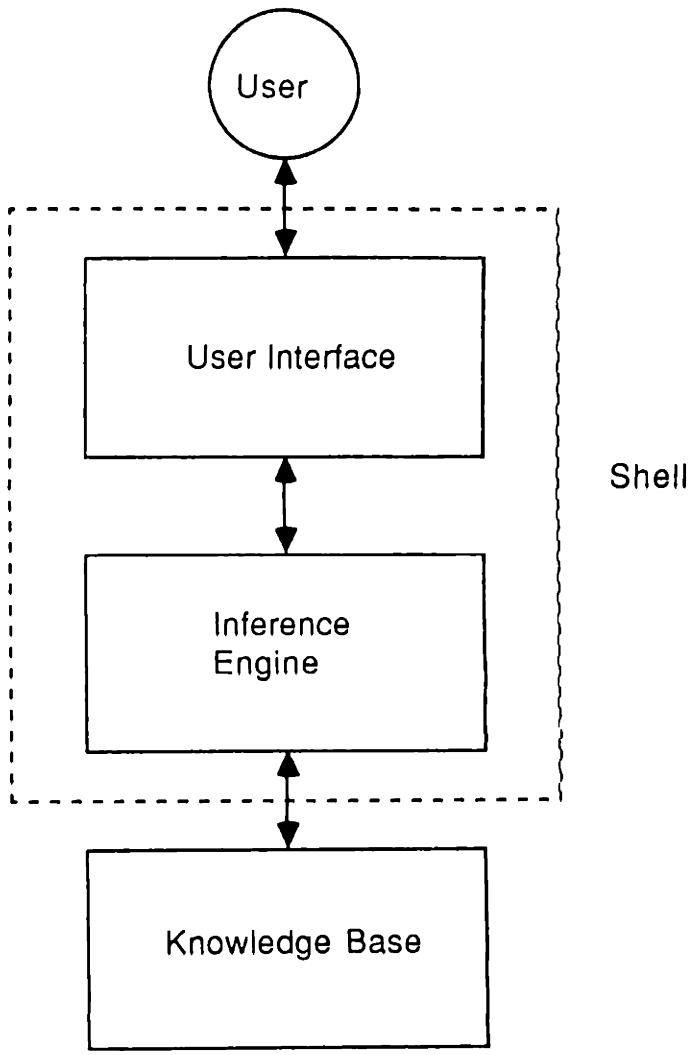
We also have the following procedural rule which tells us not to place anything which is difficult to move and anything which needs to expand adjacent

difficult_to_move(x) $\wedge$ needs_to_expand(y) $\rightarrow$ ~adjacent (x,y)

We use logical inference procedures to determine that department B should not be placed adjacent to heavy machineB. Figure 4.1 shows the structure of a knowledge based system. The function of the inference engine is to invoke these procedures. Logic studies the relationship of implication between assumptions and conclusions. It was originally devised as a way of representing the form of arguments, so that it would be possible to check in a formal way whether or not they were valid. We can use logic to express propositions, the relation between propositions and how one can validly infer some propositions from others. So the reasoning techniques of factory designers may be expressed precisely in symbolic logic. Knowledge Based Systems are generally programmed in symbolic or logic programming

Figure 4.1

A Simple Representation of a Knowledge Based System



languages such as LISP, Prolog and OPS5.

4.4 A Knowledge Based Approach to Factory Design

In the knowledge based approach to factory design we express the knowledge of an expert in terms of IF-THEN rules which form the basis of a deduction system. It is very important that the an IF(premise) -THEN(consequence) rule links a consequence to the correct premise. Although experts have a vast store of knowledge, they are often not conscious of why they choose to do things in the way that they do. The role of the knowledge engineer is to extract the correct deduction mechanism from the expert. For example in automobile factory design, the rule

IF automobile factory , THEN assembly line layout,

might seem reasonable. After all most automobile assembly plants have sequential assembly line type layouts. The pertinent question to ask is "Why?". Traditionally automobile factories have aimed for high volume production at the expense of low product variety. If we wished to design a new automobile factory producing a large variety of automobiles at a smaller production rate, this rule might be completely inappropriate. In order to take advantage of expert knowledge in new designs rather than allow old practices to hinder new designs we must ensure that IF-THEN rules make correct associations and for the right reasons. In building a knowledge base for factory design it is very important to associate rules with the design objectives rather than with categories of layout. Categories of layout and their associated objectives are described further in Appendix 2. Design Objectives are listed in figure 4.2. Rules associated with these design objectives are listed in figures 4.3 - 4.7. Automobile industry specific rules are listed in figures 4.8 - 4.10 . Many of the automobile industry rules are special cases of the general rules. When this is the case the general rule is indicated in parentheses. We may question whether objectives such as orderly flow or safety can be characterized for input to a computer. Althougth symbolic

languages facilitate symbolic expression, which helps us in representing qualitative objectives, there must be an explicit metric by which we can compare safety of layouts if we are to represent a safety objective for computation. For example, we may consider narrow aisles or a machine with no safety guards, or layouts in which personnel share aisles with handling equipment to be unsafe features of a layout. We could compare safety of layouts, by counting the number of features associated with safe and unsafe criteria. If different features have different "safety values" we may wish to take this into account.

4.5 The Logic Based Approach and the Purpose of a Formal Model

We may express both the Generate and Test paradigm, for generating geometric configurations, and the knowledge for representing the department relationships and constraints in logic. We can also express preliminary process planning and machine selection in logic. This is demonstrated in Chapter 5. Developing a formal model in logic enables us to unify the components of the factory design problem. It enables us to formalize the relationships between the phases of preliminary process planning, machine selection and layout. It enables us to decouple the general structure of the problem from semantic interpretations of the logical models. We determine which knowledge adds to the syntactic structure of the problem, and which knowledge simply represents parameters of the problem. This helps us to understand the extent of the intractibility of the problem in a specific domain. We determine whether there is any formal difference between domain specific and domain independent knowledge.

Figure 4.2

Factory Design Objectives

FLOW

Minimize material flow

Minimize personnel flow

Attain orderly flow

Minimize handling

FLEXIBILITY

Maximize expansion flexibility

Maximize routing flexibility

Maximize machine flexibility

Maximize operation flexibility

UTILIZATION

Maximize machine utilization

Minimize inventory

Optimize space

CONSTRAINTS

Isolate undesirable effects

Accommodate existing immovables

Figure 4.3

General Rules Relating to "Flow" Objectives

Minimize material flow (volume,weight,frequency)

- receiving near to start of process
- final assembly near to shipping
- locate departments with high interaction near to each other etc.
- move heavy material the minimum distance
- locate departments with many interactions centrally
- deliver material to point of use
- combine processing with handling
- minimize movement of fragile parts

Minimize personnel flow

- minimize walking distance by operators
- minimize walking distance by supervisors
  (choose square departments)

Attain orderly flow

- minimize backtracking, ie duplicate departments or alter process plan (if using a fixed handling system e.g. conveyors)
- modular layout
- straight aisles

Minimize handling

- combine operations on same machine to minimize load-unload (this also relates to machine selection and process design)
- use gravity to move material where practicable
- minimize ratio of mobile equipment dead weight to pay load
- make sure ramps are at a low enough grade for handling equipment

Figure 4.4

General Rules Relating to "Flexibility" Objectives

Maximize expansion flexibility

- locate departments likely to expand in a good position for expansion.
- preplan for expansion in at least two directions
- design flow pattern for logical extension
- consider upward expansion
- locate "permanent equipment" with special foundations and services in suitable permanent locations
- decouple department designs to provide modularity and flexibility
- minimize partitioning - use movable ones
- support roof with columns, not walls
- plan adequate height for roof installations

Maximize routing flexibility

- modular layout
- flexible handling equipment (eg AGVs)
- standard handling fixtures - pallets
- use uniform service ducts with multiple connection points

Maximize machine flexibility (task flexibility)

- choose equipment that can perform a variety of tasks and applications

Maximize operation flexibility (task implementation flexibility)

- choose equipment that can be programmed to perform task on varieties of shapes, materials etc.

Figure 4.5

Rules Relating to "Utilization" Objectives

Maximize machine utilization

- reduce idle time of equipment,handling equipment, manpower

Minimize inventory

- standardize tooling, fixtures, raw materials

Optimize space

- optimize aisle width (wide = costly) (narrow = unsafe)
- optimize department space
- make full use of building cube
- space columns according to predicted dept size, handling equipment

Figure 4.6

Rules Relating to Constraints

Isolate undesirable effects

- noisy equipment; isolate through distance, soundproofing, special mountings
- inflammable materials; require ventilation, fire protection

Accomodate existing departments

- design around currently placed equipment which cannot be moved

Figure 4.7

Rules Relating to Work Cell Planning

Minimize flow

- integrate workplace handling system and flow into overall system
- deliver material so operator is not required to prepare or reposition it
- plan for tools, gauges, materials close to machine and operator

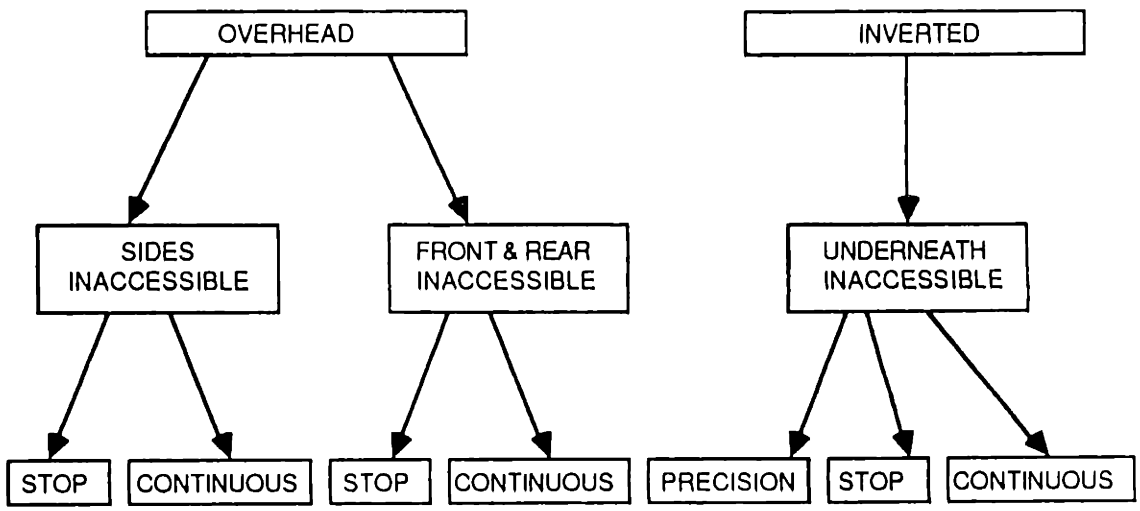
Maximize utilization

- plan for machine travel, material insertion and removal, operation movement, maintenance access
- minimise material in work area
- avoid placing material directly on floor
- arrange machinery so operator can attend more than one

Figure 4.8

Conveyor Types and Conveyor Selection Rules

Conveyor Types



Examples of Conveyor Selection Rules

1. If the front and rear of the car must be accessed, do not use an overhead conveyor which attaches to the lower front and rear of the car body.

2. If the sides of the car must be accessed, do not use an overhead conveyor which attaches to the lower sides of the car body.

3. If the car bodies must stop independently use a conveyor which can be decoupl ed from its carriers.

4. If the car body must be submerged use an overhead conveyor.

5. If the underneath must be accessed use an overhead conveyor.

6. Use precision conveyors for robot automation.

7. If the front, rear and sides must be accessed, (e.g. for painting), use an inverted conveyor.

Figure 4.9

Automobile Assembly Plant Conveyor Placement Heuristics

- center conveyors twelve feet from one set of pillars
  (ie leave just enough room for people and machinery on one side, so that there is as much room as possible before the next set of pillars)

- allow sufficient radius for conveyor turns (twelve feet)

- do not include conveyor turns as working length of the conveyor

- if the conveyor is inverted and it must be raised for an eight foot aisle, eighty feet of conveyor is required; this cannot be considered working length

- make conveyor lengths small and loop frequently so that aisle access is not restricted

- minimize conveyor length (general rule: minimize material flow)

- minimize conveyor changes (general rule: minimize variety of material handling equipment)

- leave space at the end of conveyor turns so that they can be extended if necessary (general rule: routing flexibility)

- flow lines must avoid pillars

- design layouts which can accommodate new conveyors

- do not block access to railroad tracks with conveyors; that is place departments using metal nearest to railroad tracks (general rule : minimize flow)

Figure 4.10

Automobile Assembly Plant Design Rules

Space utilization

- leave space for material storage and for expansion

- mount large tanks of liquid on ground floor (esp if acid etc) with catchment tank underneath.

Flexibility

- make layout changes specific to current model year flexible (this is facilitated by an original flexible design)

- standardize machine mechanisms, for example indexing mechanisms, for ease of maintenance (applicable to industries with a lot of specialized equipment)

Temporal rules

- if machinery cannot be installed in the 4 week changeover period install in existing unused space (better to have a modular layout with routing and expansion flexibility if possible)

Constraints

- place extraction booths around silicon bronze grinding (general rule : isolate undesirable effects)

- place extraction booths around paint spraying (general rule : isolate undesirable effects)

- avoid moving extraction booths, primer tank, overhead welder water tanks (general rule : design around existing constraints),(general rule: expansion flexibility - locate "permanent equipment" in "permanent locations")

CHAPTER 5
A MODEL FOR FACTORY DESIGN

Overview

Chapter 1 introduced the three phases of factory design; preliminary process planning, machine selection and layout.

Chapter 2 discussed existing models of factory design, most of which considered only the layout phase. It was found that each model made a constraining assumption which reduced its expressive power for accurately describing the problem at hand. Most notably, department shapes could not be expressed in the iterative improvement and graph theoretic approaches. Department shapes could be expressed in construction algorithms, but these suffered from being deterministic and offering only one solution. This provokes the question, is the factory layout problem expressed in realistic terms computable? A more general question is, is the factory design problem computable?

In this chapter we address these questions by developing a formal model of each phase of factory design. We show that each phase of factory design can be expressed as a model of the first order predicate calculus. The first order models are restricted enough in their expressiveness that they can be represented by a finite conjunction of sentences in the propositional calculus. We can therefore prove that the factory design problem is decidable using the results of the validity proofs of section 3.3.5. We show how the formal model which describes all factory designs can be extended to evaluate those designs.

The model reveals some interesting dependencies and independencies between the phases. We prove that knowledge about the functional requirements of the product itself cannot help the factory designer to solve the layout problem. Only the characteristics of the products and machines are relevant. This has unfortunate implications for reducing the complexity of the layout problem, but beneficial implications for the generality of a software model for factory design. A large body of industry independent knowledge

can be expressed in terms of product characteristics such as weight and fragility, and machine characteristics such as size, noise generating and fume producing, rendering such a model easily transferable from industry to industry and firm to firm.

We will show how an interpretation of the layout model forms a basis for a complete process planning system. Once we have a layout of machines, and knowing the capabilities of the machines, we can conclude whether or not a part can be made in the factory, and which machines could be used to manufacture it.

We conclude with a discussion of domain dependence and independence, evaluating whether or not we can define a meaningful distinction between the two, or whether they just overlap in some murky continuum. We discuss Godel's Incompleteness Theorem which implies that there are limits to the computational power of domain independence which preclude the design of an all-encompassing domain independent expert system. An increase in expressive power is not concomitant with an increase in solution power.

5.1 A Formal Model for the Three Phases of Factory Design

5.1.1 A Uniform Representation

Each phase of factory design can be expressed as a model in predicate calculus. The domain of each model represents the set from which choices are made at each stage. We represent distinct members of each domain by a domain alphabet. When the domain would otherwise contain several members which represent a different manifestation of the same underlying structure, we use a domain descriptor alphabet to describe members represented by the domain alphabet. For example if a part is represented by a geometry, and it comprises three different materials, we use a domain alphabet to represent the geometry, and a material alphabet to represent the material description of each element of the geometry. We also introduce a special alphabet which consists of two symbols, a connector and a disconnector. These are used to represent connectivity or separation of

members of the domain.

The alphabets of the formal model are summarized below.

Domain alphabet

$\Sigma = \{g_1, g_2, g_3, \ldots\ldots\ldots\ldots\ldots\ldots g_n\}$

Special alphabet

$S = \{\bullet, \#\}$.

• represents connectivity of domain elements

represents separation of domain elements.

Domain descriptor alphabets

$D = \{d_1, d_2, d_3, \ldots\ldots\ldots\ldots\ldots d_n\}$.

Table 5.1 summarizes the alphabets used in each phase of the formal model. The preliminary planner requires a complex domain representation, using the descriptor and special alphabets. The domain consists of geometry elements described by material and process alphabets. The selection and layout phases require only a simple domain representation. Their domains consist of a database of machines and a set of grid coordinates for assigning those machines respectively. We shall consider in more detail why we do not need complex representations for these domains as we develop the phase models.

5.2 A Formal Model for Preliminary Process Planning

The preliminary process planner selects tentative process plans for the products to be made in the factory. The inputs are the part attributes, which are found on the part drawing. These include the part geometry, materials which make up the geometry and the process operations which produce physical properties, such as hardness.

Table 5.1

Domain Descriptions for Logic Models

Formal Structures		Phase		
Symbol	**Concept**	Preliminary plan	Equipment selection	Factory layout
Σ	Domain alphabet	Raw material grid	Machine database	Factory grid
	Special alphabet			
•	Connector	Material connectivity	Machines belong to same machine set	Space connectivity in factory
#	Disconnector	Discrete parts	Machines belong in different factories	Discrete factories
D	Domain Descriptor Alphabets	Material $\mathbf{D}_m$		
		Internal transform-ations $\mathbf{D}_p$		

5.2.1 Problem representation

In a CAD model, coordinates and equations represent geometric surfaces. Non geometric parameters, such as surface roughness, are assigned to geometric surfaces. At the simplest level of representation, a part may be considered to consist of a set of square elements, with an edge length of the minimum tolerance of any machine in the system (see figure 5.1). We consider operations on individual elements. If we use a hierarchical representation, which allows a layer of elements to be considered as a 'pass', and a set of passes to be considered as an operation, the complexity of our representation increases, but the complexity (in terms of run time) of the computer program decreases. By using heuristics, a CAD system or a Computer Aided Process Planning system can group elements into blocks of elements in order to represent the model more powerfully for the purpose of computation. For a lucid description for the purpose of proof , we choose the simplest model.

In the preliminary process planner, the domain alphabet represents the geometry elements of figure 5.1 described above.

$$\Sigma_{PLANNER} = \{g_1, g_2, g_3, \ldots\ldots\ldots\ldots\ldots\ldots g_n\}$$

In the special alphabet S = {•,#},

- • represents connectivity of material,
- # represents disjunction of material.

Two descriptor alphabets are use to describe the geometry:

1. A material descriptor alphabet,

$$D_m = \{\Delta, m_1, m_2, m_3, \ldots\ldots\ldots\ldots\ldots\ldots m_n\};$$

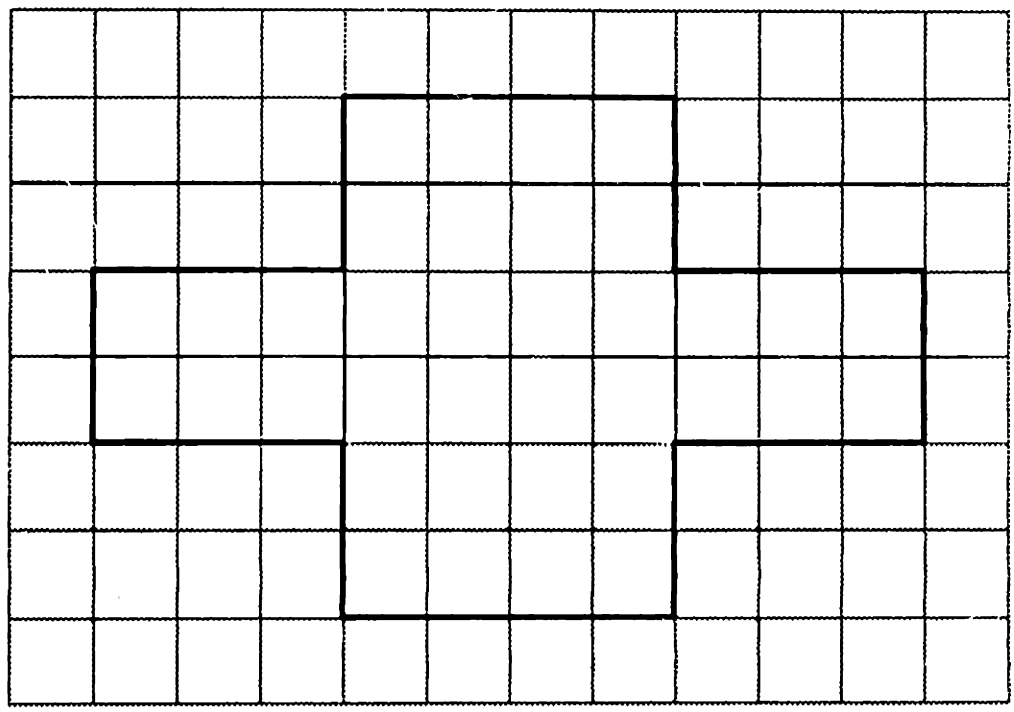
Each member of this finite alphabet to represents a different material. The Δ symbol represents absence of material.

2. A property descriptor alphabet,

Figure 5.1

A part shown in its grid representation

Each element in the grid represents the minimum tolerance capability of any machine in the factory. The grid is shown enlarged for the purpose of illustration.



$$D_p = \{p_1, p_2, p_3 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots p_n\};$$

Each member of this finite alphabet signifies that a process causing internal change, such as heat treatment, has been performed. The processes which alter properties are noted on the part drawing. They may be considered as input to the planning problem.

A part is represented by a word over all the alphabets. Figure 5.2 shows a raw material block and a part machined from the raw material. The raw material geometry is represented by an input word of the form:

$$(\#m_1g_1 \bullet m_1g_2 \bullet m_1g_3 \bullet m_1g_4 \bullet m_1g_5 \bullet m_1g_6\ \#).$$

Likewise, the final part is represented by an output word:

$$(\#m_1g_1 \bullet \Delta g_2 \bullet m_1g_3 \bullet m_1g_4 \bullet m_1g_5 \bullet m_1g_6\ \#).$$

Letters from the descriptor alphabets precede the geometry letter representing the element of material which they describe. The entire description of a part is contained between two disconnector symbols. We define a *connected gemetry set*, C, to be the set of all geometry letters between two consecutive disconnector symbols.

We need to define a set of rewrite rules representing operations which transform the raw material input word into the finished part output word.

5.2.2 Operation Classes - Descriptions

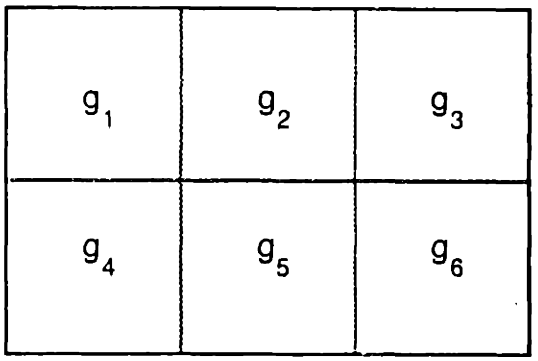
At the highest level, operations may be described by the following functions which act on the descriptors of the geometry alphabet:

Material removal R: $pmg \rightarrow \Delta g$ (1)

Figure 5.2

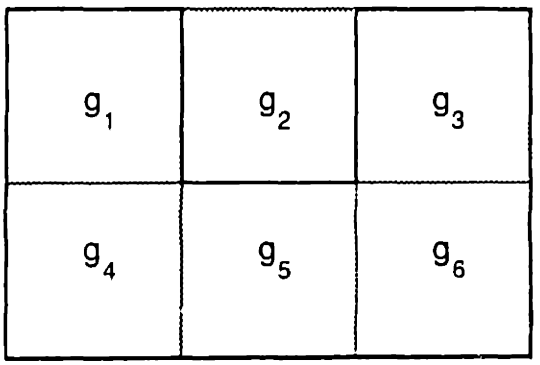
Word Descriptions of a Piece of Raw Material and a Finished Part

(a) raw material



$$\#m_1g_1 \cdot m_1g_2 \cdot m_1g_3 \cdot m_1g_4 \cdot m_1g_5 \cdot m_1g_6 \#$$

(b) finished part



$$\#m_1g_1 \cdot \Delta g_2 \cdot m_1g_3 \cdot m_1g_4 \cdot m_1g_5 \cdot m_1g_6 \#$$

Material addition	$A: \Delta g \rightarrow mg$	(2)
Internal change	$I : mg \rightarrow pmg$	(3)
Assembly	$A: g_1 \# g_2 \rightarrow g_1 \bullet g_2$	(4)
Disassembly	$D: g_1 \bullet g_2 \rightarrow g_1 \# g_2$	(5)

where $g \in \Sigma$, $m, \Delta \in D_m$, and $p \in D_p$. A distinction is drawn between disassembly and material removal, and assembly and material addition. In removing material from the part, removal operations create material which has no distinct structure, such as swarf. We need not represent the removed material. Disassembly operations, such as unbolting, generate more than one part of distinct structure which must be modelled. Some operations, such as the cutting of a bar must be represented by a combination of removal and disassembly operations, since two distinct pieces of bar are created, and also swarf is generated in the cutting process. We distinguish between material addition and assembly for similar reasons. Material addition represents operations such as painting or plating, where material of no distinct structure is added to a part. Assembly represents operations such as bolting, where two or more parts are joined together. Geometric change operations such as forging can be represented by a combination of the material addition, removal and internal change operations.

If there exists a sequence of rewrite rules transforming the input word into the output word then this represents a process plan for the part. To transform the the raw material of figure 5.2(a) into the part of figure 5.2(b), the following rewrite rule may be applied:

$$R: m_1 g_2 \rightarrow \Delta g_2$$

This system of words and rewrite rules is a special case of a semi-Thue system described in Section 3.1.4.. The rewrite rules apply only to the

descriptor alphabets and the special alphabet.

The chosen representation language consisting of a geometry alphabet, special alphabets and descriptor alphabets is not as simple as it might be. We could use a representation with no geometry alphabet, the element of the geometry being implied by the position of its material descriptor in the part word. Similarly we could omit the connector symbol, and search for the nth material descriptor symbol to find the nth piece of the geometry. The part and operation representation could be made complete by using only a binary alphabet {a,b}. However, this would require long strings of symbols in both the word and the rewrite rules.

The connector and geometry symbols are used for clarity and for brevity. They are not necessarily superfluous. There must be a means for referencing distinct pieces of the geometry. In a binary alphabet this could be achieved only by using long strings of letters. The geometry alphabet is included to this end. The connector • symbolizes the connectivity of each element of the material. Yet it recognizes each element as independent in terms of the operations it can undergo, because each element is defined in terms of the minimum tolerance. The disjunction # symbolizes the independence of two disjoint parts, before say an assembly operation.

5.2.3 Feasible Operations

The operation classes of section 5.3.2 are, by design, very general. There may be several constraints on a particular geometry element which prevent practical completion of such operations. We therefore must refine the operation description by representing these constraints. We introduce *feasible operations* which recognize constraints such as accessibility of material.

To determine accessibility of a piece of the geometry in terms of other pieces of the geometry, a neighbor relation is defined:

Neighbor $N: g_i \rightarrow g_j$ where $g_j \in B_i$, the set of neighbors of g_i (6)

e.g. $B_1 = \{g_2, g_3, g_4, g_5, g_6, g_7\}$,
$N(g_1) = g_2$

This relation determines all the elements adjacent to the prescribed element. It is illustrated in figure 5.3. To determine the description of a geometry element two functions are defined:

Material $M: g \rightarrow m$ (7)

Property $P: g \rightarrow p$ (8)

The material function determines which material descriptor precedes a given geometry element. Similarly, the property function determines which internal change operations have been applied to the geometry element. Constraints on the general operations which, if satisfied, produce feasible operations are described below. A feasible material removal operation is subject to constraints on the accessibility of the element:

Removal $\forall g \{ R(g) \leftarrow \tilde{}(M(g) = \Delta) \wedge \exists(N(g)) \{M(N(g)) = \Delta\} \}$ (9)

That is, in order to remove element g, it must not be empty already, and it must have at least one neighbor which is empty. Note that elements on the grid boundary have empty edges which are not defined by elements. We could also express this special boundary case in logic.

A furnace heat treatment operation can not be applied to one element of the part without being applied to the other elements in the part:

Heat treatment $\forall(g \in C) \{ I(g) \leftarrow \forall(g' \in C)\{ I(g') \} \}$ (10)

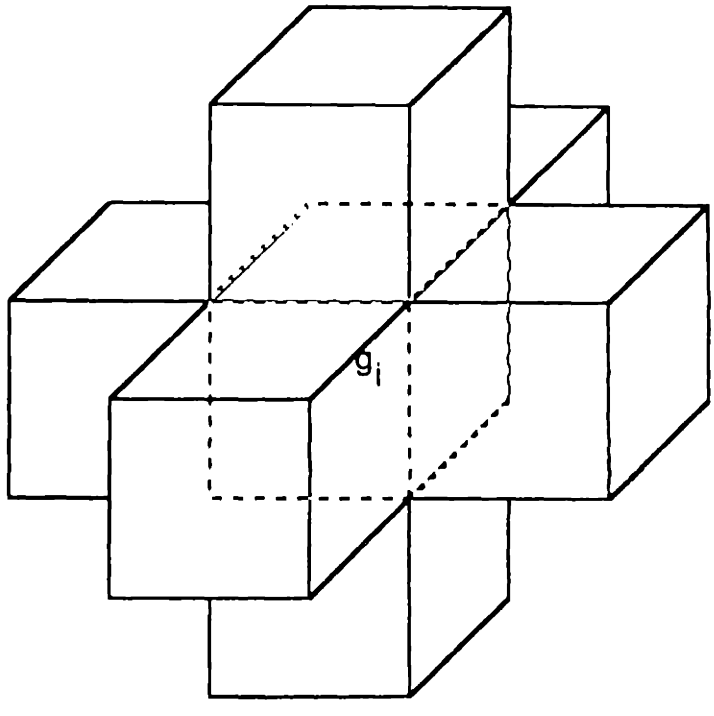
That is, if one element of a connected geometry set, C, undergoes heat treatment, all elements must undergo heat treatment.

Figure 5.3

The Neighbor Relation

Illustration of the neighboring elements to the central element, generated by the neighbor relation.

$$N : g_i \rightarrow g_j$$



The following sentence identifies that two parts are not connected:

Disassembly

$$\forall g_1 \forall g_2 \forall C_1 \forall C_2 \{D(g_1,g_2) \leftarrow (g_1 \in C_1) \wedge (g_2 \in C_2) \wedge^{\sim} (C_1 = C_2)\} \qquad (11)$$

We read the disconnector like a cartesian operator.
(i.e. $(g_1 \cdot g_2 \cdot g_3)\#(g_4 \cdot g_5 \cdot g_6)$ implies $g_1\#g_4$, $g_1\#g_5$, $g_1\#g_6$, $g_2\#g_4$....$g_3\#g_6$).

The above discussion illustrated how general rules for process planning may be written in logic. The effect of heuristics in process planning expert systems is to augment sentences defining feasible operations, with sentences defining good procedures. Figure 5.4 illustrates levels describing operations which are still more detailed than feasible operations and which may have a large amount of industry and product specific information.

5.2.4 A Logic Model for Preliminary Process Planning

A preliminary process plan is represented by a list of pairs of states and feasible operations which transform the raw material into the finished part:

[(input,turning),(stateA, grinding),.....,(final,polishing)].

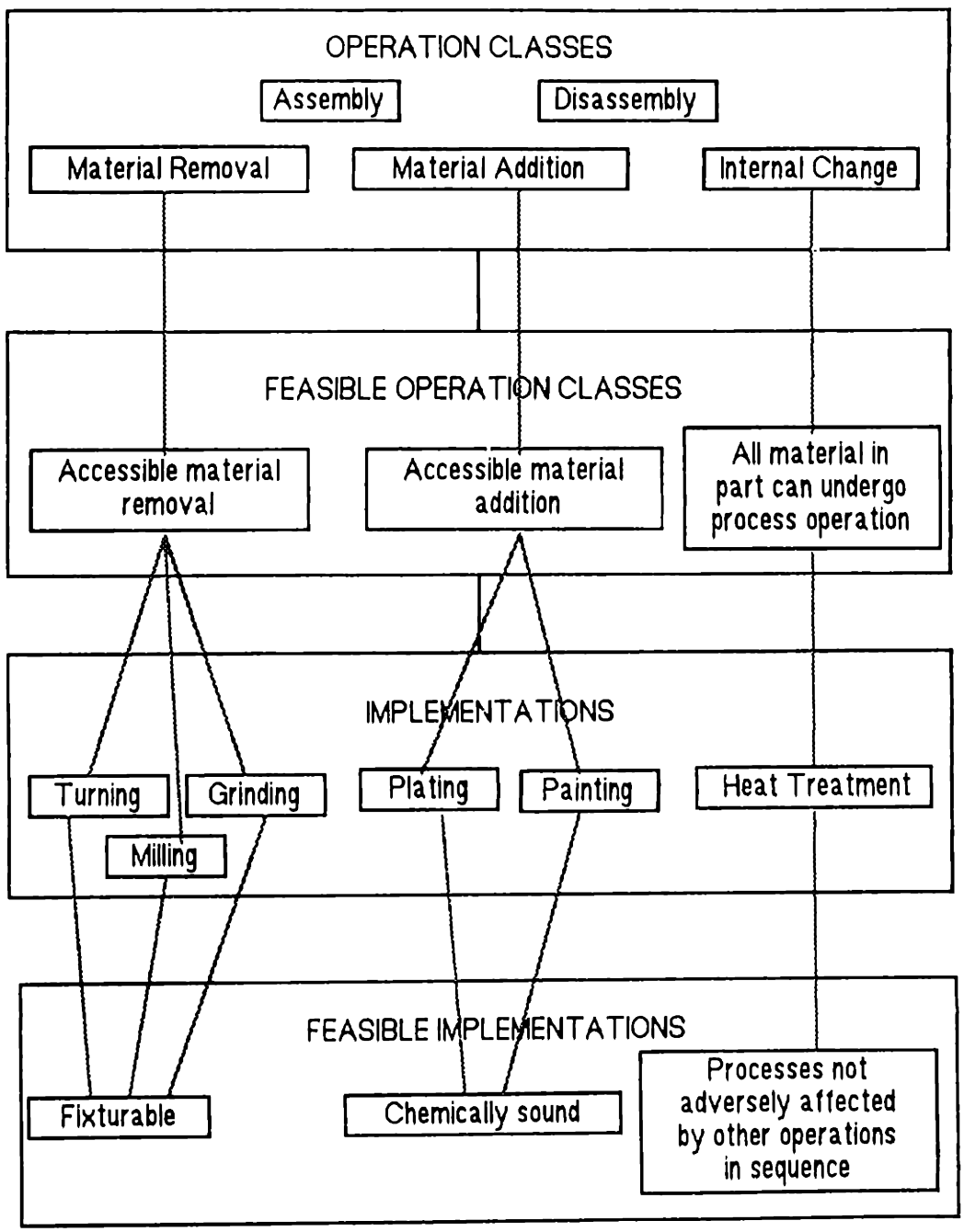
In the previous section we showed how feasible operations and states may be represented. In developing the formal model we assume the existence of a predicate "canDo" which determines whether or not there is a feasible operation from a given input state to a given output state.

```
canDo(Current,NextState) ←                                    (12)
                    removal(Current,NextState) ∨
                    addition(Current,NextState) ∨
                ... heatTreat(Current,NextState).
```

The feasible operations are a function of the states. To be precise operations

Figure 5.4

Hierarchy of Operations



are predicates which map a pair of states into the values *true* or *false*.

We define in logic a model, which, given an input state of the raw material and the output state of the finished product, develops a preliminary process plan. The domain of the model is the set of states. First we present a model which simply determines the existence of a plan. The model is defined in equation (13). In the models we adopt the Prolog convention of beginning variable names with uppercase letters and constant names with lower case letters. The initial, final and intermediate states are represented by S_i, S_f and S_n respectively. The model is quantified over all states.

$$\forall S_i \forall S_f \forall S_n$$
$$\{ \text{planExists}(S_i,S_f) \leftarrow S_i=S_f \vee \text{canDo}(S_i,S_n) \wedge \text{planExists}(S_n,S_f)\} \quad (13)$$

This statement assigns appropriate predicate constants to the following wff:

$$\forall x \forall y \forall z \; p_1(x,y) \leftarrow x = y \vee p_2(x,z) \wedge p_1(z,y) \quad (14)$$

The model determines whether or not a plan exists from the initial (raw material) state to the final state. The model consists of a boundary condition and a recursive structure. In words the model behaves as follows : "There is a plan from the initial state to the final state, iff *either* the initial state and the final state are equal, *or* if we can do an operation from the initial state to an intermediate state, and there is a plan from the intermediate state to the final state". "planExists" is a predicate. By definition it returns only the value true or false. It does not make explicit the assignment of states which it used to determine truth or flasehood. We wish to make these states explicit. We also wish to make explicit the operation by which canDo transforms each state. Therefore "canDo" is redefined as follows:

$$\text{canDo}(\text{Current},\text{NextState},\text{Op}) \leftarrow \quad (15)$$
$$(\text{removalOp}(\text{Current},\text{NextState}) \wedge \text{Op} = \text{removal}) \vee$$
$$(\text{additionOp}(\text{Current},\text{NextState}) \wedge \text{Op} = \text{addition}) \vee$$

$$\ldots(\text{ heatTreatOp(Current,NextState)} \wedge \text{Op} = \text{heatTreat }).$$

We redefine "planExists" as a function, "plan", which maps the input and output states into a sequence of states and operations, rather than a predicate which maps them into the values true or false:

$$\text{plan: } (S_i,S_f) \rightarrow [(S_i,Op),(S_2,Op_2),\ldots\ldots\ldots\ldots|(S_f,\text{stop})]^{\dagger}$$

The function plan is defined by the following logic model:

$$\begin{aligned} \forall S_i \forall S_n \forall S_f \forall Op \ \{ \quad & \text{plan } (S_i,S_f) \\ \leftarrow \quad & S_i=S_f \wedge \text{ plan } (S_i,S_f) = (S_f,\text{stop}) \\ & \vee \text{ canDo}(S_i,S_n, Op) \wedge \\ & \text{plan}(S_i,S_f) = [\ (Si,Op)|\ \text{plan } (S_n,S_f)] \end{aligned} \qquad (16)$$

This model is only satisfied by sequences of state and operation pair assignments which all satisfy the canDo predicate. We can define the "plan" function as a predicate "planPred", which has the function "plan" as one of its arguments. It returns the value true when the function "plan" returns a plan, and false when it does not. :

$$\text{planPred: } (S_i,S_f, \text{plan}(Si,S_f)) \rightarrow \text{true or false}$$

The predicate "planPred" is defined in the following logic model:

---

†The sequence of states and operations is written as a list of the form [head | tail]. This notation allows the list $[(S_i,Op),[(S_2,Op_2),[(S_f,\text{stop})]]]$, to be represented more neatly as the head and tail of the list,

$[(S_i,Op_1), (S_2,Op_2) \mid (S_f,\text{stop})]$.

$$\forall S_i \forall S_n \forall S_f \; \forall Op \quad planPred(S_i, S_f, plan(S_i, S_f)) \qquad (17)$$
$$\leftarrow \quad S_i = S_f \wedge \; plan(S_i, S_f) = (S_f, stop)$$
$$\vee \; canDo(S_i, S_n, Op) \wedge$$
$$plan(S_i, S_f) = [(Si, Op) \mid plan(S_n, S_f)] \wedge$$
$$planPred(Sn, Sf, plan(Sn, Sf))$$

We can use the predicate definition as an axiom in a refutation theorem prover which derives (proves) or validates plans. To validate a plan we present the clause :

$$\sim planPred(s_i, s_f, [(s_i, op), (s_2, op_2), \ldots\ldots, (s_f, stop)]).$$

If this is a valid plan, the empty clause is derived. To derive or prove plans from the database of states and operations we present the clause:

$$\sim planPred(s_i, s_f, plan(s_i, s_f)).$$

All sequences of states and operation assignments which represent valid plans will derive the empty clause.

The Prolog coding of the model of equation (17) is given below.

```
doPlan (Current, Final, (Final,stop)):- Current=Final.          (18)
doPlan (Current, Final, [(Current,Op)|RestOfPlan] ):-
            canDo (Current,NextState,Op),
            doPlan (NextState,Final, RestOfPlan).
```

If canDo cannot find an operation from the current state, then there cannot be a plan which succeeds through the current state, so the current variable assignment fails. In theorem proving a new variable assignment from the domain is tried. In a programming implementation backtracking occurs to a previous state and the search resumes.

