1995

Intelligent graph-search techniques: an application to project scheduling under multiple resource constraints

Mohammad Reza Zamani

University of Wollongong

Recommended Citation
NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
INTELLIGENT GRAPH-SEARCH TECHNIQUES: AN APPLICATION TO PROJECT SCHEDULING UNDER MULTIPLE RESOURCE CONSTRAINTS

A thesis submitted in fulfilment of the requirements for the award of the degree

DOCTOR OF PHILOSOPHY

UNIVERSITY OF WOLLONGONG

by

MOHAMMAD REZA ZAMANI
B.S., M.S. (OR)

Department of Business Systems

MAY 1995
“There is an art of reading as well as an art of writing and there should exist a science of thinking.”

- Alkhwarizmi (the ninth-century mathematician and the word algorithm is derived from his name)
DECLARATION

This is to certify that the work presented in this thesis was carried out by the author in the Department of Business Systems of the University of Wollongong, Australia and has not been submitted for a degree to any other university or institution.

Mohammad Reza Zamani
ACKNOWLEDGMENTS

The work toward this thesis was carried out at the University of Wollongong over the period from 1991 to 1995 with the full support of the Department of Business Systems. However, it would never have succeeded without the eight years in which I had the chance to gain experience both as a MS student and a lecturer at CSP (Centre for Systems Planning, Industrial Engineering Department, Isfahan University of Technology). I wish to thank all my former colleagues especially my MS adviser Professor Aashtiani, a brilliant top-ranked MIT graduate, who, as the former head of CSP, initiated my overseas PhD study and supported it all the way.

Ever since I began to take my PhD at Wollongong University, I have become more and more grateful for the support and encouragement from the Department of Business Systems. During these years, because of the kindness and the helpfulness of the staff in this department, I never felt alone as some overseas students did. I sincerely thank all the staff of the department who have been so kind and helpful. I would like to express thanks and appreciation to my adviser Dr. L. Y. Shue for his always open-door policy and his guidance throughout the course of this research. During this long period of time, nearly every day he supervised the progress of this research and provided me with invaluable advice. I would also like to express the deepest gratitude to Professor Graham Winley, the head of the Department of Business Systems, who kindly provided all kinds of facilities that were needed for the accomplishment of this research. Whenever I have met him, he has given me encouragement which has motivated me to keep the quality of this research as high as possible. In addition, he, as a mathematician and operations research practitioner, with his dedicated assistance on providing subtle comments on this thesis, highly enhanced the quality of this research.

These acknowledgments cannot be concluded without expressing thanks to my wife and my two daughters who have tolerated my non-stop work on this research over such a long period of time.
ABSTRACT

The topic of graph-search techniques is of common interest for both Operations Research (OR) and Artificial Intelligence (AI) communities. Solutions found by OR techniques are optimal and precise but may be obtained at the price of high computation time. On the other hand, those found by AI techniques have no guarantee of being optimal but are usually achieved quickly. In this research, these techniques of AI and OR are investigated and a learning search method integrating the features of both fields is presented. The main characteristics of this method are its capacity to learn along the search process and its ability to trade speed with precision to whatever degree desired.

In order to develop this method, we utilise a technique developed by Korf (1990) and expand its learning capability to generate optimal solutions. Then, by the use of a threshold parameter, the features of Korf's AI algorithm and this new OR method are integrated to represent a new learning technique. Through an intensive programming task, the correctness of this integrating method is presented by testing it on three different graphs with different characteristics. A mathematical proof is also developed to show that solutions found by this method are always guaranteed to be within a prescribed range of the optimal solution. This range is determined by the user and can be any positive real number.

The generality of this method allows its application to all graph-search problems including all such combinatorial problems. To show how efficiently combinatorial problems can be solved by this method, it is applied to the problem of Project Scheduling Under Multiple Resource Constraints, which is one of the hardest types in the scheduling area. It is shown that the method is able to solve a benchmark problem of this type manually and that it requires only 25 backtracks to find its optimal solution and only 1 backtrack to find a solution guaranteed to be within the range of 3 units of the optimal one. It is emphasised that this method is superior to all previous techniques which required tens of seconds of computer time to solve the benchmark problem.
PUBLICATIONS ARISING FROM THIS THESIS


PREFACE

This thesis makes three contributions: classifying graph-search efficiency factors, developing two intelligent search techniques, which improve their performance during the process of problem solving through a learning mechanism, and finally showing their efficiency by applying them to a benchmark scheduling problem. These three contributions are presented in five chapters.

In Chapter 1, it is pointed out how proper representation of problems along with appropriate control can increase the efficiency of a search. It is described how these means have been expanded by a number of researchers, and their limitations are examined. The importance of appropriate problem representation and restrictions placed on it by the Von-Neuman structure of today's computers are presented. Control methods for search are classified into six different classes. It is emphasised that a lucky guess which contributes to the efficiency of a search algorithm in one special problem has some chance to be efficient in other areas, and that by classifying known methods according to similarities in their structures, it is possible to identify strategies which often lead to success.

Chapter 2 can be considered as the main contribution of this thesis. In this chapter, two intelligent search techniques are presented, the first one finding optimal solutions and the second one finding solutions guaranteed to be within a desired range of the optimal one. The first technique, LBA* (Learning and Backtracking A*), has been developed by incorporating a backtracking process into an efficient search technique called LRTA* developed by Korf [1990]. LRTA* represents a major research direction which takes into consideration the effect of learning in the search process by utilising initial heuristic estimations and continuously improving them as the search process continues. Despite its power of learning, LRTA* cannot utilise the updated heuristic estimations in the same problem in which they are obtained, and those updated values can only be used in the following problem solving trials. Besides, it neither guarantees optimality nor near-optimality of solutions found. Contrary to the LRTA*, in the LBA* updated heuristic values can be used in the same problem in
which these values have been obtained. A proof has been provided that under very
general circumstances the application of LBA* leads to finding optimal solutions.
As stated above, the problem with LRTA* is that when it finds a solution it gives no
measure of how far this solution may potentially be from the optimal one. Although
LBA* removes this problem and finds optimal solutions, its application to some
combinatorial problems can take a long time. The second technique, LCBA*
(Learning and Controlled Backtracking A*), is an improvement over the first one,
LBA*, that addresses this problem and trades accuracy with speed. LCBA* accepts a
parameter as input and finds a solution within the range specified by the parameter
from the optimal solution. The speed of LCBA* depends on the magnitude of this
parameter. For instance, if this parameter is very large, LCBA* works exactly like
LRTA* and finds a solution without any guarantee of its quality. On the other hand,
if this parameter is selected as zero, LCBA* works like the first technique, LBA*, and
finds an optimal solution by doing all the necessary backtracks. The difference
between LCBA* and many other guaranteed accuracy methods is that this method is
not case sensitive and can be applied to all graph-search problems. A proof is
presented that under very general circumstances the application of this method leads
to guaranteed accurate solutions. If LRTA* is considered as an algorithm in the field
of artificial intelligence, which only finds solutions without any guarantee of their
quality, and LBA* is considered as an algorithm in the field of operations research,
which finds optimal solutions, then LCBA* can be considered as an integrating
approach between these two fields. At the end of Chapter 2, LCBA* is further
improved and two different versions of it are presented.
In Chapter 3, a description of how LCBA* can be applied to solve the problem of
"project scheduling under multiple resource constraints" is provided. This problem
has been identified as the most complex and general problem in the field of
scheduling and occurs not only within industrial organisations but in many business
enterprises as well. Its computational complexity, due to subsuming many scheduling
problems, has transformed it into one of the most difficult to solve problems. At first,
available approaches to solve this problem are described, emphasising that even modest-sized projects will have an enormous number of possible schedules which nullifies control mechanisms used in these approaches. Then it is described, in detail, how this complicated problem can be solved by the general search method of LCBA*.

In Chapter 4, the performance of LCBA* in dealing with a benchmark project scheduling problem is emphasised. How LCBA* manages to find the optimal schedule of this project, which has thousands of potential schedules, in only 25 backtracks and how it finds a solution guaranteed to be within the range of 3 units of the optimal one, 39, in only 1 backtrack, are described. The importance of this performance is further highlighted by the fact that one of the best available methods has taken tens of seconds of computer time to solve this benchmark problem.

In Chapter 5, after the concluding remarks, some suggestions regarding the future directions of this research are put forward. It is suggested that a computer package based on LCBA* to solve this scheduling problem be developed, emphasising that the development of such a program can put a stop to a dead-end that for decades has kept this problem intractable. Further, it is suggested that some modifications be made to LCBA* so that it can be applied to the areas such as theorem proving, diagnosing, and even explaining or justifying results.
TABLE OF CONTENTS

1. APPROACHES TO CLASSIFYING GRAPH-SEARCH EFFICIENCY FACTORS 1
   1.1 Introduction 1
   1.2 Problem Representation 3
   1.3 Intelligent Control Methods 15
      1.3.1 Systematic Search 16
      1.3.2 The Complexity of Search Techniques 18
      1.3.3 Control 21
         1.3.3.1 Heuristic 22
         1.3.3.2 Wise Pruning 24
         1.3.3.3 Reduction 27
         1.3.3.4 Abstraction 29
         1.3.3.5 Learning 31
         1.3.3.6 Transformation 35
         1.3.3.7 Hybrids 39

2. THE DEVELOPMENT OF TWO INTELLIGENT SEARCH METHODS 41
   2.1 Introduction 41
   2.2 Learning and Backtracking A*(LBA*) 44
   2.3 Learning and Controlled Backtracking A* (LCBA*) 55

3. APPLICATIONS OF LCBA* TO PROJECT SCHEDULING PROBLEMS UNDER MULTIPLE RESOURCE CONSTRAINTS 75
   3.1 Introduction 75
   3.2 Historical Approaches 78
      3.2.1 Exact Methods 78
         3.2.1.1 Linear Programming 80
         3.2.1.2 Enumerative Techniques 82
      3.2.2 Heuristic Methods 106
3.3 The Application of LCBA* 115

4. THE APPLICATION OF LCBA* TO A BENCHMARK PROBLEM 129

5. CONCLUSION 138

REFERENCES 142

AMENDMENT 152
1. APPROACHES TO CLASSIFYING
GRAPH-SEARCH EFFICIENCY FACTORS

1.1. Introduction

Bellman (1978) showed that many examples of thinking by computers could be regarded as searches in the sense of tracing a path through a network. Generally speaking, thinking involves problem solving and problem solving is the process of transforming an initial situation into a desired one, the goal. This transformation process takes place in a problem space and continues with a search. Searching is an important part of problem solving which involves topics such as scheduling, categorising, retrieving, diagnosing and even explaining or justifying a result. Operations research (OR) and artificial intelligence (AI) are two highly computer dependent disciplines which are deeply concerned with problem solving, decision making and therefore with searching.

AI emphasises goal seeking whereas OR focuses on optimal methods. However, the complexity of many, if not most, real-world decision problems has exposed the limitation of these disciplines and causes them to seek solution approaches that integrate these tools (Brown and White 1990). Perhaps, Simon (1987) is the most significant authority on AI who calls for integrating problem solving techniques toward more powerful strategies of problem solving. The main purpose of this chapter is to provide a useful insight on the issue of how to make a search process efficient.

Since the efficiency of searching, in the first place, involves the issue of how to represent the problem, our first section is problem representation. In this section we consider the restrictions put on the methods of representing problems by the Von-Neuman architecture of today's computers. High speed computation with no power of direct connection between members of a very limited inactive memory in this structure forces users to use a representation quite different from what is used in the mind. In this section by defining data structure as a set of operations acting on a common variable, we relate the representation process entirely to control issues. State-space, problem-reduction and network-based representation as three important
representation techniques are investigated and the importance of appropriate representation in the efficiency of searching is emphasised.