Figure 5.5 illustrates a sample interaction of "doPlan" for an example database of feasible operations. Figure 5.6 shows how the directed graph representing possible plans of the figure 5.5 database may be represented as a tree. Figure 5.7 illustrates the order in which the tree is searched in depth first backtracking search, the search procedure employed by Prolog. Using a recursive structure, with backtracking, we may query the database until we find all possible plans.

If the database of figure 5.5 is modified to include operations which return the material to a previous state, such as

canDo(a,b,plate)
canDo(b,a,mill)

there are cases where the generative model will not halt. Figures 5.8 and 5.9 show how by repeatedly plating a surface and then milling off the plating we can generate an infinitely long plans, or end up in a infinite loop so that we do not find a plan at all.

There should be no optimal plan which returns to the same state. Recall that a state consists of a geometry described by its material composition and its properties. Any plan which returns to the same geometry, material and property state must have redundant operations within the loop of operations which returns the part to a former state. In order to avoid redundancy and unbounded growth and looping of plans, we redefine the model to check that the plan visits each state only once. As we generate the plan we must be able to check candidate new states against states which exist in the partial plan. We need a means to identify the input state to start the plan, a means of adding operations to the plan and a means of detecting membership in the plan. "simplePlan" is a function which generates only those plans with no loops. It is represented by the logic models of equations (19) and (20). In equation (19) "simplePlan" creates the initial plan represented by operation "start". It then calls the function "checkPlan" defined in (20). The function

Figure 5.5

A Programming Representation of the Preliminary Process Planning Model

(a) Feasible Operation Database.

```
canDo(a,b,mill).
canDo(b,c,grind).
canDo(c,d,drill).
canDo(a,c,grind).
canDo(c,e,turn).
```

(b)The "doPlan" predicate for generating plans

```
%Boundary condition for plan generation

    doPlan(Current,Final,(Final,stop)):- Current=Final,!.

%Recursive structure for plan generation

    doPlan(Current,Final,[(Current,Op)|Tailplan]):-
                canDo(Current,NextState,Op),
                doPlan(NextState,Final,Tailplan).
```

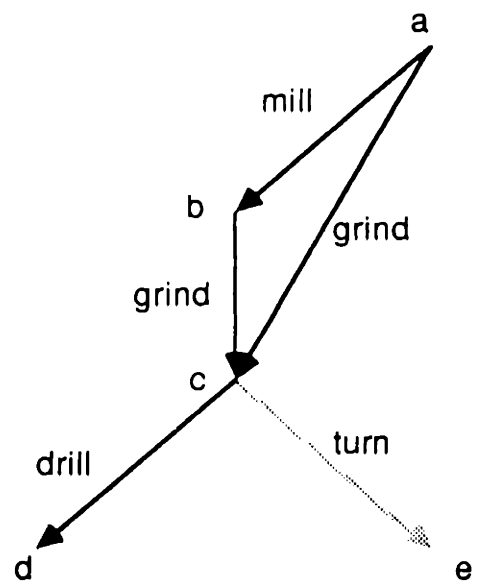
(c) Sample interaction exhibiting plans between initial state a and final state d

```
| ?- doPlan(a,d,Plan).

Plan = [(a,mill),(b,grind),(c,drill)|(d,stop)] ;

Plan = [(a,grind),(c,drill)|(d,stop)] ;

no
```

Figure 5.6

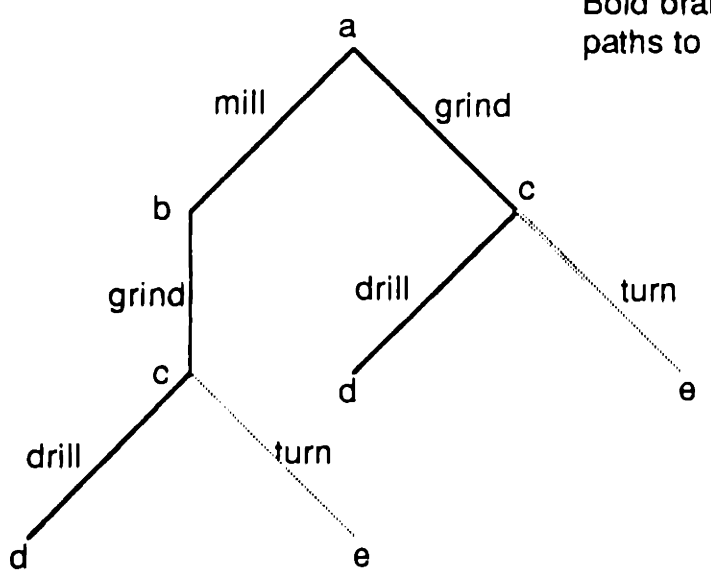
Graph and Tree Representations of the search for a Plan.

(a) Illustration of directed graph of possible plans



Bold edges indicate paths to goal state d

(b) Illustration of tree of plans



Bold branches indicate paths to goal state d

Figure 5.7

Illustration of Search Path and Backtracking in Preliminary Process Planning

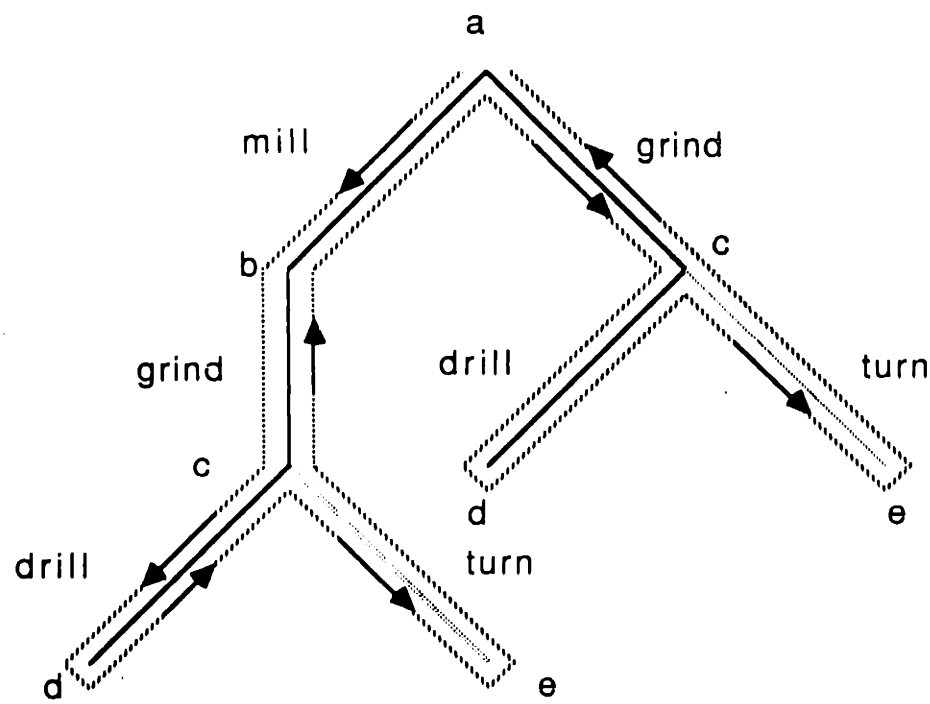


Figure 5.8

A Database which Generates Infinitely Long Plans

(a) Feasible Operation Database.

```
canDo(a,b,mill).
canDo(b,c,grind).
canDo(c,d,drill).
canDo(a,c,grind).
canDo(c,e,turn).
canDo(b,a,plate).
```

```
| ?- doPlan(a,d,Plan),

Plan = [(a,mill),(b,grind),(c,drill)|(d,stop)] ;

Plan = [(a,mill),(b,plate),(a,mill),(b,grind),
        (c,drill)|(d,stop)] ;

Plan = [(a,mill),(b,plate),(a,mill),(b,plate),
        (a,mill),(b,grind),(c,drill)|(d,stop)] ;

Plan = [(a,mill),(b,plate),(a,mill),(b,plate),
        (a,mill),(b,plate),(a,mill),(b,grind),
        (c,drill)|(d,stop)] .

yes
```

Figure 5.9

A Database which Sends the Program into an Infinite Loop.

(a) Feasible Operation Database.

```
canDo(a,b,mill).
canDo(b,a,plate).
canDo(b,c,grind).
canDo(c,d,drill).
canDo(a,c,grind).
canDo(c,e,turn).
```

```
| ?- trace.

yes

| ?- doPlan(a,d,Plan).
(1) 1 Call: doPlan(a,d,_0) ?
(2) 2 Call: a=d ?
(2) 2 Fail: a=d
(1) 1 Back to: doPlan(a,d,_0) ?
(3) 3 Call: canDo(a,_65644,_7) ?
(3) 3 Exit: canDo(a,b,mill)

(4) 3 Call: doPlan(b,d,_8) ?
(5) 4 Call: b=d ?
(5) 4 Fail: b=d
(4) 3 Back to: doPlan(b,d,_8) ?
(6) 5 Call: canDo(b,_65662,_12) ?
(6) 5 Exit: canDo(b,a,plate)

(7) 5 Call: doPlan(a,d,_13) ?
(8) 6 Call: a=d ?
(8) 6 Fail: a=d
(7) 5 Back to: doPlan(a,d,_13) ?
(9) 7 Call: canDo(a,_65680,_17) ?
(9) 7 Exit: canDo(a,b,mill)

(10) 7 Call: doPlan(b,d,_18) ?
(11) 8 Call: b=d ?
(11) 8 Fail: b=d
(10) 7 Back to: doPlan(b,d,_18) ?
(12) 9 Call: canDo(b,_65698,_22) ?
(12) 9 Exit: canDo(b,a,plate)

^CAction (h for help): ?

[ execution aborted ]
```

"checkPlan" is a modified version of "plan". It has three arguments, the input (or current) and final states, and the unary function "planTo(S_i)" which determines the plan as far as the current input state S_i. In "checkPlan", after assignments have been made to the next operation and state in "canDo", the member predicate is used to check that the state has not already been assigned in the plan so far. In this way all state and operation assignments which cause the plan to loop are rejected. "checkPlan" is recursive. It appends the current state-operation pair to the plan so far, and then recalls "checkPlan" with the next state in the plan. "checkPlan" is defined in a more procedural and less elegant manner than plan. This is necessary to avoid infinite looping.

$$\forall S_i \forall S_f \quad \{ \text{simplePlan}(S_i,S_f) \leftarrow \text{checkPlan}(S_i,S_f, [(S_i,\text{start})]) \} \quad (19)$$

$$\begin{aligned}
&\forall S_i \forall S_n \forall S_f \forall Op \forall Op_1 \\
&\qquad \{ \text{checkPlan (Si, Sf, planTo (Si))} \leftarrow \\
&S_i=S_f \wedge \text{checkPlan } (S_i, S_f, \text{planTo } (S_i)) = [\text{planTo } (S_i) | (S_f,\text{stop})] \\
&\qquad \vee \quad \text{canDo}(S_i,S_n, Op) \wedge \\
&\qquad\quad \sim\text{member}((S_n, Op_1), \text{planTo (Si)}) \wedge \\
&\qquad\quad \text{planTo}(S_n) = [\text{planTo } (S_i) | (S_n,Op)] \\
&\qquad\quad \text{checkPlan}(S_i,S_f,\text{planTo}(S_i)) = \\
&\qquad\qquad [\text{planTo}(S_n) | \text{checkPlan}(S_n, S_f, \text{planTo}(S_n)].
\end{aligned}$$

The predicate member determines whether (S_n, Op_1) is a member of planTo(S_i). The predicate member is negated so that if a state is already in the plan, an operation which would return to that state is rejected. Figure 5.10 illustrates a programming implementation of the modified planning model, "simplePlan". The functions are represented as predicates in the programming implementation. "simplePlan" detects all plans which visit a state only once. The predicate append(OldPlan, (Si,Op), NewPlan) appends (Si,Op) to OldPlan to form NewPlan.

Note that the word problem of Semi-Thue systems (section 3.1.5) was undecidable because infinite looping could occur and because the word

Figure 5.10
A Programming Representation of the Preliminary Process Planning Model Which Visits Each State Only Once

(a) Feasible Operation Database

```
canDo(a,b,mill).
canDo(b,a,plate).
canDo(b,c,grind).
canDo(c,d,drill).
canDo(a,c,grind).
canDo(c,e,turn).
```

(b) Predicate "simplePlan" generates all plans without loops

```
simplePlan(Input,Final,Plan):-
        checkPlan(Input,Final,[start],Plan).


checkPlan(Current, Final, OldPlan, Plan):-
        Current = Final,
        append(OldPlan,(Final,stop),Plan).

checkPlan(Current, Final, OldPlan, Plan):-
        canDo(Current, NextState, Op),
        not member((Current, Op1), OldPlan),
        append(OldPlan, [(Current,Op)], NewPlan),
        checkPlan(NextState, Final, NewPlan, Plan).



member(X,[X|Tail]).
member(X,[H|Tail]):- member(X,Tail).

append([],L,L).
```

(c) Sample Interaction

```
| ?- simplePlan(a,d,Plan).

Plan = [start,(a,mill),(b,grind),(c,drill)|(d,stop)] ;

Plan = [start,(a,grind),(c,drill)|(d,stop)] ;

no
```

generation was unbounded. If we were to allow unbounded state generation by defining an infinite geometric grid, rather than one bounded by the largest dimension of the raw material or finished part this would also render the problem undecidable. By specifying a problem carefully and limiting the infinite tendencies of a problem it is possible to make some specific realistic problems decidable. Note that we could not do this for general classes of problems, such as the halting problem of Turing machines, because determining all possible ways in which a Turing machine could be looping is as hard as the problem itself.

In mathematical logic the domain usually consists of one class of numbers, such as the non negative integers. In symbolic logic, the domain may contain different types of objects. Objects from the domain are assigned to the individual variables. Logic programming allows a natural partitioning of the domian so that only appropriate objects are assigned to variables. For example, in figure 5.5(a), the first two variables in the canDo predicate are states, and the third variable is an operation. When the pattern matching procedure tries to match one of the canDo predicates in the database with the current values in the doPlan program, the structure of the pattern ensures that the states and operations are matched accordingly. Yet this does not give any unfair advantage over mathematical logic which should deny us the properties of mathematical logic. We could assign all the values from the domain in a random fashion, and we would simply have far more combinations to check, many of which would make no sense. The completeness properties of wffs are determined by the structure of the syntax of the wffs and the properties of finiteness or infiniteness of the domain over which they are interpreted.

The domain of the plan model is a partitioned domain consisting of operations and states. We saw that the operations were functions of the states. However since more than one type of operation may be used to get between the same pair of states, there is a choice involved in assigning operations. Therefore the preliminary process plan domain consists of states and operation types.

The model, M, is satisfied by some interpretation I, if for a given input state *y* and output state *x*, a feasible operation exists for each state transformation in some set of states which form a path from *y* to *x*. The plan generation model, M and a corresponding interpretation I of equation (19) consist of the following assignments to the *wff* of (20):

$$\forall x \forall y \quad \{ \; p_1 (x,y) \leftarrow p_2 (x, y, [(a)]) \; \} \tag{20}$$

$$\begin{aligned} \forall x \forall y \forall z \forall v \forall v_1 \quad \{ \; & p_2 (x, y, p_3 (x)) \leftarrow \\ & x=y \wedge \; p_2 (x, y, p_3 (x)) = [\, p_3 (x) | \, (y,b)] \\ & \vee \; p_4(x, z, v) \wedge \\ & \quad \sim p_5((z, v_1), p_3 (x)) \wedge \\ & \quad p_3 (z) = [\, p_3 (x) \mid (z,v)] \wedge \\ & \quad p_2 (x, y, p_3 (x)) = [\, p_3 (z) \mid p_2(z, y) \,] \; \} \end{aligned}$$

individual variables :

S_i, x	input state,
S_f, y	final state,
S_n, z	intermediate (next) state
Op, v	an operation

individual constants:

start, a	start operation to act as an anchor for the plan which is to be checked
stop, b	stop operation to indicate final state has been reached.

binary predicate constants:

member, p_5

binary function constants:

simplePlan, p_1

unary function constant:

planTo, p_3 — generates a plan from the start state to the current state (x or z).

trinary function constant:

checkPlan, p_2 — generates a plan from the raw material, state x, to final state y.

canDo, p_4 — determines whether an operation, v, exists from the current state, y, to the next state, t.

5.2.5 Summary of the Preliminary Process Planning Model, $D,\mathfrak{1}_c,\mathfrak{1}_v$

The domain, D, is the set of possible states and the set of operations. The constant assignments, $\mathfrak{1}_c$, are predicates determining whether an operation exists between states and the individual constants representing the start and stop states. The variable assignments, $\mathfrak{1}_v$ are states and operations from the domain.

The output of the plan generation, is an ordered set of processes. In fact all possible plans can be produced and we can choose the best plan by using heuristics, or perhaps some kind of optimizing function. A well designed expert system would try to search the space of all possible plans in such a way that the best plan was produced first. We shall examine how this might be done in Chapter 7.

5.2.6 Proof of Decidability of Plan Generation

We contemplate whether this plan generation scheme is computable. That is, can we guarantee that we will find a plan if one exists, and that if one does not exist, the program will fail, or might it run forever? In section 3.3.7 we noted the completeness of deduction systems for the first order predicate

calculus. This allowed logic programming languages based on first order predicate logic using refutation theorem provers which are complete for Horn clauses. However the completeness of deduction systems depends on the validity of the wffs which are the axioms of the deduction system. If the planning model is to serve as an axiom of a deduction system, for each instance of variable unifiication, it must determine correctly and conclusively, whether this instance represents a plan or not. If the validity of the wffs on which the deduction systems are based is not decidable, the completeness of the deduction system serves no purpose. The closure properties of theorem provers do not help us to deduce facts from undecidable wffs. The planning wff was defined in first order logic. In general the validity of first order wffs is undecidable (section 3.3.5.2). Therefore we pose the question, is the planning problem decidable?. If we can show it is a restricted form of the first order predicate calculus, or that it can be represented by a finite conjunction of sentences of the propositional calculus, then we can determine that planning is decidable.

Theorem : The plan generation problem represented by the model of (19) is decidable.

We prove the above theorem by showing that the plan generation problem can be represented by a first order logic sentence whose domain contains a finite number of individuals. That is, we use Hilbert and Ackerman's result (section 3.3.5.2).

Proof :
The plan generation problem may be represented by the model, M of (19). The domain of the model is the set of states and operations. A state comprises geometric, material and property information.

A part is defined on a grid whose boundary is equal to the outer dimensions of the raw material or the finished part, whichever is larger. The grid spacing is equal to the minimum tolerance which is machinable. That is the spacing is finitely small so that the number of grid elements is finite. The geometry grid is

represented by a finite alphabet

$$\Sigma_{PLANNER} = \{g_1, g_2, g_3, \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots g_l\}$$

A geometric boundary is implied by the grid boundary. This prevents repeated material addition operations generating new geometric elements which could render the number of states unbounded.

The material and property alphabets, D_m and D_p respectively, describe the contents of the grid.

$$\text{Material alphabet } D_m = \{D, m_1, m_2, m_3, \ldots\ldots\ldots\ldots\ldots\ldots\ldots m_n\}$$

$$\text{Property alphabet } D_p = \{p_1, p_2, p_3 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots p_r\}$$

Lemma: The number of possible states is finite and enumerable.

Proof of Lemma:
Let the lengths of the alphabets be as follows:

$$|\Sigma_{PLANNER}| = l$$

$$|D_m| = n$$

$$|D_p| = r$$

The number of material choices for a single grid element = n.

The property alphabet is used to describe process operations such as heat treatment which achieve a set of properties for a part. Each process operation is unique and is applied only once. The order in which the processes are applied may produce different properties.

The possible set of process operations on a part is given by the set, $P^{\#}$, which contains all permutations and combinations of D_p. If $D_p=\{p_1,p_2\}$, then

$$P^{\#} = \{\text{Process combinations and order}\}$$
$$= \{p_1, p_2, p_3, p_1p_2, p_1p_3, p_2p_1, p_2p_3, p_3p_1, p_3p_2\}.$$

The length of the set $P^{\#}$ is equal to the number of ways of choosing one operation plus the number of ways of choosing two operations and so on.

$$|P^{\#}| = r + \frac{r!}{2!\,(r-2)!} + \frac{r!}{3!\,(r-3)!} + \ldots + \frac{r!}{r!\,0!}$$

$$= \sum_{0}^{i} C(r,i) = 2^r$$

Therefore the number of material and process combinations which a single element could assume is

$$n \times 2^r$$

The number of possible states of the part is given by all possible combinations of the descriptions of the geometry elements:

$$|\{\text{states}\}| = (n \times 2^r)^l$$

All the states may be enumerated by proceeding through the alphabets systematically .

Hence the set of all states is finite and enumerable.
End of Lemma Proof.

Let the set of states be denoted S, having length s.
The set of operation types is a finite set of length t, denoted by

$$O = \{o_1, o_2, \ldots o_t\}.$$

Feasible operations are defined by rules on the input and output states.There are a finite number of states and operations, and each state may only be visited once. The number of ways of choosing a pair of states from the set of all states is given by

$$\frac{s!}{2!\,(s-2)!}$$

This calculation of the number of combinations excludes diiferent permutations of a pair of states, (if we choose (a,b) we exclude (b,a)), and it excludes repetition of a state within a pair, such as (a,a).

Therefore the number of possible operations and state pairs evaluated by the canDo predicate is given by

$$|\{ S_i, S_n, Op \}| = \frac{s!}{2!\,(s-2)!} \times t$$

A plan consists of a list of state and operation pairs

$$[(S_i, Op_1), (S_2, Op_2), \ldots\ldots\ldots\ldots, (S_f, stop)]$$

We prevent the plan from visiting any state which has already been visited. Since there are a finite number of operations and states, there are a finite number of permutations and combinations of operations and states. That is, there are a finite number of different candidate plans and all of the candidate plans are finite.

Therefore every possible interpretation of states and operations can be systematically generated. "simplePlan" can be written out as a finite sequence of propositions by assigning all of the states and operations from

an interpretation. Since the validity of the propositional calculus is decidable (section 3.3.5.1), the plan generation problem is decidable.
End of Proof.

The preliminary process planning model was discussed in the context of material removal and addition operations such as machining and plating. But it also serves to model assembly and disassembly. Since any part contains a finite number of smaller parts to be assembled we need only address the looping problem. We represent each unit part as a letter in the basic alphabet. If we allow assembly and disassembly of the same part more than once in such a way that we can loop between states, then assembly is undecidable. But if we restrict an assembly model to allowing only one visit to each state of the part then all asssembly plans can be represented by a finite sequence of propositions, and assembly is decidable. Process planning can be represented by a similar model. [Olsen and DeVries, 1987] showed that process planning was decidable.

5.3 A Formal Model For Machine Selection

Given a set of products to manufacture, the volume of production of each product and a preliminary plan for each product, we must select a set of machines.The machine selection problem is to find a set of machines which can perform all operations required by the preliminary plans of all parts.

The relation mapping operations, o, to machines, v,

$$\text{op-map} : o \rightarrow v,$$

is not many to one. Both lathes and milling machines can perform a drilling operation. Also in a machine database accessed by an expert system there may be many different types of lathes and milling machines to choose from, with different ranges of feeds and speeds. Therefore the machine selection process is non-trivial, even though the processes are known.

5.3.1 A Logic Model for Machine Selection

The domain of machine selection is a machine database which may be described by the alphabet

$$\Sigma_{SELECTOR} = \{v_1, v_2, v_3, \ldots\ldots\ldots\ldots v_n\} \text{ where } v_i \text{ is a machine.}$$

In the connector alphabet S = {•,#} ,

• represents grouping of machines within a factory
represents separation of machines in different factories.

Further description of the machines needed to determine whether an operation may be performed on a machine includes details such as the maximum dimension of a part which can be held in the machine and maximum rates of material removal. These parameters are specific to each machine. Therefore they cannot be described by a general alphabet . In the preliminary planner we used descriptor alphabets to avoid the repetition of representing each element in material 1 and each element in material 2 and so on. For machine selection a single domain alphabet is sufficient.

The predicate indicating a mapping of a set of processes to a set of machines is ,

selection: O→M,	where,	O is the set of operations {o},
	and	M is the set of machines {v}.

We now assume that the members o, of O are parameterized operations represented by triples (s_i, Op, s_e), where s_i is the input state of the part to the operation, and s_e is the output state of the part from the operation. This information was available in the preliminary process plan. This predicate "selection" is defined by the following logic model:

$\forall o\ \forall v\ \forall O\ \forall M$ { selection (O,M) $\leftarrow$ $\forall(o \in O)\ \exists(v \in M)$ { op-map(o,v) } }. (20)

In words, "M is a set of machines for the set of operations, O, if for every operation in the set, there is a machine in the machine set, which can perform the operation". The predicate op-map performs the discriminatory function undertaken by canDo in the preliminary plan model. For example,

$$\text{op-map}((s_i,\text{turning}, s_e), \text{Machine}) \leftarrow \quad (21)$$
$$\text{lathe(Machine)} \wedge (\text{chucksize(Machine)} > s_i).$$

It determines which machines may perform which operations.

5.3.2 Summary of the Model, D, $\mathfrak{I}_c$ $\mathfrak{I}_v$

The domain, D of the selection model is the database of machines. $\mathfrak{I}_c$ consists of predicate constants *selection* and *op-map.* An interpretation $\mathfrak{I}$ of the selection model is obtained by assigning machines to the individual variable,v. The operations, o are constants assigned from the chosen interpretation of the preliminary planning model.

5.3.3 Proof of Decidability of Machine Selection

Our optimization criteria might be to choose the interpretation of the selection model which has the smallest number of different types of machines. Or we may wish to choose the interpretation with the lowest total machine cost. However our ability to find the optimal interpretation depends upon the computability of the MachineSelection model. Can we guarantee that a program of the model will find a selection if one exists, and that if one does not exist it will fail , or might it run forever? If Machine Selection is decidable, then we can find all possible machine combinations satisfying the set of processes to be performed. If we can find all the combinations, then we can systematically enumerate all of them to determine which one is the optimum.

Theorem : The machine selection problem defined by equation (20) is

decidable.

Proof :

Define a machine database from which to make the selection decision:

$$\Sigma_{SELECTOR} = \{v_1, v_2, v_3, \ldots\ldots\ldots\ldots v_k\} \quad \text{where } v_i \text{ is a machine.}$$

Given a set of operations, O, from all plans

$$\forall o\ \forall v\ \forall O\ \forall M\ \{\ \text{selection}\ (O,M) \leftarrow\ \forall (o \in O)\ \exists (v \in M)\ \{\ \text{op-map}(o,v)\ \}\ \}.$$

If the number of machines types in the database, $|\Sigma_{SELECTOR}|$, is finite and the number and length of the plans, $|P|$, is finite, then the number of operations in the operation set $|O|$ is finite, and the number of different machine combinations, is finite and enumerable. The maximum number of different machine set combinations for a given operation set is given by

$$n + \frac{n!}{2!\,(n-2)!} + \frac{n!}{3!\,(n-3)!} + \ldots + \frac{n!}{n!\,0!}$$

where n is the number of machines in the database or the number of operations in the operation set, whichever is smaller. Note that the set M denotes the machine types which will be in the factory. It does not specify the exact number of each machine type.

Therefore all interpretations of the machine set can be systematically generated. For each interpretation the selection model can be written out as a finite sequence of propositions. Since the validity of any wff of the propositional calculus is decidable, (section 3.3.5.1), machine selection is decidable.

End of Proof

An exhaustive search for a suitable machine set might search all of these combinations. But the search space can be reduced substantially by

intelligent search techniques. A software implementation of machine selection is described in Chapter 7. This will be discussed further in Chapter 7.

5.3.3 Extending the Description of the Selection Model

We could define further functions to express the optimization problem. For example, we may wish to minimize the number of different machines required. We could express the number of machines of type j required for each interpretation as follows:

$$N_j = \sum_{i=1}^{n} (q_{ij} t_{ij} / c_{ij}),$$

where

q_{ij} = number of parts with a process i mapped to machine j

t_{ij} = time taken for each process i on machine j

c_{ij} = time available per production period for operation of machine j,

We could write the above in a set of logical sentences and also in a logic programming language.

5.4 A Formal Model for Layout

5.4.1 Introduction to the Layout Model

The layout problem in its entirety consists of placing machines, handling equipment paths, tool cabinets, etc., to meet the space, flow and adjacency related objectives. The layout phase is usually simplified by laying out departments, and then laying out the workstations within the department. A difficulty arises due to the interaction between interdepartmental and intradepartmental flows. If the department shapes are defined to give optimum internal flow, the interdepartmental flow may be sub optimal. If the deaprtment shapes are flexible so that optimal space usage and interdepartmental flows are obtained, the internal flows may be sub optimal. The approach of laying out departments also assumes that the decision of

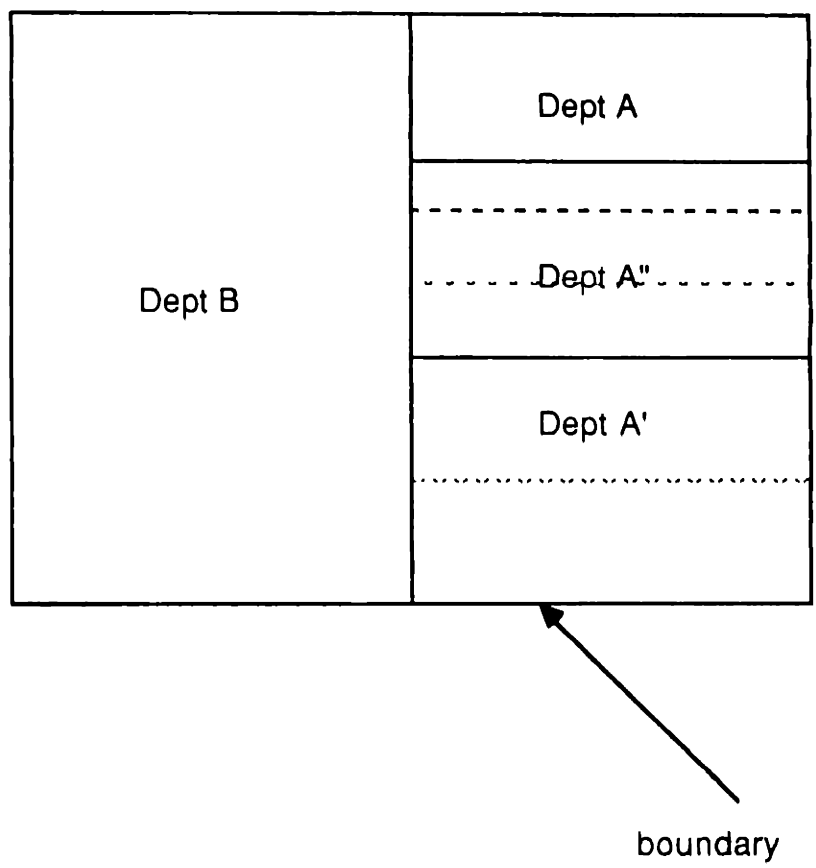
which machines to group in which departments, and how many machines should be grouped as a department is clear cut. It becomes evident that the abstract approach is satisficing and is employed to limit the complexity of the problem. The complete layout problem is formulated by treating each object which receives or sends a flow as a separate object. Each object is represented by a shape which includes a border around the object to allow for machine travel, maintenance access etc.. The space, flow and adjacency related objectives will orient the objects into the most suitable department configurations.

The layout problem can be represented as optimizing the arrangement of the objects which constitute the factory, subject to the optimizing constraints. The constraints are important because, otherwise, the problem is to lay out a given number of shapes in space. This problem is undecidable because it is unbounded. Even if a factory boundary is defined, unless the shapes fit exaclty into the boundary the problem is undecidable. In fact it is uncountably infinite. Consider the object pair within the boundary in figure 5.11 . Object A could be moved to location A'. It could also be moved to A", inbetween A and A'. No matter how close is A to A' there is always a location between them. This is because no matter how many points on a scale we define, there is always an uncountably inifinite number of points constituting the remaining space. This follows from Cantor's result that the set R of real numbers is not countable.[Brady, 1977]. But we can appeal to practicality for the argument that to the factory designer any movement less than say one inch is insignificant, in particular because it would not cause any other object to be moved, since each object has a border for maintenance access as discussed previously. If we consider the layout problem bounded by a predefined border, where the only valid object placements are on a one inch grid, then the number of different object placement locations within the boundary is finite. Furthermore, the bordered area can be much larger than the combined area of all the objects, so that it does not affect the near optimal solutions. The optimizing constraints which aim to minimize flow and space will cause the optimal solutions to be those which cluster the objects together so that the boundary does not pose an active constraint.

Figure 5.11

Illustration of the Uncountably Infinite Possible Placements of Department A;

Wherever Department A and A' are placed, another department can always be placed in between



It is possible that we could prove the continuous space case decidable, if we use these constraints to develop an equation similar to that of the n body problem. In this problem n bodies apply an attractive force on each other (from center to center). No bodies are allowed to overlap, and we consider only with relative positions of objects. Otherwise the same configuration could be located in space in infinitely many orientations.

For a graphic depiction of the problem, consider the objects of a factory linked together by large springs, the spring constant of each spring representing the flow between objects. Initially the objects are randomly oriented. Then in a manner similar to the simulated anealing approach [van Laarhoven, 1987] we connect springs with a tensile strength representing 1/100th of the actual flow. We disturb the configuration, and when it reaches an unstable equilibrium the tensile strength of the springs is increased and the configuration is disturbed again. That is, we use a suitable cooling schedule (in this case a suitable spring tightening schedule) such that eventually the departments will be optimally located by the springs representing the full flow in the stable equilibrium position. It is conjectured that such an equation could be formulated and would converge.

5.4.2 A Logic Model for Layout.

The layout problem is to determine the best arrangement of a set of machines on a finite grid within a given perimeter. A large perimeter is chosen so that it does not pose a constraint on the optimal solution.

The factory grid is represented as a spatial alphabet

$$\Sigma_{LAYOUT}=\{c_1,c_2,.....c_n\}.$$

Let

M = the set of machines, {v}, chosen by the Machine Selection Module,

(since we are to lay machines out in the factory we assume that the number of each type of machine has been chosen by an optimizing extension of the machine type selection model, and that each individual machine is a member of the set of machines).

D = the set of dimensions, {(w,h)}, (width and height) of the machines in M,

F = the coordinates of the factory, a quadruple, (left,bottom,width,height).

C = the set of coordinates, {(l,b)}, at which the lower left corner of each machine in M is placed.

g = the grid spacing.

Let each member c of the alphabet, Σ_{LAYOUT} , represent a pair,(l,b), at which the lower left corner of each machine in M may be placed. Then the factory layout domain is the set of grid coordinates,

$$\{(l_1,b_1),(l_2,b_2),(l_3,b_3).....(l_n,b_n)\}$$

The length of the set of grid points in the factory is:

$$|\Sigma_{LAYOUT}| = \frac{\text{width x height}}{g^2}$$

The set of pairs of machine, and placement coordinate and dimensions, is written {(m,(l,b,w,h))} .

The problem is to assign the grid coordinates in the domain to the machines in the machine set, such that the area covered by the machines is identified by the set of machines and locations,

$$L = \{(m_1,(l_1,b_1,w_1,h_1)), (m_2,(l_2,b_2,w_2,h_2)).....\ (m_k,(l_k,b_k,w_k,h_k))\},$$

subject to constraints that no machines overlap, and that all machines lie within the factory boundary. A placement convention is adopted. For example, the grid coordinate represents the lower left corner of each machine.

The layout function maps a set of machines, and their width and height to a set of placements

$$\text{layout} : \{ (v,(w,h)) \} \rightarrow \{(l,b)\}.$$

The following logical model defines the function "layout". It has a boundary condition and a recursive structure. The boundary condition checks to see whether all machines assignments have been made. The recursive structure assigns coordinates to the current machine, and then recalls the function "layout" to assign coordinates to the remaining machines in the set. It also checks that the whole machine lies within the factory boundary, through the predicate *within*, and that it does not overlap any of the assignments which have already been made through predicate *anyOverlap* . The model which describes layout is defined below. It is quantified over the domain of coordinates. The machines and their dimensions are constants assigned from the chosen interpretation of the machine selection model. Because this is a general model and it is appropriate for any machine set, machines and their dimensions are denoted as variables, (M,(W,H)), but in fact their choice and has already been made.

$$\begin{aligned}
&\forall L\ \forall B\ \forall W\ \forall H\ \forall \text{Machines}\ \forall M\ \forall \text{Left}\ \forall \text{Bottom}\ \forall \text{Width}\ \forall \text{Height} \quad (22)\\
&\quad \{\ \text{layout}(\text{Machines}) \leftarrow\\
&\quad (\text{Machines} = \Delta) \wedge \text{layout}(\Delta) = \Delta\\
&\quad \vee\ \text{member}(M, \text{Machines}) \wedge\\
&\quad \text{dimensions}(M, (W,H)) \wedge\\
&\quad \text{placement}\ (M, (L,B)) \wedge\\
&\quad \text{within}((L,B,W,H), (\text{Left}, \text{Bottom}, \text{Width}, \text{Height})) \wedge\\
&\quad \text{layout} = [(M,(L,B,W,H)) \mid \text{layout}(\text{Machines} - M)] \wedge\\
&\quad \sim(\text{anyOverlap}((M,(L,B,W,H)), \text{layout}(\text{Machines} - M)))\ \}
\end{aligned}$$

The *anyOverlap* predicate is defined recursively in equation (23). It is quantified over all variables, (denoted by upper case letters). It checks that an assignment does not overlap with the first of the remaining assignments

through a predicate *overlap*. It then checks that is does not overlap with any remaining assignemnts in the list by recalling the predicate *anyOverlap*. The predicate *overlap* is not defined here. It simply checks to see that rectangles defined by two sets of coordinates do not intersect.

$$\begin{aligned} \text{anyOverlap}((M,(L,B,W,H)), \text{layout}(\text{Machines})) \leftarrow \quad & (23) \\ \text{member}((M_1,(L_1,B_1,W_1,H_1)), \text{layout}(\text{Machines})) \wedge \\ (\text{overlap} ((M,(L,B,W,H)), (M_1,(L_1,B_1,W_1,H_1))) \vee \\ \text{anyOverlap}((M,(L,W,B,H)), \text{layout}(\text{Machines} - M_1)) . \end{aligned}$$

5.4.3 Summary of the Layout Model $D,\mathfrak{I}_c,\mathfrak{I}_v$

D, the domain is the set of coordinates on the factory grid. $\mathfrak{I}_c$ consists of predicate constants, such as, *placement*, *anyOverlap*, and *within*, and the function constant *layout*. $\mathfrak{I}_v$ completes the interpretation of the model by assigning coordinates from the domain to each machine.

5.4.4 Proof of Decidability of Factory Layout

Theorem: The factory layout problem of the model of (22) is decidable.

Proof:

We show that the number of possible layout configurations is finite and enumerable. The number of grid points is

$$|\Sigma_{LAYOUT}| = \frac{\text{width x height}}{g^2}$$

The length of the set of machines is |M|.

The number of grid points is finite and the number of machines is finite. Therefore the number of possible different assignments of machines to locations is finite. It is bounded from above by

$$= |\Sigma_{LAYOUT}|^{|M|}$$

Therefore all interpretations of the coordinate set can be systematically generated. For each interpretation the layout model can be written out as a finite sequence of propositions. Since the validity of any wff of the propositional calculus is decidable,(section 3.3.5.1), layout is decidable.

End of Proof

The optimization problem is to find the best layout which meets the constraints stated, subject to given optimization criteria.

5.4.5 Extending the Description

For each part we have an operation sequence

$$o_1, o_2, o_3 \ldots\ldots\ldots o_n$$

and each operation is mapped to a machine

$$o_1 \rightarrow m_1,\ o_2 \rightarrow m_2, o_3 \rightarrow m_3, \ldots\ldots, o_n \rightarrow m_n$$

Since we know the volume of production of each part, the plan sequence, and the machine on which each operation is to be performed we can determine the flow between machines. We could develop a handling equipment selection model similar to the machine selection module to determine which sets of handling equipment are suitable. With suitable cost functions describing the handling equipment we can evaluate the cost of each layout interpretation. These logic models together with some control procedures form the basis of a computer system for factory design described in Chapter 8.

5.5 Implications of the Factory Design Model.

5.5.1 Dependencies and Independencies of the Phases of Factory Design

The formal model for factory design consists of three phases

(i) preliminary planning,

(ii) selection,

(iii) layout.

Table 5.2 shows the domain, D, and the assignment of the constants and variables, $\mathfrak{l}_c$ and $\mathfrak{l}_v$ for each phase. Figure 5.12 depicts the information flow from phase to phase. We draw comparisons between table 5.2 and figure 5.12. The mapping relations which link each phase are expressed through the predicate constants of $\mathfrak{l}_c$. These mapping relations map variables from the domain on to the interpretation of the previous phase, expressed as individual constants in $\mathfrak{l}_c$, in order to develop an interpretation for the new phase. In the first phase, planning, the individual constants are the part and raw material specifications, obtained from the part drawing.