Intelligent control methods is the title of the second section. In this section, systematic searching is first discussed. Systematic searching focuses on the methods for exploring a tree structure and can be considered apart from the control topic which focuses on how the problem solver chooses to shift attention among its subprocesses. Clearly, there is no solid boundary dividing searching from control. However, for the sake of clarity of discussion we distinguish between these two topics.

Next the complexity of search techniques is discussed. Search techniques for a category of problems usually suffer from combinatorial explosion and unfortunately many practical problems are grouped in this category. Fields of AI and OR have proposed various control means for resolving this combinatorial explosion and these are considered in a subsection titled control. As part of the control concept, some properties of the objects involved in a search problem representation are used to devise an algorithm that does not examine all possible solutions; and instead examines only the promising ones.

We consider the concept of control as a means of mixing some characteristics of the human mind with the speed of computers to tackle the problem of complexity in searching. Control stems from knowledge on the one hand and from cleverness on the other. At first, we present the complexity of the issue; then, we classify present control means in six different classes: Heuristic, Wise Pruning, Planning, Abstraction, Learning, Reduction and Hybrid.

A heuristic mechanism helps the most promising candidates to be selected. Associated with heuristic is a function called the evaluation function. The exact evaluation function can define a path to the goal without any search effort and this fact illustrates the key role of this function in the search efficiency. The addition of an evaluation function to a searching process incorporates the knowledge of a human expert into the solution of the problem. Our discussion of heuristic aims to consider
how to inject this knowledge into the evaluation function in order to give proper control of the search. We gather all control techniques which are concerned with pruning in a class called wise-pruning. We show what they have in common and how they can increase the search efficiency. The process of searching can be performed as a series of sub-processes so that in each of them an intermediate state is sought. This is the main theme of reduction. In investigating reduction methods we show how determining intermediate states can drastically reduce the search effort. Abstraction is the next class to be discussed. Any kind of hierarchical approach used in constraining and directing a search process is included in the abstraction class. We introduce abstraction as an important characteristic of humans in modelling problems and we show that the Von-Neuman structure of today's computers questions whether this subject should be considered as a part of problem representation or control.

The key to the solution of many unsolved problems in control is "learning" which is discussed in a subsection. Control methods can complement each other to make searching more efficient and this idea is discussed under the title of hybrid. The chapter concludes with remarks about the search efficiency.

1.2 Problem Representation

The first step in solving any problem is to represent it. This representation has a crucial role in the process of problem solving and may cause the problem to be solved easily or with a lot of difficulty. Nilsson (1980) states that search efficiency is not so much due to the methods of control as it is dependent on the way in which a problem is represented. As in the mind, a large vocabulary assists profound thinking; in the machine, a sophisticated scheme for representing a problem will lead to an efficient search. The issue of how to select a proper representation for a problem is an interdisciplinary subject that attracts and draws heavily on the academic areas of philosophy, mathematics, engineering, computer science and specifically artificial intelligence and operations research. Different academic areas based on their views of problem representation present their special definition of problem representation. However, even in AI which is concerned more with stating relationships between
symbols as a means for representation, nearly every particular practitioner has his or her own definition of problem representation. The problems which mathematics usually represents are in continuous space, whereas the problems which are usually represented by AI are in discrete space. Operations research is at the middle point between AI and mathematics; in the sense that problems represented in this field vary from being in discrete space to being in a continuous space. Mathematics, OR and AI have different criteria for good representation of problems. Criteria for good representation focus on various aspects of problems. From the point of view of understandability and easy coding, the AI perspective on problems has a remarkable advantage over other perspectives. On the other hand, from the point of view of efficiency and flexibility the OR perspective is unique. Mathematical representation converts the search process into a system of equations which are usually nonlinear. Since one of the functions of mathematics is to formulate precise questions, the mathematics perspective of problems from the point of view of abstraction and precision challenges all other approaches. What matters is that all these schools of thought regard searching efficiency as an important criterion.

To some extent, simplification and abstraction are used to represent any search problem. All representation systems use some symbols for representing thoughts and some system for representing relationships. These symbols and relationships represent a simplified and abstract model of the underlying thoughts. These differences between the abstract model and the underlying thoughts become very distinct when an attempt is made to represent a problem for a computer to solve. The natural representation of a search problem will need to be transformed subject to constraints in order to be in a form suitable for a computer to process. However, symbols and relationships between them still play key roles. Today the term "representation" is as vague as terms like "velocity" and "acceleration" were four hundred years ago. There is a chance that further consideration of the question of whether a computer can think will assist in clarifying this concept. However, as Bellman (1987) stated, the
definition of none of the terms used in the question of whether a computer can think is precise.

The Mind and the machine in spite of their own special capabilities have their own special restrictions; and we do not have any expectations that machines will do what minds can do. Merely, we are seeking active programming to make machines search more efficiently and thus more quickly. To do this, we should consider two different mechanisms which computers and minds apparently use in representing and solving search problems. We know that the human mind is the result of thousands of years of transformable experience and that problem representation and searching are not treated as two separate issues. For instance, it cannot be determined whether symmetry in many games or real problems is a part of problem representation or a search method. However, while considering the Von-Neuman structures of today's computers we make a sharp distinction between a central processing unit (CPU) and a main memory in which the problem will be represented. From memory, information either in the form of data or instruction is transformed to the CPU one element at a time; one of few operations possible by CPU is performed, and the result is stored back in the memory. This separation between CPU and memory, the small number and type of operations performed by the CPU at each cycle, and the sequential control structure are major limitations imposed on computers, when compared with minds. On the other hand, the CPU has a power which operates millions of times faster than the mind does (Kelin and Methile 1990). In brief, the search system of the mind has low speed computation with high power of direct connection between a large active memory. Whereas, the search system of a computer has high speed computation with no power of direct connection between members of a very limited inactive, but reliable memory. So, an important hurdle preventing us from representing the problem of searching in its natural form in the memory of the computer stems from two facts: first, differences between what passes in mind and its simplified, absolute representation with symbols and relationships; and second, communication between
humans and computers. The first hurdle is not addressed here and we focus on the second.

Based on the language selected for a program to be written in, two kinds of distinct representation can be used: declarative representation and procedural. In declarative representation, the sharp distinction between a program and data has been removed, but the issue of virtuality of this distinction still remains. Semi-natural symbols help this representation seem more understandable; and therefore from this point of view this representation has an advantage over procedural representation. The application of this representation is concentrated more on data base queries, and diagnostic and object recognition used in robot systems (Firebaugh and Morris 1990). In this representation, a description of facts and information about objects and their properties and their relationships are represented. The concepts of semantic networks, property lists (frames), production systems and hierarchical schemes make an important contribution (Linsay 1988).

The key concept of this representation is knowledge. Raw data when refined and processed yield information that is used in answering a user's queries. Upon further refinement and addition of control methods, this information can be turned into knowledge. In other words, knowledge can be presented as a meaningful structure of information. The appropriate representation of knowledge has lead to the success of a branch of AI called expert systems. Sometimes these systems are considered the most significant practical product of thirty years of AI research (Firebaugh and Morris 1988). These systems can advise, diagnose, analyse and categorise using a declarative representation of a problem (Kelin and Methlie 1990). Expert systems belong to a broader branch of problems known as knowledge-based systems; and "automated advisers", "computerised assistant" and "virtual consultant" are its other names.

Searching in this kind of representation is an internal function of a compiler and is used to relate various semi-natural symbols together in order to reach a desired goal. Through this kind of representation, the search problem is represented symbolically,
and the user does not have any explicit control on the search procedures. A built-in control system in the compiler controls the order in which production rules are considered; and its important role, as with any other control system, is resolving conflicts regarding the conduct of the search.

The reason for not having any explicit control on the search procedure in this kind of representation stems from the fact that control is something beyond numbers and strings and lies in active connections expressed through sophisticated data structures. So this concept is partly due to methods in which compilers translating this representation are based on and partly due to inseparable ingredients of a descriptive representation.

In spite of all the advantages we have stated for descriptive representation, it has one major disadvantage in the sense that control is partly a fixed function of a compiler and cannot explicitly vary according to the particular characteristics of a given problem. Meta knowledge is a remedy for this, however, it still is a remote goal (Ginsberg and Geddis 1991). This restriction has caused declarative representation to have a narrowly restricted problem domain and to become a tool for data bases and diagnostic systems. We believe that heavy dependency of AI on this kind of representation without special attention to meta knowledge can dilute the admirable goal of making machines intelligent into simply the development of data base and diagnostic systems. This could give AI a reverse direction when compared with other branches of science which develop from simplicity to complexity. It is worth mentioning that still there is no widely-accepted theory for declarative representation. Some systems work well for certain applications, but not others. That is the reason that many AI researchers feel uncomfortable in considering expert systems as a branch of AI (Firebaugh and Morris 1998).

Procedural representation, on the other hand, encompasses control strategies explicitly and is a reflection of the declarative. In this case, the program is considered as a sequence of instructions modifying the values of variables. Each program uses a set of operations acting on one or more common variables. Due to the fact that any set
of n common variables can be reduced to a single variable whose values are n-plets, we only need to consider the case of a single variable.

For the purpose of clarity, we define a data structure in this representation as a finite set of operations that act on a common variable. The general form of these operations is not necessarily a primitive of the programming language used; such operations may be given names, for instance, "select" or "expand". So an algorithm for a search problem is first expressed in terms of operations for an appropriate structure and then operations on this data structure themselves are programmed. The programming of these operations depends heavily on the representation of the common variable. As a sophisticated machine without a good operator is useless, an efficient procedure without appropriate representation is also useless.

Standard data structures related to searching can be divided into three categories: sequential type, set type and priority queue type.

In the first category, there are structures for which the relevant operations act on a finite sequence of objects. Stacks and queues are some instances of this type which are widely used with regard to depth-first and breadth-first searching respectively (Berlioux and Pierre 1990).

In set-type data structures, the operations act on a finite set of objects. Dictionaries are the main structures characterised by the operation of "belonging" in this type. There are many ways of programming dictionaries: arrays, hashed addresses, trees and many hybrid methods (Jun-ichi and Aove 1991). Trees are one of the common ways of representing search problems, starting at the top with the initial conditions and branching every time a decision is made.

Priority queue type as the third category is concerned with associating priority to objects. Many heuristic search methods use this structure. Relevant operations in this type act on a finite set of objects based on their priorities. Two major terms referring to the way in which this structure is programmed are "heap" and "list of objects with the same priorities" (Berlioux and Pierre 1990). Heaps are complete binary trees where the value of each node is not less than the value of its children. Therefore the object
with highest priority stands at the root of this tree and is easily accessible. The two operations of "remove" and "enter" are done with the complexity of $O(\log n)$ where $n$ is the number of objects in the heap. "List of objects with same priority", on the other hand, includes an array of pointers to the objects with the same priority. Working with this structure implies that all possible priorities are given in advance. For example, by using as a series of integers such as $1, 2, 3, 4, \ldots, m$, the priority queue can be programmed as $m$-linked stacks consisting of objects with the same priority with an array of pointers pointing to the first object on each of these linked stacks. "List of objects with same priority" is efficient if the number of priority levels ($m$) is much less than the number of objects ($n$) and the "heap" is proposed when the number of priority levels is of the same order as the number of objects (Berlioux and Pierre 1990). The most efficient implementation of a full complement of the priority queue operation that is currently known, the Fibonacci heap, was developed by Fredman and Tarjan (1987) and a full description of it can be found in Kingston (1990). This heap operates on the basis of a binomial queue considered by Vuillemin (1987) and has been designed for efficiently melding two priority queues into one. Like all powerful tools, this heap is based on very simple principles fundamental to binomial trees. A binomial tree, say $B_r$, is defined inductively by setting $B_0$ as a tree with just one node, and defining $B_r$ as linking to copies of $B_{r-1}$. The complexity of inserting a node into a Fibonacci heap is of $O(1)$ and of deleting the minimum node from it is of $O(\log n)$; whereas the complexities of both inserting and deleting in an ordinary heap are of $O(\log n)$.

Sequential, set and priority queue types can be combined to provide hybrid structures. For example, a set of sequences or a sequence of sets can be considered. It is worth mentioning that "list of objects with same priority" regards priority queue type as a sequence of sets.

By considering any search system as a problem of finding paths through graphs, all such problems can be represented by the data structure of a graph. This type of structure can only have one single operation (successor-generator) that associates
each node with a sequence of its successors. Representing a graph in a complete form in many searching problems will cause node explosion. Therefore implicit representation, in which a graph is represented through giving the procedure for the successor-generator operation, plays a key role in this kind of representation.

Another form of representation for searching problems is the re-writing system (Nilsson 1971). This system consists of rules of the form " A can be rewritten as B", where A and B are a set of characters selected from a given alphabet. Then the problem of searching can be stated as: given two set of characters x and y from a given alphabet, can y be rewritten as x by applying the given rules of rewriting? Declarative representation can be considered as a complicated form of this easy kind of procedural representation.

In representing search problems, it is worthwhile to distinguish between two groups, state-space representations and problem-reduction representations (Pearl 1984). In some search problems, the problem statement is seeking a sequence of operators, whereas in others an identification of a strategy is required. Finding a strategy is much more complicated than finding a sequence of operators because strategy is not just simply applying a sequence of operators but a prescription for choosing appropriate operators in response to any possible external event such as the result of a decision of another decision maker or the outcome of a test.

These differences can be better explained by taking two examples. Before stating these examples it is noted that using problems taken from real life as examples would take many pages of detailed description, therefore two puzzles were selected. The puzzles can be stated in a paragraph and still have the same complexity and richness as real problems.

The first example is the jug problem (Klein and Methlie 1990): there are two unmarked jugs, one of them with the capacity of three gallons and another with the capacity of four; the gaol is to take two gallons of a liquid using these jugs. The second example is the coin problem (Pearl 1984): there are twelve coins, one of
which is known to be lighter or heavier than the rest. The goal is to find the unusual coin in no more than three tests using a two-pan scale.

In the first problem only a sequence of operations is sought. For example, filling the first jug or pouring the contents of the first into the second. Hence, the solution to this problem is represented as a path in a graph. Each node of this graph represents a state of this problem, the contents of these jugs. Virtually, any kind of data structure can be used to describe states. These include lists, trees, vectors and two-dimensional arrays. In this particular simple problem, a two-element vector is used to represent the contents of the jugs. Arcs in the graph represent operators. Applying these operators changes one state into another state. Thus, they can be regarded as partial functions whose domain and range are a set of states. They are partial functions, because an operator may not be applicable in some states. So if the jugs are in the initial condition (initial state) and the goal of the problem is to reach a special condition (goal state), a sequence of operations (arcs) is sought that connect the initial state to the goal state. This kind of representation is usually called state-space representation.

This representation as whole can be considered as a data structure which has a common variable representing states and a set of operations affecting this variable and changing its value. The initial state and the properties of the goal state are regarded as input. As stated, lists, trees, vectors, arrays and other structures for representing this variable can be used.

In the second problem, on the other hand, a weighing strategy is required. The reason is that every test may result in one of three possible outcomes: balancing, tipping to right and tipping to left. This strategy should guarantee that whatever the results of the tests are, the unusual coin should be identified in, at most, three tests. The solution consists of a prescription describing what to weigh first, and for all possible results of this test what to weigh second, and for all possible outcomes of the second test what to do third.

This strategy is composed a piece at a time; Initially, a first weighing is chosen, then the selection of the second weighing is based on each outcome of the first and the
third weighing is based on various outcomes of the second. So, in contrast to the first problem in which the solution is represented by a path in a graph or a sequence of actions in real world, in this problem the solution is represented as a tree or a set of reactions in the real world. This kind of representation usually is called a problem-reduction representation.

It should be noticed that in a problem-reduction representation, in spite of treating subproblems as individual nodes of a graph, it is required that all these subproblems be solved before the parent problem is considered to be solved. A graph used for representing problem-reduction representation is called an AND/OR graph. In this graph, changes in the problem situation caused by external factors are represented by AND links and alternative ways of reacting to these changes are represented by the OR links (Pearl 1984).

This representation can be used even for solving non-strategic problems. In this case the general theme is to repeatedly establish sub-problems until eventually the problem is reduced to a set of trivial primitive sub-problems. In this case, maybe, there is no need for searching at all. For example the well-known problem of the Tower-of-Hanoi can be represented in this form: decomposing the movement of n disks from the first peg to the third, to three different subproblems where one of these is trivial and the two others can be further decomposed. Reduction operators applicable to a problem are not unique and often many of them can be found, each producing an alternative set of subproblems. Some of these subproblems are solvable and some are not. However, operators should be tried which can produce a set whose members are all solvable. Therefore, even in the case in which non-strategic problems are represented, the problem of searching may appear again.

Theorem proving can be a good example for representation in this form (Nilsson 1971). In summary, this representation method can be considered as a data structure which has a common variable representing problem description and a set of operators that transform the problem description into a set of subproblem-descriptions which are usually simpler to solve.
It can be claimed that state-space representation employs a trivial kind of decomposition that decomposes the original problem into a similar problem with different initial state. However, in problem-reduction representation, the transformation is such that the solutions to all reduced problems are somewhat easier than the solution of their parent problem (Kelin and Methlie 1990). In other words, this representation is guided by some type of planning mechanism to repeatedly reduce a state-space search problem into easier search problems until trivial problems are produced.

Problem-reduction representation is more suited to two types of problems: first, a problem in which the final solution has to be represented as a tree; and second, a problem, without any "OR" link, where expertise can be used to repeatedly reduce it to simpler problems until trivial subproblems are produced. The hybrid cases using these categories are problems where expertise cannot be used to reduce them to a set of simpler problems, but it is known that if this problem is supposed to be reduced into a set of simpler problems what alternatives exist. Logical reasoning, theorem proving, and symbolic integration belong to this category. On the other hand, state-space representation is more suited to problems for which no reduced form can be found except transforming them from one state to another state.

Since a search problem involves the task of finding or constructing an object with given characteristics, by selecting either a state-space or problem-reduction representation, it is necessary to define a symbol structure as well as a set of operators acting on this structure. The selection of this symbol structure along with a set of operators acting on it, has a significant effect on the efficiency of the search effort.

To represent each object in a searching problem, a symbol-structure is used which can be utilised to reject its corresponding object but not a large group of objects. A symbol-structure can become more powerful by being able to be used for a large set of objects and this can provide flexibility in problem representation. This kind of representation can provide an early detection of some subsets that contain no solutions and thus reduces the size of the candidate solution space. Subset
representation instead of object representation implies more sophisticated operators. These operators should act on a subset-symbol and produce another one.

To appreciate the subset-symbol structure, let us consider an encyclopedia without any order of entries and see how ordering entries can improve the search efficiency. This example is an instance in key-search problems (Juni-ichi 1991). Objects in this encyclopedia while being ordered alphabetically show some kind of subset-symbol structure, enabling one to eliminate a chunk of entries while looking for a particular one. So in this case what happens in a key-search can be used in a graph-search. Pearl (1984) claims that most OR practitioners consider the process of searching as a huge subset of potential solutions that have to be reduced finally to a single element.

Regarding searching as an intelligent constructive process is something that AI researchers rely on (Firebaugh and Morris 1990). Even when the process of searching is regarded as the intelligent construction of a solution to a problem, each time a new part of a given symbol-structure is completed, the subset of solutions represented by this structure is narrowed. Therefore, the process of narrowing down the solutions can be facilitated by representing the subset of solutions as a structure.

Besides state-space and problem-reduction, there are many other trivial representations and the most important of them is the network-based representation. In many search problems, the path which is being sought should satisfy some given conditions and these conditions can be represented in a variety of ways. Network-based is one of the important approaches to this problem (Detcher and Pearl 1988). It is known that many search problems can be regarded as the assignment of values to variables subject to a set of constraints. Truth maintenance systems, map colouring and electronic circuit analysis are some instances of this group. To present an example of this kind of representation, we consider the "constraint satisfaction problem" (CSP). This problem involves a set of n variables, \( X_1, X_2, \ldots, X_n \), which have domains \( D_1, D_2, \ldots, D_n \). An n-ary relation on these variables is a subset of the cartesian product of these domains. Changing this n-ary relation into a set of binary constraints between each pair of variables, which is a subset of their constraint
product and represents these relations by means of a network of binary constraints, is the central theme of this representation. This network can be represented by a constraint graph in which variables are represented by nodes and constraints are represented by arcs, with each constraint specifying a set of permitted pairs of values so that variables participating in the same constraint will be mutually adjacent (Freuder 1985). This representation can be subjected to many refinement processes, with each process improving the representation. Improved representation has been shown to reduce the search effort for finding solutions (Laveen and Kumar 1988).

Finally, we emphasise the importance of appropriate representation in the reduction of search effort, noting that representation is partly dictated by a problem specification and is partly a matter of choice. Regardless of the type of representation used for a problem, there is room for changing the ways objects are represented. Therefore the first issue arising is what type of representation to use and the second issue is how to present a problem appropriately using that kind of representation. In any case, the idea of using variables in the state description and thus using a structure expressing a subset of states, instead of just one state, can simplify significantly a search technique.

The definition of data structure as a set of operations acting on a variable emphasises the need not to see representation as separate from control issues and highlights the importance of problem representation in control processes.

1.3 Intelligent Control Methods

AI and OR have been very concerned with searching and the means of controlling it. Searching focuses on methods for exploring a tree structure whereas the control issue focuses on how a problem solver chooses to shift attention among subprocesses. The boundary dividing control from search is not solid, but the distinction is useful for discussion purposes.

One might expect that substantial advances in control methods have been made over the years. However, considering the fact that for a well-known game like chess the brute-force tree search in which no control mechanism is used is still the dominant
method shows that as yet, after so much experimentation, control mechanisms are not fully understood (Brook 1991). For a moment suppose that a brute-force search which is used usually by computer programs to play chess is equipped with all the control methods used by a champion. Considering the fact that the brain's "clock cycles" are typically milliseconds long compared with the microsecond to picosecond cycle of computers, the time spent making a decision for every move would decrease significantly. However, it still seems to be a remote goal to combine the characteristics of search in the human mind with the speed of computers in order to tackle this problem. There is no doubt that the huge differences between today's computer architecture and the human mind plays a key role in preventing the transformation of appropriate control means to brute-force search methods.

1.3.1 Systematic Search

Searching focuses on methods for exploring a tree structure. These methods should be able to traverse a tree systematically, showing which part of the tree has already been explored and managing to explore the rest of it. For the sake of simplicity, we limit our description to searching in trees; recognising that some modification is required to make tree algorithms work for graphs. At the beginning of a tree, there is a single node called the "root" which describes the initial state or the original problem description in state-space or problem-reduction representations respectively. By applying any operator on any existing node, the search tree grows.

Breadth-first and depth-first are two extreme points in the continuum of search methods. In depth-first, the most recently generated nodes are expanded and in breadth-first the nodes are expanded in the order they are generated.

In the depth-first method, expansion priority is given to the nodes at the deeper levels of the tree. In other words, each node selected for expansion has all its successors generated, before another node is expanded. When a node is expanded, one of its recently generated children is selected for expansion and this process continues until, for some reason, the process is blocked. In this case, the process again resumes from the deepest level of nodes expanded.
Based on the consideration of computer memory, a version of this method called "backtracking" is popular among programmers. In this version, when a node is selected for expansion, only one of its successors is submitted for expansion. This process continues until progress is blocked. In this case the program backtracks to the previous node and continues the search. Storage economy is the main advantage of this version of a depth-first search.

In breadth-first search, on the other hand, expansion priority is given to the nodes at the shallowest levels of the tree. It first explores all nodes in level one, then goes to level two and so on. Unlike depth-first search, breadth-first search is guaranteed to find the shallowest possible solution. Only by storing all expanded nodes can this method traverse the tree, and thus no version like backtracking can be derived from it. Consequently, this method is associated with large storage requirements.