The part specifications are determined from its functional requirements and constraints. The functional requirements are defined as:

The minimum set of independent requirements for a product or process that completely characterizes the design objectives.[Suh, 1984].

Design may be characterized as a mapping process from a set of functional requirements {FRs} to a set of design parameters {DPs}, or product characteristics. This is represented as

$$\{FRs\} = [DM]\{DPs\},$$

where [DM] is called the design matrix. [DM] describes the transformation process. The process of design is to determine the product parameters from the functional requirements.

The preliminary planner showed that given the product characteristics we can determine a process plan. The machine selector showed that we can

Table 5.2

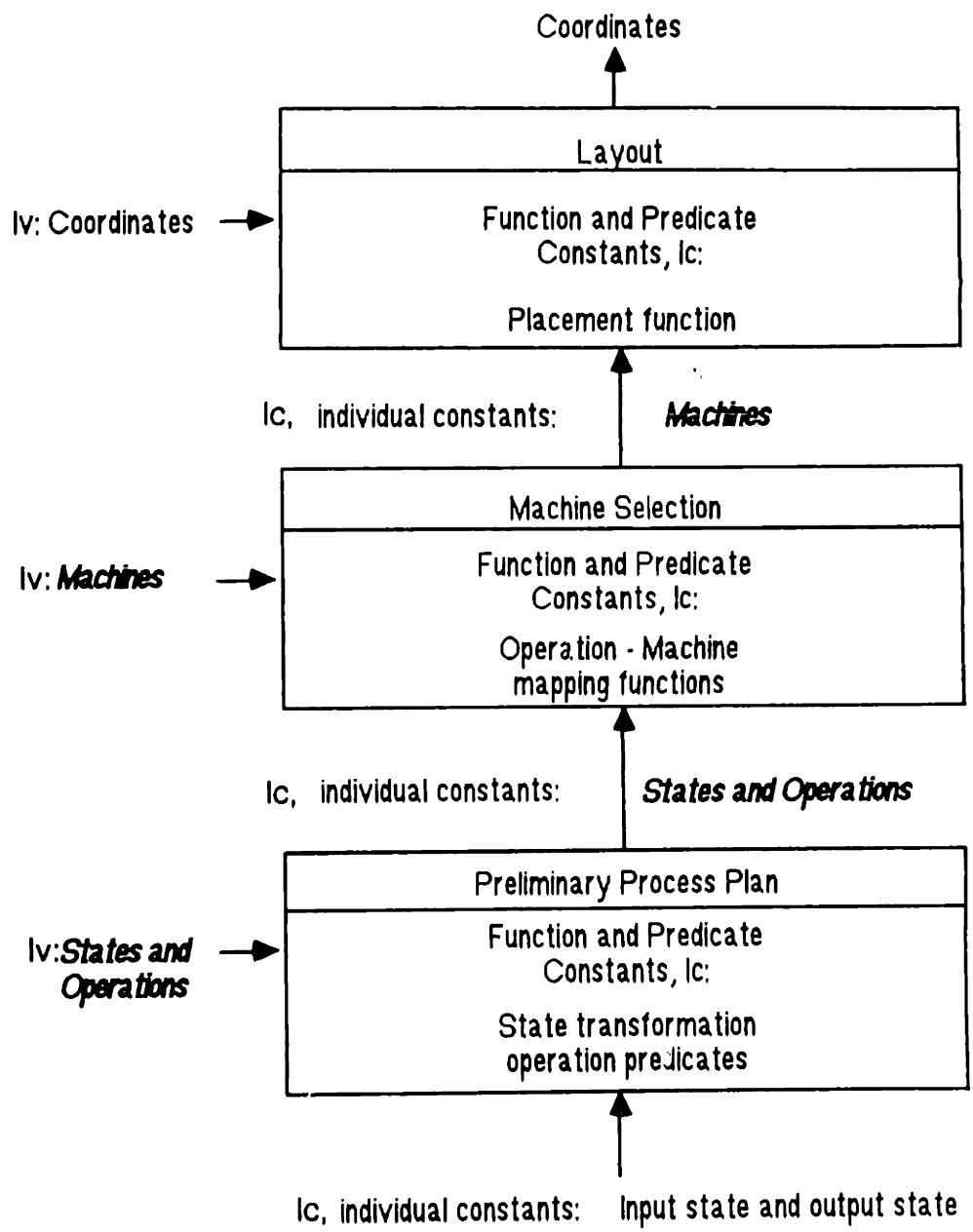
Constant and Variable Assignments in the Models of Factory Design

Formal Structure	**Phase** ← Factory Design →		
Interpretation	Preliminary plan	Equipment selection	Factory layout
Domain	Set of states and operations	Database of machines	Coordinates on grid
Ic	Operation function	Process-machine map function	Locate function
Iv	State	Machines	Coordinates

Figure 5.12

Dependencies Between the Phases of Factory Design

The variables assigned from the domain of the previous model are assigned as individual constants in the next model



determine which machines to select given the preliminary plan, and the layout module showed how these choices manifested themselves in a physical layout.

Theorem : A preliminary plan can be determined solely from the specifications of a product. Extra information from the functional requirements not embodied in the specifications can not be used to determine the preliminary plan.

Proof :
The specifications or design parameters of a product can be determined from its functional requirements.

$$\begin{bmatrix} FR_1 \\ FR_2 \\ \vdots \\ FR_n \end{bmatrix} = \begin{bmatrix} X & 0 & X & X \\ 0 & X & 0 & X \\ X & 0 & X & X \\ X & 0 & 0 & X \end{bmatrix} \begin{bmatrix} DP_1 \\ DP_2 \\ \vdots \\ DP_n \end{bmatrix}$$

Any functional requirement which is not manifested in the specifications has no bearing on the manufacture of the product. Suppose there is a functional requirement with hidden information which could aid the preliminary plan. We can decompose such a functional requirement into two parts, FR_i, which is related to the design parameters already stated and FR_j, which is not. If the information in FR_j is relevant to the manufacture of the product, then it can be represented by coupling through the design matrix to a further design parameter, DP_j.

$$\begin{bmatrix} FR_1 \\ FR_j \\ : \\ FR_n \end{bmatrix} = \begin{bmatrix} X & 0 & X & X \\ 0 & X & 0 & 0 \\ X & 0 & X & X \\ X & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} DP_1 \\ DP_j \\ : \\ DP_n \end{bmatrix}$$

If the information does not affect the manufacture, then it is coupled through the design matrix to the further design parameter, DP_j, in the equation below by a row of 0s.

$$\begin{bmatrix} FR_1 \\ FR_j \\ : \\ FR_n \end{bmatrix} = \begin{bmatrix} X & 0 & X & X \\ 0 & 0 & 0 & 0 \\ X & 0 & X & X \\ X & 0 & 0 & X \end{bmatrix} \begin{bmatrix} DP_1 \\ DP_j \\ : \\ DP_n \end{bmatrix}$$

But in this case the informational content of any such requirement is irrelevant. Hence a preliminary plan can be determined solely from the specifications of a product.
End of Proof

Corollary:
A factory design can be determined solely from the specifications of a set of products, and a database of machines. Extra information from the functional requirements of machines or products not embodied in the specifications can not be used to determine the factory design.

This result is related to the result of [Kim,1985], [Kim,1986], that the purpose of a product, (its functional requirements), cannot be deduced from its structure, (its specifications). Its implication is that even if we could determine the functional requirements from the specifications, they would not aid the factory

design process. Even if we view design as an iterative process, and several designs and preliminary plans are developed in conjunction before the design and plan are finalized, at each stage of the iteration, the design and planning process are linked only by the specifications, otherwise they are independent.

Implications of Theorem :

We might expect that to design a factory we would use industry specific knowledge about the parts being produced, the processes and the machines. However our theorem suggests a limit to the amount of domain dependent knowledge which is applicable at each stage of the design process. Only knowledge manifested in the product specifications is relevant for generating the preliminary plan, (figure 5.13). The preliminary plan incorporates all of the information embodied in the product specifications, together with information describing how to proceed through the intermediate states from the raw material to the product defined by the product specifications. The preliminary plan expresses the functional requirements of the equipment which is to be selected to make the part, (figure 5.14). Once the equipment is chosen, its functional requirements are irrelevant from the perspective of the layout designer. In the layout phase, the only relevant information relates to the spatial characteristics of the machines and handling system, and to the proximity relationships between them, which are determined by the operation sequence of the plan and machine characteristics such as *noisy* or *dust producing*.. These machine characteristics can be regarded as parameters of the layout problem. We can write general rules which relate to machine characteristics. For example, *isolate vibrating machines from precision machines.*

5.5.2 Domain Dependence and Independence.

If we examine the rules gathered from an experienced automobile factory designer, listed in figures 4.8 - 4.10, we find that many layout rules are more specific cases of the general rules in figures 4.3 - 4.7. For example *minimize conveyor length*, and *place receiving next to transportation* are cases of the minimze flow rule expressed as follows:

Figure 5.13

Illustration of the Dependence of a Plan on the Product Specifications

All information in the functional requirements of a product which is relevant to its manufacture can be expressed through the product specifications.

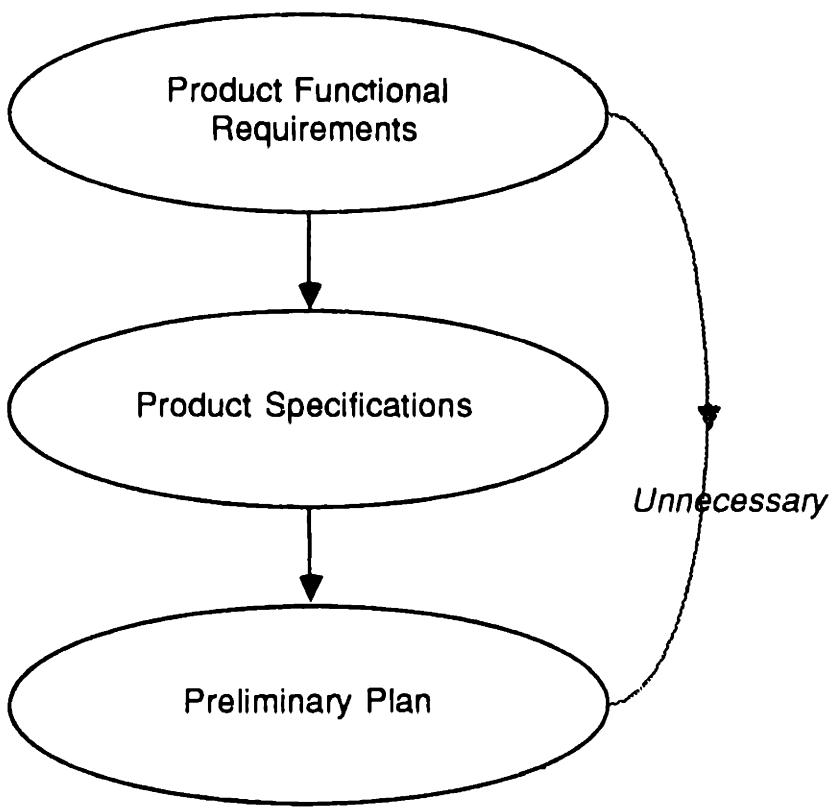
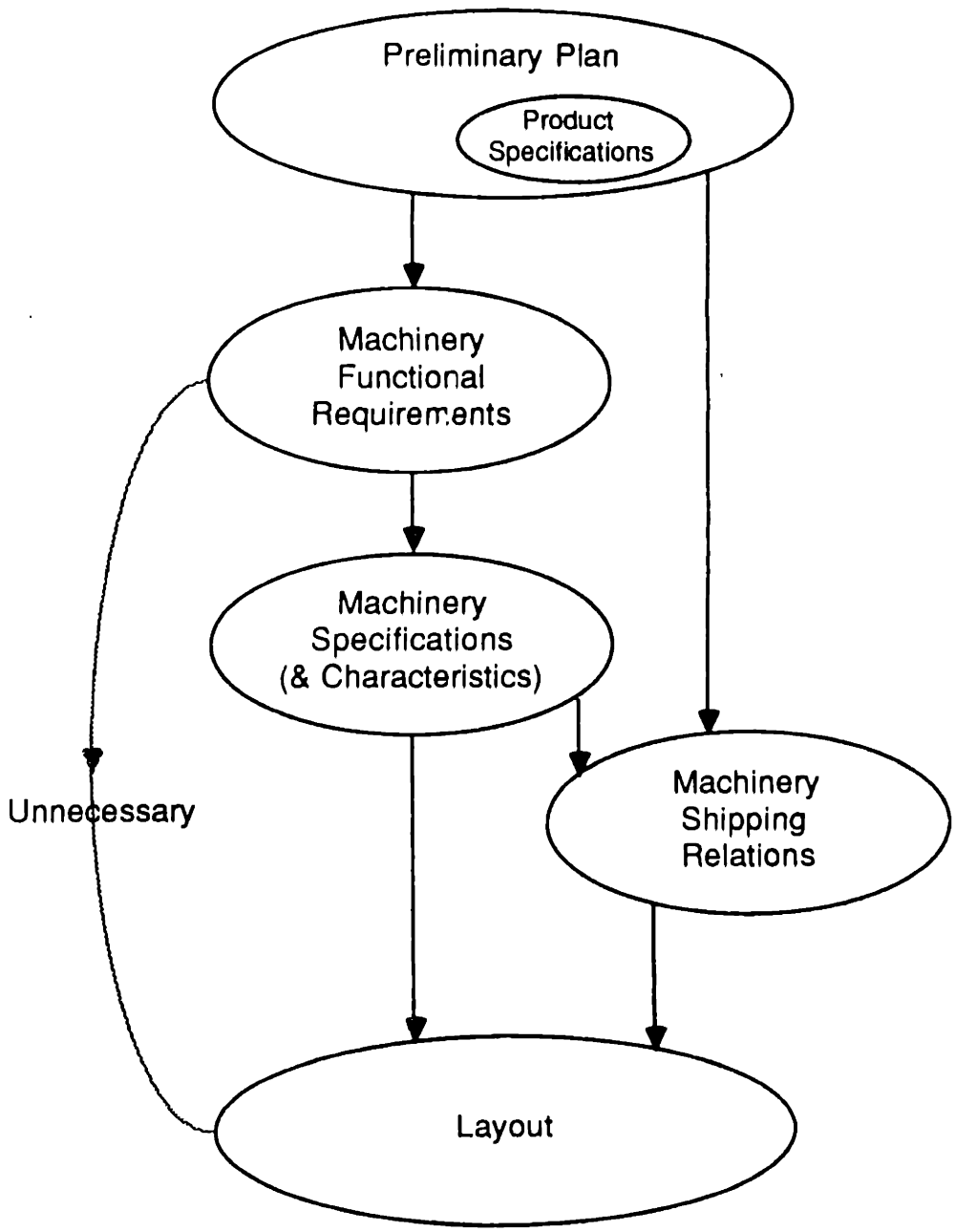


Figure 5.14

Illustration of the Information affecting the Layout Phase.

All information required for the layout phase can be expressed as a relationship between machines or a machine characteristic. The functional requirements of a machine do not aid the layout process.



IF DeptA ships Flow-weight to DeptB,
THEN place Dept A adjacent to DeptB.

The priority of placement depends upon the magnitude of the parameter *Flow-weight*. Similarly the rule *avoid moving extraction booths and large tanks* expresses the need not to place these items near to anything which might need to expand, causing them to have to move. It may be expressed as follows:

If Dept A needs to expand and DeptB cannot be moved,
THEN do not place Dept A adjacent to DeptB.

In Chapter 7 we discuss the way in which we organize the attributes of each department such as its dimensions and characteristics into frames so that they may be accessed simply and new features added easily.

The conveyor selection rules of figure 4.8 relate to the product and process specifications. Figure 5.14 leads us to expect this. Rules which relate directly to the product specifications, (machine selection rules) are more likely to be industry specific, because product characteristics which relate to the process plan are not so easy to generalize as machine characteristics which relate to layout. It can clearly be seen from figure 4.8 that these rules relate to the product. However the conveyor placement rules of figure 4.9 could equally well apply to other factories using conveyors. They do not relate to the product because they are layout rules. If a conveyor is a floor conveyor which cannot be crossed then it is desirable to make the conveyor lengths small and to loop them frequently so that aisle access is not restricted. This rule relates to minimizing flow. It also gives insight into how we should calculate travel distances between departnments. It states that if two departments are linked by a conveyor then we cannot consider the edge between them to be an aisle through which we can transport material. Some of the other conveyor placement rules are rules which relate to details of placement such as

avoiding pillars which support the factory roof. These detailed rules would probably be incorporated after the layout had been made by performing incremental adjustment to the placements. The temporal rule, *If machinery cannot be installed in a four week changeover period then install in existing unused space,* produces a sub-optimal layout but satisfies the existing constraints and time constraints. Some rules such as this one reflect firm policy. Ideally we design flexible enough layouts so that such events do not arise. Most of the automobile factory machine selection and layout rules fall into the following categories.

Machine and conveyor selection.

Layout : Conveyor placement
Machines sharing resources should be near to each other
Maximize space and machine utilization rules
Minimize material flow rules
Isolate undesirable effects
Constraints - Immovable objects.

Rules in the final five categories may be cast in general terms. The domain simply determines the parameters which we assign. The only domain dependent categories are conveyor placement and machine selection. The weight and size of an automobile demand substantial conveyor systems (or AGV's) and certain types of machines. But the placement of these machines follows a general set of rules. Some rules relate specifically to the conveyor systems. Many of the conveyor placement heuristics may be generalizable to other conveyor intensive factories.

The class of a factory layout, which lies in the range from batch production to dedicated assembly line production (see Appendix 2), is not a product related phenomenon, but rather an objective related phenomenon. It reflects the priority of objectives such as *maximize machine utilization* and *maximize flexibility.*

5.5.3 Parameter knowledge and Constraint Knowledge

We may consider the generality of the layout rules beneficial in that they may be applied to many different domains. However application of domain specific knowledge may make the problem more tractable. If there are no rules which are domain specific, assignment of parameters to domain independent rules may make a specific case of a problem easy to solve. For example, in the branch and bound method, if one partial placement has a much lower cost than the others, then extensions of other partial placements may not need to be investigated. Knowledge of the "flow parameters" is thus used to solve the problem. But can we really consider a parameter to be knowledge, if it only constrains the problem by virtue of its applicaion in a method such as branch and bound? It was stated in the summary of background work (Section 2.3) that knowledge based systems had not used industry specific layout knowledge. Systems such as IFLAPS, [Tirupatikumara, 1985], have used "parameter knowledge". The problem associated with coding industry specific layout knowledge is that it requires a well defined general layout system before the detailed layout knowledge for a specific industry can be added. Most prototype systems have not reached the level of sophistication which has allowed layout knowledge to be added. The system presented in chapter 7 is designed to illustrate a logic based approach to layout using a Generate and Test method. The general structure allows industry specific knowledge to be incorporated, and suggested industry specific layout rules are proposed, but they are not yet encoded and therefore it is not claimed that this is a "knowledge based" layout system.

We define two types of knowledge, *parameter knowledge* and *constraint knowledge*. Parameter knowledge is the knowledge which describes a general parameter of the problem such as the relationship between two departments. If the knowledge about departments is *decisive*, that is they must be adjacent, or they must not be adjacent, or they are constrained to fixed positions in an existing factory, then this knowledge is constraint knowledge. It acts as a direct constraint on the problem which helps to limit

the number of true interpretations of the model which we must compare for the optimal solutions. If the knowledge is *indecisive* then it expresses a parameter which is one of many factors in an optimization problem. For example each department flow is knowledge instrumental in deciding where to place the departments, but one department flow cannot be considered without addressing how this placement affects other placements and therefore other flows. This knowledge does not reduce the intractability of the problem. If we simply aim to minimize flow then all flows between departments are a parameter of the problem.

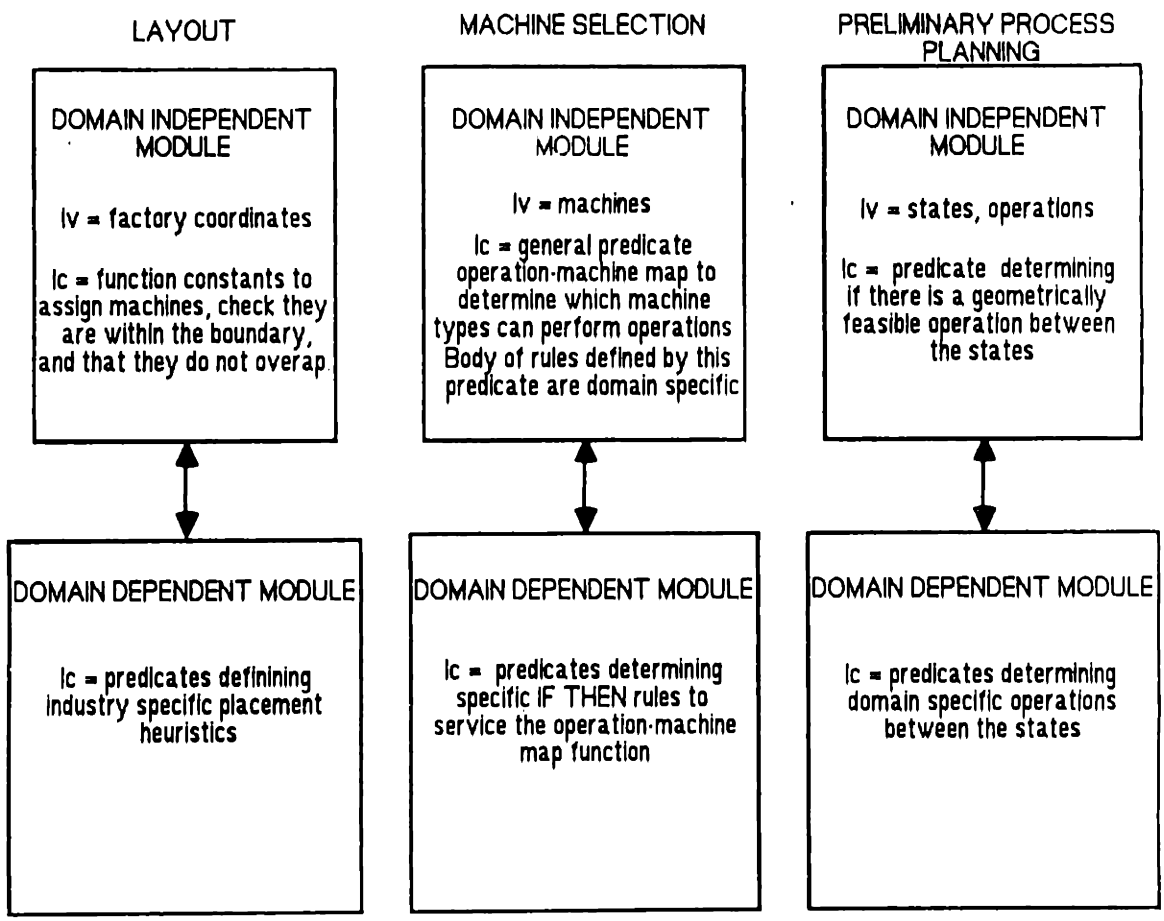
5.5.4 Expression of domain dependence and independence in the formal model.

The three models represented in table 5.2 are domain independent models which express the computability of each phase. Extensions to the models which enable us to determine which is the better interpretation, may be considered industry or application domain dependent, since what is judged to be a good interpretation depends upon the domain. Yet we may consider "minimize flow" to be a domain independent extension since all factories aim to minimize flow. Minimizing flow is one of several objectives common to many domains. The priority with which we address minimizing flow over other objectives will depend upon the domain. We consider these objectives to be domain independent because they may be represented in a software system by domain independent code which may be accessed by any domain specific module and assigned an appropriate priority. Further extensions to the model which relate solely to a particular industry are considered domain specific. Extensions which relate to a particular factory are case specific.

In terms of the formal structures which comprise the logic model, the domain of the logical model, D, and the variable assignments $\mathfrak{1}_v$ to be made are independent of the application domain. The individual constants and constant predicates, $\mathfrak{1}_c$, express both application domain dependent and independent knowledge. Therefore the formal representation cannot be completely decomposed in terms of domain dependence and independence. Figure 5.15

Figure 5.15

Domain Dependence and Independence of Logic Constructs



illustrates in terms of logical constructs, what would be represented in the domain dependent and independent modules of computerized systems for the three phases of factory design.

Our references to application domain dependence and independence so far have all referred to the field of factory design. Domain independent rules apply to all factories. Domain dependent rules apply only to the domain in question, say automobile factories.

5.5.5 Prospects for a General Expert System - Incompleteness.

If we were to build a general expert system with a domain independent structure so general that the assignment of the variables, $\mathfrak{l}_{\nu}$, would include assignment of predicate and function variables as well as individual variables, then our formal model would have to be represented in second order predicate calculus. However the increase expressive power afforded us by second order predicate calculus has a fundamental limitation implied by Godel's Incompleteness Theorem. The set of solutions is incomplete. That is, there are truths about the system which are expressible but not provable in this axiomatic theory. That is since we cannot necessarily find all solutions we cannot necessarily guarantee that we have found the optimal solution.

5.5.6 Implications of the Factory Design Model for Process Planning - Soundness and Completeness

Figure 1.4 showed the relationship of factory design to process planning and scheduling. Once a factory is designed and operating it dictates to a large extent the design and the resulting effectiveness of process plans and schedules, especially with regard to operation sequencing and queueing times. Since the factory design model can be expressed in propositional calculus it has the properties of soundness and completeness described in Chapter 3. The propety of soundness means that if a process plan is input to the factory model with the question, can the factory make a part with this plan, the model will be able to answer decisively, yes or no. The property of completeness, means that if a part description is input to the model, the model will be able to determine every possible operation sequence by which the

part could be made.

Summary of Chapter 5

This chapter showed that the three phases of factory design, preliminary process planning, machine selection and layout are all computable. A logic model was developed for each phase. The interdependency of the three models was shown. An interpretation of individual variables in the preliminary process planner was represented in the machine selection model as a set of individual constants. An interpretation of individual variables in the machine selection model was represented in the factory layout model as a set of individual constants.

It was shown that there are limits to the amount of domain dependent knowledge which can be used to lay out a factory. The functional requirements of the products to be made in the factory do not affect the layout stage. The layout stage depends only upon knowledge about machines, the flows between them, their characteristics and spatial constraint knowledge.

Two types of knowledge were described, parameter knowledge and constraint knowledge. Constraint knowledge reduces the number of solutions which are possible. Parameter knowledge merely expresses placement preferences and only reduces the number of configurations which we may wish to search by virtue of techniques such as Branch and Bound and A* (see Chapter 6). Unfortunately a large proportion of factory layout knowledge is parameter knowledge.

It was seen that we cannot decouple domain dependent and independent knowledge in terms of in terms of the classes of formal logical constructs D, $\mathfrak{l}_c$ and $\mathfrak{l}_v$.

When a large amount of factory layout knowledge is generalizable, a software model is easily transferable from domain to domain. But if there is little domain specific layout knowledge to constrain the number of different interpretations, evaluation of the optimal solution may be intractable.

CHAPTER 6

COMPLEXITY OF FACTORY LAYOUT

Overview

In chapter 2 we described factory layout as a combinatorial optimization problem. In chapter 4 we introduced the NP-complete class, to which such problems often belong.

In this chapter we prove that the Quadratic Assignment Problem is an NP complete problem in the number of departments, and that the general factory layout model introduced in Chapter 6 is NP-complete in the number of grid points.

We consider the implications of this by examining the size of the search space for these grid representations. We then consider how we might intelligently reduce the search space of possible layouts, by generating locations which appear promising, rather than searching a fixed grid. Having examined the possible representations, we examine the search techniques in the light of what we know about the likely distribution of good solutions to the layout problem. We examine the expected number of nodes expanded by these search techniques.

The favored approach is to evaluate solutions on their structure rather than on a cost evaluation function. The favored search procedure is a hybrid search strategy based upon informed depth first search with backtracking. We consider the best strategy for guiding our depth first search by examining the conjunct ordering problem.

6.1 NP-completeness of Factory Layout

6.1.1 Introduction

In Chapter 4 we introduced the concept of NP-completeness and discussed the nature of this class of problems which includes many combinatorial optimization problems. We now present a proof of NP completeness for the

General Factory Layout Problem introduced in the formal model of Chapter 5. This model was the basis of our decidability proof which showed that the problem was finite and hence computable. It was built from the tools of logic to explicate the finite properties of the factory object domain and its implications for knowledge based systems techniques. Having proved the finite properties of the problem we now recast this model in the mold of a known Operations Research (OR) model. This allows us to examine the complexity of finding an *optimal* solution, whereas before we were concerned with whether all solutions could be found (completeness). We could also express the optimization model in logic, but an OR formulation is preferred because this facilitates comparison with existing OR models such as the QAP.

6.1.2 Conditions for NP-completeness

Recall that a problem is NP-complete if

(a) the problem is in NP
(b) all other problems in NP polynomially transform to the problem.

For a problem to be in NP, we require that, if x is a *yes* instance of the problem then there exists a concise (of length bounded by a polynomial in size of x) certificate for x, which can be checked in polynomial time for validity.

6.1.3 Formal Definition of The Optimization Problem for Factory Layout

The Quadratic Assignment Problem was introduced in chapter 2. In chapter 5 we showed that the Factory Layout Problem (FLP) could be represented as a problem of assigning factory objects to a uniform grid. The grid spacing was much smaller than the smallest factory object and the boundary of the factory enclosed an area much larger than the combined area of the factory objects, so that it did not pose a constraint. We define the FLP as an extended version of the QAP.

In the QAP objects were assigned to grid elements, and all objects were the same size as grid elements. In the FLP an object covers many grid elements. The lower left corner of an element is its origin, or its grid point. In the FLP when we *assign* an object to an element, the lower left corner of the object is placed at the element's grid point. In addition to defining assignment we define the condition of covering. An object covers all elements whose grid points it overlaps. We now define the Factory Layout Problem as an extended version of the QAP. We refer to departments rather than objects, because in practice there would be so many objects to enter into a computer that the layout engineer is likely to work with departments, and then lay out the objects within each department.

Definition: the Factory Layout Problem (illustrated in figure 6.1)

Let

g = the grid spacing.

n = number of grid elements.

d_{jl}= the cost per movement or interaction over the distance from element j to element l.

f_{ik}= the number of moves per time period in the work flow from department i to department k.

a_{ij} = 1 if lower left corner of department i is assigned to grid point j,

0 otherwise.

Let,

the grid point of element j be its lower left corner, represented by coordinates (x_j, y_j),

the grid point of element j' to which department i is assigned be its lower left corner, represented by coordinates (x_i, y_i),

the length in the x direction of department i be l_i,

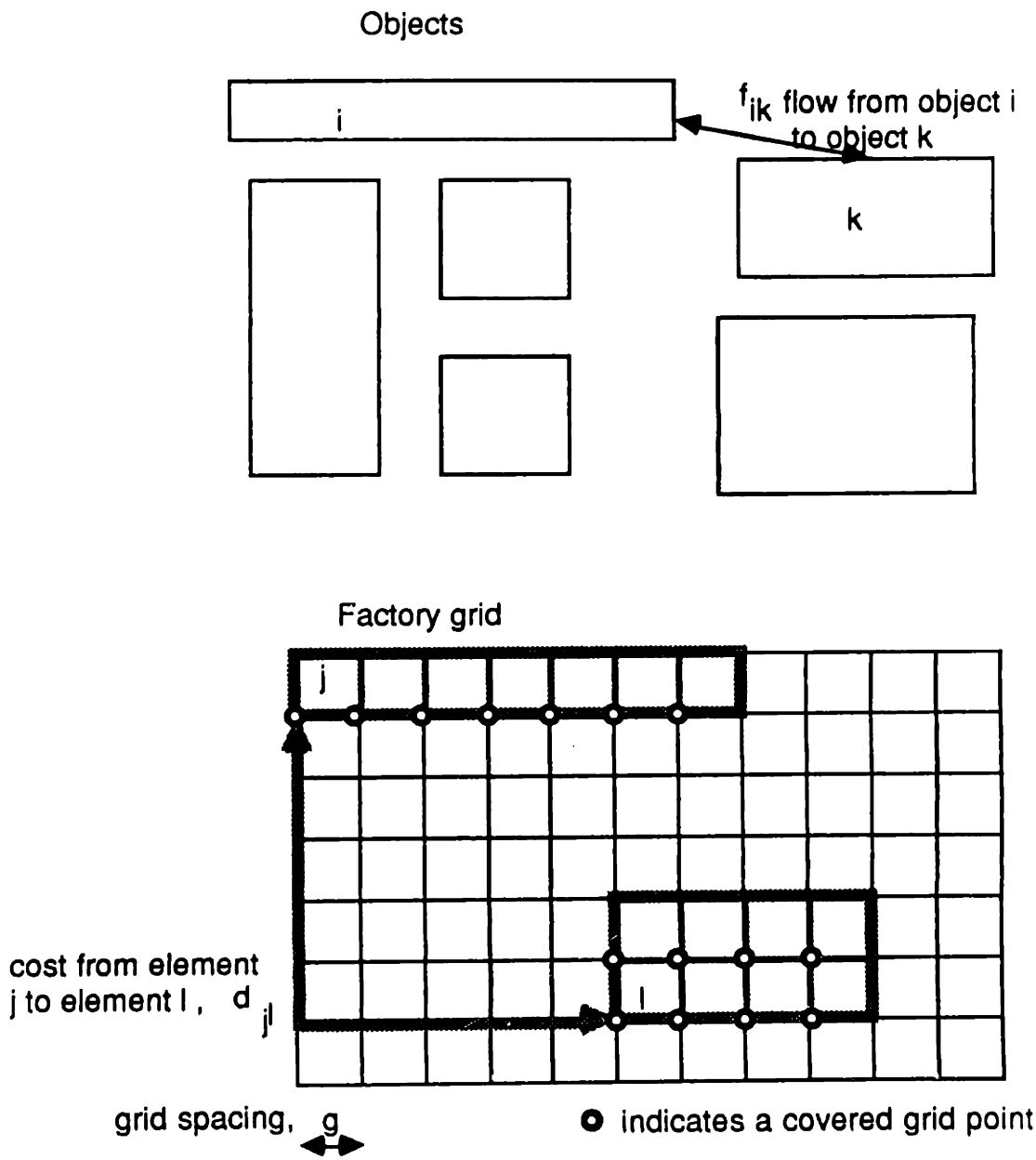
the height in the y direction of department i be h_i.

Department i assigned to element j' covers element j if

Figure 6.1

The Factory Layout Problem

Illustration of assignment of objects i and k
to elements j and l respectively.



$x_i \le x_j < (x_j + l_i)$ and

$y_i \le y_j < (y_i + l_i)$.

Let

c_{ij} = 1 if department i covers grid point of element j,

0 otherwise.

The cover condition is not satisfied if the grid point is not defined, (i.e. it lies outside the factory boundary).

The problem is to

$$\text{Minimize} \quad 1/2 \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \sum_{l=1}^{n} f_{ik} d_{jl} a_{ij} a_{kl} c_{ij} c_{kl} \qquad (1)$$

subject to

$$\sum_{i=1}^{n} a_{ij} \le 1, \qquad j=1,2,\ldots,n,$$

(at most one department may be assigned to location j)

$$\sum_{i=1}^{n} c_{ij} \le 1, \qquad j=1,2,\ldots,n,$$

(at most one department may cover a location i.e. no overlapping)

$$\sum_{j=1}^{n} a_{ij} \le 1, \qquad i=1,2,\ldots,n,$$

(department i may be assigned to at most one location)

$$\sum_{j=1}^{n} c_{ij} = \text{ceiling}(l_i/g) \cdot \text{ceiling}(h_i/g) \qquad j=1,2,\ldots,n,$$

(department must cover as many locations as its area dictates. This condition ensures it fits inside the factory boundary)

x_{ij}=0 or 1, i=1,2,...,n, j=1,2,...,n.

(Ceiling denotes the rounding upwards of a fraction to the nearest whole integer).

The QAP is a special case of the FLP in which all departments are the same size as grid elements. The cover conditon does not arise in the QAP because all departments cover only the grid point to which they are assigned.Therefore $c_{ij} = a_{ij}$. If we can show that the QAP is NP complete, then we can prove that the FLP is NP complete, because the QAP is a special simple case of the FLP and it can be transformed to the FLP representation in polynomial time.

6.1.4 Proof of NP-completeness for the QAP

In order to prove that the QAP is NP-complete we transform a known NP-complete problem, *the Hamiltonian Path Problem*, to the QAP. This proof is based upon that of [Sahni and Gonzales, 1979].

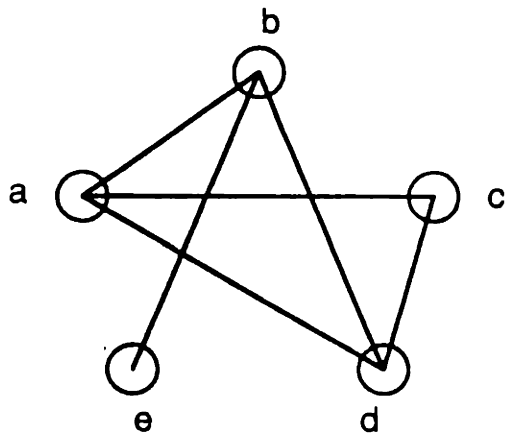
The problem of finding a Hamiltonian pathmay be stated as follows: Given a graph G(N,A), where N is the set of nodes in the graph and A is the set of edges in the graph, does it contain a path which visits each vertex exactly once?

The analogy between the two problems is depicted in figure 6.2. The nodes of the graph in figure 6.2(a) are represented by locations in the QAP of figure 6.2(b). In figure 6.2(c) values of 1 are assigned to edges in the original graph, and values of ω are assigned to virtual edges of the original graph. A valid Hamiltonian path travels only the edges of value 1, and does not travel any edge more than once (figure 6.2(c)). This is represented in the QAP by defining a fixed path between locations and assigning departments to the path such that all flows between adjacent departments along the path have value 1, (figs 6.2(b),(d)). If a flow of ω appears on this path, this is equivalent to a non valid edge being used in the graph to attain the Hamiltonian.

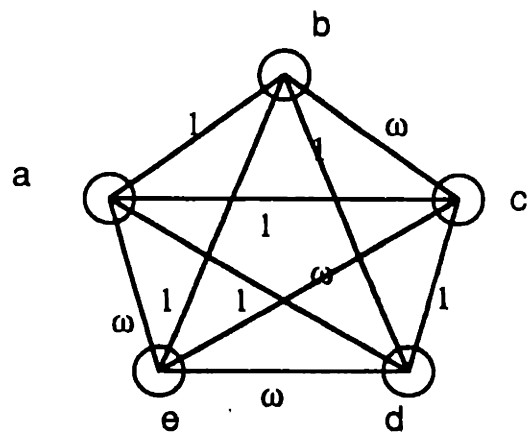
Figure 6.2

Representing the Hamiltonian Path problem as a QAP

Hamiltonian Path Problem:
Find a path of length n - 1 (where n is no. of nodes in graph),
such that each node is visited only once.

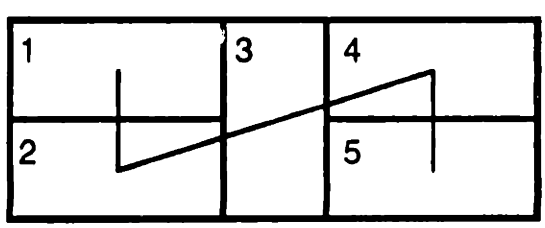


7.2(a) a graph

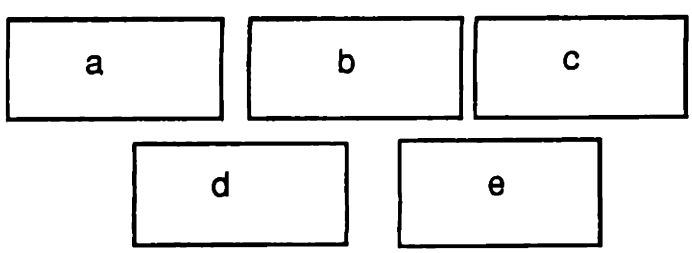


7.2(c) graph with costs to represent valid edges of 7.2(a)

The Hamiltonian as a QAP: Find an assignment of objects to locations such that the cost of the predetermined path in figure (b) is n -1.



7.2(b) predetermined path in layout representation



7.2(d) flow costs between objects to be placed, corresponding to costs of fig 7.2(c)
(only 1 values are shown)

Proof:

Let

G(N,A) be an undirected graph (as in figure 6.2(a)) with n nodes, where n = |N|.

The following QAP is constructed from G.
The flow from department i to department k,

$$f_{ik} = \begin{cases} 1 & \text{if } (i,k) \in A, \\ \omega & \text{otherwise.} \end{cases} \qquad \text{where} \quad 1 \le i,k \le n.$$

(figure 6.2(c),(d))

In the QAP any department is allowed to ship to any other department. Therefore we define costs for all possible edges. We assign cost 1 to flows between departments if a corresponding edge exists in the original graph G. We assign cost ω to flows for which G does not have a corresponding edge.

The transportation cost from location j to location l,

$$d_{jl} = \begin{cases} 1 & \text{if } l = j+1 \text{ and } j<n \\ 0 & \text{otherwise} \end{cases} \qquad \text{where} \quad 1 \le j,l \le n,$$

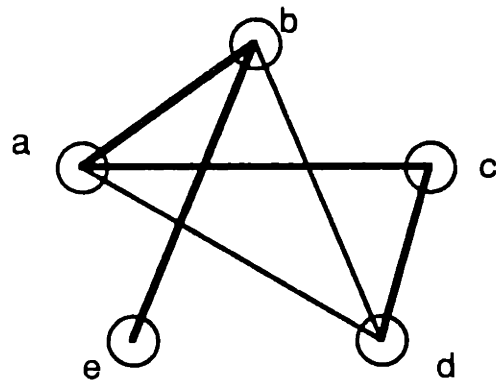
That is, we define the locations to be linked by a path which visits each location only once (figure 6.2(d)). A cost of 1 is assigned to neighboring locations in the path. Cost 0 is assigned to non neighboring locations. By doing this we exclude any flow from the final summation which does not occur in the path.

If we can assign departments to the locations such that all the flows of weight 1 and no flows of weight ω occur along the path, the total cost of the assignment is n - 1 and we have a Hamiltonian path in our original graph G. Figure 6.3 illustrates existence of a Hamiltonian path in the graph of figure 6.2(a).

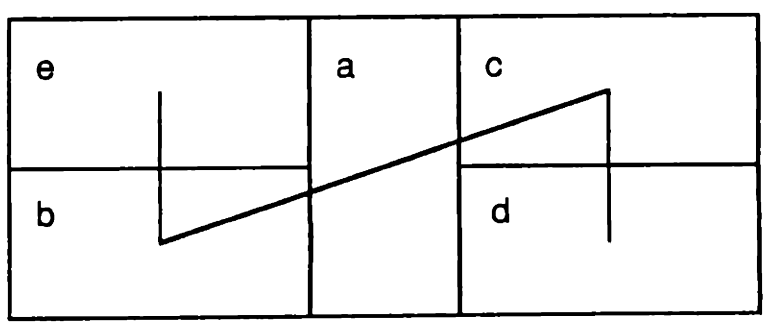
Figure 6.3

A Solution to the Hamiltonian Path Problem of Figure 6.2

Solution: Hamiltonian Path is e - b - a - c - d



Solution : Assignment is e-1, b-2, a-3, c-4, d-5



If the graph does not have a Hamiltonian, then the only assignment of departments that we can make to the fixed path of locations must have more than one flow of cost ω, i.e. the path has used an illegal edge of graph G and is not Hamiltonian.

(In the minimization of the sum of the product of D and F, sets of transportation cost between locations and flow between locations respectively, the least cost assignment has all pairs (0, ω) and (1,1)).

The total cost $f(\gamma)$, of an assignment γ of departments to locations is

$$\sum_{i=1}^{n-1} f_{ik} d_{\gamma(i)\,\gamma(k)}, \qquad (2)$$

where $k = i + 1$,

when $\gamma_{(i)}$ is the index of the department assigned to location i. The condition k=i+1 defines the path between all locations which visits each location only once.

If G has a Hamiltonian $i_1, i_2, \ldots i_n$, then the assignment $\gamma_{(j)} = i_j$ has a cost

$$f(\gamma) = n - 1.$$

If G has no Hamiltonian path then at least one of the values $d_{\gamma(i)\,\gamma(i+1)}$ must be ω and so the cost becomes $\geq n + \omega - 2$.

Therefore the Hamiltonian Path problem can be polynomially transformed to the QAP. Since the Hamiltonian path problem is NP-complete, the QAP is at least as hard as the Hamiltonian. Any assignment of locations can be validated in polynomial time. Therefore the QAP is NP-complete.

End of Proof

6.1.5 Proof of NP-completeness for the Factory Layout Problem

Proof:

Any QAP can be transformed to a FLP in polynomial time. This is so because the QAP is simply a special case of the FLP in which all departments are the same size as the grid elements. That is the FLP is at least as hard as the QAP. Furthermore, any layout can be validated in polynomial time. Since the QAP is NP complete, so is the FLP.

End of Proof

6.2 Are there solutions for intractable problems?

Although we have shown that the class of Factory Layout problems is intractable, it does not mean we should not explore further. The Simplex algorithm for Linear Programming has also been shown to be NP-complete, and yet it is used in practice to solve large problems in linear time. Although the class may be intractable, using domain specific knowledge we may be able to eliminate large protions of the search space. We consider four facets of the problem which must be explored.

1. Representation:
   What is a realistic practical representation?
   How does the search space vary when using this representation?

2. Problem Domain:
   What is the nature of the particular problem?
   Are there likely to be a large number of near optimal solutions?
   Are all the solutions at the same depth of the search space?

3. Optimizing vs Satisficing:
   Should we look for an optimizing or a satisficing solution?

4. Search:

What is the best way to search for the solution?

6.3 Choosing a representation

We must first compare the number of nodes in the search space for plausible representations of the layout problem.

6.3.1 Quadratic Assignment Problem.

This is a special case of assigning departments to a grid. Each department is assigned to one element of the grid (see figure 2.5(a)).

Let D= number of departments to be assigned.

Then the search may be represented by a tree of depth D, and uniformly decreasing branching factor (figure2.5(b)). In each level of the tree a new department is placed. The branching factor of the tree at each level represents the number of available locations, which decreases as departments are assigned to grid elements.

The number of possible configurations, C, is:

$$C=D! \tag{3}$$

and is equivalent to the number of leaf nodes. This is equal to the number of distinct paths of length D, through the tree.

The total number of nodes expanded to generate all possible configurations is

$$N = \sum_{j=1}^{D} \frac{D}{(D-j)!} \tag{4}$$

where (D - j) represents the depth of the node layer from the root of the tree. In the third layer of the tree when D is 3 for the 9 department case, there 9x8x7 nodes. This is given by 9! / (9-3)! . N represents the summation of the number of nodes in all the layers of the tree.

6.3.2 Non Uniform Assignment Problem
(A literal Computer representation of the FLP)

A more general case of assigning departments to a grid would not limit the department shape and area to the shape and area of the grid element. The grid elements are generally smaller than the departments, such that assigning the corner of a department to a grid coordinate is more akin to placing departments in continous space. We call this Non Uniform Assignment (NUA).

We assume that the total area of the departments is approximately equal to the area within the factory boundary, and that all the departments fit neatly together within the boundary so that we do not encounter a packing problem. The Non Uniform Assignment Problem is depicted in figure 6.4.

Let

Average department area = a

Average department side length = $\sqrt{a}$

Grid Spacing = g

Then the average no. of grid points per department area, n_a, is given by

$$n_a = a/g^2$$

The total number of grid points in the factory is n_aD. The first department has a choice of n_aD grid points. Once it has been placed, the second department has a choice on average of n_a (D - 1) grid points an d so on. The search tree for the NUA problem is shown in figure 6.5.

Figure 6.4

The Non Uniform Assignment Problem

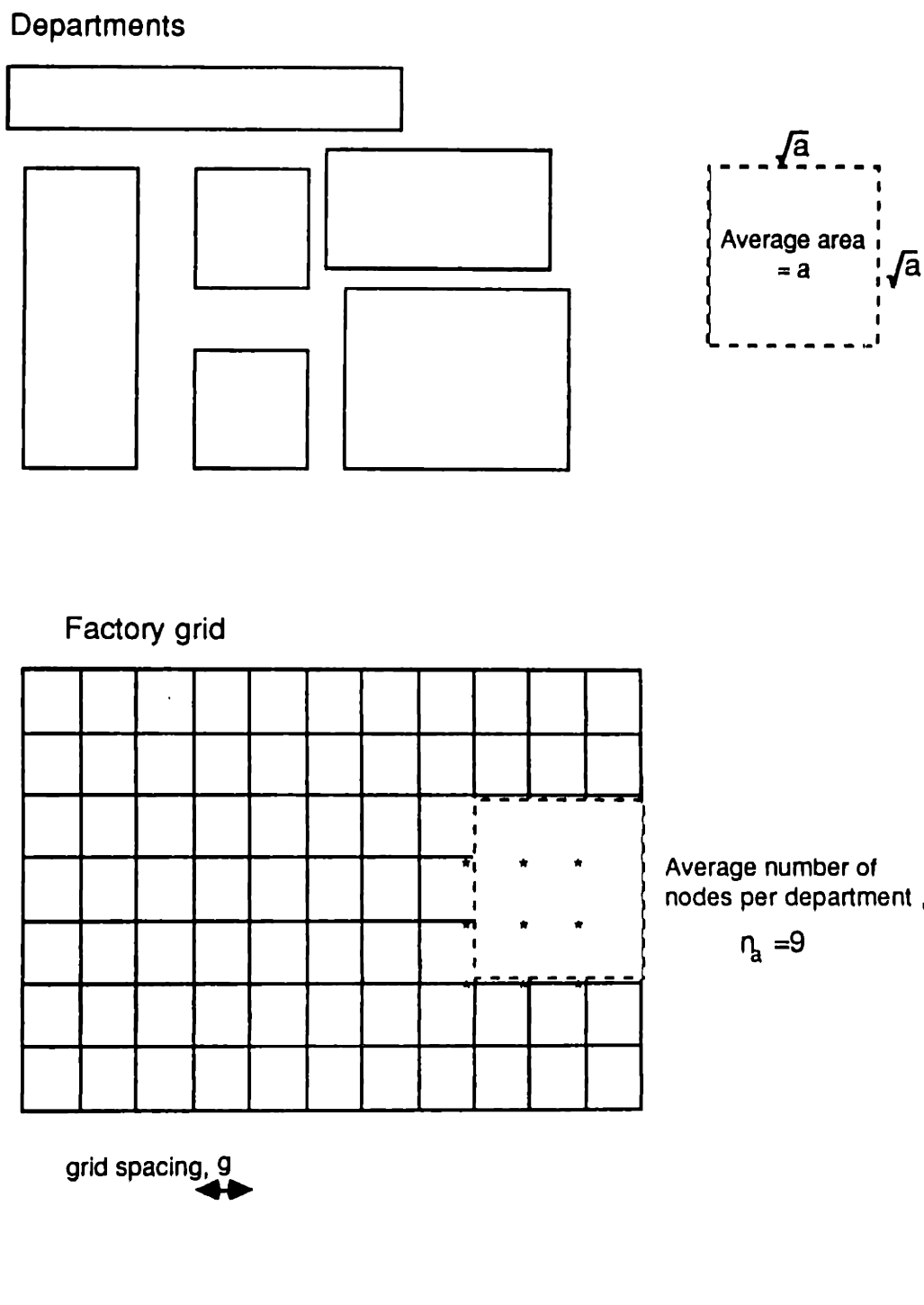
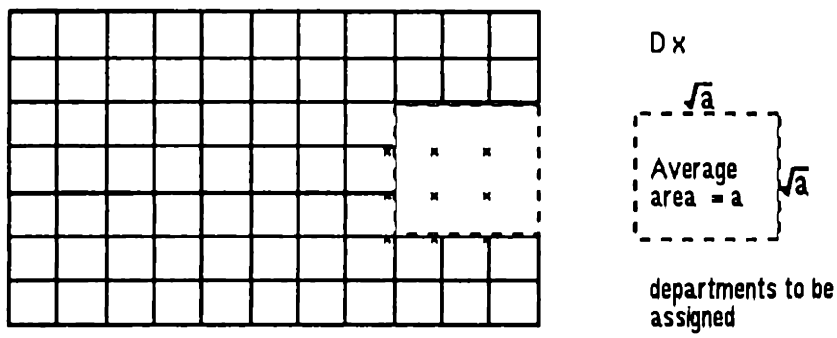


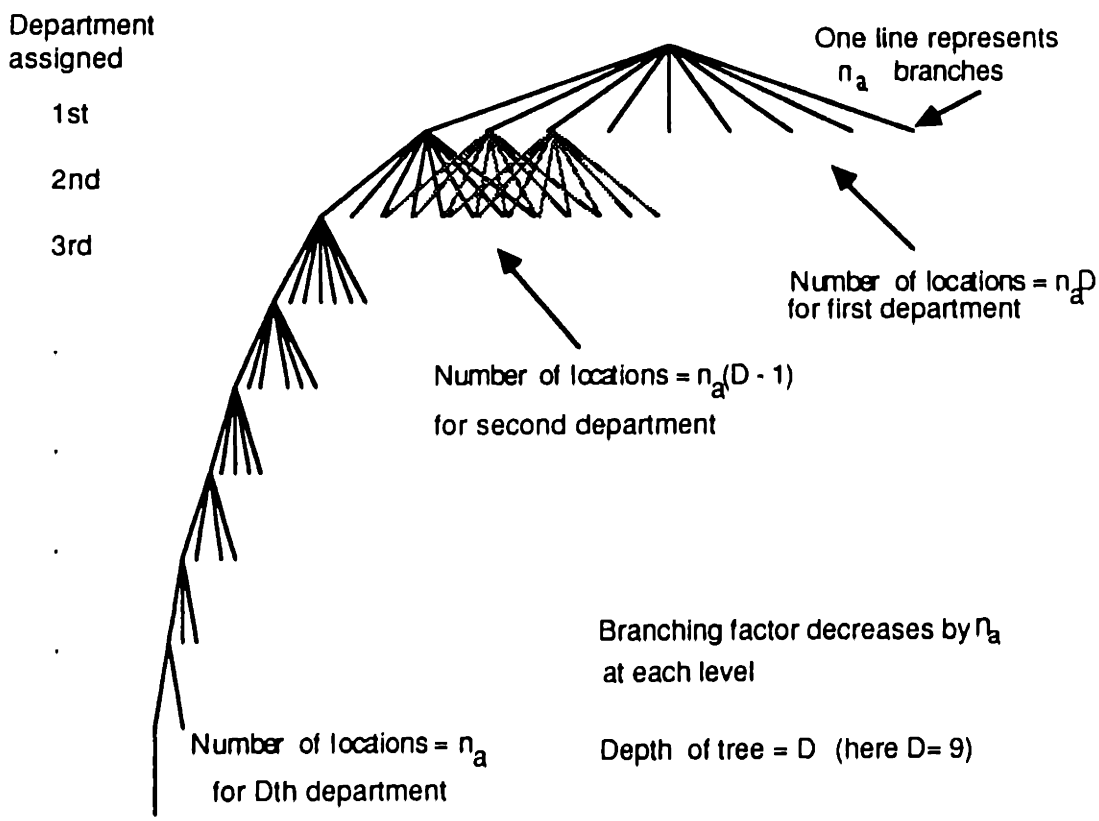
Figure 6.5

Representing the Non Uniform Assignment Problem for Search

(a) The Non Uniform Assignment Problem Grid and Average Department



(b) Search Tree for the Non Uniform Assignment Problem



The number of possible configurations, C, is given by

$$C = n_a D \bullet n_a(D-1) \bullet n_a(D-2) \bullet \ldots\ldots\ldots\bullet n_a(D-(D-1))$$

$$C = n_a{}^D D! \tag{5}$$

The total number of nodes expanded to generate all possible configurations is

$$N = \sum_{j=1}^{D} \frac{(n_a D)!}{(n_a (D - j))!} \tag{6}$$

NB If the factory area and the sum of the departments are not approximately equal, we can represent the factory area by a virtual number, D', of average sized departments. Then

$$C = n_a D' \bullet n_a(D'-1) \bullet n_a(D'-2) \bullet \ldots\ldots\ldots\bullet n_a(D'-D)$$

$$C = \frac{n_a D'!}{(D' - D)!}$$

6.3.3 Practical Implications of the Rate of Growth of the Search Space for the Grid Representations.

Even the search space of configurations for the simple formulation of the QAP is exponential in D. An optimizing solution of the QAP is computationally infeasible for more than 15 departments. If we improve the representation to allow location of different sizes and shapes of departments on a discrete grid, the number of possible configurations grows by a factor of $n_a{}^D$. A reasonably expressive grid spacing would have say five nodes along the side of an average sized department, giving, n_a = 25. The more expressive

representation would be computationally infeasible for a smaller number of departments than the QAP.

6.3.4 Implications for Improved Sequential Machines and Parallel Machines

An increase of computing power by a factor of ten would only allow us to increase by one the number of departments for which we could solve the QAP. Table 6.1 [Lewis and Papadimitriou] illustrates that the size of the largest instance solvable by a polynomial time algorithm will be multiplied by a constant between 1 and 10, if we can achieve a ten fold increase in computing power. By contrast, exponential algorithms will experience only an additive increase in the size of instances which they can solve.

The 'depth' of a computation to find a configuration is at most D, the number of departments to be placed. The number of configurations which are generated is D!. Parallel machines may therefore seem to offer an advantage over sequential machines. In theory the costs of all configurations could be computed in parallel in the time it takes a sequential machine to evaluate one configuration. However the space required by a parallel machine would grow exponentially with the number of departments.

6.3.5 The Generate and Test Approach

There are two things to bear in mind. The size of the search space is not a given, in one sense, because we may use heuristics to decide which parts of the search space to evaluate. And in the second sense, because we may generate an appropriate search space as we proceed, rather than beginning with a fixed grid. This leads us to consider a generate and test method of layout.

6.3.6 Adjacent Location Assignment Problem

The layout problem is, in essence, a problem of accommodating as many department relationships as possible. (If there were no relationships we could solve the linear assignment problem). The intractability arises from the conjunctive nature of the problem, which is to satisfy several relationships

Table 6.1

The Growth of Polynomial and Exponential Functions

[Lewis and Papadimitriou]

Function	Approximate values		
Polynomial			
n	10	100	1000
$n \log n$	33	664	9966
n^3	1000	10^6	10^9
Exponential			
2^n	1024	1×10^{30}	1×10^{301}
$n^{\log n}$	2099	2×10^{13}	8×10^{29}
$n!$	3,628,800	10^{158}	4×10^{2567}

Table 6.2

Polynomial-time algorithms take better advantage of technology

Function	Size of Instance solved in one day	Size of instance solved in one day in a computer 10 times faster
Polynomial		
n	10^{12}	10^{13}
$n \log n$	1×10^{11}	1×10^{12}
n^3	10^4	2×10^4
Exponential		
2^n	40	43
$n^{\log n}$	79	95
$n!$	14	15

simultaneously. The majority of relationships will express desired proximity; the overall tendency is for departments to attract and cluster , rather than to repel. If we select a department for placement, according to its relationships with other departments already placed, then a placement which minimizes distance will always be on the perimeter of the placed departments, assuming that the perimeter of the placed departments is largely convex (see figure 6.6). The clustering tendency should render this convex property inevitable, especially if we choose to place departments with a large number of neighbors first.

This motivates a method of location generate and test based upon perimeter placement.

We assign departments to perimeter nodes of placed departments.
Let,

n_p = average number of perimeter nodes per department, and
g = grid spacing.

Assuming square departments, the perimeter length is given by

$$p = 4\sqrt{a} \tag{7}$$

and the average number of perimeter nodes is

$$n_p = \frac{p}{g} = 4\frac{\sqrt{a}}{g} \tag{8}$$

while the average number of area nodes is

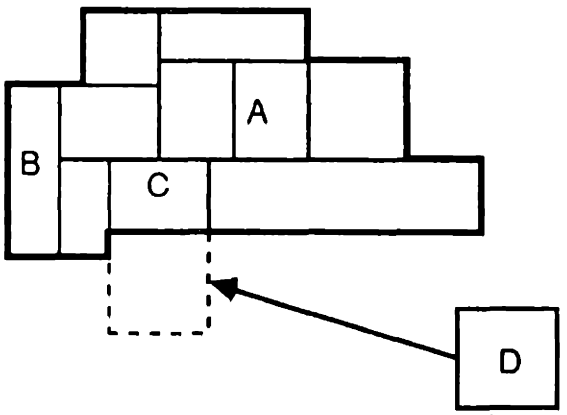
$$n_a = \frac{a}{g^2} \tag{9}$$

We derive the ratio of perimeter nodes to area nodes as a function of the

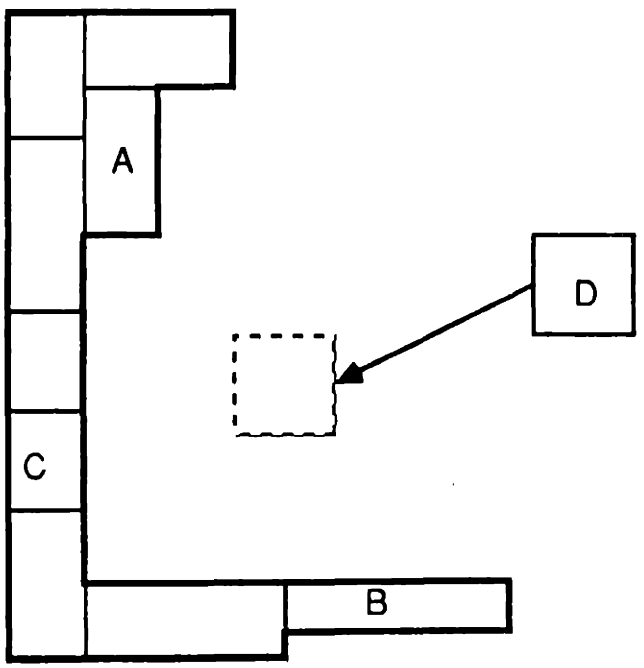
Figure 6.6

Illustration of Convex and Concave Perimeters

(a) Placement on a largely convex perimeter



(b) Placement for a concave perimeter



average area of a department and the grid spacing. The ratio of perimeter nodes to area nodes is given by

$$\frac{n_p}{n_a} = \frac{\sqrt{a}}{4g} \qquad (10)$$

Perimeter nodes and area nodes are depicted in figure 6.7. When the average department side is greater than four times the grid spacing ($\sqrt{a} > 4g$), there are more area nodes per department than perimeter nodes.

Let us assume that a criterion for selecting the next department for placement exists (e.g. department with maximum flow to the departments already placed), and assume that the first department is a fixed reference point.

Then the number of configurations possible using adjacency placement is equal to the product of the number of ways of placing the second department and the third department, through to the Dth department.

$$C = n_p1 \cdot n_p2 \cdot n_p3 \cdot \ldots\ldots\ldots\ldots \cdot n_p(D-1)$$

$$C = n_p^{D'}\ D'!\qquad \text{where}\quad D' = D-1 \qquad (11)$$

This is a pessimistic estimate. It assumes that the number of perimeter nodes increases as the sum of the individual perimeter nodes of the placed departments. This is illustrated in figure 6.8. Since the departments will cluster, a more optimistic and realistic estimate is to assume that the the perimeter grows as a function of the total area of the placed departments. An optimistic estimate with predetermined department ordering is

$$C = \frac{4\sqrt{a}}{g} \cdot \frac{4\sqrt{2a}}{g} \cdot \frac{4\sqrt{3a}}{g} \ldots\ldots\ldots\ldots \frac{4\sqrt{d'a}}{g}$$

Figure 6.7

Area Nodes and Perimeter Nodes

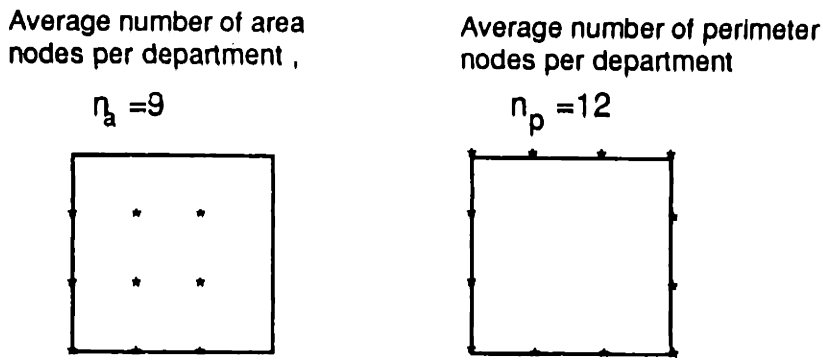
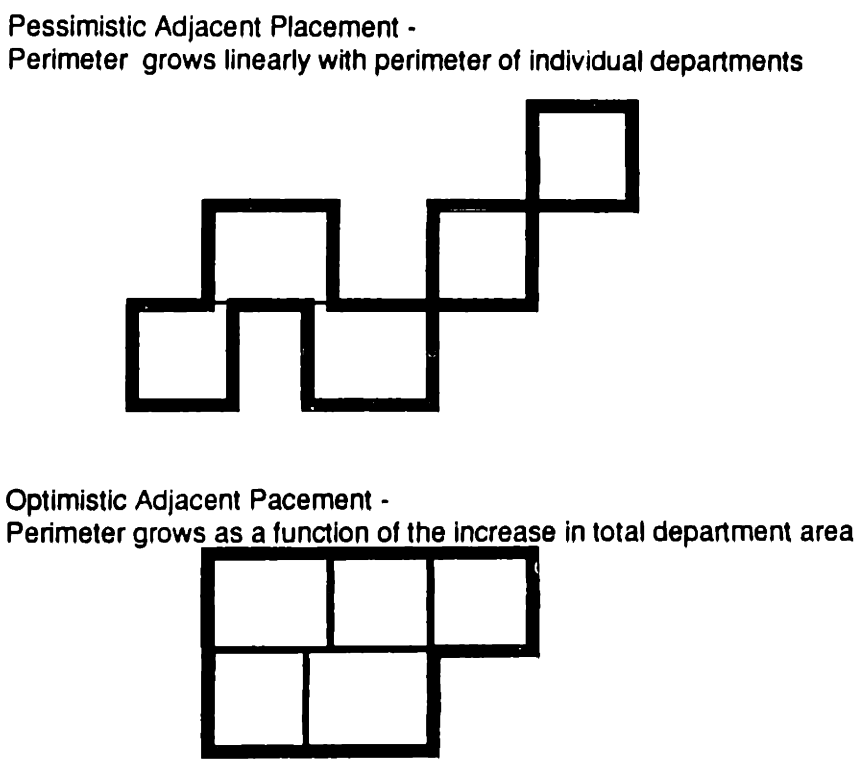


Figure 6.8

Illustration of Optimistic and Pessimistic Estimates of Perimeter Length



$$C = \left[\frac{4}{g}\right]^{D'} \sqrt{D!\, a^{D'}} \qquad (12)$$

It is illustrated in figure 6.8. The adjacent location search generates a tree of depth D', of increasing branching factor. The rate of increase of the branching factor, b_j, (where j is the depth of the branch) depends upon our assumption about the rate of growth of perimeter nodes. Taking the optimistic scenario,

$$b_j = 4\frac{\sqrt{ja}}{g}$$

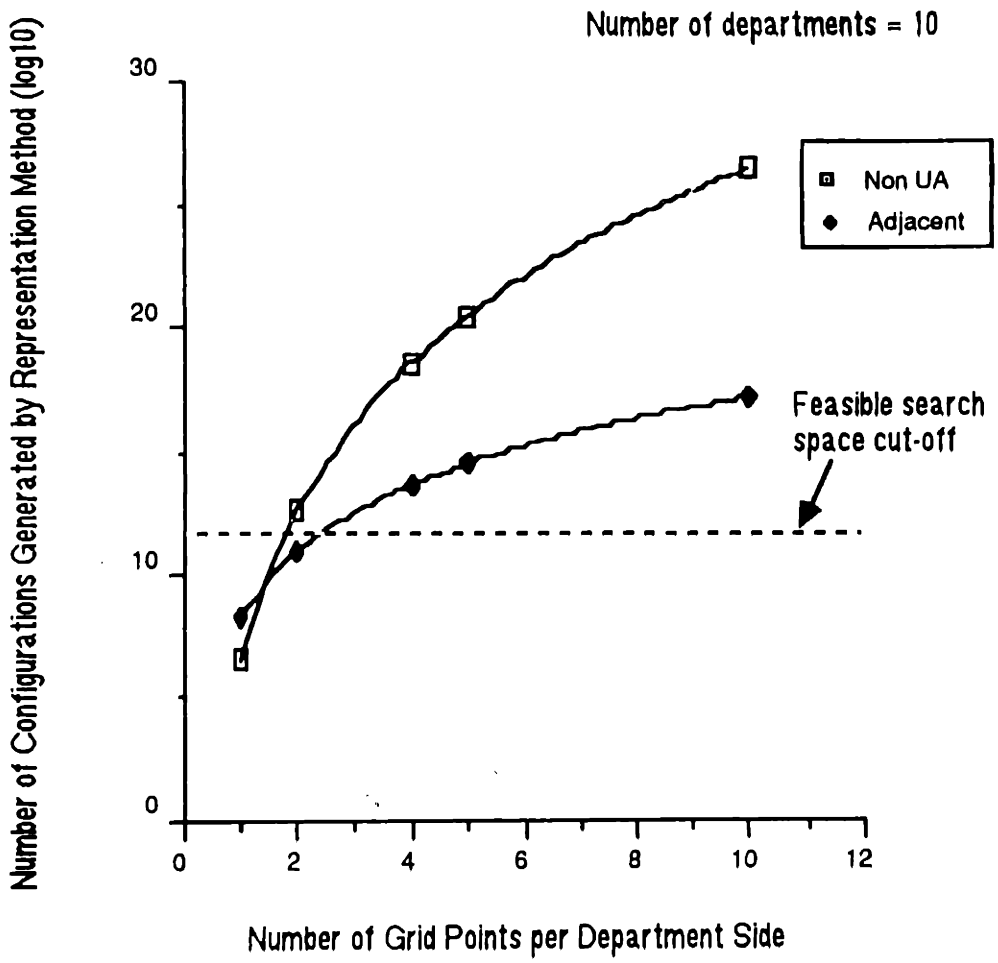
We compare the number of configurations for *non uniform assignment* and *optimistic adjacent placement*

$$C_{NUA} = \left[\frac{a}{g^2}\right]^{D} D! \quad \text{and} \quad C_{Adjacent} = \left[\frac{4}{g}\right]^{D'} \sqrt{D'!\, a^{D'}}$$

The net effect is that the number of configurations for the *non uniform assignment* (NUA) grows as $a^D D!$, whereas *optimistic adjacent placement* (Adjacent) grows as $\sqrt{a^{D'}D'!}$, assuming g is unity. Figure 6.9 shows how the size of the solution space, (the number of configurations) varies for the ten department case as we vary the number of grid points per average department side. If $\sqrt{a}/g > 2$, which realistically it should be to take advantage of the non uniform assignment grid, it is more efficient to use adjacent placement. (Note that adjacent placement does not generate every conceivable placement, but that by our reasoning it generates all the useful ones). But when $\sqrt{a}/g > 2$ the number of configurations exceeds the number which a standard computer could calculate in one day for the ten department case. We therefore make one further assumption to reduce the number of configurations:

Figure 6.9

Variation of Solution Space with Grid Point Spacing for Two Representation Methods



6.3.6.1 Desired Adjacency Placement

Assume that if we can not place a department next to its desired neighbor then we satisfice and place it in any available location. If we try to place each department on the perimeter nodes of just one placed department , the number of possible configurations is

$$C_{Adjacent1} = n_p^D = \left[4\frac{\sqrt{a}}{g} \right]^D \qquad (13)$$

This is a pessimistic estimate. It assumes that we always search all the perimeter nodes of the placed department and that there is no information to advise us that some may already be covered.

Figure 6.10 shows how the size of the solution space, (the number of configurations) varies for the ten department case as we vary the number of grid points per average department side. The figure compares three representations, NonUniform Assignment (NonUA), the optimistic adjacency placement (Adjacent) and adjacency to one placed department (Adjacent1). Figure 6.11 shows how the size of the solution space varies for a fixed number of grid points per average department side (one), as we vary the number of departments.

The maximum number of configurations that could be considered in one day by a computer is of the order of 10^{12}. This cut off point is shown on all figures. Even for single adjacency search and only two grid points per side, the largest number of departments which can be considered is approximately fifteen, (figure 6.11). If we wish the solution to be at all interactive, requiring a solution time of less than one day, and we wish to take advantage of more expressive representations, we must either sacrifice optimality or search more intelligently or both.

Figure 6.10

Variation of Solution Space with Grid Point Spacing for Three Representation Methods

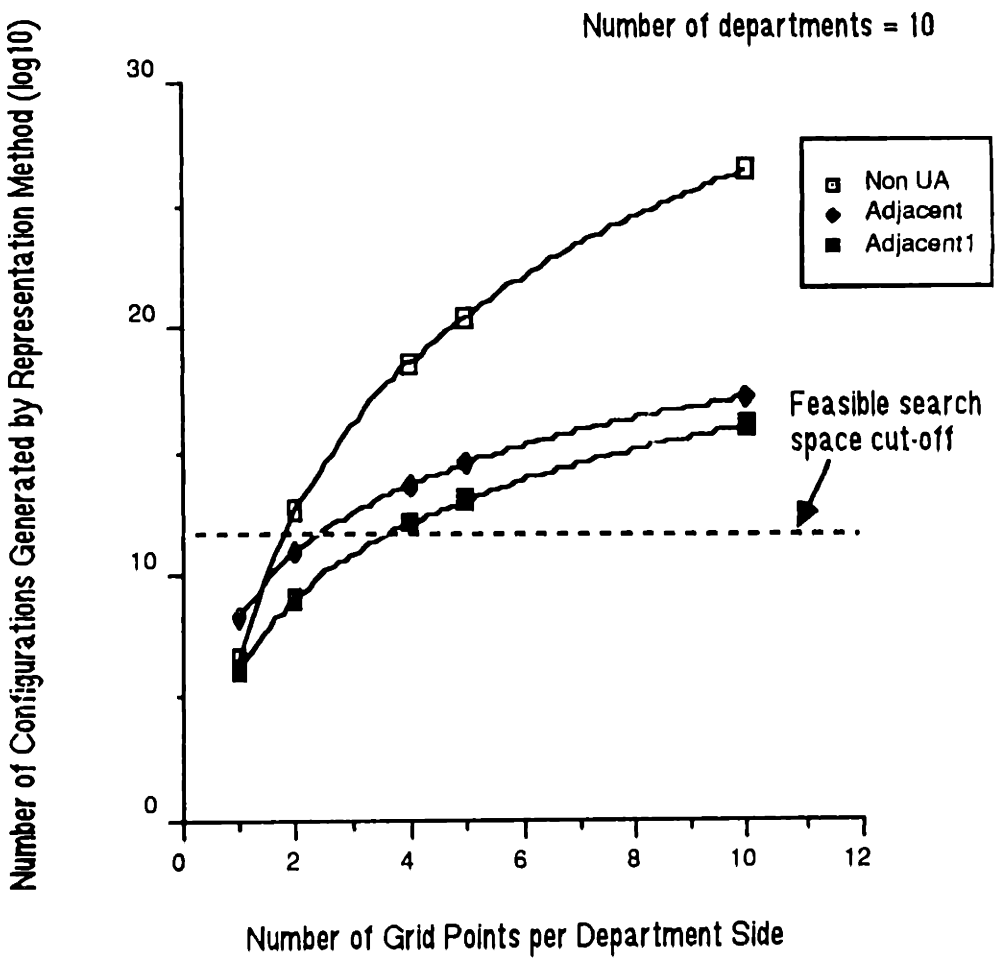
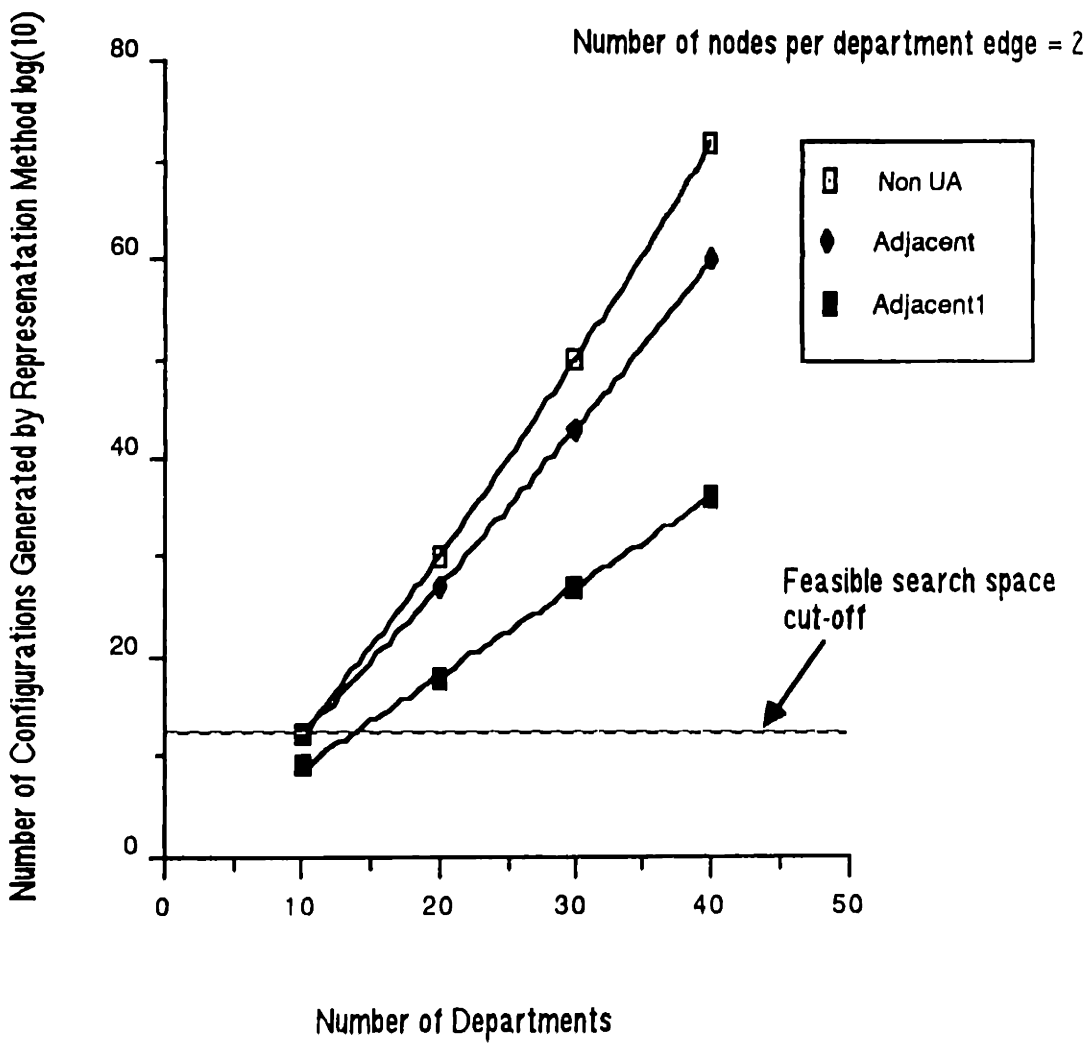


Figure 6.11

Variation of Solution Space with Number of Departments for Three Representation Methods



6.4 Search Techniques

In this section we consider search techniques for pruning the search tree, and evaluate their appropriateness for factory layout. The simple technique of hill climbing was discussed in section 2.2.3.4.

6.4.1 Breadth First Search

Figure 6.12(a) shows the path of a breadth first search through a tree. The nodes representing solutions are f and j. The breadth first search strategy visits all nodes on a shallower depth of the tree before proceeding to nodes at a deeper level. The shorter solution (the path to node f) is found before the longer one (to node j). This is a good strategy if solutions are likely to occur in the shallower depths of the tree.

6.4.2 Depth First Search with Backtracking

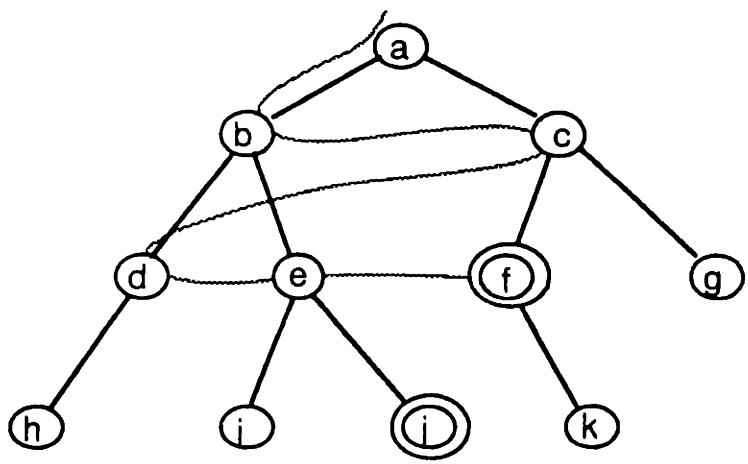
Figure 6.12(b) illustrates the depth first search strategy on the same tree of figure 6.12(a). The depth first strategy always visits a deeper node until it reaches a terminating node. If this is not a solution node it backtracks to the nearest decision point with unexplored alternatives and proceeds depthward again. It finds the leftmost solution first. For this reason the longer solution path to j is found before the path to f. This strategy is preferred if the solutions are all distributed at deep levels of the tree, especially if the branches of the tree are arranged in such a way that the more promising paths are situated towards the left of the tree. This is known as informed depth first search. The tree is shown with all the nodes generated. In Depth First Search with backtracking the nodes are generated as the solution proceeds. At each decision point, only one node is generated. If this node fails another node is generated.