From a computation point of view, the difference between depth-first and breadth-first methods is in the data structure used. In depth-first this structure is a stack and in breadth-first this structure is a queue. Some situations are suitable for breadth-first and others for depth-first: when a solution exists in a shallowest level surrounded by paths that may go to a much deeper depth, the application of breadth-first can have many advantageous over that of depth-first. On the other hand, breadth-first expands every node up to the depth of the shallowest goal and this may be unnecessary on many occasions. Regardless of the storage requirement, the trade-offs between these two extreme methods are exploring every node up to the depth of shallowest goal versus getting trapped in one of the long paths surrounding the goal.

These two methods can be applied to the problems represented in either a state-space or problem-reduction representation; although for problem-reduction representations some small adjustment is needed. This adjustment is mainly in the procedure employed to determine the termination condition. The termination condition in state-space representation involves the property of a single node, whereas in problem-reduction representation it involves the property of successors. For this purpose, the procedure in the problem-reduction representation will label nodes "solved" or
"unsolvable". In other words, finding a solution-tree in a problem-reduction representation is associated with generating a sufficient part of an AND/OR graph to demonstrate that the start node is "solved". Search in this kind of representation terminates successfully as soon as the start node can be labelled "solved" and it terminates unsuccessfully as soon as the start node can be labelled "unsolvable".

Minimax and Alpha-beta procedures are common routines used in determining "solved" and "unsolvable" nodes. Minimax, completely, separates the process of generating a search tree from the process of labelling, whereas the Alpha-Beta procedure does these tasks simultaneously. Since labelling needs at least some part of the tree to be generated to maximum depth, some kind of depth-first search is usually employed when using the Alpha-Beta procedure. This procedure has an advantage over the minimax procedure because, by labelling as unsolvable some nodes in the course of the process, it will not generate their successor nodes that always are generated by the minimax procedure.

It should be noted that either breadth-first or depth-first techniques, whether they are used in state-space or problem-reduction representation systems, cannot solve the problem within a reasonable time except for very small and shallow trees. Control methods are employed to tackle this problem. However, in order to investigate remedies for the problem one needs to understand how these difficulties arise. We investigate this issue in the following sub-section.

1.3.2 The Complexity Of Search Techniques

Combinatorial explosion associated with search techniques has been identified as a chief obstacle to effective problem solving. Application of search techniques to hundreds of important problems has lead to the notion of combinatorial node explosion (Basse 1988). The size of a tree is an exponential function of its depth, thus the search-tree can grow catastrophically.

Even for a tree of depth of 6, should twenty operators be applicable to any node, as many as sixty-four million \((20^6)\) leaves are produced. This number is equal to the number of nodes which should have static values in a chess game with only six half-
moves or three complete moves. This is worse than looking for a single needle in a haystack; since in looking for a needle one at least knows what to look for. The reason is that the static evaluations allocated to each node such as 2 for knight, 3 for rook and so on are arbitrary and depend on the opinion of the human expert and may well be far from the actual values.

In spite of the emergence of high-performance computers, available resources including memory and CPU in relation to providing solutions to the node explosion problem are less significant than they appear at first glance because when the depth of variation is increased by just a few levels, these resources need to be increased thousands of times, something which is impossible in practice. For instance, adding 2 moves or four half-moves to a chess problem will multiply the number of nodes by one hundred and sixty thousand \((20^4)\). Therefore, a modest increase in computer power is of little significance in coping with the problem of node explosion.

NP (Non-deterministic Polynomial) is a notation used to describe a class of problems for which there exists no known search technique whose worst-case time complexity is bounded by a polynomial. The NP-complete notation represents a subset of the hardest of NP problems such that the solution of one of these hard problems in polynomial bounded time complexity proves that all NP problems have a polynomial bounded solution.

Remarkably, this class contains hundreds of important combinatorial problems from logic, language theory, data base, automata theory, graph theory and scheduling (Goldberg and Pohl 1984). For these problems, even the best known algorithms would require many years or centuries of computer time for moderately large inputs. However, there is not one single problem in NP for which it has been proved that no polynomial bounded search technique exists. This itself gives hope that advanced control can provide some headway in tackling the problem of search complexity. As well as this, NP-complexity is usually a worst-case analysis for the whole of the problem and does not cover all typical instances which, on the contrary, are easy to solve.
In order to find the source of the hard problems, Cheesman and Kanefski (1991) show that for many NP problems one or more "order parameters" can be defined with hard instances occurring around special critical values. These critical values form a boundary separating the space of the problem into two different regions. In one of these regions, the density of solutions is high, making it relatively easy to find a solution. In the second region the density of solutions is low, also making it relatively easy to find a solution. For, in the latter case the low density of solutions creates a strong attraction region; and thus any reasonable algorithm is likely to find the solution easily. Moreover, in the case of there not being any solution, backtrack searching cuts off the potential solution path early.

Really hard problems occur between these two regions. The reason is that on the one hand, since there are not a lot of solutions the chance of finding one is significantly increased, but on the other hand, the number of solutions is not so rare that it initiates a strong attraction to the correct one. This problem is discussed through phase-transition and is of significant importance (Huberman and Hogg 1987). It helps in understanding the complexity of the system at a macroscopic level in terms of a few order parameters describing the treatment of the system as a whole.

For instance, a Hamilton path (HC) in a graph is a path (cycle) that passes through every node exactly once. The simple question of whether a given graph has a Hamilton path is a NP-Complete problem. In a fully connected graph, all node-orderings are a Hamilton circuit; and thus finding a HC is as simple as writing an arbitrary node ordering. For an unconnected graph there is no HC and if the average connectivity is low, it can be determined early that no HC in that particular graph exists. For some critical degree of connectivity between these two extremes, the probability of having a HC changes steeply from almost 0 to 1. Theory predicts that the transition will occur at average connectivity of $\ln(n) + \ln(\ln(n))$, where $n$ is the number of nodes of the graph, and as Cheeseman and Kanefski (1991) show empirical results confirm this fact.
Therefore, we investigate how the computational cost of finding a HC (if one exists) varies with connectivity, the answer is that phase transition in cost occurs at the same point at which the probability of having an HC drops to zero within a given numeric significance. In other words, the critical connectivity separates two different regions: a region where there is nearly zero probability of there being a HC and a region with a high density of HCs. In both these regions the backtrack algorithm works quickly and it is between these regions that hard problems occur.

Finding the minimum weighted Hamilton circuit in OR and AI is called the travelling salesman problem (TSP). Even for this optimisation problem some order parameters can be found. Cheesman and Kanefski (1991) show that standard division of the cost matrix is one of these order parameters.

Therefore, most of the search problems, in spite of their NP completeness, are not so difficult to solve and can be tackled with appropriate control means. In the following, these control means are discussed briefly.

1.3.3 Control

Search techniques usually suffer from combinatorial node explosion; and simple searching fails when faced with the tremendous search space requirement usually encountered in typical applications. Since the emergence of AI and OR, early researchers were concerned that search procedures must be equipped with some kind of control mechanism to cope with the combinatorial nature of these problems. While OR developed "dynamic programming" and "branch and bound" to solve classical optimisation problems like TSP, AI used similar methods to handle puzzles and theorem proving problems (Goldberg and Pohl 1984). If we regard searching as tracing a path through a graph then absolute control can define this path without any search effort. Therefore, a situation involving a huge search effort can be regarded as the result of a lack of appropriate control. Control stems from knowledge about the problem and the application of cleverness to this knowledge. For instance, for chess champions it is quite impossible to keep track of all possible moves. They, instead, apply their own experience to evaluate a few more significant moves.
However, the problem of getting this type of knowledge into machines in a representation which can be recognised by today's computers has been described as the "Knowledge Bottle-neck" and its solution is still a remote goal for researchers. Intelligent control techniques applied to search are highly oversimplified models of this very complex issue.

Up to now, control methods have been incorporated in very different ways in search procedures. We classify these methods into seven different classes: Heuristic, Wise-pruning, Reduction, Abstraction, Learning, Transformation, and Hybrids.

1.3.3.1 Heuristic

The concept of heuristic search as an aid to problem solving was first introduced by George Polya (1945). From then on various definitions have been offered by many AI and OR researchers with some of these definitions covering all the control methods we can employ in a search problem (Firebaugh and Morris 1988). Some computer scientists even argue that heuristic programming better describes the field of artificial intelligence, pointing out that it is a major problem solving technique used in AI and hence a key characteristic of AI programming. By accepting this definition that heuristic search constitutes rules of good practice, rules of expertise and rules of intuition, we simply broaden its range to all control methods. Here we adopt the definition presented by Pearl (1984) as follow:

"It is the nature of good heuristics both that they provide a simple means of indicating which among several courses of action is to be preferred and that they are not necessarily guaranteed to identify the most efficient course of action, but do so efficiently often."

Associated with the heuristic is a function called the evaluation function which incorporates the knowledge of human experts into a problem. These functions are based on a variety of ideas understood by experts. In this regard, the simplified distance between an arbitrary node and the goal is an idea that is predominantly used by researchers. Another idea which can incorporate expertise is the probability that a node is on the best path; unfortunately this idea has received limited mention (Pearl
Many other different ideas can also be used to develop these functions. For instance, a chess champion can evaluate two different configurations on a chess board with two different scores despite not being able to describe why these scores have been selected. As Pearl (1984) states it is so hard to articulate the mechanism to help discover these functions and little attempt has been made toward this goal. The thinking process that is used in suggesting such functions is very similar to that of the symbolic proof in mathematics, where once applied it will leave no memory trace of the intermediate steps. Generally, considering what is termed the "relaxed problem" has been used as a source of devising such heuristic functions. Here "relaxed" means that some constraints are removed from the initial problem, leaving a problem which is usually easier to solve than solving the original, but, with answers somewhat similar to that of the original problem.

One important point about these relaxed problems is that not every relaxed problem is simpler than the original one and in some cases the complexity of searching for the solution is higher than that associated with the original. The reason is simply that removing constraints makes the search graph shallower but increases the number of available operators applicable to states. Hence, there is no guarantee that using a relaxed problem will reduce the search effort. This phenomena can lead to using additional constraints as another source of creating heuristic functions. These constraints should be able to drastically reduce the average number of available operators for each state without making the search graph too deep.

In general, modifying the search graph in a such way that solutions change slightly but search effort is reduced drastically is the usual way of devising heuristic functions. Therefore relaxed or over-constrained problems must be both simple and close to the original.

The most common heuristic algorithm is the A*-algorithm and much research regarding the effectiveness of heuristic functions has been done on that algorithm. In the most recent model called IIS introduced by Gheoweth and Davis (1991), heuristics are modelled as random variables and the model is used to determine what
properties the heuristic must have in order for A* to have an average polynomial versus exponential time complexity. The graph used in their research is a uniform b-arry tree, with bi-directional arcs each of unit cost and a single goal, N units away from the initial state. Their studies assume that heuristic errors are independent and identically distributed random variables. By using empirical data, it is claimed, contrary to common belief and previous experiments, that accuracy is not critical, and that the key issue is whether or not heuristic values are concentrated near a rapidly growing "critical function". This kind of work can be of importance in regard to thoroughly understanding heuristic guidance mathematically which should be a challenging objective for OR and AI researchers.

1.3.3.2 Wise Pruning

Even by using the best heuristic functions, search trees expand very fast, as fast as millions of nodes per second; and, what is worse, every node potentially can generate other nodes. Many methods have emerged to implicitly eliminate large groups of potential solutions to a problem without explicitly evaluating them. This elimination can be done either temporarily or permanently and either with or without damaging the optimal solution. We place all these elimination methods in the "wise pruning" class. At the two extremes of this class are "branch and bound" and "hill-climbing" methods.

The "branch and bound" method is the most important of the wise pruning approaches and it is used extensively in OR. The attractiveness of this method stems from the way in which it is able to reduce the size of the solution set permanently without damaging the optimal solution. However like many other control methods it is a general algorithm that should be merged with the structure of the specific problem at hand to form an implementable algorithm. From the implementation point of view, the alfa-beta cut-off method in AND/OR graphs can be considered as one of these methods. Suppose a branch and bound method is ninety-nine point nine percent efficient, in the sense that it can eliminate that percentage of leaf nodes from an exponential tree. Still, point one percent of an exponential number is, again, an
exponential number. Therefore, the use of this method is limited to moderate and even simple problems. For instance, a binary-integer program with a few hundred or so variables is the most difficult problem that can be dealt with by this method whereas, in practice, there are binary integer programs with many more than thousands of variables.

At the other extreme, is the hill-climbing method which is used extensively in AI. It is the simplest control method and prunes all branches except the most promising ones; and hence it does not guarantee any optimal solution. It is so named because it is like a climber wishing to reach a mountain peak, selecting the direction of the steepest ascent regarding his current position and ignoring all others. In exchange for losing the chance of finding any proper path, this method requires no computational effort. It expands a node, inspects its recently generated successors and chooses the best successor to expand next, retaining no further reference to the node. An uninformed evaluation function in this method can lure the search into deep or even infinite node explosion. On the other hand, a well informed evaluation function can decrease the chance of losing the solution. It should be noted that as soon as a local optimum is found (a node more valuable than all its successors) this method stops without reaching any further solution and the only way to cope with this problem is to begin from an arbitrary fresh node.

This method in spite of its simplicity and usefulness has many shortcomings. However, if used with a well informed evaluation function it can perform brilliantly. MACSYMA, a mathematical expert system with the capability of performing over 600 distinct mathematical operations and over half a million lines of LISP that represents over 100 person-years effort of programming, uses the hill-climbing method as a control technique for searching (Firebaugh and Morris 1988).

Branch and bound and hill-climbing methods lead to the cut off of some branches permanently, whereas there are methods in which a number of branches can be ignored temporarily. The only reason for doing so is memory saving and in exchange for this saving some extra execution time is required. The backtracking method is the
simplest method in this class. From a memory efficiency point of view, the backtrack method provides an improvement over the simplified depth-first search. However, despite the gain in memory efficiency this method causes a great deal of redundant and unnecessary effort. One important point about backtracking is that if it is used to search for the values of a set of variables which are subject to a set of constraints, the order in which variables are chosen (variable order) plays as an important role as the "horizontal order" which is determined by the evaluation function (Freuder 1985).

Iterative deepening (ID), devised by Korf (1985), is another method which in included in this class. This method was also designed as a remedy for memory exhaustion. It is based on a depth-first search which only maintains the current path from the root to the current node, and hence uses space that is linear with search depth. ID performs a series of depth-first searches and branches are pruned as soon as the cost of a node on them exceeds a certain amount called the "threshold" and this amount changes for each iteration. At the first iteration, it is initialised to an arbitrary amount that does not over-estimate the minimal solution and for each succeeding iteration it is the minimal cost value that exceeds the threshold of the previous iteration. Since the cost used at each iteration is a lower bound on actual cost, the first solution for expansion is an optimal one. Special cases of iterative deepening include depth-first-iterative deepening (DFID), and iterative-deepening-A* (IDA*).

ID methods and branch and bound are quite contrary to each other. In branch and bound, an upper bound is fixed on actual cost whereas in the ID method a lower bound is considered. The major drawback of the ID method is that all nodes expanded in one iteration should also be expanded in all subsequent iterations and that is a price of temporarily pruning off branches. ID expands more nodes than a method known as "best first search" (BFS) which includes breadth-first and A* as general algorithms to expand nodes based on some particular priorities. These differences have been examined by Vampy, Kumar, and Korf (1991).
1.3.3.3 Reduction

The process of searching in some problems can be performed as a series of sub-processes, each seeking a separate goal which when recombined attain the ultimate goal. A problem whose complexity is $a^n$, $n>1$ (exponential) if separated into two similarly sized sub-problems whose solution when recombined solves the original problem, yields the complexity of $2a^{n/2}$.

Within this class there are some important ideas of which multi-directional search is the most important. Let us consider a simple example, the bi-directional search. Common sense tells us that working backward and forward can often be useful. In all instances both an explicit initial state and a goal state are known and what is desired is a chain of operators connecting these two states. For instance, in theorem proving one starts with some clauses and obtains the null clause. $A^*$ is an example of a unidirectional search proceeding from initial state to goal.

Assuming that operators are reversible, by simultaneously exploring the state space both from the initial state and the goal state we have a bi-directional search. In the worst-case the bi-directional search is twice as expensive as a unidirectional search, but as Goldberg and Pohl (1984) state when it works there is an exponential reduction of effort. This approach can also be expanded to multi-directional searching by determining some intermediate nodes along the optimal path and reducing the problem to sub-problems, each much simpler than the original.

The reduction technique can be applied recursively to the generated sub-problems. The effectiveness of this technique depends on dividing the problem into equally sized sub-problems and combining the solutions to the sub-problems into a solution to the original problem. The Davis-Putnam procedure is one of the most efficient algorithms representing this method. Before describing this procedure let us look at the "conjunctive normal form" (CNF) formula as a seemingly simple problem in logic which is NP-complete and to which this procedure can be applied. Logical variables are blocks of this problem and may be assigned values of either true or false. Should $x$ be a logical variable then $\neg x$ is the negation of $x$. A literal is either a logical variable
or the negation of a logical variable. A clause is a sequence of literals separated by the boolean operator "or" (\(\lor\)). A logical expression in CNF is a sequence of clauses separated by the boolean operator of "and" (\(\land\)). For instance \((x \lor y) \land (y \lor \neg z)\) is an example of logical expression in CNF. Assigning the values of true or false for variables in the expression so that the expression will have a value of true is a CNF-satisfactory problem.

This problem, in spite of its easy appearance, is an NP-complete problem. The root of the difficulty lies in the necessity to check all possible assignments of true or false to the variables. This means checking \(2^n\) assignments in the case of \(n\) variables. By the Davis-Putnam procedure, in a CNF formula, a variable is chosen from which two sub-problems are derived, based on the assignment of true or false to that variable. The sub-problems are such that the original formula is satisfied if and only if either one of the two sub-problems is satisfied. In the worst-case, the sub-problems may each have only one less clause than the original formula, a situation that results in exponential complexity. However, like quicksort, on average the division is very effective, leading to a polynomial average time bound for the procedure when a uniform distribution of the problem parameters is assumed. The reason why the average-case like this is not widely used is that in the average-case analysis, some assumptions regarding probability distribution of the problem parameters is needed and this may be more difficult than the worst-case analysis.

Solving a high-level, complex problem by breaking it down to sub-problems is natural to humans and is very powerful. Using recursion, one solves a large problem by breaking it down to smaller instances of the same problem and combining the solutions to solve the original solution. However, this approach is not without its shortcomings. The Fibonacci number construction problem is a prime example. Whereas a simple iterative procedure can solve this problem in a few iterations of a simple loop, on the contrary, breaking it down into simple problems will increase the solution time exponentially. This is because it repeats a lot of work which would not have been necessary if the results of the computation had been stored. The dynamic
programming algorithm is a remedy for this which, instead of recomputing the
solution of sub-problems, stores the results and looks them up when it needs the
results. This method is one of the most efficient ways to solve optimisation problems
in OR. Dynamic programming is well suited to some recursive algorithms which
solve many sub-problems redundantly.
Like branch and bound and many other control methods this method also is a general
algorithm that can be merged with the structure of the specific problem at hand to
form an implementable algorithm. The way in which this method merges with the
structure of a problem plays a key role in the efficiency of a search process. For
example, in the shortest path problem, one can use many instances of this method,
where some of these instances are much faster than others.
One might think that since dynamic programming uses an n*c table where n is
number of objects and c is the number of stages and only uses a few operation to
compute each table entry, its running time is always of O(nc) which is not
exponential. As Basse (1988) states this belief is wrong because c is exponentially
larger than datum c in the input which is represented by the ln(c) bits. However in
many cases dynamic programming drastically reduces the complexity of the search
problem.
Briefly, the control methods which reduce problems to simpler problems are
classified in "reduction" class, particularly dynamic programming which by storing
solutions to sub-problems rather than re-computing them trades space for speed.
1.3.3.4 Abstraction
An hierarchical approach is strategy human beings use often to solve problems. This
strategy can also be used as a control method for constructing and directing the
searching process where states and operators are divided into a series of hierarchical
categories, beginning with the most important and ending with the most trivial. The
Von-Neuman structure of today's computers poses the problem of whether to consider
this issue as part of problem representation or control methods since introducing
states in a hierarchical order can be as much related to the representation of a problem
as it is related to the control of searching among states. The main concept of abstraction is to ignore the lower levels of detail until the higher level details have been solved. By using this method, therefore, concentration is focused on states based on their hierarchical priorities. Abstraction like reduction reduces the search effort by breaking the search space into various sub-spaces thereby reducing the complexity of the problem to the sum of the complexities of sub-spaces whereas previously it had been a product of those complexities.

Consider the following example of abstraction that has been presented by Korf (1987) and has two levels of hierarchies. Suppose there is a network of roads and intersections in a transportation system which can be grouped into two categories: major and minor. Each major intersection is connected to other major and minor intersections by major roads and can be considered as a centre for some of the minor intersections. Notice that it is not necessary to have a major road between every pair of major intersections but simply to have the set of major intersections connected by a set of major roads. Now consider the problem of finding a short path between two of these intersections. This problem can be regarded as one of connecting both of these two intersections to their related major centres and then connecting these two major intersections to each other. This instance shows a two level hierarchical problem where intersections were divided into two categories, and each major intersection has a set of minor intersections, as its neighbours. This arrangement of minor and major intersections, therefore, can drastically reduce the effort of searching. In this simple problem, the hierarchy of intersections could be based on various levels of majority, each level having some of the lower levels in its region. This idea can readily be generalised to multiple hierarchical levels of abstraction.

Ignoring low level details and concentrating on the essential features of the problem will lead to filling in details based on their priorities. This is analogous to the way in which an artist draws a picture, starting with a major framework of the picture and ending by filling in minor details. The importance of abstraction was first mentioned by George Polya (1957) and has been of significant importance for AI researchers.
The first explicit use of it was the planning version of GPS (Newell and Simon 1972). Sacerdotis (1974) has developed ABSTRIPS which is another important approach to abstraction and Korf (1989) presents a quality analysis of abstraction in problem solving. The question Korf addresses is: "How much search efficiency is gained by the use of abstraction and what is the optimal level of detail for each level of abstraction". A model of abstraction is first formalised and a special case of single level abstraction is considered; then attention is turned to the general case of multiple level abstraction. All analysis Korf has done is based on the average case. The main result is that an abstraction hierarchy can reduce the amount of search time for a problem of size n from linear, \( O(n) \), to logarithmic, \( O(\ln(n)) \). He reports that for a single level of abstraction the optimal size for abstract space is the square of its base space, \( n \), and that this reduces search time from linear in size of space, \( O(n) \), to the square root of the space, \( O(\sqrt{n}) \). In the case of a multiple hierarchical level of abstraction, he reports that the optimal hierarchy has a logarithmic number of levels with a constant ratio between their size. He also claims that since the number of states, \( n \), is often exponential to the size of the problem, then abstraction hierarchies can reduce this exponential complexity to linear complexity.

Abstraction is a common method which humans use to solve problems and can be used as a control method to guide the search process very efficiently noting that it is highly dependent on the representation of the problem.

1.3.3.5 Learning

If a particular search method can improve its efficiency in successive runs, it is said that the method has a "learning" capability; and if this learning improves search efficiency we regard it as a control means. Since the very early days of computer science, learning has been regarded as an important feature of computers and today it is considered a key to the solution of many unsolved problems. Most human experience can be described as learning and this is why machine learning is a vague topic and difficult to define. Learning is a very complicated phenomena and includes many different aspects. Although, there are as many definitions of learning as there
are AI researchers, one of the most comprehensive is given by Simon (1983) as follows:

"Learning denotes changes in a system that are adaptive in the sense that they enable the system to do the same task or tasks from the same population more efficiently and more effectively the next time."