6.4.3 Best First Search

A* is a best first search algorithm. It is a modification of breadth first search

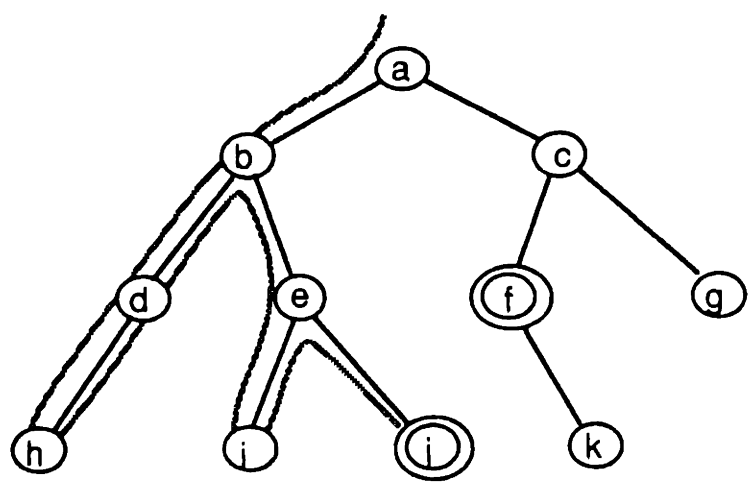
Figure 6.12

Breadth First and Depth First Search



(a) Breadth First Search

a is the start node, f and j are goal nodes. The order in which the breadth first strategy visits the nodes in this tree is : a,b,c,d,e,f. The shorter solution [a,c,f] is found before the longer one [a,b,e,j].



(b) Depth First Search

a is the start node, f and j are goal nodes. The order in which the depth first strategy visits the nodes in this tree is : a,b,d,h,e,i,j. The solution found is [a,b,e,j]. on backtracking, the other solution is discovered: [a,c,f].

which uses a heuristic estimator to evaluate which node should be expanded. The heuristic estimator is given by

$$f(n) = g(n) + h(n) \qquad \text{where} \quad g(n) \geq g^*(n) \quad h(n) \geq h^*(n).$$

The function g(n) evaluates the cost of the solution from the start node s to the node n. The function h(n) estimates the cost from intermediate node n to a goal node. We expand the node with the lowest total value of f(n). If the estimate h(n) is close to the optimal value, and many of the possible search paths have much larger values than the optimum, then the nodes on these search paths will not be expanded before the optimal solution is found.

An A* algorithm that uses a heuristic function h which always underestimates, such that for all nodes n in the state space

$$h(n) \leq h^*(n)$$

is admissible (that is, it always produces an optimal solution (minimizing)).

6.4.4 Hybrid Strategies

The strategies of hill climbing, backtracking and best first search can be viewed as three extreme points in a continuous spectrum of search strategies. It is convenient to characterize search strategies along the following dimensions[Pearl, 1983]:

Recovery of pursuit: the degree to which a search strategy allows recovery from bad choices to reaccess previously suspended alternatives.

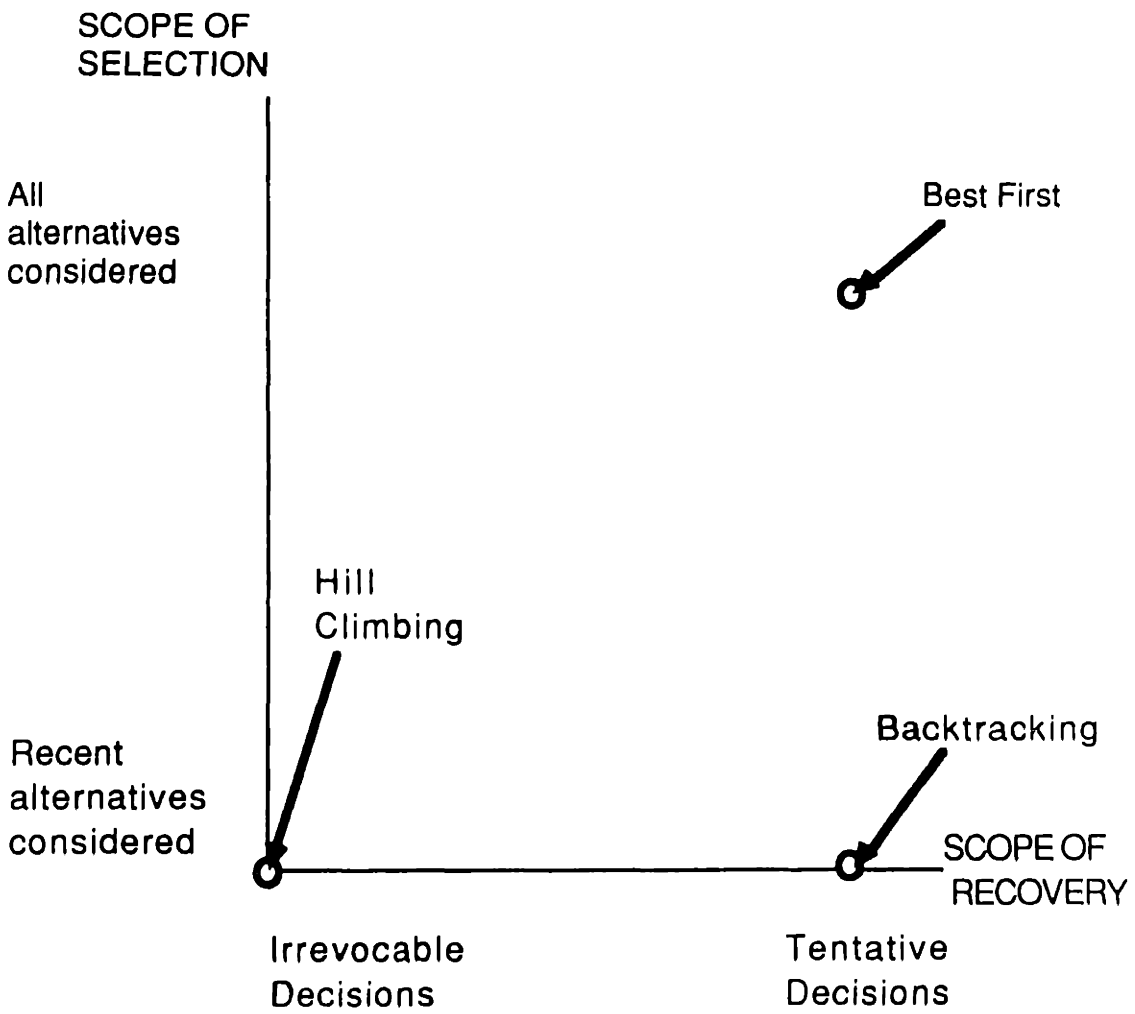
Scope of selection: The number of alternatives considered in each decision. This is linked directly to storage requirements.

The extreme cases are illustrated schematically in figure 6.13.

Hybrid strategies address the need to cut down the storage requirements of

Figure 6.13

Hill Climbing, Backtracking, and Best First Search as Three Extremes in the Space of Hybrid Strategies



Best First strategies at the expense of narrowing the evaluation scope. They allow some local search to determine a good candidate for depth first search. They prune the search space in such a way that an optimal solution is not guaranteed, but they generally produce good solutions in linear average time. Implementation of such a strategy for factory layout is discussed in section 6.8.

6.5 Expected number of nodes expanded by search techniques

The search space sizes presented for comparison of representations do not consider the probability of node survival. Using optimizing techniques with evaluation functions, we can rule out many candidate search paths. Pearl defines the expected number of nodes expanded by the various search techniques discussed above. This is done by estimating the probability at each level of the tree that a fruitless node will be expanded. Calculations are done on these probability estimates. Even though the search spaces for A* and backtracking grow exponentially with the number of departments, the search space size for a practical problem may be realistic if the modifying probabilities of node survival are small enough. Unfortunately for complex problems it is very difficult to estimate the probabilities. It will be some function of the number of desired adjacencies and the number of departments already placed. The search spaces for hill climbing and staged search are linear in the number of departments. We consider the search space for staged search in section 6.8 .

6.6 Distribution of Solutions in the Factory Layout Problem and Appropriateness of Search Techniques

Branch and Bound procedures have been found computationally infeasible for layouts of more than 15 departments. Therefore in any realistically sized search space we must look to satisficing solutions. In the factory layout problem, this means that since all the solutions lie at the same depth of the tree, where all the departments are placed, we must sacrifice search of the breadth of the tree in order to reach the leaf nodes of the tree. It is difficult to

generalize the probable distribution of good solutions to the layout problem. It depends upon the flow characteristics of the departments. However we can argue informally in terms of flow cost, that for a large number of departments where there are no very large dominating flows there are likely to be a large number of solutions spread evenly across the width of the leaf nodes. If there are large dominant flows, then the solutions will be clustered in one area of the leaf nodes which descend from partial solutions where the dominant flow departments were placed adjacent. In this case the probability of finding these solutions through random search is small but if we use heuristics to guide the search then they will certainly choose these large flow departments for adjacent placement early in the search procedure. Therefore guided depth first search of the tree should lead us to good solutions.

6.7 Analysis of the factory layout problem

For a simple analysis of E(Z) for a factory layout problem, consider the tree shown in figure. A choice of placement is either bad or good. Since we have some method of choosing the department and placement location, we asume that the probability of a good placement is reasonably high. Say

$$P(good) = 0.8$$
$$P(bad) = 1\text{-}P(good) = 0.2$$

The probability that we pick the optimal configuration is very small

$$P(optimal) = P(good)^d$$

For d= 20, P(optimal) is 0.06

The probability of a near optimal configuration scoring at least S,

$$0 \leq S \leq d$$

is given by the binomial distribution:

$$P(\text{score} \leq S) = B_{p,D}(S) = \sum_{i=0}^{S} C_i^D \, p^i (1-p)^{D-i}$$

Figure6.14 shows a binomial distribution for 20 departments.

The probability that S>14, that is that more than 14 of the department choices are good is greater than 95%. The skew of the binomial depends on the probability p. For p=0.5 the distribution would be symmetric about 10.

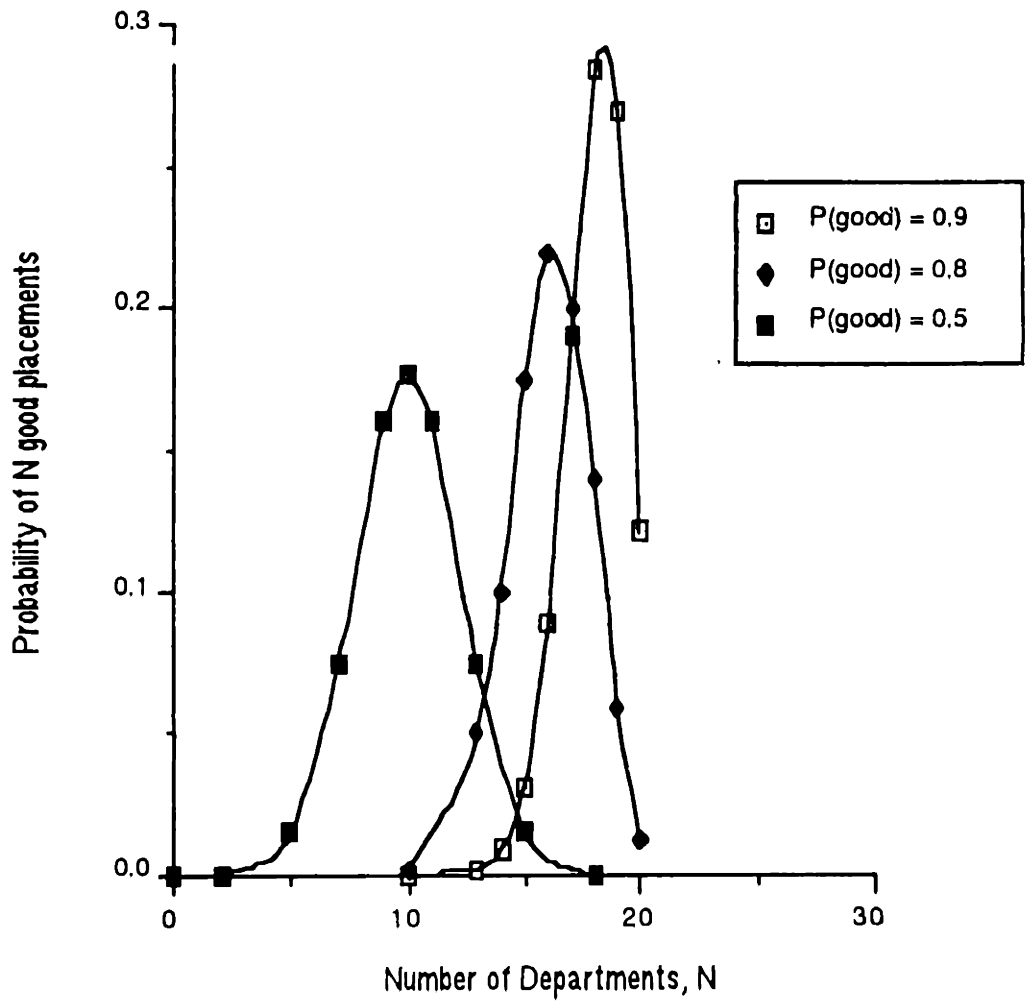
Thus even if we have no facility to backtrack, if we have some reasonable criteria for selection and placement of the next department, we should be able to generate a reasonable layout. This is why deterministic programs, such as CORELAP have had some success, especially when they were modified to allow interaction.

The preceding argument assumed that the effects of a bad choice were local and that a good layout was one with mostly good choices. However what may seem to be a good choice for a high flow department, placed early in the layout procedure (at a high level of the tree), may have adverse affects on the placement of several lower priority departments, ancestors, whose cumulative effect is greater than that of the high flow department. (This is why one might want an interactive capability). The combinatorial character of the layout problem renders detection of such cases as difficult as the layout problem itself.

We can reduce the search space significantly however, if we assume that a poor department placement will manifest itself in failure to successfully place an ancestor within a certain number of levels. This is tantamount to allowing a limited amount of local search to verify a placement before committing to it. Since our selection criteria for departments will place those with the highest priority first, (we shall refer to a high priority department as being more

Figure 6.14

Binomial Distributions for a 20 Department Layout



senior), if we set the limit on backtracking t four levels, we prevent a department failing five levels lower than a senior department from attempting to reconfigure it. In such cases, the department is placed in a non optimal location. This kind of hybrid strategy has the net effect of increasing the probability P(good), and therefore increasing the skew of the binomial distribution, so that the probabilty of a good solution is higher. It is known as Staged Search.

6.8 Staged Search

Staged search with a limit of three levels of backtracking is illustrated in figure 6.15.

The maximum number of nodes searched is given by

$$N = b^n(D - n)$$

where n is the number of levels of backtracking allowed and b is the constant branching factor. This is linear in D, the number of departments. For level dependent branching factor, b_i, the maximum number of nodes searched is given by

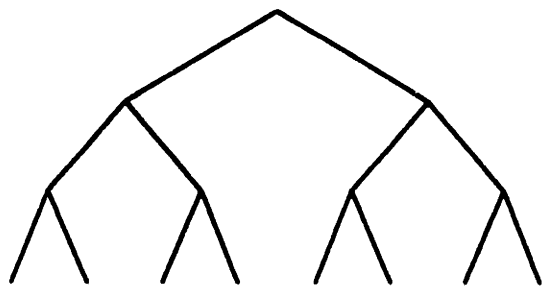
$$N = \sum_{i=1}^{D} \prod_{j=i}^{i+n} b_j$$

Figure 6.16 shows how the search space grows with the number of levels of backtracking for the twenty department case for two constant branching factors (bf), bf=32 and bf=16. The branching factor represents the number of perimeter nodes of existing departments at which we could place the next department in the sequence. With n, the number of levels of backtracking, equal to 5, only $O(10^7)$ nodes are expanded for a constant branching factor of 32. It is likely that checking for successful placement five levels deep before committing to the department placement in the twenty department case,

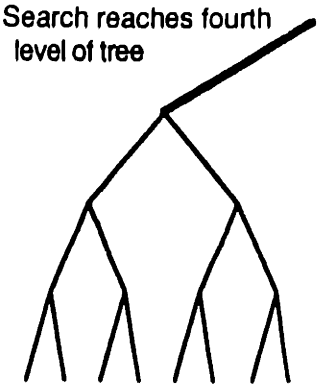
Figure 6.15

Illustration of Staged Search in a Tree of Depth 5 with 3 Levels of Backtracking.

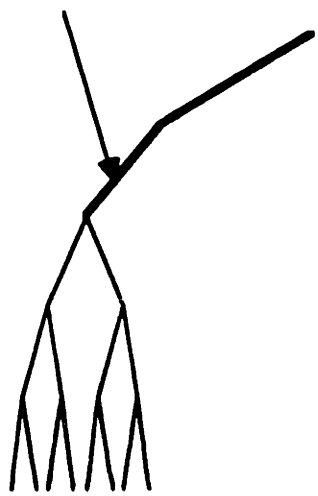
(a) First three levels of tree



(b) Search reaches fourth level of tree



(c) Bold lines indicate irrevocable decisions. At this stage, backtracking cannot alter this placement choice.



(d) Dashed lines indicate portion of search space which was never explored.

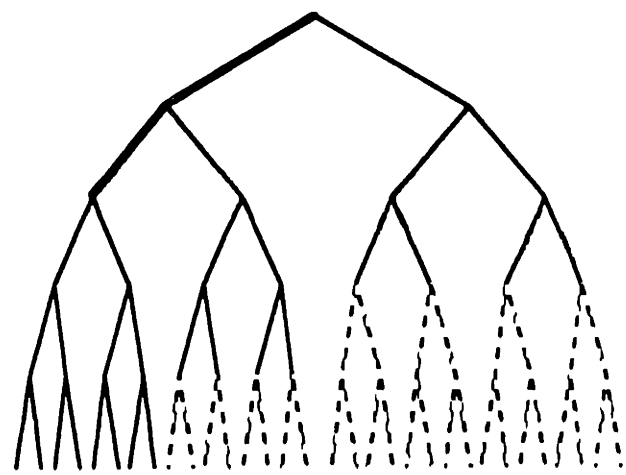
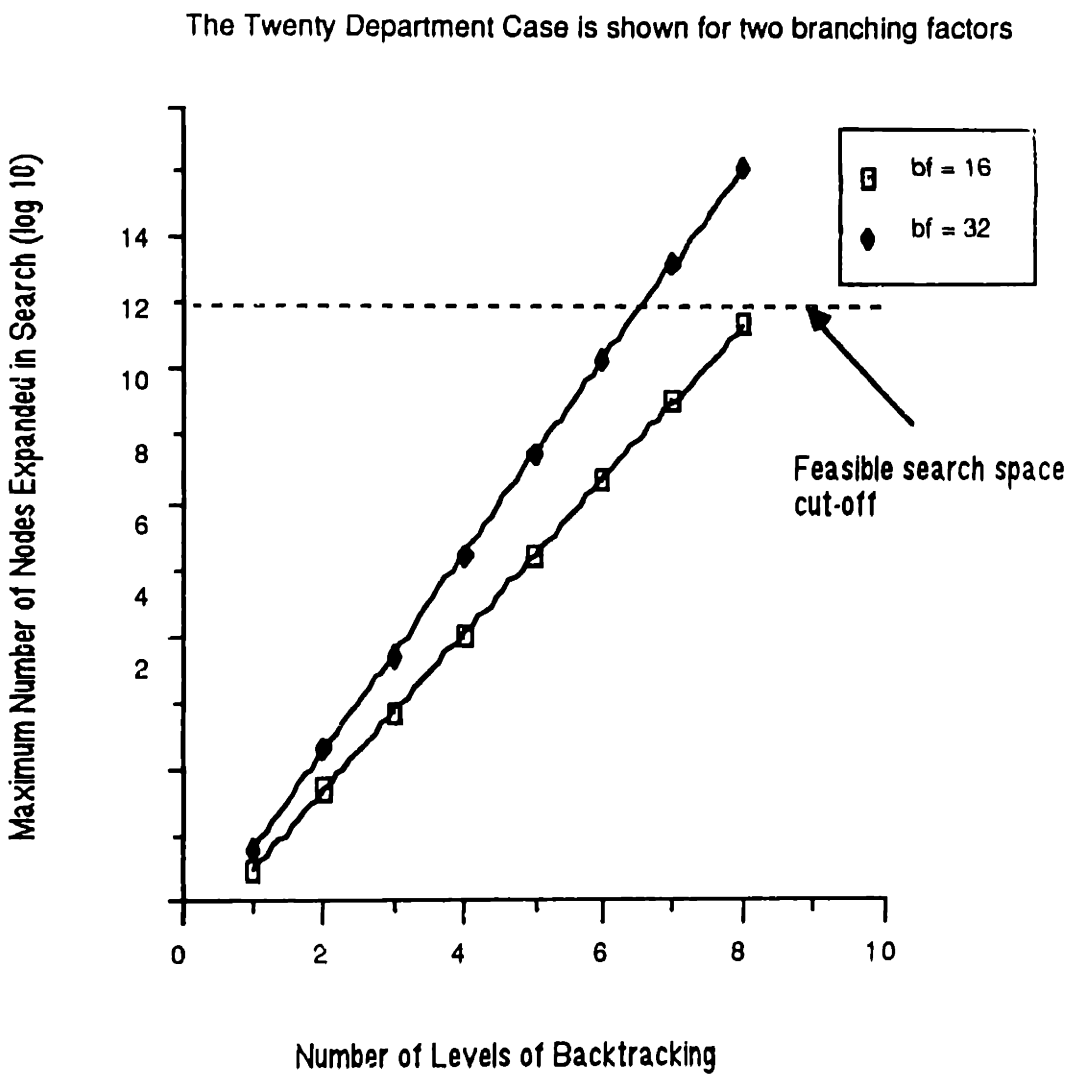


Figure 6.16

Growth of Search Space with Number of Levels of Backtracking for a Hybrid Strategy



should improve the probability of a good placement and the final solution significantly.

Unfortunately this method has disadvantages too. All the partial paths in the backtracking region of the current node must be stored. This storage amounts to b^n nodes. It requires a considerable amount of control structure to be programmed, by nature of its hybrid qualities. Implementation in any kind of programming language with a built in breadth first or depth first search mechansim is awkward. Logic programming languages such as Prolog, (with a built in depth first search strategy) , are designed to allow expression of logic sentences in a programming style. They do not lend themselves to manipulating the logical sentence structure for control purposes. When facilities, such as the cut (see section 3.4.5), are provided out of programming necessity, they can alter meaning of the program, and violate the properties of the logic model which we set out to express as a program. Ideally we want a program which finds all solutions, but presents the best ones first. Then we attain the properties of soundness and completeness, but we do not have to generate several poor solutions before we generate a good one. A hybrid strategy with such properties would require impractically large storage requirements and a more complex control structure than a simple satisficing hybrid strategy.

6.8.1 Forward strategies versus Backward strategies.

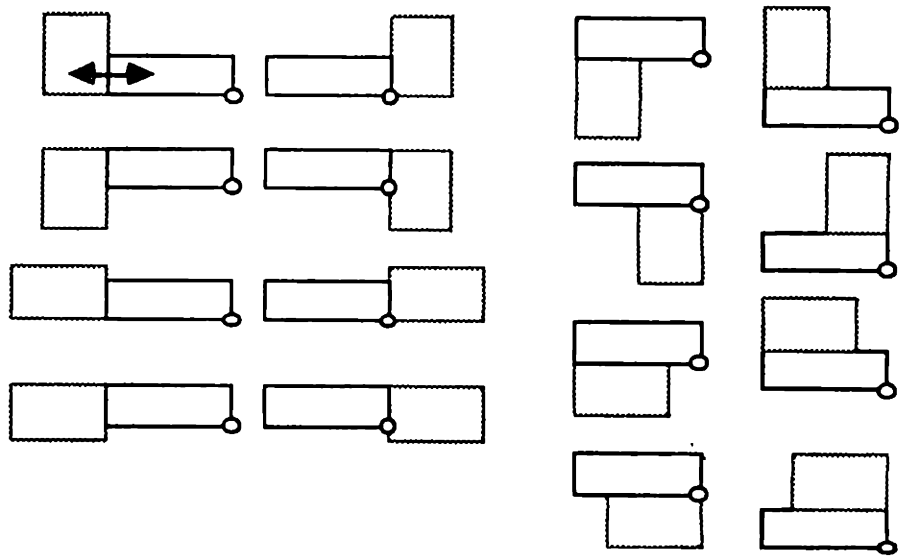
So far all the search techniques that we have discussed use *backward strategies*. *Forward strategies* allow us the benefits of staged search with none of the problems of the control structure.

In a forward placement strategy, a set of placement methods are stored, as shown in figure 6.17, and we try the appropriate method of placement for the adjacencies desired. We store all the ways in which two departments which need to be adjacent may be placed by aligning their edges (figure 6.17(a)). We store the way three department relationships, in which two departements which ship to each other and also to a third department may all be placed

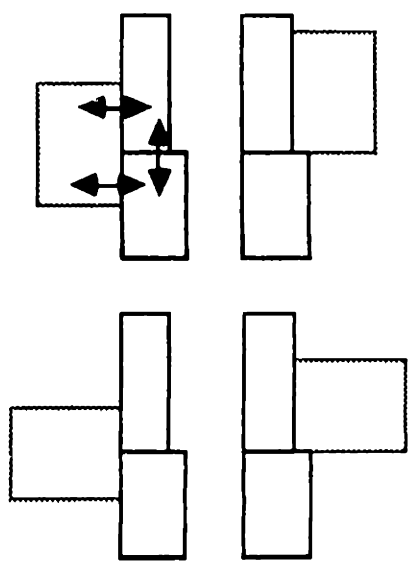
Figure 6.17

Adjacent Department Placement for Forward Strategies

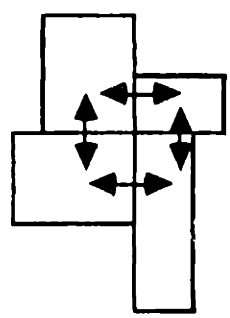
Two Departmnet Adjacency Placement Strategy -
Align Department Edges



Three department Adjacency Placement Strategy



Loop Adjacency Strategy



Arrows denote flows between adjacent departments

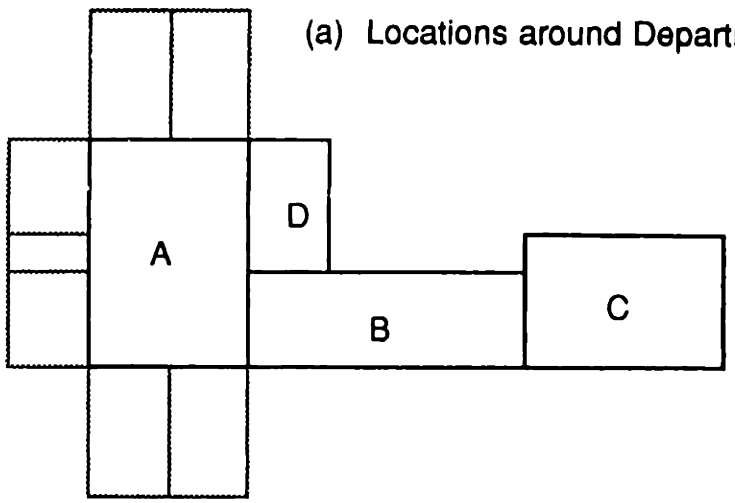
adjacent (figure 6.17(b)). etc. If three way adjacency fails for three departments, A,B and C which all need to be adjacent to each other, we try to achieve a pair of two way adjacencies, in which A is adjacent to both B and C but B and C are not adjacent to each other. Failing that we try to achieve just one two way adjacency.

That is we use local backtracking as in staged search, until we find a solution. Because of the ordering of the placement methods, the first solution that we find is the best. This technique allows us to compound a set of locally optimum configurations into a layout, as in staged search, without storage of partial solutions.

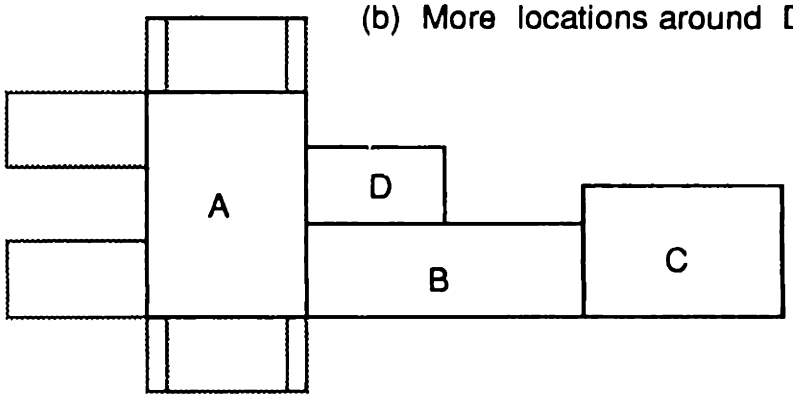
In a backward strategy there are no predetermined placement methods. We only compound placements of the two department adjacency type. We place single departments next to departments which have already been placed. We evaluate the best placement by number of adjacencies achieved, or minimum flow cost to departments already placed. We may wish to retrieve previous placements of lower cost. For example in figure 6.18 we would like to achieve the best placement of department D given that A, B and C are already placed. D ships goods to A, B and C. Since we do not have three or four way adjacency stored , we try to place D next to one department, say A. There are fourteen possible distinct placements of which two achieve incidental adjacency with B as well as A (see figures 6.18(a) and(b)). Since using only two way adjacency we cannot predict whether by chance we shall achieve a good third or fourth adjacency which makes one placement better, we must search through all possible single adjacency placements and evaluate them. We must store at least one partial solution, (the best one so far), for comparison. To achieve still a better solution with D adjacent to C as well as A and B we would have to alter the placement of C, as in figure 6.18(c). That is we would have to have stored all partial solutions to C so that we could backtrack to find all the methods of changing C. If we used a forward technique which stored four department placement methods this would not be necessary.

Figure 6.18

Backward Placement Strategies -
Searching for The Best Location for Department D

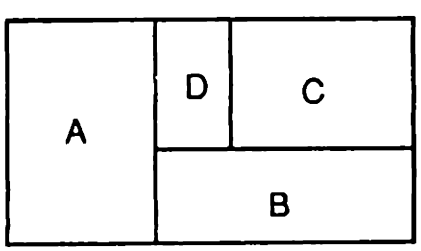


(a) Locations around Department A



(b) More locations around Department A

(c) The largest number of adjacencies to D are achieved by backtracking to relocate C



The limits to the forward technique become apparent.

1.In addition to defining three way placement methods with department A we could attempt to find four and five department adjacencies to A and to each other But this would mean storing a large number of complex placement methods against which to check. The complexity of the placement methods increases with the number of departments.

2. Instead of local optimization among department A's subordinates, we could attempt to satisfy the subordinates of A's subordinates when we place A 's subordinates. If we extend this forward strategy to the point where all departments are considered, this would amount to storing as many predetermined configurations as there are configurations found by backtracking search.

6.9 Evaluation of the Partial Solution - Measure vs Structure of the Layout.

We may evaluate a partial solution by measuring the flow cost, or we may keep track of the structure of the solution, the number of adjacencies achieved. Departments are either adjacent or not adjacent. If we ask the question "Are all adjacencies achieved?", the answer is a definitive "yes" or "no". This enables us to define *perfect solutions*, those which achieve all desired adjacencies. Note that we reduce the amount of information by reducing department relationships to a binary one, adjacency or non-adjacency. All non-adjacencies are equally bad, and all adjacencies, no matter which faces are adjacent, are equally good. In order to examine the complexity of the search in the factory layout problem, defined in logic, it is useful to examine the properties of search for a perfect solution. The search for a perfect solution is easier than for an opimal one because the value of a perfect solution is known, (all adjacencies achieved), whereas the value of an optimum solution in terms of flow cost is not.

6.10 Conjunct Ordering

6.10.1 How do we 'inform' backtracking search intelligently.

It was determined that the most suitable combination of representation and search technique for the factory layout problem is an adjacency placement generate and test method using informed depth first search with backtracking. This motivates the question, how do we inform the search?. That is how do we decide which department to try and place first. Nominally we discussed informed depth first search in terms of placing the departments in order of size of flow to other departments. However, although our objective may be to minimize flow, placing the highest flow departments first may not be very efficient.

Note that backtracking search will find every possible configuration subject to the imposed constraints. The ordering of the nodes for expansion simply determines which solution we find first. If we simply state that all departments must be placed, we find all possible arrangements which fit within a given boundary. If we state that all departments must be adjacent to all other departments to which they transport material, then we are asserting a constraint which finds only perfect solutions. That is we extend the logic model of section 5.4.2. However it is likely that, in a factory with a reasonable amount of interconnectivity, such a strict constraint would arrive at no solutions. We may wish to compromise by trying to place a department next to its desired neighbors, allowing only n levels of backtracking, after which we satisfice and place the department in the best nearby location. Note that the solutions developed by this approach will depend upon the department placement ordering, because the backtracking cut off prevents generation of all solutions. In essence, we have imposed a procedural constraint. Alternatively we may use a forward strategy for staged search. This placement method will generate all solutions with backtracking.

We can gain considerable insight into the layout problem by assuming that there exists a solution in which all desired adjacencies are accomplished, and posing the question, "Which is the best way to order the departments for

placement such that the solution is found most efficently? ".

If we are to find all desired adjacencies, then the flow cost between departments is irrelevant. It is important that a department is next to all of its neighbors. In order to minimize the amount of backtracking, we aim to place objectives which are likely to conflict as near as possible to each other in the tree, so that backtracking is as localized as possible. Departments with common adjacencies are most likely to conflict because of the clustering phenomena. Therefore rather than ordering departments according to flow it may be more appropriate to order them according to common neighbors. We explore the best method of informing the search by considering further the conjunct ordering problem.

6.10.2 The Conjunct Ordering Problem

Definition: A conjunctive problem is a set of propositions which share variables which must be satisfied simultaneously.

6.10.2.1 The Cheapest First Heuristic

The conjunct ordering problem is to find the ordering of propositions which minimizes the solution search space. One heuristic commonly used to order propositions is the *cheapest first heuristic*, illustrated by the following example. | P(X) | = 1000 indicates that there are 1000 possible solutions to the proposition P(X).

Example:

| P(X) | = 1000

| Q(Y) | = 2000

| R(X,Y) | = 100,000

| R(x,Y) | =100

$$| R(X,y) | = 10$$

Upper and lower case letters denotes uninstantiated and instantiated variables respectively. Using the cheapest first heuristic with no variables instantiated the cheapest propostion is P(X). With X instantiated to x, the next cheapest proposition is R(x,Y). Choosing this ordering of literals, the size of the solution space is

$$|P(X)| * |R(x,Y)| = 1000 * 100 = 100,000.$$

But if we had chosen a more expensive literal first, Q(Y), we could have chosen a cheaper second literal, R(X,y) = 10, and the overall solution search space would be smaller:

$$|Q(y)| * |R(x,Y)| = 2000 * 10 = 20,000$$

The cheapest first heuristic was referred to by [Kowalski,1979] as the procrastination principle, because it tends to pick off the easy parts of the problem instead of directing the problem solving effort towards the heart of the problem.

6.10.2.2 Reducing the Number of Conjunct Orderings

For a set of n literals there are n! possible orderings. Two theorems due to [Smith and Genesereth, 1985], enable us to reduce the number of possible literal orderings which should be searched in order to find the optimal ordering. Before discussing these theorems we introduce the assumptions and definitions upon which they are based.

6.10.2.2.1 Assumptions and Definitions

The theorems are based upon the average number of solutions to a given conjunct. This will be defined shortly. Average numbers of solutions or proabilities are used so that we can predetermine the cost of a total conjunct

sequence before any of it has been implemented. That is, a *Static Ordering Assumption* is made.

The Static Ordering Assumption is that an adequate improvement in system performance can be obtained by ordering each specific set of conjuncts right before giving it to the generate and test engine. A problem solver can be represented as a *conjunct ordering step* followed by a *generate and test step.*

Let,

s = a set of conjunctive goals, and

t = an ordered set of conjunctive goals

The conjunct ordering step is expressed in logic as follows: Find t such that BestOrdering(s,t).

BestOrdering(s,t) $\Leftrightarrow$

Ordering(s,t)$\wedge$ $\forall$x(Ordering(s,x)$\Rightarrow$Cost(t)$\leq$Cost(x)).

The cost represents the average number of nodes explored by a conjunct sequence. We assume that there are no special capabilities such as selective backtracking, and that no inference is required in generating the partial solutions at each node.

If t = <p_1,. . . p_m> is a sequence of conjuncts, the average number of solutions to conjunct sequence t is given by

$$\text{Numsol}(t) = \prod_{i=1}^{m} \text{Avgnumsol}(p_i|t_{1,i-1})$$

where Avgnumsol($p_i|t_{1,i-1}$) denotes the average number of solutions to proposition p_i under variable bindings to the sequence $t_{1,i-1}$. It is equivalent to the average branching factor at that level. Intuitively Numsol(t) calculates the average number of leaf nodes in the search space.

The average cost of producing all the solutions is given by summing the cost of producing each layer of nodes. It represents the total number of nodes expanded.

$$\text{Cost}(t) = \sum_{i=1}^{m} \text{Numsol}\,(t_{1,i})$$

Numsol gives the total number of leaf nodes in the search tree. *Cost* sums the costs associated with producing each node in the search space. The characteristics of the cost equation can easily be seen. If the search space continues to expand dramatically with each conjunct the net cost is proportional to the total number of solutions. If the space expands and then undergoes substantial reduction, as in most conjunctive problems, one of the intermediate terms will dominate. This dominant intermediate term corresponds to the widest portion of the search tree.

6.10.2.2.2 The Adjacency Restriction

Theorem: (The Adjacency Restriction) [Smith and Genesereth, 1985] Suppose that there is a sequence, t, of M conjuncts

$$t = < t_{1,i-1}, c, d, t_{i+2,M} >$$

and t is an optimal ordering of the conjuncts. ($t_{1,i-1}$ and $t_{i+2,M}$ respectively denote the sequences of conjuncts which precede and follow the two intermediate conjuncts c and d, which are in the positions i and (i + 1) in the conjunct ordering). Then for any adjacent pair of conjuncts c and d, the number of solutions to the second conjunct, d, under the bindings to the preceding conjuncts $t_{1,i-1}$, must be greater than the number of solutions to the conjunct c, under the bindings to the preceding sequence $t_{1,i-1}$. Formally,

$$\text{AvgNumsol}(c, t_{i-1}) \leq \text{AvgNumsol}(d, t_{i-1}).$$

This theorem is proved in [Smith and Genesereth, 1985].

The adjacency restriction is much weaker than the cheapest first restriction. It does not imply that a conjunct should be less expensive than all subsequent conjuncts, only that it should be less expensive than its immediate successor. Useful corollories of this theorem are:

Corollary 1: The most expensive conjunct is never the optimal one to do next.

Corollary 2: The search for conjuncts to immediately follow a conjunct c, can be limited to those which more expensive than c at the time that c was selected.

In the simple example above the Corollary 1 of the adjacency restriction advises us that R should not be selected first. However it cannot tell us which of P and Q is the best to solve first.

6.10.2.2.3 Place Reducing Conjuncts First

A second theorem due to Smith and Genesereth is

Theorem: (Reducing Conjunct First). [Smith and Genereseth, 1985].
When the cheapest conjunct, c, will reduce the search space rather than expand it (i.e. the average number of solutions is less than one), this conjunct should be chosen first. The resulting ordering will be the optimal ordering if the solution is non-trivial. (If the solution is trivial the conjunct sequence which should have preceded c in the optimal solution also has a cost less than one, and the ordering with c first has a cost no greater than twice that of the optimal ordering). This theorem is proved in [Smith and Genesereth, 1985].

We consider the conjunct ordering problem for factory layout.

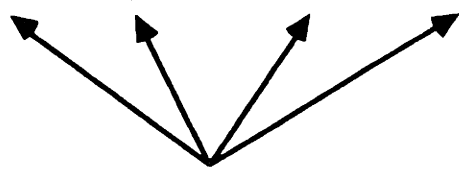
6.10.3 Conjunct Ordering and Factory Layout

6.10.3.1 The Nature of Adjacency Requirements

Consider the conjunct ordering problem for finding a perfect solution to the layout problem, one in which all desired adjacencies are met. The conjunct ordering problem is to find the optimal ordering of the adjacency conjuncts. The optimal ordering minimizes the search cost. Assume that the departments are to be assigned to a grid, as in the QAP. Consider the following conjunct ordering:

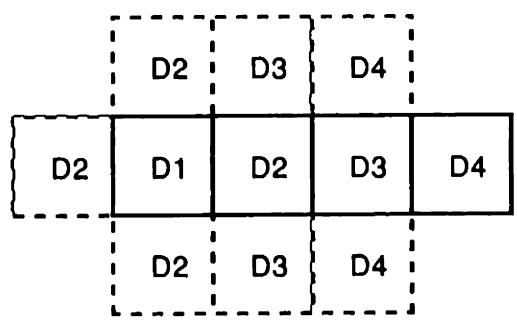
loc(D1)∧ adj(D1,D2)∧ adj(D2,D3)∧ adj(D3,D4)∧ ... ∧ adj(Di,Dn)

Numsol = 1 * 4 * 3 * 3



Branching factors at node

Assume that department D1 is constrained to a fixed location. The number of locations in which we can place D2 adjacent to D1 is 4. (See sketch below). Since D2 is adjacent to D1, there are three free faces on D2 on which to locate D3. Once we have chosen a location for D3, there are three free faces on which to place D4.



Given that locations have been assigned to departments D1,D2, D3 and D4 in the first four conjuncts, consider the next conjunct :

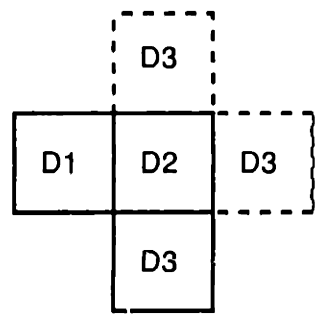
1. adj(D4,D5)

   Since D4 is attached to D3 there are three possible faces on D3 to

locate D4. A sequential relationship between departments in which each department is attached to a previous department in a production line manner arises if we have a string of adjacencies of this kind. The number of solutions grows by at most three with each additional department.

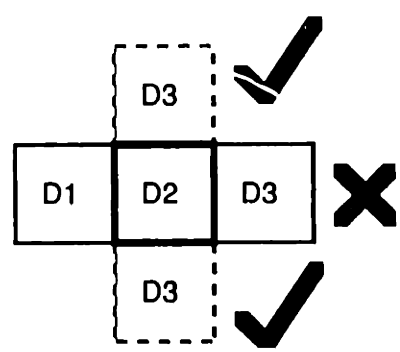
2. adj(D1,D3)

This adjacency is not possible given the previous conjuncts. That is, the branching factor is zero and the search terminates without a complete solution.

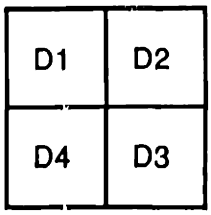


3. adj(D1,D4)

The average branching factor is less than one. This adjacency condition reduces the search space by a factor of 2/3. The linear arrangement of D1,D2,D3 cannot satisfy this adjacency condition.



A successful configuration is shown below



Note that D1and D4 were both already placed. A theorem overleaf proves that requiring adjacency of departments which are already placed can never increase the search space.

4. adj(D6,D7)

This adjacency requirement is independent of the previous conjuncts. It expands the search space by the number of spaces available in the grid where a pair of departments could be located.

In general, the more departments we place, the lower becomes the probability of a successful adjacency,

(1) because of the clustering problem of many departments with adjacency requirements.

(2) because as the number of departments approaches the number of locations, edge effects limit the the cluster configurations.

6.10.3.2 The Reducing Property of Placed Departments

The Perfect Factory Layout Problem may be expressed as the problem of satisfying the conjunct sequence

$$adj_1(D1,D2) \wedge adj_2(D2,D3) \wedge .... \wedge adj_r(Di,Dn),$$

which represents all of the desired adjacencies, in the order adj_1 to adj_r.

Theorem: (Placed Department Adjacency). In the Perfect Factory Layout Problem, any adjacency requirement of two departments which have already been placed,

$adj_m(D_a,D_b)$ s.t. $adj_i(D_a,D_x)$, $adj_k(D_b,D_y)$ $1 \leq i,k < m$

has a branching factor of less than or equal to 1.

Proof: (Placed Department Adjacency)
We distinguish between two types of placement in the Perfect factory layout problem, adjacent and not adjacent. If the adjacency requirement $adj(D_a,D_b)$ has been made implicitly or explicitly before in the conjunct sequence, then the requirement $adj_m(D_a,D_b)$ has a branching factor of one. If the adjacency requirement $adj(D_a,D_b)$ has not been made before then it acts as a constraint on the previous solutions disallowing all of the previous solutions in which this constraint was not met. Therefore it must reduce the number of solutions.
End of Proof.

This proof exploits the fundamental property of adding an additional constraint which can be shown to act in a limiting manner on the solutions already presented. Further proofs would rely much more heavily on the topological properties of the chosen representation.

6.10.3.3 Implications of Theorems for Search Strategies

We examine the implications of the above proof and the two proofs of Smith and Genesereth for heuristics in factory layout. We postulate that the most dramatic reduction of the search space occurs if we always choose the department with the lowest probability of successful placement.

Consider two cases of department A in which it has a low probability of successful placement:

(1) adj(DeptA,DeptB) where a large number of departments are already adjacent to DeptB.

(2) adj(DeptA,DeptC) $\wedge$ adj(DeptA,DeptD) $\wedge$ adj(DeptA,DeptE) where

DeptC, DeptD and DeptE are already placed.

In both cases the low probability of placement is related to the clustering phenomena. We can achieve this clustering phenomena by the following strategies.

Choose as the first department for placement, the one with the largest number of desired adjacencies. Departments which desire adjacency with a placed department are called target departments. Try to place all of the first department's targets adjacent to it. Try to satisfy the desired adjacencies among the target departments.
Then,

(a) choose the target placed department with the most desired adjacencies and try to place its targets adjacent to it, and so on,
or
(b) choose as the next department for placement the one with the most desired adjacencies to the departments already placed and try to satisfy its adjacencies and so on.

In each heuristic we use the properties of the third theorem (Placed Department Adjacency). If a desired adjacency exists between two placed departments which were not explicitly placed adjacent, then this adjacency requirement will reduce the search space.

According to the second theorem, (Cheapest Reducing Conjunct First), if an adjacency conjunct is a reducing conjunct, and it is the cheapest reducing conjunct, then it should be the next conjunct to be implemented. We may use an order scale, such as :

P(DeptA adjacent to 3placed depts) < P(DeptB adjacent to 2 placed depts)

to estimate the cheapest conjunct.

By trying to select the department with the lowest probability of successful placement we are in essence using a cheapest first heuristic. We cannot prove that either of these heuristics gives the optimal conjunct ordering. However, this heuristic does not violate the adjacency restriction of the first theorem. As stated previously the conjunct ordering problem is a combinatorial optimization problem. The adjacency theorem helps to reduce the number of possible combinations, but it cannot address the basic intractibility of such problems. Since Numsol(t_j) depends on the preceding conjunct sequence we always have to evaluate a number of partial conjunct sequences (for example as in the branch and bound method) before we can find an optimal conjunct ordering. In this interpretation of the cheapest first heuristic the cheapest conjunct has the smallest number of solutions because it is the most difficult problem to solve. It attempts to force a cluster so that the hardest problems will be solved first and then the easier ones can be accommodated around them. It does not appear to procrastinate. To draw an analogy with box packing, where the aim is to fit several objects of different sizes into a box, an effective strategy is to put the large objects in first and slot the small objects around them.

The nature of the solution space is to expand first and then contract. At first, there are so few placed departments (two or three) that there are a number of ways of placing one adjacent to the other. But as more departments are added the number of configurations starts to decrease. The width to which the search tree grows before it begins to contract depends upon the interconnectivity of the departments.

The implications of the theorems for the perfect layout problem are as follows:

1. If there are not many conflicts it is easy to find a solution.

2. If there are conflicts, they should be clustered together at the start of the search space in order to limit it.

3. If there are a large number of conflicts, we will determine that there is no

solution quickly.

Preliminary checks to ensure that a department has enough space on its own perimeter to accommodate desired adjacencies will eliminate needless search.

6.10.3.4 Implications of Theorems for Satisficing Hybrid Search Strategies in Layout Software.

The methods of forward and backward staged search aim to achieve perfect solutions in local parts of the search space. If we allow n levels of backtracking, or specify placement methods for configurations of n departments, we try to find the most departments which can be placed adjacent within the local search space. We limit the backtracking by choosing those departments with most solutions first. In the forward strategy we try to place more complex adjacency patterns among larger groups of departments first; we try three way adjacency before two way adjacency. If we do not try to place a department's targets adjacent, when we place that department, it may become surrounded by unrelated departments. We may be unable to backtrack to achieve an adjacent placement if backtracking is restricted to n levels. In this case we must satisfice in placing the targets.

While strategies which place targets will achieve the most adjacencies with limited resources of backtracking, these strategies do not address the desirability of one adjacency over another. If a department pair has a very large flow between them, it is important to place them adjacent. But if these departments do not have many other targets they will not be placed early in the procedure. However, since they do not have many targets, the probability of their successful adjacent placement is high. In the software implementation of chapter 7, we consider how to combine strategies of many desired adjacencies and important adjacencies.

Summary of Chapter 6

In this chapter we proved that the Factory Layout Problem was NP - complete in the number of grid squares. It was known that the QAP of more than fifteen departments was intractable for the purpose of computation. It was therefore not surprising that the more expressive grid formulation of the FLP was intractable for smaller number of departments, the measure of intractability increasing with the fineness of the grid. However it was also shown that generate and test methods, which generate and test only locations neighboring placed departments, were also intractable.

It was determined that a hybrid search strategy was most appropriate for finding satisficing solutions. With large numbers of departments, such a strategy would have to focus on depth first search with limited probing breadthways because all solutions to the layout problem lie at the lowest level of a tree of depth, D, equal to thenumber of departments. These hybrid strategies find solutions in linear time.

A forward strategy in which placement methods are predetermined is the most appropriate strategy for coding in a logic programming language because it does not require a sophisticated control structure.

Three conjunct ordering theorems imply that the most effective way to achieve a solution with a large number of desired adjacencies, is to try to cluster departments; place a department with a large number of desired adjacencies first. Try to place all of those departments with a desired adjacency to a particular department at the same time as that department is placed.

CHAPTER 7

SOFTWARE

Overview

This chapter describes the software implementation of a Factory Design Advisor (FDA). It also introduces a conceptual software architecture which underlies the software structure.

The solution procedure is divided into three phases:

Selection of machines and machine groupings within departments,
Selection of desired adjacencies,
Placement in space.

There are domain independent rules and domain dependent rules associated with each phase. We discuss how adjacency conjuncts, derived from the rules, are ordered for implementation in the context of the complexity discussion of chapter 6.

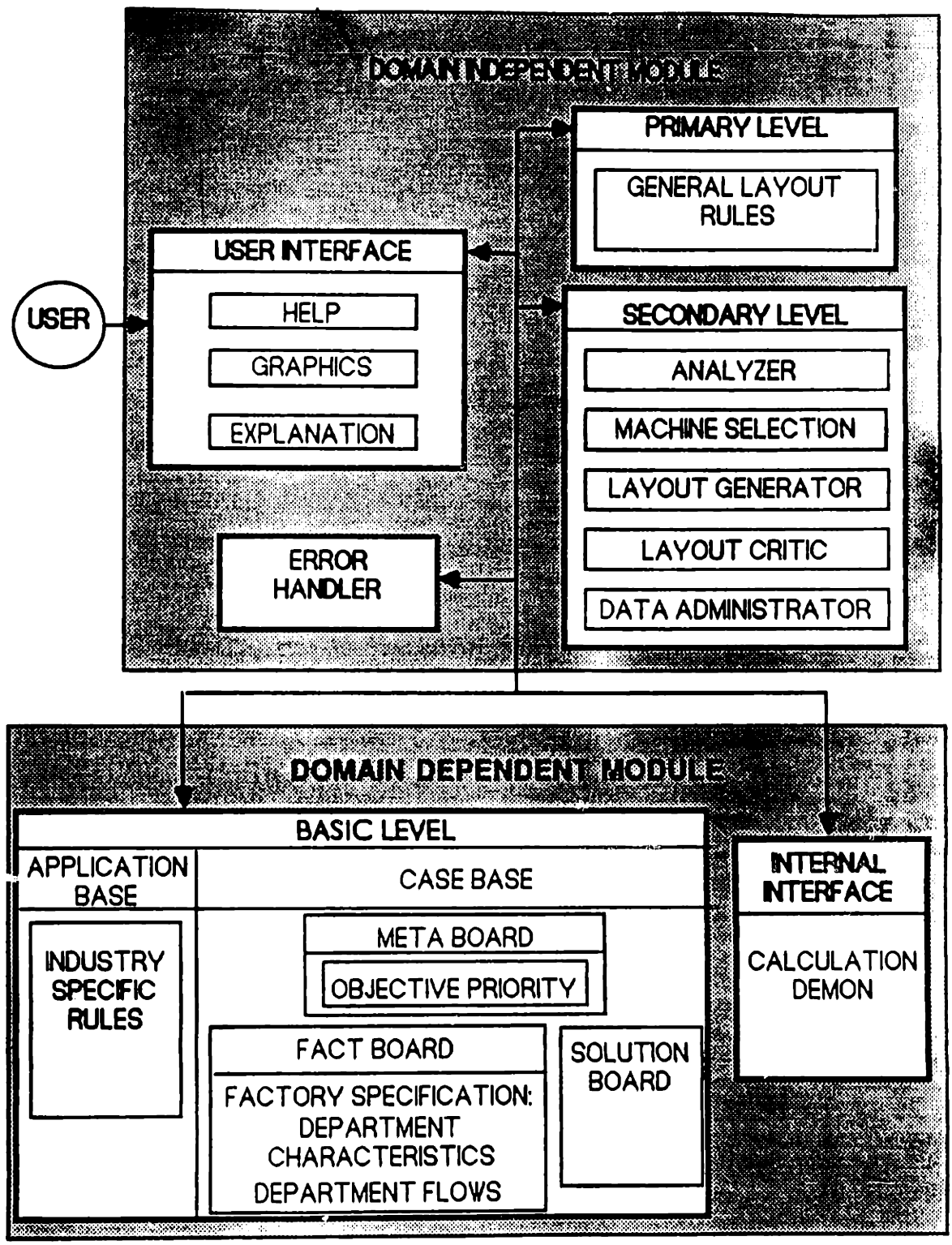
We discuss implementation of the FDA in software, presenting examples for a five department factory, to illustrate the features of the program, and an automobile factory, to illustrate a practical application. We examine the benefits and limitations of the logic based programming language Prolog, in which the system is programmed.

7.1 Conceptual Architecture

The software architecture is shown in figure 7.1. It is based upon the general architecture proposed by [Kim and Suh, 1985]. It consists of a Domain Independent Module and a Domain Dependent Module; this modular structure facilitates system modification, portability and expansion to other domains.

Figure 7.1

Conceptual Software Architecture



7.1.1 Domain Independent Module

The Domain Independent Module, illustrated in figure 7.2, can be decomposed into four parts:

(i) Primary Level
(ii) Secondary Level
(iii) User Interface
(iv) Error Handler

The Primary Level contains both declarative and procedural general knowledge, while the Secondary level contains procedures for generating the layout, critiquing the layout and checking data consistency. The user interface provides for friendly, intelligent interactions with the user. The error handler addresses problems which arise during consultation, or attempts by the user to drive the system beyond its intended capabilities.

7.1.2 Domain Dependent Module

The Domain Dependent Module, illustrated in figure 7.3, consists of a basic level and an internal interface. Within the Basic level, the Application Base contains procedural and declarative knowledge about the domain, in this case, automobile assembly, while the Case Base keeps track of the layout under design. The Case Base consists of (i) a Fact Board to keep track of input assertions such as the factory specifications, the characteristics of the departments and the department flows, (ii) a solution board to keep track of the evolving design, including assertions deduced from the fact board, and (iii) a Meta Board to post information such as the objective priority. The internal interface serves as a gateway to the domain independent module. The calculation demon is an agent which when triggered by an external stimulus invokes the Secondary Level procedures in the Domain Independent Module.

Figure 7.2

Domain Independent Module

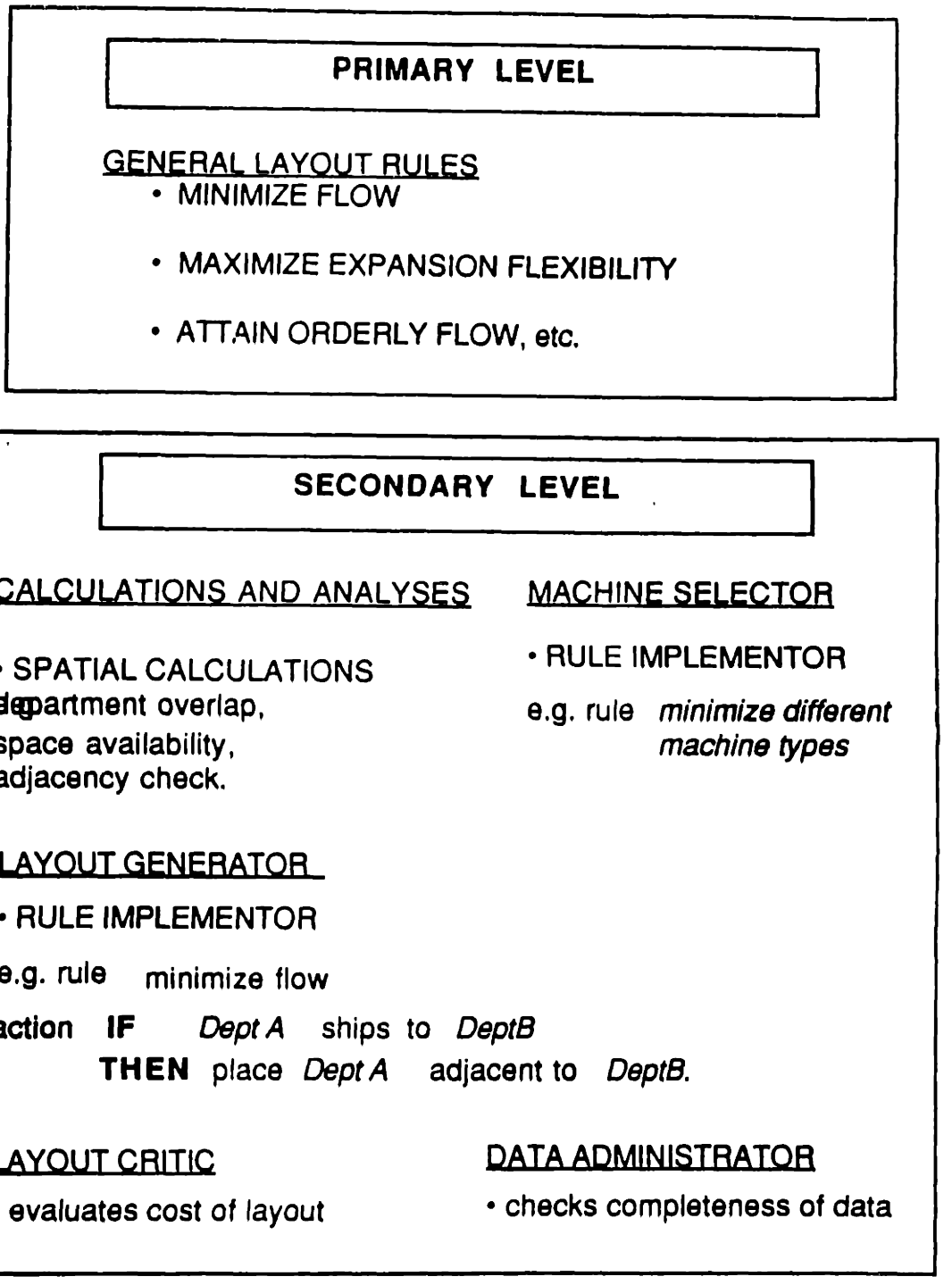


Figure 7.3

Domain Dependent Module

INTERNAL INTERFACE

- GATEWAY TO DOMAIN DEPENDENT MODULE
- CALCULATION DEMON
  - AN AGENT WHICH INVOKES THE SECONDARY LEVEL PROCEDURES WHEN TRIGGERED BY AN EXTERNAL STIMULUS

BASIC LEVEL

APPLICATION BASE

- INDUSTRY SPECIFIC RULES

e.g. Minimize conveyor changes.

Make conveyor lengths small so aisle access not restricted.

Flow lines must avoid pillars.

CASE BASE

- FACT BOARD - CONTAINS FACTORY SPECIFICATION
- SOLUTION BOARD - KEEPS TRACK OF EVOLVING DESIGN
- META BOARD - PROVIDES CONTROL INFORMATION SUCH AS OBJECTIVE PRIORITY

7.2 Domain Dependent Input

The factory specifications form the bulk of the domain dependent input. In the conceptual architecture this information relates to the Fact Board in the Case Base. We may wish to consider several different objective orderings. The objective orderings affect the manipulation of data in the factory specifications. They are posted on the Meta Board in the conceptual architecture.

7.2.1 Factory Specifications (Case Base - Fact Board)

The department data is input in an input file in frames as follows:

```
assembly    ::
            [department_type:       [welding],
            dimensions:             [100, 50]
            features:               [expanding,dustextraction],
            op_access_type:         [side,roof,front,back]].
```

A department which needs to expand may have *expanding* as a feature. The feature *reinforcedfloor* would indicate that a department could not easily be moved. Features are of two types, similar features and hostile features.

A *similarfeature* is one which is common to a group of departments and expresses desired closeness of departments in the group. For example, it is desirable to have all the welding departments close to each other since they all require the same services of water cooling and smoke extraction. Since the features water cooling and smoke extraction always apply to welding departments, we may simply express *welding* as a feature. The similar features which are to be considered in the current layout are input at the following prompt:

```
effect ?: welding
effect ?: dustextraction
```

For each similar effect the program checks the department frames to determine which departments should be placed adjacent.

The *hostile features* express that two departments, each having one of a pair of hostile features, should not be placed together. The hostile features which are to be considered in the current layout are input in pairs at the following prompt:

```
hostile effects ?: expanding-reinforcedfloor
hostile effects ?: noisy-quiet
hostile effects ?: vibrating-precision
```

For each pair of enemyfeatures the program checks the department frames to determine which departments should not be placed adjacent.

Shipping relationships are input in the input file as follows :

```
ships(sideframe,bodyframing, 44)
ships(bodyframing,assembly,110)
ships(assembly,primerdip,190)
```

The dimensions of the factory within which the departments are to be placed are input as follows:

```
factory(100,200).
```

7.3 A Meta Language and the Incorportion of Industry Specific Rules

A meta language is defined in order to access the knowledge stored in the department frames:

```
attributes(Dept,Attribute_name,Attribute_list):-
        Dept:: List,
```

```
member(Attribute_name,Attribute_list,List).
```

To determine if the assembly department had any notable features we would apply the above rule as follows:

```
?- attributes(assembly,features,Attribute_list).
```

This would return

```
Attribute_list = [expanding,dustextraction,welding].
```

The advantages of the meta language and flexible frame structure are twofold:

(i) If a department does not have any attributes of a particular type, e.g. features, this slot may simply be omitted from the department frame. This flexible frame structure ensures that only relevant information need be supplied.

(ii) The flexible frame structure can be used for storing industry specific data. An attribute such as access_type is peculiar to automobile factories. It denotes which parts of the car must be accessed in order to perform a particular operation. This information is used in determining which conveyors are appropriate for the car body at a given operation. This slot would not apply in other types of factories, so it would be omitted.

There are some industry specific rules which cannot be expressed in the above manner. They usually relate to details of placement within a department. For example, *conveyors should be placed a certain distance from pillars to allow room for machines.* These rules do not affect the department placements. They are not incorporated in the prototype version of the FDA. They would be implemented in a fourth "fine tuning" phase, after the Layout Phase.

7.3.1 Objectives as Meta rules

Implicitly, the objectives form the heads of meta rules:

IF *Minimize Flow*
THEN *Establish desired department adjacencies to minimize flow*

The body of the meta rule then forms the head of a rule establishing facts about departments or relations between departments:

IF *Establish desired department adjacencies to minimize flow*
THEN *Place DeptA adjacent to DeptB*
Place DeptC adjacent to DeptE
Place DeptF adjacent to DeptA.....etc.

Meta rules fall into three categories:

(i) rules relating to facts about the material handling system choice and the departments, such as the size and number of turning departments desirable. We might choose one large department, or several small ones to allow routing flexibility.

(ii) rules relating to adjacency relationships between departments.

(iii) rules relating to the placement of departments in space.

7.4 Phases of the Solution Procedure

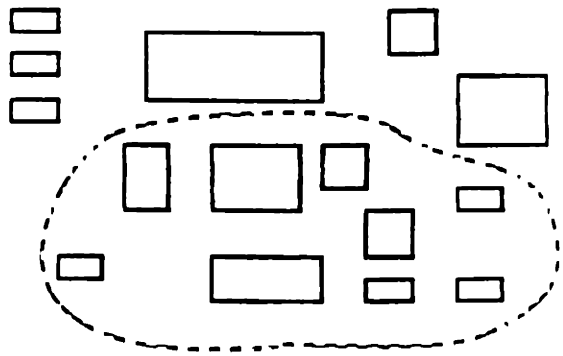
The adjacency relationships possible will depend upon the size and number of each type of department. For example, four turning departments with two lathes each will achieve more departmental adjacencies than one turning department with eight lathes. By determining which adjacencies are possible before proceeding to the spatial layout, unnecessary search in the spatial layout process is eliminated. The search procedure is therefore divided into three phases, illustrated in figure 7.4:

1. Make department and material handling system choices

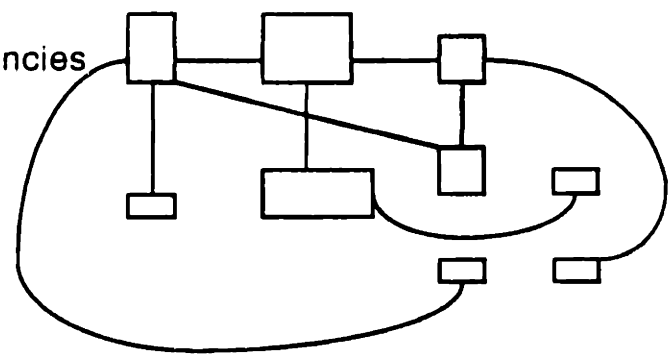
Figure7.4

Phases of The Solution Procedure

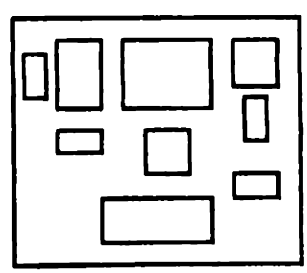
Phase 1 :
Make department and material handling system choices



Phase 2 :
Establish desired adjacencies



Phase 3 :
Establish spatial relationships



2. Establish desired adjacencies
3. Establish spatial relationships.

We could determine desired adjacencies and then choose the department divisions in order to satisfy the adjacencies. Or, rather than divide the search into phases the program could backtrack between the first and second phases before each is complete. However, it is easy to imagine how the program could satisfy every adjacency requirement by choosing a large number of small departments, but that this may not lead to a good layout.

A factory designer proceeds by deciding how much flexibility is needed and by choosing an appropriate number of departments and appropriate handling equipment. The designer then tries to lay out the factory according to the selection of departments and handling equipment. If a good layout cannot be found an alternative department division may be considered.

Prior to choosing the department divisions and the material handling system, decisions on machinery and inventory levels are made. These decisions will be affected by the importance of machine and operation flexibility. They will affect the choice of department shape and the ability to subdivide departments. These preliminary choices are not incorporated in the prototype version of the system software. There is no capacity for sub-division of the departments. All departments must be distinct. Each department name and shape is input into the department frame by the designer.

Examples of meta rules in the three phases are given below, (those coded in the protoptype system are asterisked) :

Choice Related Meta Rules

Maximize machine flexibility } Objectives relating to preliminary
Maximize operation flexibility } choices, not made by the prototype FDA.
Minimize inventory }

Minimize varieties of handling equipment*

Maximize routing flexibility
Maximize machine utilization

Adjacency and Minimum Flow Meta Rules
Minimize material flow*
Minimize personnel flow
Attain orderly flow
Maximize expansion flexibility*
Isolate undesirable effects*

Spatial Meta Rules
Optimize space
Accommodate existing immovables*

The ease with which the other objectives may be coded will be discussed in section 7.7.3.6.

7.5 Rule Types and the Solution Procedure

The search space is pruned by limiting the amount of backtracking through staged search. We seek to order the rules intelligently so that the most promising paths are explored in the limited search space.

Rules, consisting of objectives, O, and actions, A, may be classified as follows:

Independent $(O1 \rightarrow A1) \wedge (O2 \rightarrow A2) \wedge ... \wedge (ON \rightarrow AN)$

Alternative $(O \rightarrow A1) \vee (O \rightarrow A2) \vee ... \vee (O \rightarrow AN)$

Cooperating $(O1 \rightarrow A) \wedge (O2 \rightarrow A) \wedge ... \wedge (ON \rightarrow A)$

Competing $(O \rightarrow A1) \wedge (O \rightarrow A2) \wedge ... \wedge (O \rightarrow AN) \wedge \sim(A1 \wedge A2 \wedge ..... \wedge AN)$

Conflicting $(O1 \rightarrow A) \wedge (O2 \rightarrow \sim A)$

In factory layout, an action places two departments adjacent to each other. The OR construction for *alternative* rules indicates that objective O may be satisfied by one of several actions. The AND construction for *cooperating* rules illustrates that several objectives are satisfied by the same action. The *competing* rules illustrate the nature of combinatorial optimization problems. One objective, such as *minimize flow,* requires that several departments be placed adjacent. If not all of the adjacencies can be met, the departments compete with each other in order to determine the best arrangement. Competing rules express that somewhere we will find a conflicting rule, but until we have implemented the search procedure, we do not know where the conflict will arise. They also express that the conflict arises not only from two actions but from the combination of several actions. Conflicting rules arise when one objective requires that an action be performed and another objective requires that that action not be performed.

Conflicting rules will always invoke backtracking and cause failure. In order to obtain a solution, a preference of one rule over another must use expressed, and the lower priority rule retracted. Since rules are related to objectives, the objective preference may be used to determine rule preference. Conflicting rules can be determined after each phase, when the bodies of the rules have the same format:

IF Minimize Flow	THEN DeptA adjacent to DeptB
IF Maximize expansion flexibility	THEN DeptA not adjacent to DeptB

After each phase of rule invoking the conflicting rules are determined and marked. If the first rule succeeds, the second is retracted, so that backtracking will not occur. If backtracking occurs for some other reason and the first rule cannot be satisfied, the second rule is automatically satisfied, since it is the negation of the first.

Competing rules are expressed above as actions related to one objective which cannot all be fulfilled. They need not relate solely to one objective.

Competing rules may also be ordered by objective preference.

7.6 Rule Ordering

7.6.1 Sequential objective satisfaction

Within each phase we could order the rules according to the priority of the objectives. For example,

O1 > O2 > O3

where the inequality denotes prioritization,

O1: adjacent(deptA,deptB),
adjacent(deptB,deptC),
adjacent(deptB,deptD).

O2: adjacent(deptA,deptC),
adjacent(deptB,deptF).

O3: adjacent(deptC,deptG),
adjacent(deptB,deptA).

We would then create a partial solution for the first objective, backtrack to meet the requirements of the second objective, and so on until no more objectives could be satisfied. However this method would require alot of backtracking; each objective addresses the same set of departments, and each time a new objective is added it checks that the current department placement is satisfactory, or it modifies it. In a system with limited backtracking, if the first objective demanded a large number of adjacencies, the backtracking cut-off would prevent the satisfaction of adjacencies required by the lower priority objectives.

7.6.2 Parallel objective satisfaction

This suggests that it would be better to treat each department as an object with an attached set of rules:

Rules relating to departmentA

adjacent(deptA, deptB) : Minimize flow

Rules relating to departmentB:

adjacent(deptB, deptA) : Minimize flow

adjacent(deptB, DeptC) : Minimize flow

adjacent(deptB, DeptF) : Minimize varieties of handling equipment

...

When a department is placed, we should also try to place its subordinates. Note that we cannot also try to place the subordinates of the subordinates and the subordinates of those subordinates and so on, because to do this in an optimal manner is as difficult as the layout problem itself and would require the same amount of search. But we may choose a nominal placement order, for example, the number of adjacencies required by a department, or size of the total flow to and from each department, such that we have

deptA > deptB > deptC.

Then when A is placed we try to place its subordinates, and to place them in such a way that their relationships with each other are satisfied. When B is placed, we try to place its subordinates.

By grouping rules relating to a department in this way, we keep conflicting and competing departments close together so that backtracking is local, and so that backtracking between related departments occurs within the backtracking cut-off However if all the subordinate departments have a large number of other desired adjacencies in their rule sets, it will be impossible to keep all of the related departments together, and some important adjacencies may be excluded by the backtracking cut-off. In this case it may be better to simply order all of the adjacency pairs in order of importance, and to choose as the next department for placement the one with the most important adjacency to one of the departments already placed. This is only a slight modification of sequential objective satisfaction. It is likely to result in the compromise which achieves the most important adjacencies, but does not

achieve a large number of desired adjacencies.

7.7 Software Demonstration

7.7.1 Knowledge Administration

The input file for an automobile assembly plant body shop is listed in Appendix3 figure1. The FDA relies upon this data so it must be consistent with the expectations of the FDA. A predicate *datacheck* determines whether all the departments input have been assigned dimensions. Since all other attributes in the department frame are optional, the program does not check to see if they exist for all departments. Predicates *checkattribute* and *checkdepartment* list all departments defined for a given attribute and all attributes defined for a given department respectively. A sample interaction of these predicates which form the Knowledge Administrator is shown in figure 7.5. Initially the datacheck predicate fails because the department "rearcomp", for rear compartment manufacture, does not have dimensions assigned. In fact "dimensions" is misspellt in the input file. When this mistake is corrected, *datacheck* succeeds.

7.7.2 Conveyor Selection

The size of the machine selection search space was discussed in the formal model of Chapter 5.

Given a set of m conveyors,

$$C = \{c_1, c_2, c_3 \ldots\ldots\ldots\ldots c_m\},$$

then the sets of possible conveyor types which can satisfy the departments are, in order of increasing size,

$$\{c_1\},\{c_2\},\{c_3\}\ldots\ldots\{c_m\},\ \{c_1c_1\},\ \{c_1c_2\},\ \{c_1c_3\}\ldots\ldots\ldots\{c_1c_2c_3\ldots\ldots c_m\}.$$

These represent the leaf nodes in a search space of size 2^m.

Figure 7.5
A Sample Interaction with the Knowledge Administrator

```
| ?- datacheck.

rearcomp has no dimensions assigned
no

| ?- checkdepartment(rearcomp).

rearcomp has the following attributes:

dept_type = ubody
dimonions = [15,10]
no


| ?- datacheck.

All departments have dimensions assigned
yes


| ?- checkattribute(features).

The following departments have features :

ubodyweld has features = [reinforcedfloor]
assembly has features = [expanding,dustextraction,welding]
primerdip has features = [reinforcedfloor]
no


| ?- checkdepartment(primerdip).


primerdip has the following attributes:

dept_type = process
dimensions = [20,20]
opaccess = [underneath]
features = [reinforcedfloor]
no

| ?- checkdepartment(office).

No such department exists
yes
```

$$\text{Search Space Size} = m + \frac{m!}{2!\,(m-2)!} + \frac{m!}{3!\,(m-3)!} + \ldots + \frac{m!}{m!\,0!}$$

$$= \sum_{0}^{m} C_{(m,r)} = 2^m$$

Figure 7.6 shows an interaction with an automobile factory database. The attributes predicate is used to determine which departments have expressed the operation access attribute, "opaccess". This attribute determines which part of the car must be free of conveyor attachments in order for the operations of the department to be performed. From the rules of figure 4.8, which were encoded, we determine which conveyors cannot be used if certain access is required. Hence we are able to determine the good conveyors for each department, through the predicate "goodconv".

We can use the knowledge of the number of conveyors which satisfy each department to substantially reduce the search for a satisfactory conveyor set for all departments. If there are any departments which are only satisfied by one conveyor, then this conveyor must be included in the final set. If any departments are only satisfied by either of two conveyors, then one of these two must be included in the final set.

We form a list of conveyors for "one-conveyor-departments", "two-conveyor-departments", "three-conveyor-departments" and so on. This list is generated by the predicate "allconvsbyN" of figure . The search for a satisfactory conveyor set begins with conveyors which must be included in the set because they are one-conveyor-department conveyors. Then the set is increased by adding conveyors from the two-conveyor-department list. If these sets fail conveyors from the three-department-conveyors list are added and so on. We try all the combinations which might work, and we try them in increasing size so that the smallest sets are found first. We may eliminate without trial all sets which do not include the one-conveyor-department

Figure 7.6 (part 1)
A Sample Interaction with the Conveyor Selector

```
| ?- attributes(Dept,opaccess,Opaccess).

Dept = bodyframing
Opaccess = [side,roof,front,back] ;

Dept = assembly
Opaccess = [side,roof,front,back] ;

Dept = primerdip
Opaccess = [underneath] ;
no


| ?- cantaccess(Conveyor,Cantaccess).

Conveyor = overheadside
Cantaccess = [side] ;

Conveyor = overheadfb
Cantaccess = [front,back] ;

Conveyor = inverted
Cantaccess = [underneath] ;
no


| ?- goodconv(Department,Conveyor).

Department = bodyframing
Conveyor = inverted ;

Department = assembly
Conveyor = inverted ;

Department = primerdip
Conveyor = overheadside ;

Department = primerdip
Conveyor = overheadfb ;
no


| ?- allconvsbyN(Conveyors).

Conveyors = [[inverted],[overheadfb,overheadside]] ;
no
```

Figure 7.6 (part 2)

A Sample Interaction with the Conveyor Selector

```
| ?- ratconv(ConveyorChoiceSet).

ConveyorChoiceSet = [inverted,overheadfb] ;

ConveyorChoiceSet = [inverted,overheadside] ;

ConveyorChoiceSet = [inverted,overheadfb,overheadside] ;
no

| ?- assignconv(Department,Conveyor,ConveyorChoiceSet).

Department = bodyframing
Conveyor = inverted
ConveyorChoiceSet = [inverted,overheadfb] ;

Department = assembly
Conveyor = inverted
ConveyorChoiceSet = [inverted,overheadfb] ;

Department = primerdip
Conveyor = overheadfb
ConveyorChoiceSet = [inverted,overheadfb] ;

Department = bodyframing
Conveyor = inverted
ConveyorChoiceSet = [inverted,overheadside] ;

Department = assembly
Conveyor = inverted
ConveyorChoiceSet = [inverted,overheadside] ;

Department = primerdip
Conveyor = overheadside
ConveyorChoiceSet = [inverted,overheadside] ;

Department = bodyframing
Conveyor = inverted
ConveyorChoiceSet = [inverted,overheadfb,overheadside] ;

Department = assembly
Conveyor = inverted
ConveyorChoiceSet = [inverted,overheadfb,overheadside] ;

Department = primerdip
Conveyor = overheadside
ConveyorChoiceSet = [inverted,overheadfb,overheadside] ;

Department = primerdip
Conveyor = overheadfb
ConveyorChoiceSet = [inverted,overheadfb,overheadside] ;
no
```

conveyors or one of the two-conveyor-department conveyors and so on.

This technique of finding the smallest number of machines, conveyors or tools to satisfy a selected set of tasks is known in manufacturing as rationalizing. Several companies have gone about rationalizing tooling in order to streamline inventories and to standardize tooling and operations. In this demonstration rationalizing is applied to conveyor selection, but it could equally well be applied to tooling selection, fixture selection, machine selection and so on.

The most efficient way to rationalize the different conveyor types is to satisfy the hardest departments to please first, and then to progressively add more machines to satisfy less demanding departments, until all departments had been satisfied.

The predicate "ratconv" rationalizes the conveyors (figure 7.6 (part 2)). The smallest conveyor sets which satisfy all departments are found first. All conveyor sets which can satisfy all the departments are found through backtracking. The "assignconv" predicate assigns a conveyor to a department from a chosen conveyor set. If a conveyor set is not specified, all conveyor choice sets are determined from the input file using the "ratconv" predicate, and all possible assignments from all of the choice sets are found through backtracking.

7.7.3 Layout Generation.

In performing the layout generation we assume that the size and number of departments has been determined by the selection module. The layout algorithm could be applied to machines or departments. All layout candidates are treated as objects with associated attributes of dimensions and features. What they represent is irrelevant to the layout module. But the performance of the module in terms of solution time will decrease as the number of departments increases.

7.7.3.1 Candidates for Placement

Existence of a department is determined by using a predicate *unplaced*. This predicate first determines whether a department exists in the input file. It then determines whether the department has been placed. If it has not been placed it is a candidate for placement.