Whereas some AI researchers such as Botuniic (1984) emphasise that machines should learn as a human does, others such as Garbonell, Michalaski and Mitchel (1986) claim that there is no reason to believe that human learning methods are the only possible means of acquiring knowledge and skills. In any case, there are some special limitations in machine learning and other limitations in human learning. Human learning is comprehensive, expensive, slow and cannot be copied; on the other hand, learning by machines is partial, cheap, fast and can be copied.

Generally, learning involves a collection of problem instances with some common structure. This collection can consist of just one instance that is to be solved repeatedly or more instances that somehow are related to each other, and information about one can help to solve others more efficiently. We say that learning is a control method if the fixed cost of learning plus the marginal cost of solving problems using what has been learned is less than the original cost. Therefore, we view learning as a control method that results from a cost-effectiveness trade-off in storing information about the search process. This cost-effectiveness restriction is necessary; since through a huge amount of memorising at a very high cost, one may accidentally find a valuable piece of information and claim that a learning method has been used.

In real life, the process of searching in the human mind is based on very little precise information because one can remember the cases and act on memory. In other words, when prior experience is indexed cleverly, no search effort is needed. Even some researchers in AI such as Homnad and Kristian (1989) have claimed that real thinking has nothing to do with logic at all. They claim that real thinking means retrieval of correct information at the right time. However, retrieval requires a systematically organised previous experience and this is a costly process. With this
view the most difficult part of searching is the creation of labels that will allow one to retrieve a path which has already been encountered and memorised systematically. This control method is supported either by the memorised paths which are capable of conducting a search effortlessly or changing dynamic memory. In this class we briefly discuss four basic approaches to this control method: Learning Heuristic (Ernst and Banerji 1977), Learning Macro (Korf 1985), Learning Admissible Heuristic (Davis and Gregor 1991), and Learning Real Time Heuristic (Korf 1990).

Learning Heuristic:
Learning Heuristic is an approach first used by Simon and Shaw but formalised by Earnest and Banarji (1979). The model we will discuss has been simplified by Beas (1988). The essence of this approach is the narrowing of the state space in order to reach the goal more efficiently. It is assumed that nodes are represented by a set G and this set is the intersection of a number of sets:

\[ G = G(1) \cap G(2) \cap \ldots \cap G(n) \]

These sets, G(i), are to be chosen in such a way that a large number of operators relevant to G(i) are irrelevant to other sets in the sense that they do not effect the membership of their states. It is intended to reduce the search effort by putting the states into G(1), G(2), \ldots in sequence. To this end, the set of operators, O, are partitioned as following:

\[ O = O(1) \cup O(2) \cup \ldots \cup O(n) \]

with the property that every operator in O(i) is relevant only to G(i). This simple mechanism creates a powerful control over the search method and makes it very efficient. It is noted that in some cases learning how to decompose the sets "O" and "G" may be as difficult as the original search itself.

Learning Macro:
Learning Macro is an approach used by Korf (1985). It is very powerful and similar to methods which one uses in every day life. For example, in finding a route from an unfamiliar place toward one's home, as soon as one reaches a familiar intersection, the remaining problem is just applying previous information and no search from this
point on is needed. Korf (1985) defines the term macro as "a sequence of primitive operators being able to change a state of a problem toward the goal". If in a search process one reaches a state which has associated with it special macro-operations then it is possible to immediately improve the situation toward the goal. Therefore, progress toward the goal is made by getting to a state in which the problem-solver knows the associated macro operators. Learning Macro is applied to problems where from various initial states a particular goal is sought; for instance, in a problem like Rubik's Cube in which one may reach the final arrangement from various initial positions. Macro-operators can temporarily violate partial solutions which have been found in the construction of the final solution. In other words, during the application of a macro, the previous partial solution may be disturbed; however by the end of the macro application this previous partial solution will be reconstructed.

The power of this approach can be demonstrated by the fact that all 181440 solvable initial states of the eight puzzle can be solved without any search by using only 35 macros as shown by Korf (1985). Moreover for the $4 \times 10^{19}$ initial states of the $3 \times 3 \times 3$ Rubic Cube. Korf shows that only 238 macroes are required. The disadvantage of this approach is that it does not address how the macro operators are learned in the first place, and learning these macros is still regarded as an art.

Learning Admissible Heuristic:

This method outlined by Davis and Bramanty-George (1991) uses an $A^*$-algorithm to return high quality solutions while solving a set of problems using a non-admissible heuristic. The essence of this method is to change heuristic guidance during the search process as new information is learned. This method of learning helps an over-estimating heuristic function converge to an admissible heuristic which contains the insights of the original non-admissible one. The application of this learning system is for cases where optimal or near-optimal solutions are desired and where a strong underestimating heuristic is not available. This leaning process leads to the quality of solutions returned by $A^*$ being steadily improved. As more problems are solved, a dynamically changing approximation to heuristic values is learned and as it is learned
it is also used to guide the search. Variations of this learning technique are developed depending on the amount of computation per problem that the user wishes to invest.

Learning-Real-Time-A*:  
This method, developed by Korf (1990), can be used in real-time search which is one of two classes of search algorithms; the other being off-time search. Off-time algorithms such as A* compute an entire solution before executing the first step in the path. Real-Time algorithms, on the other hand, perform sufficient computation to determine a plausible next move for the problem solver, execute that move in the real world, then perform further computations to determine the following move, and so on, until the goal is reached. Real-time algorithms may not find optimal solutions, but can guarantee a move in constant time. This learning process is called Learning-Real-Time-A* (LRTA*) which builds and updates a table containing heuristic estimates of distances from each state in the problem space to the goal. Initially, the entries in the table correspond to underestimated heuristic evaluations, or zero if none is available. While searching for the goal, the value of these heuristic estimations are updated and become more and more accurate until they eventually become the exact distances to the goal.

The four basic methods classified here as "learning control" are the result of cost-effectiveness trade-offs in storing information about the process of searching. The development of these kinds of learning methods has a significant influence on search efficiency and can be of major interest to both OR and AI researchers.

1.3.3.6 Transformation  
There is an old proverb that says "knowledge is of two kinds, the first kind is a subject which one knows and the second kind is information about where a subject can be found". Transformation like reduction actually uses this fact. The difference between reduction and transformation is that the former uses this fact recursively and the latter uses it once. When a problem is a special case of another problem whose solution is already known, or at least it is known to be easier than the original problem, a transformation can be useful. This transformation generally can occur in
the two directions of induction and deduction. In the first direction a problem is transformed from a general problem into a special case and in the second direction from a special problem to a general case. It is clear that the second direction does not guarantee attaining a solution. Following are two examples that clarify what we mean by transformation.

Our first example is a graph colouring problem, a constraint satisfaction problem in which each variable can be a number of colours and some binary constraints forbid particular pairs of variables from having the same colour. The goal is to see if there is an assignment of colours to the variables using only K colours. Many practical constraint satisfaction problems, such as time-table construction, can be mapped into this problem. Cheeseman and Kanefski (1991) present three transformation rules for this problem by which some hard problems are transformed to easy to solve problems. They have proved that if the transformed problem is K colourable (or not) the original problem will be K-colourable (or not). The first rule concerns the removal of unconstrained nodes, nodes with less than K-constraints, which can always be coloured. The second rule concerns the removal of subsumed nodes; it says that a node n can be removed if there is a node m that is connected to every node that n is connected to. The reason for this is clear: any colour that will work for node m will also work for node n (provided that n is not connected to m). Finally, the third rule concerns merging nodes that have the same colour and recognising these nodes is very important in transforming a problem to a trivial one. If any nodes are fully connected to a clique of size K-1, then these nodes can be merged into a single node, because they must have the same colour.

Cheeseman and Kanefski (1991) also show that these rules can be applied in any order and typically the application of one rule creates a situation where other rules become applicable. They found that the application of these three simple rules transformed all their carefully hand-constructed hard graphs to trivial cases. They state that:
"This kind of problem simplification by pre-processing is often over-looked in discussions of algorithms, yet it can make apparently hard problems trivial."

Our next example, in transformation, is the application of a special case to a general case. Consider the problem of finding what is called a knight's tour on a large-chess board. This is the path traced out by the knight when it is allowed to move in its customary manner but not land on a square more than once. In the 19th century, Warnsdorf's rule was discovered for finding the knight's tour (Goldenberg and Pohl 1984). The rule states that the knight has to be moved to an unvisited square that has the fewest next moves. The rule implicitly suggests that the corner squares are the best places for starting points since they have the fewest in and out moves and so are most easily disconnected from the main graph. Knowing that this rule has worked for a special case of the Hamilton-graph, it can be applied to the general problem. Application of a special case to the general case cannot guarantee attaining any solution; but in this case the Hamilton graph problem is an NP-complete problem and sometimes one has to resort to these methods.

Recognising the fact that one search problem is a special case of the another is not often an easy task. To emphasise this fact, we use following example. Consider the following finite set of numbers, \( S = \{ 2, 4, 1, 8 \} \). We define "binary-adding" of \( S \) as a series of adding operations between members of \( S \), each operation combining two of the members of \( S \) into one member. The cost of any operation is the result. For instance, when 1 and 8 are selected from \( S \) and combined to 9 the cost of this operation is 9. The goal is to achieve a minimum cost series of operations which result in reducing members of \( S \) to just one member. For example, if we select 2 and 1, and combine them to 3, up to this point the cost is 3; now if we select 2 and 8, combine them to 10, up to this point the cost is ten plus 3 which is 13; and at last if we combine 10 and 13, the final result is 23 and the cost of all operations is 23 plus 13 which is 36. Various combinations of methods result in various costs. It is clear that the optimum solution for this simple problem is obtained by selecting two members of \( S \) with the minimum amount, combining them into one, putting the result
in $S$ and continuing this step until only one member of $S$ is left. By induction it can easily be proved that this result is optimum.

Now consider the following problem. Suppose $S$ is a finite set of symbols, for example $S = \{w, x, y, z\}$. A "prefix encoding" of $S$ is an arrangement of a binary string to each symbol based on a binary tree whose left branches are labelled zero and the right ones are labelled one. Each symbol is considered as a leaf of this binary tree. The code of any symbol is the set of zeros and ones on the path leading from the root of tree to that special node. If an encoding is represented by such a tree, it is clearly unambiguous because no code word is a prefix of any other. Now suppose symbol $x$ occurs $t(x)$ times in the string that is to be coded optimally. The length of $x$'s code word is $l(x)$, the depth of $x$ in the decoding binary tree. So the length of the encoded string is,

$$L = \sum_{x \in S} t(x)l(x).$$

Now the problem is to find an optimal prefix encoding which results in minimising $L$. Notice that if the weights are all 1, then the optimal tree is a complete binary tree. This seemingly difficult problem is an instance of the above "binary-adding" problem.

The algorithm, due to Huffman, for finding the optimum encoding tree for this problem works as follows. At the heart of this algorithm is a forest, a set of trees, $F$, initially composed of symbols and each as an individual tree has just one node. Associated with each tree in $F$ is a weight which is just the sum of the weights of the leaves of that tree. The Huffman algorithm selects two trees of minimum weight from $F$ and combines them into one. This step will be repeated until only one tree is left. The Huffman Algorithm has been proved to find the optimal solution and it has been described by Kingston (1990) as a miracle of simplicity. It is difficult to believe that such a seemingly difficult search problem has such a simple solution. It is also interesting to notice that all that has been done in solving this problem is to recognise similarities between the two problems mentioned and hence the application of the solution of the former to the latter.
In general using transformation, the solution of one problem is utilised as a "black box" to solve similar problems.

1.3.3.7 Hybrids

The control methods classified here can assist each other to make the search process more efficient. For instance, in a search problem, heuristic and learning along with abstraction can be used simultaneously. For example, heuristic is often used along with wise pruning and sometimes this process occurs so naturally that a designer does not notice it.

Even various methods discussed in the same class can be used simultaneously; and the number of combinations can be very high. For instance, in the class of wise pruning methods, there can be many hybrid techniques. Pearl (1984) considers hill-climbing (HC), backtracking (BC), and best first (BF) as three major elements which are located in a two-dimensional space. The first dimension is the recovery of pursuit (R) and the second dimension is the scope of evaluation (S).

By the dimension R he means the degree to which the search allows recovery from a disappointing avenue to reach previously suspended alternatives; and by the dimension S he means the number of alternatives considered in each decision. Along the R dimension he finds HC at one extreme, permitting no recovery, and BT and BF, at the other extreme, keeping all suspensions recoverable. Along the S-dimension he found HC and BC focusing narrowly on the set of recently available alternatives, whereas BF examined the entire set of available alternatives including those recently generated as well as those suspended in the past. This provides him with an opportunity to see a continuous spectrum of ways to limit the R and C dimensions and thus many hybrid ways of node selection.

Various heuristic functions also can be used serially to solve the same problem. As mentioned, in many optimisation problems, one has to resort to near optimal solutions. The closeness of an answer to the optimal solution in many of these problems is highly dependent on the heuristic functions used. Since various heuristic
functions can lead to various answers, one can make use of different heuristic functions and select the best answer.

The drawback of this method is that usually the number of ways these heuristic functions are formulated is limited and thus one cannot hope to test many of them. In order to cope with this difficulty, various computational components of a heuristic function can have various weights and these weights can be changed to produce different functions. Scheduling problems are special cases that often use this method and in many cases it works very well.

We have just seen how different methods in one class can be combined, whereas many techniques in other classes like heuristic, wise pruning, reduction, abstraction, learning and transformation can be used to produce hundreds of hybrid methods.

Search efficiency involves creative activity and there is no simple recipe for success. Indeed there are many important search problems for which no acceptable algorithms in terms of the running-time on computers are known. It is not clear whether brilliant ideas for solving them have been missed or there is no efficient search technique for them at all. It is not clear, also, to what extent a brilliant idea embedded in a control method can contribute to the efficiency of a solution. However, by classifying known methods according to similarities in their structure, it is possible to identify strategies which often lead to success. The reason is simply that a lucky guess which contributes to the efficiency of a search algorithm in one special problem has some chance of being efficient in other areas.
2. THE DEVELOPMENT OF TWO INTELLIGENT SEARCH METHODS

2.1 Introduction

In this chapter, we develop two search techniques where one finds optimal solutions and the other finds solutions guaranteed to be within a pre-specified range from the optimal solution. The first technique, LBA* (Learning and Backtracking A*), has been developed based on incorporating a backtracking process into an algorithm called LRTA* (Learning Real Time A*) which was developed by Korf (1990).

LRTA* is an efficient real-time algorithm that guarantees neither optimality nor near-optimality of solutions found. The reason that this algorithm is called real-time is that existing state-space search methods are divided into the two classes of off-line and real-time algorithms. Off-line algorithms such as A* and IDA* compute an entire solution before executing the first step in the path. Real-time algorithms such as LRTA*, on the other hand, perform sufficient computation to determine each step independently until the goal state is reached. Therefore real-time algorithms cannot find and guarantee optimal solutions.

LRTA* is a search algorithm which utilises initial heuristic estimates and continuously improves them as the search process continues. This algorithm updates heuristic estimates of states by comparing them with those of their neighbours in the process of search. The initial heuristic value of every state is an evaluation of the distance of that state from the goal state and is assumed not to over-estimate the actual distance. Through repeated exploration of the state space, however, these estimates will lead to more accurate heuristic values. In this approach the optimal solution cannot be found in a single trial and moreover updated heuristic values cannot be used in the same problem in which these values are obtained. However, while starting from various states to go to the goal state, these updated heuristic values can be utilised.

By the incorporating a backtracking process into LRTA* and activating this process when updating the heuristic estimate of a state, we produce the algorithm LBA* (Learning and Backtracking A*). Contrary to LRTA*, in LBA*, updated
heuristic values can be used in the same problem in which these values are obtained. The main theme of this algorithm is to guarantee the improvement of the heuristic estimate of each state from which backtracking is done. Backtracking along with updating heuristic values help LBA* to find a solution using the circulation of updated heuristic estimates through the states. We will prove that the application of LBA* leads to an optimal solution for any state-space problem in which heuristic estimates of states do not overestimate their actual values. The contribution that LBA* makes to the general idea of backtracking is that: assuming that all numbers used in a problem are integers, the number of backtracks in any state is limited to the amount by which heuristic estimate of that individual state underestimates its actual value. This is something that, as yet, no backtracking algorithm has guaranteed.

Then we improve LBA* to LCBA* (Learning and Controlled Backtracking A*) which finds a solution within a specified bracket of the optimal solution. The application of LCBA* is useful in cases where finding the optimal solution of a combinatorial problem is very time-consuming or even impossible. In these cases, the user may wish to find a solution within a specified bracket of the optimal solution. A proof is presented for the theorem that all the solutions found by LCBA* are guaranteed to be within specified brackets of the optimal solutions. These brackets are determined by users prior to applying LCBA* to their problems and are based on the accuracy required. Then two other versions of LCBA* are introduced. The first version only improves the efficiency of LCBA* in some special cases; whereas in the second version, constraints on computer memory and CPU time automatically determine the bracket that was determined originally by the user.

Since LRTA* (Korf 1990) is the starting point from which we develop these searching techniques, its operation is briefly described.

Among the search algorithms with finite state space and heuristic estimates for every state to the goal state, LRTA* represents a major research direction which takes into consideration the effect of learning in the search process. LRTA* builds and updates a table containing heuristic estimates of the cost (distance) from each state in the
problem to the goal state. Initially, the entries in the table correspond to the initial heuristic evaluations which are assumed to be lowerbounds on actual costs. LRTA* improves the accuracy of these entries during the searching process.

From the initial state, the search process starts by comparing its heuristic estimate with the "compound values" of all neighbouring states where each "compound value" includes the estimate to the goal state and the edge cost from the current state to each neighbouring state. The neighbouring state with the minimum compound value is chosen for the next stage of expansion, and the heuristic estimate of the current state is replaced with this value to reflect a more accurate estimate. This later part represents the updating mechanism of the Korf's algorithm which can be considered as a learning process. Assuming that x is the current state of a search process, the LTRA* algorithm repeats the following steps until the goal state becomes the current state.

1. Calculate the compound value of \( f(x') = h(x') + k(x, x') \) for each neighbour \( x' \) of the current state \( x \) where \( h(x') \) is the current heuristic estimate of the distance from \( x' \) to the goal state and \( k(x, x') \) is the edge cost from \( x \) to \( x' \).

2. Move to a neighbour with the minimum compound value, \( f(x') \), and consider it as the current state.

3. Update the value of \( h(x) \) to the minimum compound value of its neighbours.

The reason for updating the value of \( h(x) \) is that since the compound value of \( f(x') \) represents a lowerbound on the actual distance to the goal through each of the neighbours, then the actual distance from the given state must be, at least, as large as the smallest of these compound values. The valuable contribution of this method to searching techniques is the improvement of the heuristic estimates of states during the process of problem solving. These improved heuristic estimates can be used in the following trials, when starting from other initial states to go to the same goal state. In a finite problem space with positive edge costs, in which there exists a path from every state to the goal, this algorithm will always be able to reach the goal. Although
there is no guarantee of optimality or near-optimality of the solutions produced by Korf's algorithm in any single problem-solving trial, the repeated trials will eventually adjust each heuristic estimate on the final path to its actual value and hence this leads to the optimum solution.

2.2 Learning and Backtracking A* (LBA*)

The fundamental feature of this algorithm (LBA*) is the repetitive application of the updating cycle which normally consists of a trial through searching, evaluation after the trial, and updating through feedback. The major difference between this algorithm and Korf's algorithm lies in the implementation of a backtracking process which occurs when updating the heuristic estimate of any state.

LBA* starts with the initial state as the current state and changes the current state until a path to the goal state is found. To search for the next state for expansion, it calculates the compound values of all neighbouring states by adding the heuristic estimate of each neighbouring state and the edge cost from the current state to that neighbouring state. The neighbouring state with the minimum compound value is selected as the current state and again computing new compound values and comparing them for selection of the next current state continues. This process of selecting the neighbouring state with the minimum compound value continues until the heuristic estimate of the current state is less than the minimum of the compound values of its neighbouring states. In this case the heuristic estimate of the current state is updated to this minimum compound value and the decision about selecting this current state to be on the path is revised by removing it from the path and backtracking to the previous state currently on the path.

The fact that a state is blocked for further expansion (dead end) may also serve as an indication for the algorithm to learn not to enter the same state again in the future. The backtracking routine is initiated following the updating of the heuristic estimate of a state. Through this backtracking process, the current state leaves the path and the previous state on the path will become the current state. In other words, whenever the heuristic estimate of the current state is updated to the minimum compound value of
its neighbouring states, this state will leave the path and the previous state on the path will became the current state. Then again re-examination of neighbouring states starts and it is likely that this re-examination will lead either to a change of direction for expansion, or to the adjustment of its heuristic value and hence one more stage of backtracking. Depending on the original estimate of the initial state, the backtracking process may retreat all the way back to the initial state as often as needed in order to update its estimate before the final path is found. The rationale for this is that the inclusion of a state in the final path is promising only when its neighbours indicate so. This algorithm can be implemented in the following manner:

**Step 0:** Apply a heuristic function to generate a non-overestimating initial heuristic estimate $h(x)$ for the distance of every state $x$ to the goal state.

**Step 1:** Put the initial state on the backtrack list called OPEN.

**Step 2:** Call the top-most state on the OPEN list $x$. If $x$ is the goal state, stop.

**Step 3:** If $x$ is a dead-end state, replace its $h(x)$ with a very large value, remove $x$ from the OPEN list, and go to step 2.

**Step 4:** Evaluate the compound value of $k(x,y)+h(y)$ for every neighbouring state $y$ of $x$, and find the state with the minimum value. Call this state $x'$. Break ties randomly. $k(x,y)$ represents the positive edge cost from state $x$ to state $y$.

**Step 5:** If $h(x) \geq k(x,x')+h(x')$, then add $x'$ to the OPEN list as the top-most state and go to step 2.

**Step 6:** Replace $h(x)$ with $k(x,x')+h(x')$.

**Step 7:** If $x$ is not the root state, remove $x$ from the OPEN list.

**Step 8:** go to step 2.

With this algorithm, the memory requirements include the storage of the most up-to-date heuristic value of each state and the OPEN list which at the end of the algorithm shows a path from the initial state to the goal state. The list keeps only the states on a path from the initial to the current state, and a current state is always on the top of the list. When the heuristic estimate of a current state is updated, backtracking is carried out by removing the current state from the list, and letting its previous state become
the current state as indicated in step 7. Step 7 is carried out after step 6 which updates the heuristic estimate of the current state. This adjustment is the result of the comparison of estimates with neighbours as indicated in step 4 and the false condition of step 5. Should the condition of step 5 be true, the algorithm will continue to expand the current state without the need for backtracking. Step 3 is arbitrary and has been used to assign a large value to the heuristic estimate of a state that has been considered a dead-end, ensuring no future visit to such states.

The way in which the heuristic estimate of a state is adjusted, as indicated in step 6, does ensure that the newly adjusted value raises the lowerbound and will never be greater than its actual value. As the search process continues, the heuristic estimates of states on the final path will finally converge to their actual values through the guidance of the edge costs. A proof that the application of LBA* will lead to finding the optimal solution is presented in the following.