7.7.3.2 Expressing Desired Adjacencies

The desired adjacencies are asserted into the program through the *subordinate* predicate as follows:

```
subordinate(DeptA, DeptB).
subordinate(DeptC, DeptD).
subordinate(DeptA, DeptD).
```

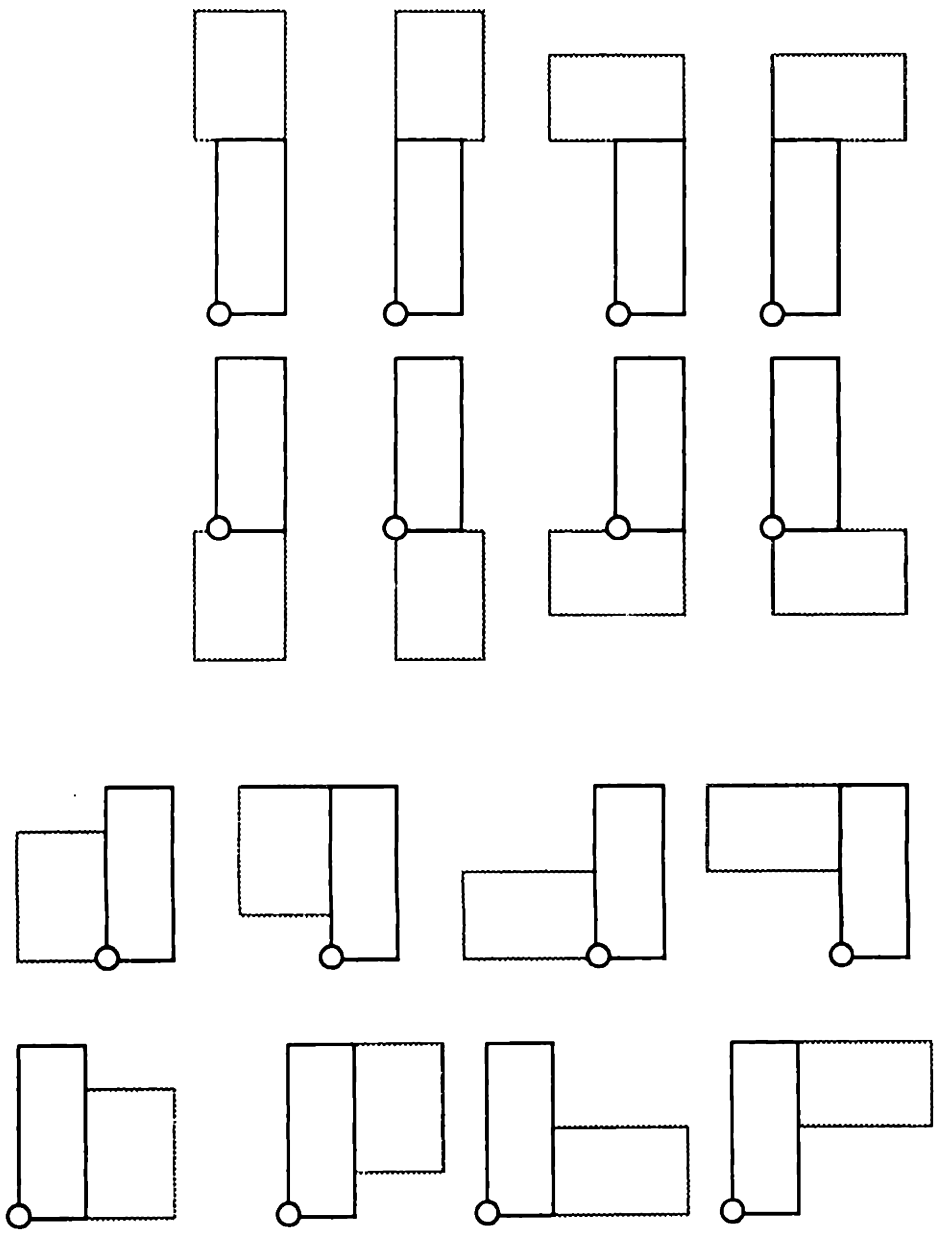
The predicate subordinate(DeptA, DeptB) expresses the fact that when DeptA is placed, DeptB should be placed adjacent to it. B is subordinate to A because the placement of B depends upon the placement of A. The subordinate relationships which are asserted and the order in which they are asserted depends upon the chosen placement strategy. Placement strategies are discussed in Section 7.7.3.4.

7.7.3.3 Location Generation and Explanation

A generate and test method is used for candidate location generation and placement. A forward strategy (section 6.8.1) is used which expresses ordinary placement, two department relationships and three department relationships. In the ordinary placement strategy, locations are generated on the perimeter of any placed department as shown in figure 7.7. Locations are generated so that they align with the edges of any placed department. The length and width of the department are switched so that in total there are sixteen ways of placing a department adjacent to a placed department. The two department placement strategy uses the same candidate geometric configurations as the default strategy, but rather than placing next to a random placed department, it attempts to find an adjacency with a specific

Figure 7.7

Locating a Department Adjacent to and Aligned with a Placed Department



department expressed in a subordinate relationship.

The number of adjacent locations which are defined directly affects the size of the search space. The more locations there are, the closer is the approximation to continuous placement. However in practice the factory designer tries to align as many edges as possible in order to allow long lengths of straight aisles. For this reason we limit our factory to aligned departments. In terms of the number of possible configurations, this equivalent to the case of perimeter placement where the number of perimeter nodes, $n_p = 16$.

If the current department cannot be placed adjacent to its parent department it will be placed as an ordinary department. Ordinary departments are those which do not have any desired adjacencies, or those which have failed in meeting desired adjacencies. Ordinary departments are simply placed on the perimeter of any department which is already placed. Once a department is placed, it is retracted from the list of unplaced departments, so that a repeat placement is not attempted.

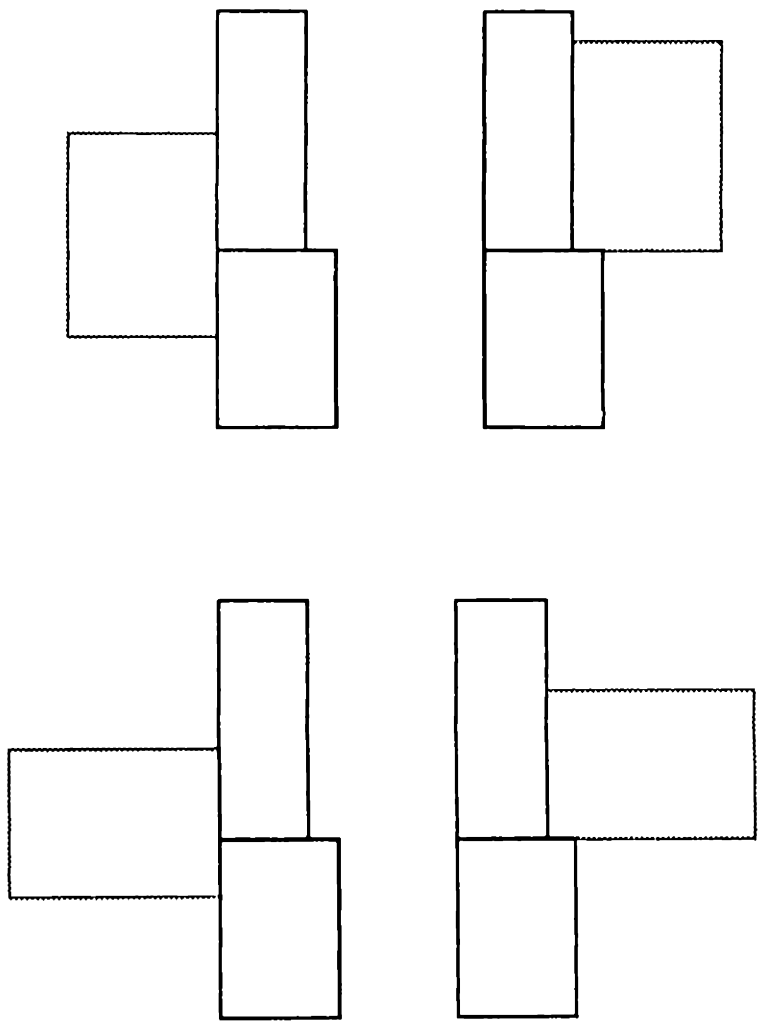
Three way adjacency works as follows. If the departments in the current subordinate pair, A and B, also have a subordinate relationship with C, then when B is placed adjacent to A we also try to place C adjacent to both A and B. There are four ways of placing the third department, once the first two have been placed. These are illustrated in figure 7.8. Note that this strategy places two departments adjacent to one department which has already been placed. It does not place one department next to two departments which are already placed. It is a forward strategy, so when it considers B it notes that B also has a relation with C. It does not determine this after placement of B and during search of a placement for C, as a backward strategy would.

7.7.3.4 Establishing Desired Adjacencies.

It is a simple matter to find all the desired adjacencies. But the order in which we try to satisfy these adjacencies will affect the order in which we find

Figure 7.8

Locating a Third Department Adjacent to Two Placed Departments



solutions to the layout problem.

There are two stages to ordering the adjacencies. First we pick a nominal ordering of the departments. This is the default order in which departments are placed if there are no adjacencies expressed. It is the order in which the departments are asserted as unplaced into the database. Nominal ordering is by the total flow of a department, or by number of targets which a department has, that is, the number of desired adjacencies.

The second stage asserts the relationships between departments. There are three types of relationships:

only ,
targets and
effects.

Only states that we do not wish to express desired adjacencies and that the departments should be placed in their nominal ordering using ordinary placement.

Targets causes the assertion of all the subordinate relationships by flow to the first department in the nominal placement ordering, then all the subordinates by flow to the second department in the nominal placement ordering and so on.

Effects causes the user to be prompted for the features which should be used to express desired adjacencies. The hostile features are input first. These express constraints. If hostile departments cannot be separated then there is no solution and a message is output to the user to explain that the problem is over constrained. Similar features express a preference, but not a constraint. If an adjacency of similar features cannot be met, then a message is sent to the user. Each time a feature is input, the desired adjacencies are asserted as subordinate relationships in the database. The solution proceeds according to the nominal ordering, until one of the departments in the nominal

ordering is expressed in a subordinate relationship. Subordinate placements to the nominal department always take priority over other nominal placements.

After selecting the nominal department ordering and the method by which subordinate relationships are to be selected, the user is offered the choice of placing any departments which are to be constrained. This allows factories to be redesigned around existing immovable departments. Otherwise the first department is located with its lower left hand corner at coordinate (0,0).

The nominal order and subordinate ordering strategy pairs are as follows:

Nominal	Subordinate	Nominal	Subordinate
Flow	Only	Targets	Only
Flow	Targets	Targets	Targets
Flow	Effects	Targets	Effects

If *targets* is chosen as the nominal placement order, then we would choose to place a department with small flows to a large number of departments first, rather than one with a very large flow to only one other department. As our discussion in Chapter 6 suggested, even though a large flow department is important, if it only has one desired adjacency, it should be easy to place it, so we concentrate on first placing the neighbors of departments with many neighbors.

If flow is the nominal placement order, we place the large flow departments first, regardless of the number of neighbors. This may be a useful strategy when some insignificant departnments have so many neighbors that they dominate the solution, and the staged search, to the exclusion of successful placement of important departments. Also because the nominal ordering by flow may be combined with target subordination, we can aim to place all the neighbors of the large flow departments as the large flow departments are being placed.

7.7.3.5 Demonstration of Placement Strategies (Flow)

A five department example is used to illustrate the placement strategies and other features of the layout program. The input file for the five department case is shown in figure 7.9. Figure 7.10 shows a description of the factory provided by the command "desc.". Both the length and depth of the factory exceed the sum of the widths or lengths of all the departments. Therefore the solution will not be constrained by the factory boundary. In figure 7.11 we try to list all of the subordinatedepartment relationships. Since no subordinate relationships have been asserted into the database, Prolog simply answers "yes". The command "placedepts" is used to invoke the layout generator. An assert order of *targets*, and a placement order of *targets* are chosen. No departments are constrained. In figure 7.11 (part 2) we interrupt the solution generation to list the unplaced departments and the subordinate relationships which have been asserted into the database. The subordinate relationships are not listed in any particular order. Figure 7.11 (part 3) shows the solution. Figure 7.11 (part 4) shows a block diagram of the first solution, and two further solutions achieved by backtracking. The command quicktest produces a layout from the current assertions in the database. It is a development tool to prevent the user having to repeat the input after interrupting the program to examine the subordinates.

The nominal placement order was

printing > packing > cutting > receiving > shipping

Printing was placed first, then the three subordinate departments to printing, packing, receiving and cutting were placed. Cutting and receiving were also related to each other by a subordinate predicate, so third placement was invoked in order to place cutting, receiving and printing adjacent to each other. After the three subordinates to printing were placed, the only remaining department from the nominal ordering was shipping. This was placed subordinately to packing.

An explanation module has been incorporated to explain the fate of

Figure 7.9

An Input File for a Five Department Layout Problem

```
receiving ::
        [dept_type :    transportation,
        features   :    [transportation],
        dimensions :    [20,80]].

printing ::
        [dept_type :    working,
        features   :    [noisy],
        dimensions :    [100,100]].

cutting ::
        [dept_type :    working,
        features   :    [quiet],
        dimensions :    [40,60]].

packing ::
        [dept_type :    working,
        features   :    [quiet],
        dimensions :    [50 60]].

shipping ::
        [dept_type :    shipping,
        features   :    [transportation],
        dimensions :    [30,70]].



ships(receiving,cutting,200).
ships(receiving,printing,20).

ships(printing,cutting,15).
ships(printing,packing,300).
ships(cutting,printing,120).

ships(packing,shipping,280).

factory(400,400).
```

Figure 7.10

A Description of the Critical Inputs for the Problem of Figure 7.9 - The Five Department Layout Problem

```
| ?- desc.

This factory has a length of 400 and a depth of 400.


It has the following departments:

Department  X-Length  Y-Length  Area
receiving         20        80  1600
printing         100       100 10000
cutting           40        60  2400
packing           50        50  2500
shipping          30        70  2100


The following shipping relationships exists:

From        To          Volume
receiving   printing        20
receiving   cutting        200
printing    cutting         15
printing    packing        300
cutting     printing       120
packing     shipping       280
yes
```

Figure 7.11 (part 1)

An Interaction for the Five Department Layout Problem: Assert Order = Targets , Placement Order = Targets

```
| ?- listing(subordinate).

yes

| ?- placedepts.


Which assert order ? : targets
(Departments by number of targets)

We find the number of targets that each department has.

Department  Targets
printing          3
packing           2
cutting           2
receiving         2
shipping          1

Nominally placement will occur in that order.


Which placement order ? : targets
(Place targets around departments)

After placing a department, attempt to place around
it departments that it ships to.  The following
departments ship to the corresponding departments:


receiving to:   printing and cutting

printing to:    receiving, cutting and packing

cutting to:     receiving and printing

packing to:     printing and shipping

shipping to:    packing


Constrained departments :

Which department ? :
```

Figure 7.11 (part 2)

An Interaction for the Five Department Layout Problem: Assert Order = Targets , Placement Order = Targets

```
^CAction (h for help): a

[ execution aborted ]

| ?- listing(unplaced).

unplaced(printing,100,100).
unplaced(packing,50,50).
unplaced(cutting,40,60).
unplaced(receiving,20,80).
unplaced(shipping,30,70).

yes

| ?- listing(subordinate).

subordinate(receiving,printing).
subordinate(receiving,cutting).
subordinate(printing,receiving).
subordinate(printing,cutting).
subordinate(printing,packing).
subordinate(cutting,receiving).
subordinate(cutting,printing).
subordinate(packing,printing).
subordinate(packing,shipping).
subordinate(shipping,packing).

yes
```

Figure 7.11 (part 3)

The First Solution to the Five Department Layout Problem: Assert Order = Targets , Placement Order = Targets

```
| ?- quicktest.

DEPARTMENT  X     Y     Length  Height
printing      0     0     100     100
receiving     0   100      20      80
cutting     -40    70      40      60
packing      50   100      50      50
shipping     50   150      30      70

When printing was placed:
        receiving was placed near it.
        cutting was placed near it.
        packing was placed near it.

When packing was placed:
        shipping was placed near it.


Analysis of flow cost:

FROM      TO         FLOW  LENGTH  COST
cutting   printing    120       0     0
TOTAL                                  0

FROM      TO         FLOW  LENGTH  COST
packing   shipping    280       0     0
TOTAL                                  0

FROM      TO         FLOW  LENGTH  COST
printing  cutting     15        0     0
printing  packing    300        0     0
TOTAL                                  0

FROM       TO         FLOW  LENGTH  COST
receiving  cutting     200       0     0
receiving  printing     20       0     0
TOTAL                                  0

The total flow cost is 0.
```

Figure 7.11 (part 4)

Block Diagrams of theSolutions for the Five Department Layout Problem: Assert Order = Targets, Placement Order = Targets

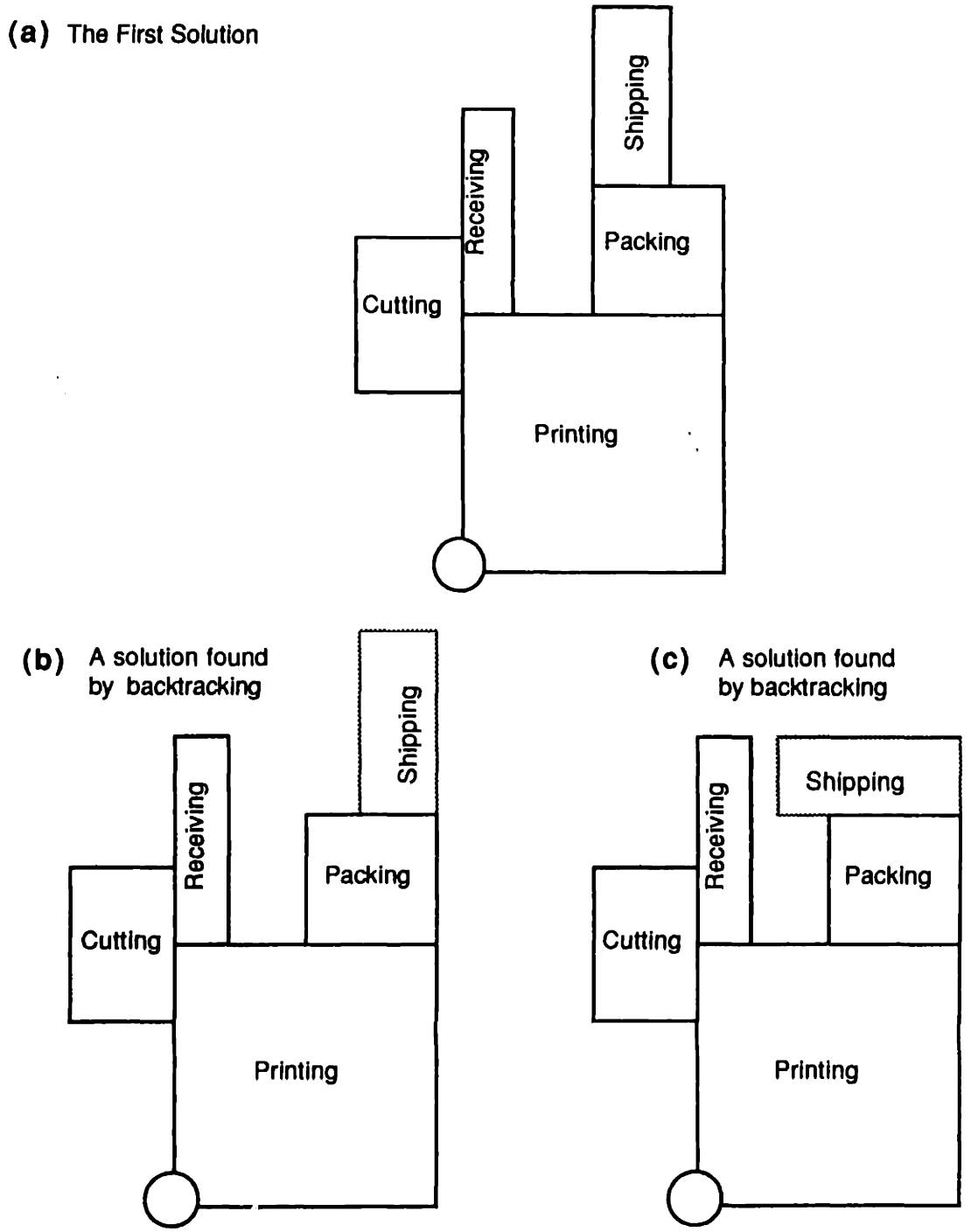


Figure 7.11 (part 5)

Solutions found by Backtracking in the Five Department Layout Problem: Assert Order = Targets , Placement Order = Targets.

```
Find another solution: Yes=>

DEPARTMENT  X     Y     Length  Height
printing      0     0     100     100
receiving     0   100      20      80
cutting     -40    70      40      60
packing      50   100      50      50
shipping     70   150      30      70


When packing was placed:
        shipping was placed near it.

When printing was placed:
        receiving was placed near it.
        cutting was placed near it.
        packing was placed near it.


Find another solution: Yes=>

DEPARTMENT  X     Y     Length  Height
printing      0     0     100     100
receiving     0   100      20      80
cutting     -40    70      40      60
packing      50   100      50      50
shipping     30   150      70      30


When packing was placed:
        shipping was placed near it.

When printing was placed:
        receiving was placed near it.
        cutting was placed near it.
        packing was placed near it.


Find another solution: Yes=> n

[ execution aborted ]

| ?-
```

departments as they are selected for placement. There are three messages relating to placement:

When Dept A was placed, DeptB had already been placed.
When Dept A was placed, DeptB was placed adjacent.
When Dept A was placed, Dept B could not be placed adjacent.

When a factory layout is complete, a cost analysis of the flow is presented. A flow cost between two departments is given by the product of the shortest rectilinear distance between them and the flow weight shipped. All adjacent departments will have a flowcost of zero. A much more sophisticated and accurate analysis could be achieved by incorporating cost functions for each flow type based on the handling equipment. This cost is presented as a rough means for comparison. The cost provides no indication of the accommodation of objectives which were expressed qualitatively. The explanation module could be extended to serve this purpose.

All desired adjacencies were achieved by this solution, so the flow cost is zero.

The interaction of figure 7.11 (part 5) shows two further solutions achieved by backtracking. These were illustrated in the block diagram of figure 7.11 (part 4(b),(c)). The early stages of backtracking simply involves moving shipping, the department which was placed last around packing. The cost remains at zero and is edited from the solution.

Figure 7.12 shows a sample interaction for the same set of departments, but this time the assert order is by flow and the placement order is by targets. Figure 7.12 (part 1) shows the selection of placement criteria. The nominal ordering is

packing > printing > cutting > shipping > receiving

The solution is shown in figure 7.12 (part 2) and illustrated in the block

Figure 7.12 (part 1)

An Interaction for the Five Department Layout Problem: Assert Order = Flow , Placement Order = Targets

```
| ?- placedepts.

Which assert order ? : flow
(Departments by flow)


We find the outward flow of each department.

Department  Total Flow
packing             580
printing            455
cutting             335
shipping            280
receiving           220

Nominally, placement will occur in that order.


Which placement order ? : targets
        (Place targets around departments)

After placing a department, attempt to place around
it departments that it ships to.  The following
departments ship to the corresponding departments:


receiving to:    printing and cutting

printing to:     receiving, cutting and packing

cutting to:      receiving and printing

packing to:      printing and shipping

shipping to:     packing


Constrained departments

Which department ? :
```

Figure 7.12 (part 2)

The First Solution for the Five Department Layout Problem: Assert Order = Flow , Placement Order = Targets

```
DEPARTMENT   X     Y     Length   Height
packing      0     0       50       50
printing    -50    50      100      100
shipping     0    -70       30       70
cutting     -50   150       40       60
receiving   -70   110       20       80


When packing was placed:
        printing was placed near it.
        shipping was placed near it.

When printing was placed:
        cutting was placed near it.
        receiving was placed near it.



Analysis of flow cost:

FROM       TO         FLOW   LENGTH   COST
cutting    printing    120        0      0
TOTAL                                    0

FROM       TO         FLOW   LENGTH   COST
packing    shipping    280        0      0
TOTAL                                    0

FROM       TO         FLOW   LENGTH   COST
printing   cutting     15        0      0
printing   packing    300        0      0
TOTAL                                    0

FROM       TO         FLOW   LENGTH   COST
receiving  cutting     200        0      0
receiving  printing     20        0      0
TOTAL                                    0

The total flow cost is 0.


Find another solution: Yes=> n


[ execution aborted ]
```

Figure 7.12 (part 3)

A Block Diagram of the first Solution to the Five Department Layout Problem:

Assert Order = Flow , Placement Order = Targets

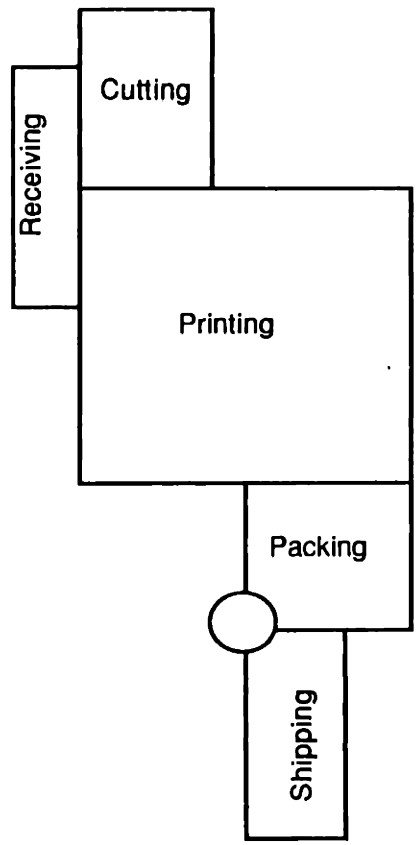


diagram of 7.12 (part 3). Packing is placed first. It has two subordinate departments, printing and shipping which are placed adjacent to it. These departments do not have a subordinate relationship with each other, so they are both placed by two-way placement with packing. The next department in the nominal placement order, printing, has already been placed, but it has subordinates cutting and receiving. Receiving is also subordinate to cutting, so when cutting is placed adjacent to printing, receiving is also placed by three way placement. All desired adjacencies are achieved and so the cost is zero.

Figure 7.13 (parts 1 and 2) shows a sample interaction for the five department case, when the option of nominal ordering by flow and ordinary placement by option *only* are chosen. *Only* placement makes no attempt to place related departments adjacent. It simply places a department in the first available location, next to any placed department. The flow cost is non zero, because three flows occur between non adjacent departments.

7.7.3.5.1 Solutions with Boundary Constraints

All department coordinates are developed relative to the department placed at (0,0). The length and width of the factory in the input file serve to prevent the total x or y length of the placed departments from exceeding the factory length and width. Backtracking occurs until placement of all departments within the perimeter is accomplished. In the three interactions of figures 7.11, 7.12 and 7.13, the factory boundary was so large that it did not pose a constraint upon placement.

Figure 7.14 shows an interaction for the five department case in which the factory is one quarter of the area of that in the previous interactions. The factory description is shown in figure 7.14 (part 1). The option selection of nominal ordering by number of targets and subordinate placement by targets, shown in figure 7.14 (part 2), is the same as that chosen for the five department case of figure 7.11. The solution is shown in figure 7.14 part (3) and illustrated in the block diagram of 7.14 (part 4). The first solution for this case is the same as the third solution found by backtracking for the case of

Figure 7.13 (part 1)

An Interaction for the Five Department Layout Problem: Assert Order = Flow , Placement Order = Only

```
| ?- placedepts.


Which assert order ? : flow
(Departments by flow)

We find the outward flow of each department.

Department  Total Flow
packing              580
printing             455
cutting              335
shipping             280
receiving            220

Nominally, placement will occur in that order.


Which placement order ? : only
(Place only this department then do the next)

When placing a department, place only that department
and make no attempt to place any other department that
might relate to it.


Constrained departments :

Which department ? :


DEPARTMENT  X     Y     Length  Height
packing       0     0       50      50
printing    -50    50      100     100
cutting       0   -60       40      60
shipping    -30   -20       30      70
receiving    50   -30       20      80
```

Figure 7.13 (part 2)

The First Solution for the Five Department Layout Problem: Assert Order = Flow., Placement Order = Only

```
Analysis of flow cost:

FROM       TO          FLOW   LENGTH   COST
cutting    printing     120       50   6000
TOTAL                                   6000

FROM       TO          FLOW   LENGTH   COST
packing    shipping     280        0      0
TOTAL                                      0

FROM       TO          FLOW   LENGTH   COST
printing   cutting       15       50    750
printing   packing      300        0      0
TOTAL                                    750

FROM       TO          FLOW   LENGTH   COST
receiving  cutting      200       10   2000
receiving  printing      20        0      0
TOTAL                                   2000

The total flow cost is 8750.

Find another solution: Yes=> n

| ?-
```

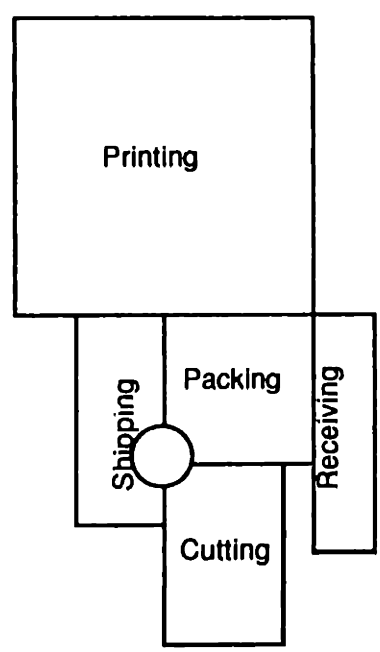


Figure 7.14 (part 1)

An Interaction for the Five Department Layout Problem Within a Constraining Boundary:
Assert Order = Targets , Placement Order = Targets.

Description.

```
| ?- desc.

This factory has a length of 200 and a depth of 200.

It has the following departments:

Department  X-Length  Y-Length  Area
receiving         20        80   1600
printing         100       100  10000
cutting           40        60   2400
packing           50        50   2500
shipping          30        70   2100

The following shipping relationships exists:

From       To        Volume
receiving  printing       20
receiving  cutting       200
printing   cutting        15
printing   packing       300
cutting    printing      120
packing    shipping      280
yes
```

Figure 7.14 (part 2)

An Interaction for the Five Department Layout Problem Within a Constraining Boundary: Assert Order = Targets , Placement Order = Targets

```
| ?- placedepts.


Which assert order: targets
(Departments by number of targets)

We find the number of targets that each department has.

Department  Targets
printing          3
packing           2
cutting           2
receiving         2
shipping          1

Nominally placement will occur in that order.


Which placement order: targets
(Place targets around departments)

After placing a department, attempt to place around
it departments that it ships to.  The following
departments ship to the corresponding departments:


receiving to:  printing and cutting

printing to:  receiving, cutting and packing

cutting to:  receiving and printing

packing to:  printing and shipping

shipping to:  packing


Constrained departments :

Which department ? :
```

Figure 7.14 (part 3)

An Interaction for the Five Department Layout Problem Within a Constraining Boundary:
Assert Order = Targets , Placement Order = Targets.
The First Solution.

```
DEPARTMENT   X     Y     Length   Height
printing     0     0      100      100
receiving    0    100      20       80
cutting     -40    70       40       60
packing      50   100       50       50
shipping     30   150       70       30


When printing was placed:
        receiving was placed near it.
        cutting was placed near it.
        packing was placed near it.

When packing was placed:
        shipping was placed near it.



Analysis of flow cost:


FROM      TO         FLOW   LENGTH   COST
cutting   printing    120        0      0
TOTAL                                   0

FROM      TO         FLOW   LENGTH   COST
packing   shipping    280        0      0
TOTAL                                   0

FROM       TO        FLOW   LENGTH   COST
printing   cutting     15        0      0
printing   packing    300        0      0
TOTAL                                   0

FROM        TO         FLOW   LENGTH   COST
receiving   cutting     200        0      0
receiving   printing     20        0      0
TOTAL                                     0

The total flow cost is 0.


Find another solution: Yes=> n

| ?-
```

Figure 7.14 (part 4)

An Interaction for the Five Department Layout Problem Within a Constraining Boundary:
Assert Order = Targets , Placement Order = Targets

A Block Diagram for the First Solution

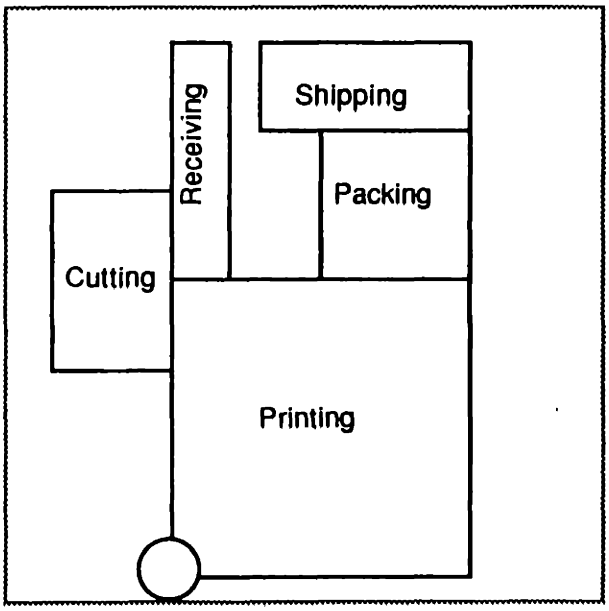


figure 4.10. The only difference between the two factory and layout specifications was the size of the factory. The first two solutions of figure 4.10 would not fit within a 200 x 200 factory so they failed and invoked backtracking.

Figure 7.15 shows a shortened interaction and block diagram for the five department case of figure 7.11, but with the factory size reduced. The nominal ordering is by flow and the placement order is by targets. Comparing the block diagrams of figures 7.11 (part 3) and figure 7.15, it can be seen that three departments, cutting, receiving and shipping, had to be moved in order to accommodate them within the factory boundary. Therefore this layout must have undertaken a more substantial amount of backtracking.

Appendix 3 shows the input file and two sample interactions for the body shop of an automobile factory. The body shop is where the major metal components of the car body are welded together. One sample interaction is for placing the departments within a very large boundary. The other is for placing the departments within a reasonable boundary. The flow cost within the reasonable boundary is smaller, simply because it forces the placements to cluster more.

7.7.3.5.2 Demonstration of Placement by Effects

So far we have only considered placement by flow. The interaction of figure 7.16 demonstrates placement by effects. The department specification is that of the input file of figure7.9, except that the department size is only 200 x 200. Figure 7.16 (part 1) shows the option selection. We choose a nominal placement order of targets, and subordination by effects. When placement by effects is chosen, the program lists the features which belong to each department. Hostile effects are entered first. These express constraints. If hostile departments cannot be separated then there is no solution and a message is output to the user to explain that the problem is over constrained. Similar features express a preference, but not a constraint. If subordinate placement is not possible, ordinary placement occurs. The hostile effect pair quiet-noisy is input. The similar effects quiet and transportation are input.

Figure 7.15

An Interaction for the Five Department Layout Problem Within a Constraining Boundary: Assert Order = Flow , Placement Order = Targets

```
| ?-  placedepts.

Which assert order ? : flow

Which placement order ? : targets

Constrained departments :

Which department ? :

DEPARTMENT  X     Y     Length  Height
packing       0     0      50      50
printing    -50    50     100     100
shipping    -20   -30      70      30
cutting     -90    50      40      60
receiving   -60   -30      20      80

When packing was placed:
        printing was placed near it.
        shipping was placed near it.

When printing was placed:
        cutting was placed near it.
        receiving was placed near it.

The total flow cost is 0.
```

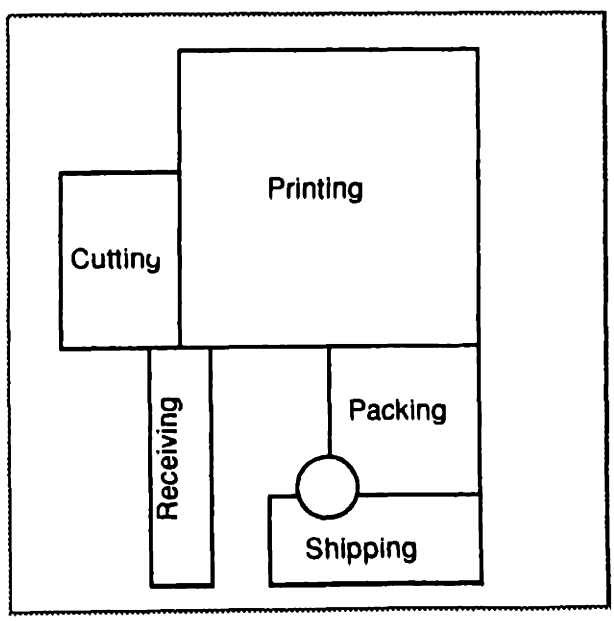


Figure 7.16 (part 1)

Placement by Effects for the Five Department Layout Problem Within a Constraining Boundary: Assert Order = Targets, Placement Order = Effects

```
| ?- placedepts.


Which assert order ? : targets
(Departments by number of targets)

We find the number of targets that each department has.

Department  Targets
printing          3
packing           2
cutting           2
receiving         2
shipping          1

Nominally placement will occur in that order.


Which placement order ? : effects
(Isolate undesirable effects)


The following features belong to the following
departments:

noisy: printing.
quiet: cutting and packing.
transportation: receiving and shipping.


Hostile effect pair ? : quiet-noisy

The following departments have effect: quiet

cutting and packing

The following departments have effect: noisy

printing

These departments will not be placed next to each other


Hostile effect pair ? :
```

Figure 7.16 (part 2)

Placement by Effects for the Five Department Layout Problem Within a Constraining Boundary: Assert Order = Targets , Placement Order = Effects

```
Which effect ?: quiet
(quiet)

Attempt to place the following departments together:

cutting and packing

Which effect ?: transportation
(transportation)

Attempt to place the following departments together:

receiving and shipping

Which effect ?:


After placing a department, attempt to place around
it departments that have similar features (such as
fume-producing or noise producing).  The following
departments have similar features:

receiving and shipping

cutting and packing


Constrained departments :

Which department ? :


<This problem is over-constrained>
yes


| ?- listing(hostile).

hostile(cutting,printing).
hostile(packing,printing).
yes

| ?- listing(subordinate).

subordinate(cutting,packing).
subordinate(packing,cutting).
subordinate(receiving,shipping).
subordinate(shipping,receiving).
yes
```

Figure 7.16 (part 3)

Placement by Effects for the Five Department Layout Problem Within a Constraining Boundary: Assert Order = Flow , Placement Order = Targets, Constrained Department = Shipping

```
Constrained departments :

Which department ? : shipping
        (shipping)

X-Coordinate: 0
Y-Coordinate: 0

Which department ? :

DEPARTMENT  X     Y     Length  Height
shipping      0     0      30      70
printing    -70    70     100     100
packing     -50     0      50      50
cutting     -90   -10      40      60
receiving   -50   -20      80      20

When packing was placed:
        cutting was placed near it.

When shipping was placed:
        receiving was placed near it.
```

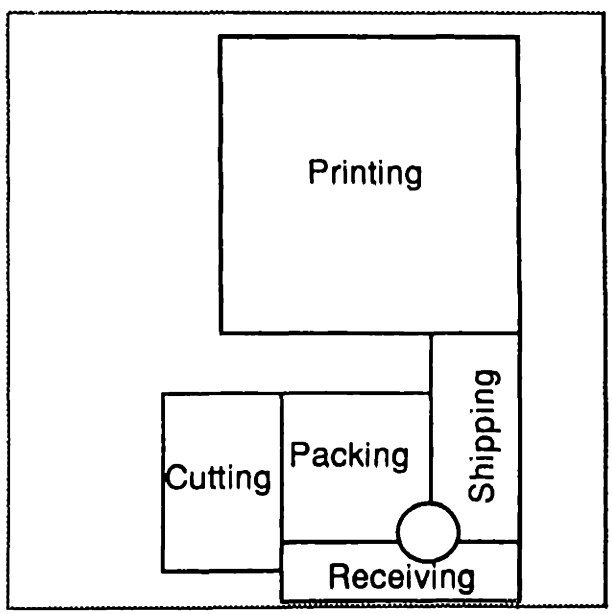


Figure 7.16 (part 2) shows that the solution fails and the problem is overconstrained. The listing of the hostile departments and the subordinate deaprtments shows why. Printing, the first department in the nominal placement order is placed at (0,0). Printing has no subordinates, but it has hostile relations to cutting and packing. The next department in the nominal placement ordering is packing. The only available locations are those bordering the only placed department, which is printing. Since printing and packing are hostile, packing cannot be placed and the solution fails. This software package finds all possible placements through backtracking for a given department ordering, but it does not have the capability to change the department ordering to suit. It is not necessary to do so if we only consider desired adjacencies. We could define hostile placement to be a priority but not a constraint, in which case the solution would just place printing and packing adjacent and then proceed. Or we could change the department ordering. The latter makes the solution ordering somewhat arbitrary. The former means that we print solutions with hostile departments adjacent, when non-adjacent solutions may exist. Therefore we prefer to print that the solution is overconstrained for the current ordering. Examination of the subordinate and hostile relationships, tells us which department does not have any hostile relations, and is therefore a good candidate for constraining as the first department. In figure 7.16 (part 3), we constrain shipping to the coordinates (0,0). A successful solution is generated in which packing is adjacent to cutting, by effect quiet, and receiving is placed next to shipping by effect transportation. Printing is not adjacent to either of its hostile departments, cutting and packing.

7.7.4 Prolog - Implementation Issues

The subordinate relationships were asserted as predicates into the database. Once a location for a department was found, the location was asserted. The assertions were not done in an arbitrary manner which could violate the properties of the logic formulation. Although the assert command does not backtrack in Prolog, all assertions were coded with appropriately placed

retract statements so that if Prolog were to backtrack the assertion would be removed. Therefore all of the subordinations and locations could have been expressed in lists. List operations automatically backtrack. A list is a function and the list function can be expressed in logic. However list processing can be inconvenient. In Prolog programming we face the choice of carrying data in lists, asserting and retracting into the database, or saving entire files. The choice is one of consultation time and processing time. Suppose that a predicate generates a long list representing the locations of all of the departments, through refutation theorem proving, and that this list will be required later in the solution for say a cost analysis. Further, suppose that there is some intermediate function which must be carried out which does not require the solution list. If we adhere to a pure logic representation then either the list must be carried from the predicate which generated the list through the intermediate function to the cost analysis function, or it must be regenerated in the cost analysis function. Carrying the extra information through the intermediate predicate is cumbersome. Regenerating the result is time consuming. Alternatively, the list could be asserted into the database and consulted by the cost analysis predicate, although this would violate the pure logic representation. The FDA used assertion and retraction as a substitute for the list function. This permitted the convenience of being able to consult the subordinate relationships from the database rather than repeatedly deducing them from the frames and shipping relationships. They were encoded in such a way that backtracking was allowed, so that the spirit of the logic representation was not violated.

7.7.5 A Critique of the Factory Design Advisor

The Factory Design Advisor allows us to express machine selection, constrained departments, flow, isolation of effects, and sharing of similar resources through adjacency of effects. Through the nominal ordering and the subordinate placement, the placement strategy appears as follows:

Nominal Order B,C,...

Rules relating to departmentB:

Dept B adjacent to Dept A : Minimize flow
Dept B adjacent to Dept C : Minimize flow
Dept B adjacent to Dept F : Minimize varieties of handling equipment

Rules relating to departmentC:

Dept C adjacent to Dept A : Share resources
Dept C adjacent to Dept B : Minimize flow
Dept C adjacent to Dept G: Minimize varieties of handling equipment

... etc.

where B and C represent two departments in the nominal ordering and the other departments represent their subordinates.

The nominal ordering could be by flow or by number of flow targets. We could also encode other nominal orderings, such as number of targets by effects. Unless we place number values on effects and flow, we cannot order the departments nominally by more than one type of objective.

In order to try and place the targets of a department as we place that department we often overrule the nominal placement ordering. In the current system the targets are not placed in any particular order. The solution could probably be improved substantially if we ordered the subordinate predicates according to size of flow, if minimize flow was an objective. In fact if we expressed an ordering of the adjacency objectives, for example,

minimize flow > group transportation > group quiet

within the subordinate list of each nominal department, then the placement of the subordinate departments would be much more systematic.

There are many extensions which we could propose to the system because the framework is very flexible. We could extend the department description to include flow faces, which tell us the flow direction of a department. Then we

could try to place the outward flow face of one department next to the inward flow face of the next. This is important in, say, automobile factories which often have long departments which have a definite flow direction associated with them. This would be one way to express an orderly flow objective. The program would always attempt to place the orderly flow faces adjacent before trying other placements.

Ordinary placement after subordination has failed could be improved by searching for the nearest location to a desired department when an adjacent location is not available. Currently it just finds any placement.

More search techniques, combined with the current forward strategy would improve the solutions generated by the algorithm. Third placement by the forward strategy only considers unplaced departments for the second and third placements. (This is illustrated in the example of Appendix 3). Some backward search techniques which would be implemented by storing partial paths would also improve the solution (section 6.7), but this could only be achieved at the expense of a more complex control strategy and more storage space. The explanation module could be extended to describe the reason for each subordination. The backtracking search allows generation of further solutions, but there are a large number of solutions and the first ones are achieved by altering the placement of only the last department. Larger variations from the first solution occur when the locations of departments placed early in the solution are altered. This could be achieved by allowing the user to intervene in the backtracking, by selecting the department which they would like to be placed in a new location. This would cause the selected department to backtrack to a new location, and all subsequent departments would then be placed in the normal manner. We can achieve the same effect in the current system by selecting constrained departments.

Summary of Chapter 7

The purpose of the FDA, as a prototype system, has been to demonstate the suitability of logic programming for expression of the factory design problem

in software. A Generate and Test Approach and domain dependent and independent knowledge were unified, through expression in a logic programming language, to demonstrate a practical extension of the factory design model expressed as a decision problem in the formal logic models of Chapter 5. The FDA, generates machinery selections and layouts. A complete and guided neighboring location generator was coded for the layout module. It used a simple forward type of hybrid search strategy. It encoded a selection of placement strategies, one of which attempted to cluster departments which were linked by material flow or sharing common resources. These placement strategies fared better than simple unintelligent placement, but could be improved by augmenting the forward type of hybrid search strategy with a limited amount of backtracking and storage of partial paths.

The FDA makes the objectives of the design explicit. The objectives which may be expressed in the current system are the preferential objectives, *minimize flow* and *place departments requiring resources adjacent* and the constraining objectives *separate hostile effects* and *constrain existing departments*. Preferential objectives simply express a preference of one solution over another, but do not reduce the number of solutions to be evaluated. Constraining objectives reduce the number of acceptable solutions by augmenting the decision problem with an additional constraint. Both qualitative and quantitative objectives were expressed.

The FDA uses a flexible frame structure which allows both domain dependent and domain independent knowledge to be expressed. The domain independent code may be completely decoupled from the domain dependent code so that the system is easily portable from domain to domain. Domain independent and domain dependent knowledge was coded for machine selection. Only domain independent knowledge was encoded for factory layout. It was suggested how this knowledge could be augmented by more detailed domain dependent layout knowledge.

An explanation module explains how the program went about laying out the departments, explaining the decison to place each department. An analysis of

the flow cost of each layout is provided. The system was validated by a case study in automobile assembly plant layout.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

A logic based approach to factory design has been demonstrated. The logic based approach to factory design consisted of

(i) a formal mathematical model of the decision problem for factory design (is a design possible?),
(ii) expression of optimization criteria in logic,
(iii) expression in logic of a Generate and Test procedure for selecting machines and generating factory layouts,
(iv) demonstration of a prototype software system for factory design, coded in the logic programming language Prolog.

Items (ii) and (iii) were not stated explicitly in formal logic, but were expressed in the code of the prototype software system, the Factory Design Advisor. This system also encoded domain dependent knowledge.

The formal mathematical model of the decision problem for factory design consisted of a preliminary process planning model, a machine selection model and a factory layout model. All three phase models represented a selection process from a finite domain of alternatives. They did not have any infinitely recursive or generative properties to render them unbounded. It could therefore be proved that each phase was computable, and that the factory design problem was computable.

The interdependency of the three models was shown. An interpretation of individual variables in the preliminary process planner was represented in

the machine selection model as a set of individual constants. Similarly, an interpretation of individual variables in the machine selection model was represented in the factory layout model as another set of individual constants.

It was shown that there are limits to the amount of domain dependent knowledge which can be used to lay out a factory. The functional requirements of the products to be made in the factory do not affect the layout stage. The layout stage depends only upon knowledge about machines, the flows between them, their characteristics and spatial constraint knowledge.

Two types of knowledge were described, parameter knowledge and constraint knowledge. Constraint knowledge reduces the number of solutions which are possible. Parameter knowledge merely expresses placement preferences and only reduces the number of configurations which we may wish to search by virtue of techniques such as Branch and Bound and A*. Unfortunately a large proportion of factory layout knowledge is parameter knowledge.

When a large amount of factory layout knowledge is generalizable, a software model is easily transferable from domain to domain. But if there is little domain specific layout knowledge to constrain the number of different interpretations, evaluation of the optimal solution may be intractable.

A complexity analysis proved that the Factory Layout Problem expressed as the problem of placing objects of various sizes on a finite grid was NP-complete in the number of grid squares. It was also shown that Generate and Test methods, which generate and test only locations neighboring placed departments, were intractable.

It was determined that a hybrid search strategy was most appropriate for finding satisficing solutions. With large numbers of departments, such a strategy would have to focus on depth first search with limited probing breadthwise because all solutions to the layout problem lie at the lowest level of a search tree of depth equal to the number of departments. It was shown that a hybrid strategy for layout would find solutions in linear time. A forward strategy in which placement methods are predetermined was the most appropriate strategy for coding in a logic programming language because it did not require a sophisticated control structure.

Three conjunct ordering theorems implied that the most effective way to achieve a solution with a large number of desired adjacencies, was to try to cluster departments which were linked by material flow or sharing common resources.

The prototype software system, the FDA, generated machinery selections and layouts. With each layout it provided an evaluation of the cost of material flow in the layout and an explanation of the layout procedure for each individual layout. The FDA demonstrated the flexibility of the Generate and Test approach and the suitability of logic programming for rule expression, frame expression, and explanation. In the layout module, it used a simple forward type of hybrid search strategy. It encoded a selection of placement strategies, one of which attempted to cluster departments which were linked by material flow or sharing common resources. It encoded domain independent and domain specific knowledge for machine selection. It encoded domain independent knowledge for factory layout and demonstrated how the domain independent knowledge could be augmented by more detailed domain dependent layout knowledge. Both quantitative and qualitative objectives were expressed.

Logic provides a formal framework for organizing what is often considered as ill structured, informal knowledge. It helps us to understand better the structure and properties of what was previously identified as informal. Using a logic based approach we were able to unify the decision problem for factory design, the optimzation problem, and domain dependent and independent factory design knowledge in a logic framework. This helped us to decouple the different types of knowledge and to examine the impact of knowledge based systems for reducing the intractability of the design problem.

8.2 Future Work

Several suggestions for extending and improving the prototype software system were mentioned in section 7.7.5. The most major improvement in performance would be obtained by coding a hybrid search strategy which allowed a limited amount of backtracking and storage of partial paths. It would be interesting to determine how much improvement was obtained in the quality of the solution with an increase in the number of levels of backtracking allowed, by using just one objective, minimize flow, and comparing the flow costs of the layouts. An experiment of this type should be performed on various different types of layout. The expansion of the search space through limited backtracking, would make the satisficing "less local" in terms of the options explored. By coding the *attain orderly flow* objective and a more sophisticated distance calculation function which prevents conveyor paths from being crossed, the system should be capable of emulating the major objectives of automobile assembly plant layout. It would then be interesting to compare the system designs with those of the layout engineer. The process of obtaining knowledge should run concurrently with the development of the software system. One of the best ways of extracting knowledge is to take a

sample layout and explanation produced by the software system to a factory designer, and to ask the layout engineer to identify what is wrong with the layout and the reasoning mechanism behind it. Then the engineer has something to which to relate his or her experience. In this way we may discover some domain dependent knowledge which was not apparent before. It would be interesting to determine whether the domain dependent objectives can capture enough of the character of the automobile assembly plant problem to generate good layouts, since the outlook for domain dependent layout knowledge was bleak. The true test of a practical software system is to determine whether it can perform a useful function for the users it is designed to serve, the factory designers, in the opinion of those whom it is designed to serve.

A more sophistcicated study of the probability of node expansion in the layout problem, based upon the number of desired adjacecies, and the number of departments already placed, may help to give better insight into the practical complexity of the problem. It may help us to develop more intelligent search strategies.

Intelligent manufacturing systems demand the integration of all of the components of a manufacturing system in order to make intelligent real time decisions on process plans and schedules and to determine when physical components of the factory or their arrangement have ceased to serve efficiently. A computer representation of a factory interpretation should form the basis of process planning and scheduling systems.

The preliminary process planning model of chapter 5 forms the skeleton of both a preliminary process planning model and a process planning model which could be programmed in logic in the same manner as the machine

selection and layout phases. Using the soundness and completeness properties of the logic model and its integration with the layout model through logic, we could determine whether a particular plan was possible and whether a part could be made in the system at all. We could generate all possible plans for a new part under the current interpretation of the layout model which represents the physical factory. The interdependence of the logic models through the interpretation of one phase being expressed as individual constants in the model of the next phase, implies that the models do not interfere with each other in terms of syntax, or in terms of the semantics of function and predicate constants. That is, the models are easy to decouple, so that integration should not pose a problem. However, an investigation into the practicality of integration of the major manufacturing functions of layout, process planning and scheduling would be very informative.

REFERENCES

Abelson, H and G.J.Sussman. *Structure and Interpretation of Computer Programs.*. MIT Press, 1985.

Apple, J.M.. *Plant Layout and Material Handling,* 3rd edition. New York: John Wiley and Sons, 1977.

Arbib, M.A.. *Theories of Abstract Automata.* NJ: Prentice-Hall, 1969.

Armour, G.C. and E.S. Buffa. A Heuristic Algorithm and Simulation Approach to Relative Location of Facilities. *Management Science*, v.9(1), pp294 - 309, 1963.

Bratko, I..*Prolog Programming for Artificial Intelligence* . Addison -Wesley, 1986

Brady, J.M.. *The Theory of Computer Science.* London: Chapman and Hall, 1977.

Clocksin , W.F. and C.S. Mellish. *Programming in Prolog.*, 2nd edition. NY: Springer-Verlag, 1984.

Davis, R. and D.B.Lenat. *Knowledge Based Systems in Artificial Intelligence.* McGraw-Hill, 1982.

DeGroot, D. and G. Lindstrom. *Logic Programming.* NJ: Prentice-Hall, 1986.

Farber, J. and E.L. Fisher. *MATHES: Material-Handling Equipment Selection Expert System.* Technical Report. NCSU-IE-85-17, Dept. of Industrial Engineering, North Carolina State Univ, 1985.

Fisher, E.L. An AI-Based Methodology for Factory Design. *AI Magazine,* Fall 1986: pp. 72-86.

Foulds L.R.. Techniques for Facilities Layout: Deciding Which Pairs of Activities Should be Adjacent. *Management Science*, v.29 (12), 1983.

Francis, R.L. and J.A. White. *Facility Layout and Location.* NJ: Prentice-Hall, 1974.

Genesereth, M and N. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufman, 1987.

Giffin, J.W.. *Graph Theoretic Techniques for Facilities Layout*, Unpublished Dissertation, University of Canterbury, Christchurch, New Zealand, 1984. (Reported in [Foulds, 1983]).

Giffin, J.W., L.R. Foulds and D.C. Cameron. Drawing a Block Plan from a REL Chart with Graph Theory and a Microcomputer. *Computers and Industrial Engineering*, v.10(2), pp. 109 - 116, 1986.

Harel, D. *Algorithmics.* Addison-Wesley, 1987.

Hillier, F.S. and M.M. Connors. Quadratic Assignment Problem Algorithms and the Location of Indivisible Facilities, *Management Science*, v.13 (1), pp.42-57, 1966.

Jackson, P *Introduction to Expert Systems*. Addison-Wesley, 1986.

Kim, S.H. *Mathematical Foundations of Manufacturing Science: Theory and Implications* . PhD Thesis, MIT, 1985.

Kim, S.H.. *A General model of Design: Formalization and Consequences.* M.I.T. Laboratory for Manufacturing and Productivity Technical Report, 1986.

Kim, S.H. and N.P. Suh. On an Expert System for Design and Manufacturing. *Proc. COMPINT '85,* ACM AND IEEE/Computer Society, Montreal, Canada, Sept. 1985.

Knuth, D.E.. *The Art of Computer Programming - V.2 Semi-Numerical Algorithms*, 2nd edition. Addison-Wesley, 1981.

Kowalski, R., *Logic For Problem Solving*. North-Holland, Amsterdam, 1979.

Lawler, E.L.. The Quadratic Assignment Problem. *Management Science*, v.9 (4), pp. 586 - 599, 1963.

Lawler, E.L., and D.E Wood. Branch and Bound Methods: A Survey. *Operations Research*, v.14, pp. 699 - 719, 1966.

Lee, R.C. and J.M. Moore. CORELAP - Computerized Relationship Layout Planning. *Journal of Industrial Engineering*, v.18(3), pp.147 -159, 1967.

Lewis, H.R. and C.H.Papadimitriou. *Elements of the Theory of Computation.* NJ: Prentice-Hall, 1981.

Malakooti,B and D'souza,G. *Computerized Facilities Planning* (ed) H.Lee Hales, Institute of Industrial Engineers, 1985.

Manna, Z. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

McNaughton, R. *Elementary Computability, Formal Languages and Automata.* NJ: Prentice Hall, 1982.

Mostow, J.. Toward Better Models of The Design Process. *AI Magazine*, Spring, 1985.

Muther, R. *Systematic Layout Planning.* New York: McGraw Hill, 1961.

Newell, A.. *Heuristic Programming Ill-structured Problems, Progress in OR,* Aronofsky, (ed.), New York, John Wiley and Sons,1969.

Newell, A. and H.A. Simon. *Human Problem Solving.* Englewood Cliffs, N.J.:Prentice-Hall, 1972.

Olsen, W.W. and W.R. DeVries. Logical Basis for Process Planning. *Proceedings of the IXth International Conference on Production Research,*

Cincinatti, Ohio, 1987.

Pearl, J. *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Mass. Addison-Wesley, 1984.

Robinson, J.A.. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, v.12(1), pp.23-41, 1965.

Rosenblatt, M. . The Facilities Layout Problem: a multi-goal approach. *Int. J. Prod. Res.*. v.17(4), pp. 323-332, 1979.

Sahni, S. and T. Gonzales. P-Complete Approximation Problems. *J. Assoc. Comput. Mach.* v.23, pp.555 - 565, 1976.

Smith, D.E and M.R. Genesereth. Ordering Conjunctive Queries. *Artificial Intelligence*, v.26, pp 171-215, 1985.

Sterling,L and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

Suh, N.P.. Development of the Science Base for the Manufacturing Field through the Axiomatic Approach. Robotics and Computer Integrated Manufacturing, V.1, No 3/4, pp.397-415, 1984.

Tirupatikumara, S.R.. *Artificial Intelligence Techniques in Facilities Layout Planning: The Development of an Expert System.* PhD Thesis, Purdue University, 1985.

Tompkins, J.A. and J.A. White. *Facilities Planning.* New York: Wiley, 1984.

Tompkins, J.A. and J.M. Moore. *Computer Aided Layout: A User's Guide*, Facilities Planning and Design Monograph Series, N0. 1, AIIE, 1978.

Ulrich,K. and W. Seering. Computation and Conceptual Design. Presented at the International Conference of Manufacturing Science and Technology of the Future, Cambridge, MA, 1987. To be printed in *Robotics and Computer-Integrated Manufacturing*, v. 4, 1988.

van Canegham, M and D.H.D. Warren. *Logic Programming and its Applications*. NJ: Ablex, 1986.

van Laarhoven, P.J.M. and E.H.L. Aarts. *Simulated Annealing: Theory and Applications*. Dordrecht, Holland: Reidel, 1987.

Winston, P.H.. *Artificial Intelligence*. 2nd ed. Mass. : Addison-Wesley, 1984.

APPENDIX 1

THE HALTING PROBLEM OF TURING MACHINES

Theorem (Turing) : The Halting Problem of Turing machines is undecidable

Proof: (Minsky)

Denote by d(M) the encoded description (program) of Turing machine M over Σ. Denote by d(M)*w the encoded description of M and input word w over Σ.

The Proof is by contradiction. Suppose there exists such an algorithm (Turing machine), A. Then for every input d(M)*w to A, we have: If M halts for input w, then A reaches an ACCEPT halt; if M does not halt for input w then A reaches a REJECT halt (figure (a)).

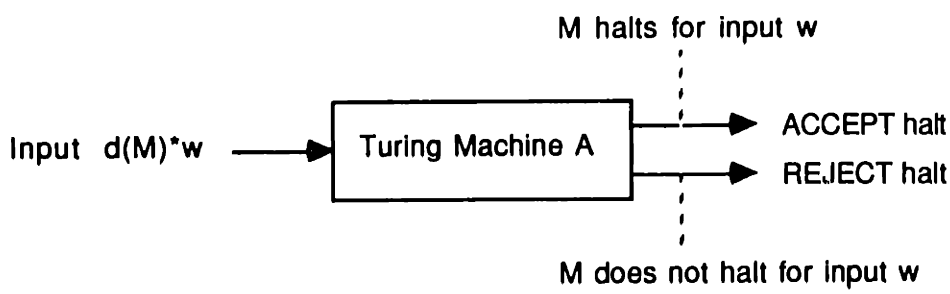


figure (a)

Now a new Turing machine B can be constructed which takes d(M) as input and proceeds as follows: First it copies the input to obtain d(M)*d(M), and then it applies Turing Machine A on input d(M)*d(M) with one modification. Whenever A is supposed to reach an ACCEPT halt, instead B loops forever. (figure (b)).

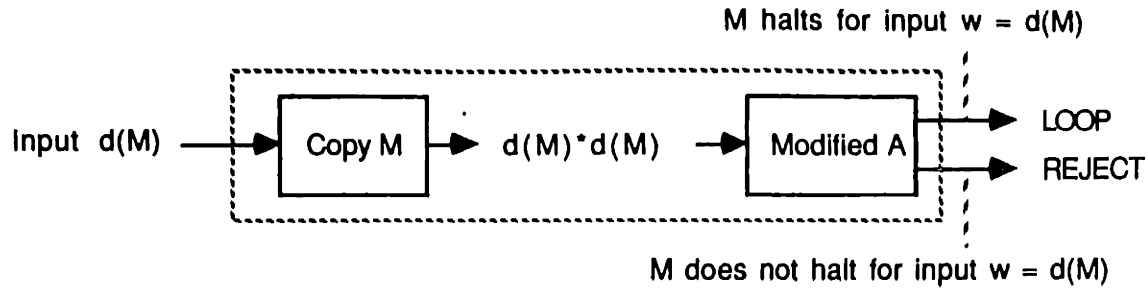


figure (b)

The above discussion holds for any Turing machine M, over Σ. Since B is a Turing Machine, let M=B; then replacing B for M in figure (b) we obtain the situation of figure (c).

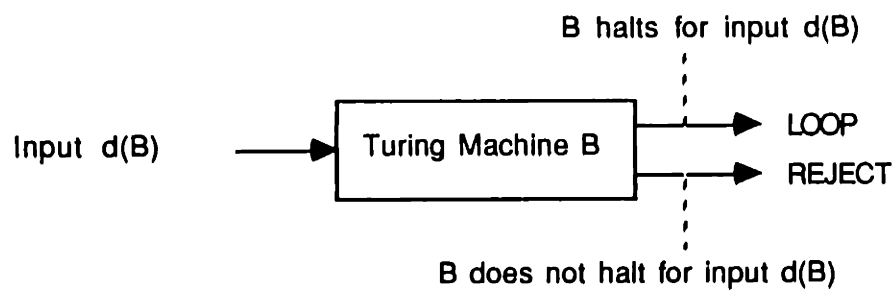


figure (c)

Thus B halts for input d(B) if and only if B does not halt for input d(B). This is a contradiction. Therefore our supposition that there is an algorithm A that determines whether an arbitrary Turing machine halts for an arbitrary word w is false. The Halting problem of Turing machines is undecidable.

End of Proof

APPENDIX 2

GENERAL CATEGORIES OF FACTORY LAYOUT

Product and process layouts represent the traditional extremes of factory layouts.

Product Layout

In what is typically known as a product layout, the departments are arranged in the sequence of the operations on the product, e.g. in traditional automobile assembly plants. The objective of orderly material flow is achieved, even if workstations have to be duplicated so that machinery is not fully utilized. This type of layout is traditionally used for high volume production of a low variety of products. A change in product design may require considerable alteration of the layout.

Process Layout

In a process layout, the arrangement of the process departments dictates the flow of the product. This is typically the arrangement in a job shop which has departments with general purpose machinery such as turning, milling and grinding departments. A large variety of products are produced in relatively small quantities. Because the bottlenecks of sequential flow do not arise, high equipment utilization is obtained, but the flow path is less orderly. The layout will accommodate the addition of new machinery easily, because the material handling system has routing flexibility.

Mixed Layouts

Typically a layout exhibits some characteristics of both process and product layouts. The Group Technology approach aims to improve the traditional

process layout by arranging machine groups for better scheduling, tooling, routing and machine setup.

APPENDIX 3

A SAMPLE INTERACTION FOR AUTOMOBILE ASSEMBLY PLANT LAYOUT

A sample interaction for automobile assembly plant layout with the Factory Design Advisor is presented in the figures of this Appendix. The input file is shown in Appendix 3 - figure 1. The departments described are :

underbody weld,
wheelhouse sub assembly,
left side frame,
right side frame,
body framing,
assembly line,
primer dip and
transportation.

These are the major departments used to weld the car body together before it is sent for painting. Transportation is represented as a long thin department. It is usually a railroad track which runs through the factory, delivering the metal components daily. Since it is not usually possible to place departments at the ends of the transportation department two solutions are presented. The solution in figure 2 of Appendix 3 is constrained within a boundary equal to the length of the transportation department in one direction, and equal to or smaller than the length of the transportation department in the other direction. This prevents placement of departments at the ends of transportation, because such placements would not fit within the factory boundary. The solution of figure 3 presents the unconstrained problem for comparison.

The constrained solution is shown in Appendix 3 - figure 2 (part 3) and illustrated in the block diagram, of Appendix 3 - figure 2 (part 5). Transportation was placed first. When the subordinate department underbody weld was placed, body framing was placed by third placement. When the subordinate department wheelhouse was placed, assembly was placed by third placement. The explanation module could be more helpful in this respect. However by examining the shipping relationships of Appendix3 - figure 2, and the block diagram, it is appparent when third placement occurs. When the subordinate department to transportation, left side frame, was placed, it was placed by two way placement. A third department which relates to both transportation and left side frame is body framing, but this had been placed by third placement with underbody weld and transportation. This demonstrates the limitations of a forward strategy with no search. Instead of noting that body framing and transportation had already been placed and searching for a location adjacent to both of them by third placement, the left side frame settles for two way placement with transportation. When the explanation module is extended to explain third placement it would note that, when left side frame was placed, body framing had already been placed. The block diagram of Appendix 3 - figure 2 (part 5) shows how all departments cluster around the transportation department. The cost analysis of Appendix 3 - figure2 (part 4) shows that only the left and right side frame were not placed adjacent to one desired neighbor. In both cases this was because a third placement was desired but not achieved. They were both placed by two way placement with transportation. The unconstrained example of Appendix 3 - figure 3 shows that, in this case, another adjacency was not met. The adjacency between body framing and assembly in the constrained solution was incidental. Both these departments were placed by third placement with two other departments. Therefore the adjacency between these departments could not be found by the forward strategy.

Appendix 3 - Figure 1

Input File for an Automobile Factory

```
ubodyweld ::
        [ dept_type      : ubody,
        dimensions       : [40,50],
        features         : [reinforcedfloor]].

wheelhouse ::
        [ dept_type      : welding,
        dimensions       :[30, 30]].

lsideframe ::
        [ dept_type      : welding,
        dimensions       :[40, 20]].
rsideframe ::
        [ dept_type      : welding,
        dimensions       :[40, 20]].

bodyframing ::
        [ dept_type      : welding,
        dimensions       :[60, 20],
        opaccess         : [side,roof,front,back]].

assembly ::
        [ dept_type      :assembly,
        dimensions       :[60,90],
        opaccess         :[side,roof,front,back],
        features         :[expanding,dustextraction].

primerdip ::
        [ dept_type      :process,
        dimensions       :[30,30],
        opaccess         :[underneath],
        features         :[reinforcedfloor]].

transportation ::
        [ dept_type      :transportation,
        dimensions       :[20,200]].

ships(transportation, ubodyweld, 44).
ships(ubodyweld,bodyframing,45).
ships(transportation,lsideframe,20).
ships(transportation,rsideframe,20).
ships(transportation,wheelhouse,2).
ships(lsideframe,bodyframing,22).
ships(rsideframe,bodyframing,22).
ships(transportation,bodyframing,11).
ships(bodyframing,assembly,110).
ships(transportation,assembly,80).
ships(assembly,primerdip,190).
ships(wheelhouse,assembly,2).
factory(2000,2000).
```

Appendix 3 - Figure 2 (part 1)

Sample Interaction for the Automobile Factory - Description

```
| ?- desc.

This factory has a length of 200 and a depth of 200.

It has the following departments:

Department      X-Length  Y-Length  Area
ubodyweld             40        50  2000
wheelhouse            30        30   900
lsideframe            40        20   800
rsideframe            40        20   800
bodyframing           60        20  1200
assembly              60        90  5400
primerdip             30        30   900
transportation        20       200  4000

The following shipping relationships exists:

From            To              Volume
ubodyweld       bodyframing         45
wheelhouse      assembly             2
lsideframe      bodyframing         22
rsideframe      bodyframing         22
bodyframing     assembly           110
assembly        primerdip          190
transportation  ubodyweld           44
transportation  wheelhouse           2
transportation  lsideframe          20
transportation  rsideframe          20
transportation  bodyframing         11
transportation  assembly            80
yes
```

Appendix 3 - Figure2 (part 2)

Sample Interaction for the Automobile Factory - Option Selection

```
| ?- placedepts.

Which assert order ? : targets
(Departments by number of targets)

We find the number of targets that each department has.

Department      Targets
transportation        6
bodyframing           5
assembly              4
rsideframe            2
lsideframe            2
wheelhouse            2
ubodyweld             2
primerdip             1

Nominally placement will occur in that order.

Which placement order ? : targets
(Place targets around departments)

After placing a department, attempt to place around
it departments that it ships to.  The following
departments ship to the corresponding departments:


ubodyweld to:  bodyframing and transportation

wheelhouse to:  assembly and transportation

lsideframe to:  bodyframing and transportation

rsideframe to:  bodyframing and transportation

bodyframing to:  ubodyweld, lsideframe, rsideframe,
                 assembly and transportation

assembly to:  wheelhouse, bodyframing, primerdip
                and transportation

primerdip to:  assembly

transportation to:  ubodyweld, wheelhouse, lsideframe,
                    rsideframe, bodyframing and assembly

Constrained departments :

Which department ? :
```

Appendix 3 - Figure 2 (part 3)

Sample Interaction for the Automobile Factory - First Solution

DEPARTMENT	X	Y	Length	Height
transportation	0	0	20	200
ubodyweld	-40	0	40	50
wheelhouse	-30	170	30	30
lsideframe	20	0	20	40
rsideframe	20	160	20	40
bodyframing	-20	50	20	60
assembly	-90	110	90	60
primerdip	-90	170	30	30

```
When assembly was placed:
        primerdip was placed near it.

When transportation was placed:
        ubodyweld was placed near it.
        wheelhouse was placed near it.
        lsideframe was placed near it.
        rsideframe was placed near it.
        bodyframing was placed near it.
        assembly was placed near it.
```

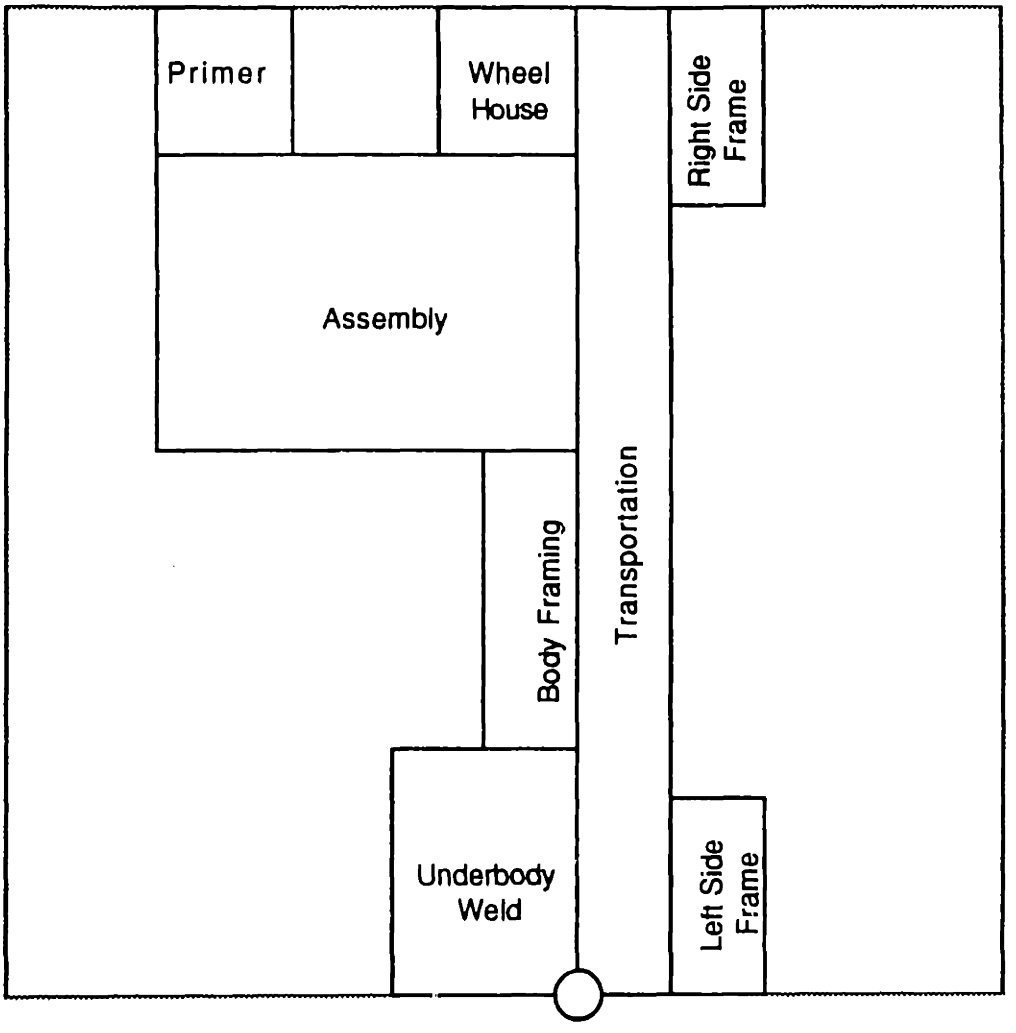
Appendix 3 - Figure 2 (part 4)

Cost Analysis for the Automobile Factory

```
Analysis of flow cost:

FROM        TO           FLOW   LENGTH   COST
assembly    primerdip     190        0      0
TOTAL                                       0

FROM          TO          FLOW   LENGTH   COST
bodyframing   assembly     110        0      0
TOTAL                                       0

FROM         TO            FLOW   LENGTH   COST
lsideframe   bodyframing     22       30    660
TOTAL                                      660

FROM         TO            FLOW   LENGTH   COST
rsideframe   bodyframing     22       70   1540
TOTAL                                     1540

FROM            TO           FLOW   LENGTH   COST
transportation  ubodyweld      44        0      0
transportation  lsideframe     20        0      0
transportation  rsideframe     20        0      0
transportation  wheelhouse      2        0      0
transportation  bodyframing    11        0      0
transportation  assembly       80        0      0
TOTAL                                           0

FROM        TO            FLOW   LENGTH   COST
ubodyweld   bodyframing     45        0      0
TOTAL                                       0

FROM         TO          FLOW   LENGTH   COST
wheelhouse   assembly       2        0      0
TOTAL                                       0

The total flow cost is 2200.

Find another solution: Yes=> n

[ execution aborted ]
```

Appendix 3 - Figure 2 (part 5)

Block Diagram for the Automobile Factory



Appendix 3 - Figure 2 (part 6)

Subordinate Departments in the Automobile Factory

```
| ?- listing(subordinate).

subordinate(ubodyweld,bodyframing).
subordinate(ubodyweld,transportation).
subordinate(wheelhouse,assembly).
subordinate(wheelhouse,transportation).
subordinate(lsideframe,bodyframing).
subordinate(lsideframe,transportation).
subordinate(rsideframe,bodyframing).
subordinate(rsideframe,transportation).
subordinate(bodyframing,ubodyweld).
subordinate(bodyframing,lsideframe).
subordinate(bodyframing,rsideframe).
subordinate(bodyframing,assembly).
subordinate(bodyframing,transportation).
subordinate(assembly,wheelhouse).
subordinate(assembly,bodyframing).
subordinate(assembly,primerdip).
subordinate(assembly,transportation).
subordinate(primerdip,assembly).
subordinate(transportation,ubodyweld).
subordinate(transportation,wheelhouse).
subordinate(transportation,lsideframe).
subordinate(transportation,rsideframe).
subordinate(transportation,bodyframing).
subordinate(transportation,assembly).

yes
```

Appendix 3 - Figure 3 (part 1)

Solution for an Automobile Factory which is not Constrained by a Boundary

```
| ?- desc.

This factory has a length of 2000 and a depth of 2000.
```

DEPARTMENT	X	Y	Length	Height
transportation	0	0	20	200
ubodyweld	-20	200	40	50
wheelhouse	-10	-30	30	30
lsideframe	-20	0	20	40
rsideframe	-20	160	20	40
bodyframing	20	170	20	60
assembly	20	-45	60	90
primerdip	20	45	30	30

```
When assembly was placed:
        primerdip was placed near it.

When transportation was placed:
        ubodyweld was placed near it.
        wheelhouse was placed near it.
        lsideframe was placed near it.
        rsideframe was placed near it.
        bodyframing was placed near it.
        assembly was placed near it.
```

Appendix 3 - Figure 3 (part 2)

Cost Analysis for an Automobile Factory which is not Constrained by a Boundary

```
Analysis of flow cost:

FROM          TO            FLOW   LENGTH   COST
assembly      primerdip      190        0      0
TOTAL                                          0

FROM          TO            FLOW   LENGTH   COST
bodyframing   assembly       110      125  13750
TOTAL                                      13750

FROM          TO            FLOW   LENGTH   COST
lsideframe    bodyframing      22      150   3300
TOTAL                                       3300

FROM          TO            FLOW   LENGTH   COST
rsideframe    bodyframing      22       20    440
TOTAL                                        440

FROM          TO            FLOW   LENGTH   COST
transportation ubodyweld       44        0      0
transportation lsideframe      20        0      0
transportation rsideframe      20        0      0
transportation wheelhouse       2        0      0
transportation bodyframing     11        0      0
transportation assembly        80        0      0
TOTAL                                          0

FROM          TO            FLOW   LENGTH   COST
ubodyweld     bodyframing      45        0      0
TOTAL                                          0

FROM          TO            FLOW   LENGTH   COST
wheelhouse    assembly          2        0      0
TOTAL                                          0

The total flow cost is 17490.

Find another solution: Yes=> n

[ execution aborted ]
```

Appendix 3 - Figure 3 (part 3)

Block Diagram for an Automobile Factory which is not Constrained by a Boundary

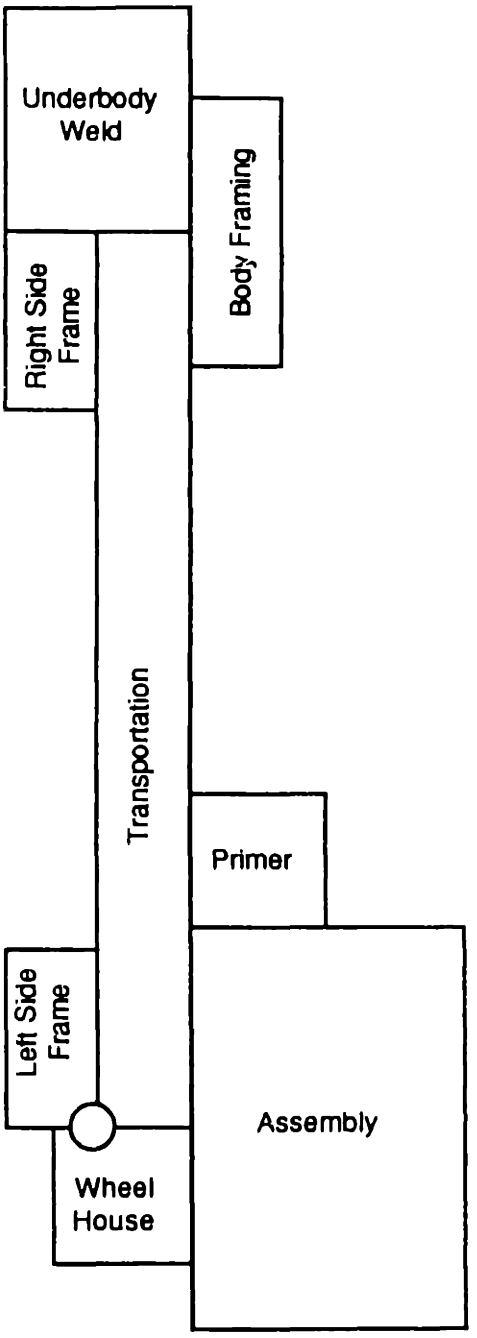


TABLE OF FIGURES

Problem Within a Constraining Boundary: Assert Order = Targets, Placement Order = Effects