**Theorem and Proof**

**Theorem 1.**

For a finite problem space with positive edge costs and non-overestimating initial heuristic values, in which the goal state is reachable from the initial state, the application of LBA* will find the optimum path.

**Proof:**

Assume the contrary, that there exits a path $P'$ to the goal state with a smaller cost than that of the path $P$ found by LBA*. Let the states on the path $P$ be denoted as $x(1), x(2), \ldots x(n)$ and the states on the path $P'$ be denoted as $y(1), y(2), \ldots y(q)$, where $x(1)$ and $y(1)$ are the same initial state, and $x(n)$ and $y(q)$ are the same goal state.

Let $y(m)$ be the first state of $P'$ which is not on the path $P$. Thus both $x(m)$ and $y(m)$ are the neighbours of their previous common state $x(m-1)$.

Let $k$ representing the edge cost from a state to a neighbouring one, the following relation must be satisfied by $P'$ to produce a smaller cost than $P$.

$$k\{x(m-1),y(m)\} + k\{y(m),y(m+1)\} + \ldots + k\{y(q-1),y(q)\} <$$
\[ k\{x(m-1),x(m)\} + k\{x(m),x(m+1)\} + \ldots + k\{x(n-1),x(n)\}. \quad (1) \]

This relation can be further modified to (2) because of the fact that the adjustment of a heuristic estimate in the algorithm will never lead to a value higher than its actual value.

\[ k\{x(m-1),y(m)\} + h\{y(m)\} < k\{x(m-1),x(m)\} + k\{x(m),x(m+1)\} + \ldots + k\{x(n-1),x(n)\}. \quad (2) \]

As indicated by steps 5 and 6 of the algorithm, the following relation is always true for a state \(x(r)\) under LBA*.

\[ h\{x(r)\} \geq k\{x(r),x(r+1)\} + h\{x(r+1)\}. \quad (3) \]

(3) can be rearranged as,

\[ h\{x(r)\} - h\{x(r+1)\} \geq k\{x(r),x(r+1)\}. \quad (4) \]

By expanding and summing the inequality (4) over the state space of path \(P\) from \(x(m)\) to the goal state, the following relation is obtained,

\[ h\{x(m)\} - h\{x(n)\} \geq k\{x(m),x(m+1)\} + k\{x(m+1),x(m+2)\} + \ldots + k\{x(n-1),x(n)\}. \quad (5) \]

With the estimate from the goal state to itself being 0, \(h\{x(n)\} = 0\), and (5) becomes,

\[ h\{x(m)\} \geq k\{x(m),x(m+1)\} + k\{x(m+1),x(m+2)\} + \ldots + k\{x(n-1),x(n)\}. \quad (6) \]

The fact that LBA*, at state \(x(m-1)\), has preferred \(x(m)\) to \(y(m)\), as indicated by step 4, also leads to the following relation,

\[ k\{x(m-1),y(m)\} + h\{y(m)\} \geq k\{x(m-1),x(m)\} + h\{x(m)\}. \quad (7) \]

With the substitution of \(h\{x(m)\}\) from (6) in (7), we have,

\[ k\{x(m-1),y(m)\} + h\{y(m)\} \geq k\{x(m-1),x(m)\} + k\{x(m),x(m+1)\} + k\{x(m+1),x(m+2)\} + \ldots + k\{x(n-1),x(n)\}. \quad (8) \]

It is obvious that (2), which is the result of the earlier assumption that \(P'\) is a better path than \(P\), contradicts (8), which is the result of the actual application of LBA*. Thus, \(P'\) does not exist, and the application of LBA* will always lead to finding the minimum path.
An Example

To show how LBA* works, we apply it to a grid problem represented in figure 2.1. This grid problem can be considered as a state space with sixteen states (cells) on which operators with the cost of 1 operate and transform them to one of their neighbouring states (cells), provided that no highlighted border (barrier) exists between the state on which the operator acts and its neighbouring state. The left bottom cell is state (1,1) which is the initial state and the top right cell is state (4,4) which is the goal state. Obviously, the purpose of the application of LBA* is to find a series of operators with minimum cost to transform the initial state to the goal state. This can be stated by the simple fact that LBA* tries to find a path with minimum cost from the initial state to the goal state.

A heuristic estimate for every state is easily constructed based on the assumption of removing all vertical and horizontal barriers. Figure 2.1 shows these estimates as well as the barriers. Now all operations, in detail, are described. At first state (1,1), the left bottom cell, as the initial state is put on the OPEN list (step1). Now the OPEN list has only one member which is the initial state. State (1,1) as the top-most state on the OPEN list is called x (step2). All neighbouring states of x, in this case only state (2,1), are evaluated and among them state (2,1), itself, with the minimum compound
value of \((1+5)\) is selected and is called \(x'\) (step4). Notice that in the compound value of \((1+5)\) the value of \(1, k(x,y),\) is the edge cost from state \((1,1)\) to state \((2,1)\) and the value of \(5, h(y),\) is the current heuristic estimate of state \((2,1)\). Since the heuristic estimate of \(x, 6,\) is not less than \((1+5)\), no updating occurs and \(x'\) is added to the top of the list. Similarly, state \((2,2)\) is added. State \((2,2)\) as the top-most state on the list is called \(x\) (step2). All neighbouring states of \(x, (2,1)\) and \((1,2),\) are evaluated and both result in the same compound value of \((1+5)\).

The tie is broken randomly and state \((2,1)\) with the compound value of \((1+5)\) is selected and is called \(x'\) (step4). Since the heuristic estimate of \(x\) which now is 4 is less than \((1+5)\), the old estimate is replaced with the compound value of \((1+5)\) and this state is removed from the list. The same process causes state \((2,1)\) with the updated heuristic estimate of \((1+6)\) to be removed from the list. Now state \((1,1)\) as the top-most state on the list is called \(x\). All neighbouring states of \(x\), in this case only state \((2,1),\) are evaluated and among them state \((2,1),\) itself, with minimum compound value of \((1+7)\) is selected and is called \(x'\) (step4). Since the heuristic estimate of \(x, 6,\) is less than the compound value of \((1+7)\), the old value is replaced with \((1+7)\). Notice that now \(x,\) state \((1,1),\) is the initial state and based on step7 when the initial state experiences learning, no backtracking happens. All neighbouring states of \(x\) are evaluated and among them state \((2,1),\) with the minimum compound value of \((1+7)\) is selected and is called \(x'\). Since the heuristic estimate of \(x\) which now is 8 is not less than \((1+7),\) \(x'\) is added to the top of the list (step5). Similarly, the states of \((2,2), (1,2), (1,3), (2,3)\) and \((3,3)\) are added. Notice that after the selection of state \((2,3)\) both states \((3,3)\) and \((2,4)\) had the same chance to be selected and that the tie was broken randomly.

Now state \((3,3)\) as the top-most state on the list experiences 2 units of learning and is removed from the list and state \((2,3)\) is called \(x\). All neighbouring states of \(x\) are evaluated and the compound values of \((1+4)\) and \((1+2)\) are obtained and among them the minimum which is associated with state \((2,4)\) is selected and state \((2,4)\) is added
to the list. The same process causes states (3,4) and (4,4) to join the list; and, finally, since state (4,4) is the goal state, the algorithm stops (step2).

After applying LBA* to this problem some heuristic estimates are updated and the optimal solution is found. Figure 2.2 shows updated heuristic estimates and the optimal path found by LBA*.

As shown, the algorithm progressed from the initial state to the goal state by updating initial heuristic estimates during the process of searching and activating a backtracking process whenever an updating occurred. Notice that even after finding the optimal path, still the heuristic estimates of many states are underestimated. For instance the heuristic estimate of state (4,1), 3, which originally underestimated the actual distance by 2 units, is still the same, indicating that this state was not visited at all during the search. The same is true for state (1,4). However, the heuristic estimates of all the states on the optimal path have been improved to their actual distances. The point is that whenever a state is close to the optimal path, there is more chance of its heuristic estimate being updated. This means that the learning process and the circulation of this "learned information" through activating a
backtracking process have led to many states not be visited even once and only the close neighbours to the final optimum path have been considered.

**Efficiency of LBA**

The fact that LBA* needs to store the heuristic estimate for every state of a problem has led to the upperbound for space complexity of LBA* being $n$, where $n$ is the number of states in the problem. In practice, however, the complexity can be lower because usually there exists a function which computes the original heuristic estimates, and it is only necessary to store in memory those values which differ from the computed ones. For instance, in the grid problem of figure 2.3, the function to compute the distance from cell $(i,j)$ to the goal state, cell $(21,21)$, is $(2*21-i-j)$. The total number of states in this problem is $21*21=441$, however, the algorithm reaches the optimal path by updating only 140 of them. To derive the time complexity for LBA*, we assume that all numbers used in the problem are positive integers.

![Figure 2.3: A sample grid problem and its optimal path found by LBA*](image)

With this assumption, the worst-case of LBA* is $n*s$, where $n$ is the total number of states and $s$ is the final cost from the initial state to the goal state. This worst-case may happen when the initial heuristic estimates for all states are zero, there is no information at all, and the edge cost from every state to its neighbours is assumed to
be one. The latter assumption will lead to only one unit increment in each updating process, and the n*s figure is the worst-case when every state has to be visited s times.

In reality, the actual worst-case will only be a portion of this figure n*s, because all states except the initial state will have a smaller "true" cost to the goal state than s. For most problems, where initial information for heuristic estimates is obtainable and edge costs vary from one state to another, the number of visits required before the final path is found could be very low.

Figures 2.4 and 2.5 demonstrate, using the example in figure 2.3, the contrast between the total amount of initial underestimation and the total number of backtracks required by LBA*. These are 1324 and 119 respectively. Comparison of these two numbers reveals that figure 2.4, on average, is ten times higher than figure 2.5. It can be seen in figure 2.5 that very few cells were subject to backtracks, and the cells with notable backtracks are located in the neighbourhood of the optimal path. For instance cells (21,17), (21,16), and (21,15) which have the largest heuristic underestimates in figure 2.4, have been backtracked infrequently compared to their heuristic underestimates.

\[ \text{Figure 2.4: A representation of the initial heuristic-underestimation of every cell.} \]
We compared the application of LBA* and the continuous application of LRTA* to find optimal solutions for these grid problems with different structures. Over the 20 square grid problems tested, LBA* consistently outperformed LRTA*. LBA* found all optimum solutions in a single problem solving trial, whereas as expected LRTA* required more than one trial.

Table 2.1 shows this comparison in detail. Columns 1, 2 and 3 show information about instances tested including their size and the barrier percent. Columns 4 and 5 indicate how LBA* has functioned; and columns 6 and 7 show the functioning of the continuous application of LRTA* to find optimum solutions. The amount of time that either LRTA* or LBA* takes to solve any of these instances is in proportion with the number of cells it visits. Therefore the ratio of column 6 to column 4 shows how much faster LBA*, compared to LRTA*, has solved the associated instance. This ratio has been shown in column 8. As can be seen, with 5 different square sizes 10, 15, 20, 25, and 30, and 4 different barrier percentages 15%, 25%, 35%, 45%, these ratios range from 3.7 to 28.7. The trend is clear that the larger the size of a problem the larger the ratio will be.

Figure 2.5: A representation of the number of backtracks required by LBA* for every cell.
Table 2.1: A comparison between performances of LBA* and LRTA* on 20 sample grid problems with different characteristics.

<table>
<thead>
<tr>
<th>Instance No</th>
<th>Size</th>
<th>Barrier Percent</th>
<th>LBA*</th>
<th>LRTA*</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td></td>
<td>Number of cells Visited</td>
<td>Update Per Visit</td>
<td>Number of cells Visited</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>15</td>
<td>26</td>
<td>.30</td>
<td>98</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>25</td>
<td>40</td>
<td>.55</td>
<td>208</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>35</td>
<td>44</td>
<td>.59</td>
<td>278</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>45</td>
<td>54</td>
<td>.66</td>
<td>270</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>15</td>
<td>42</td>
<td>.33</td>
<td>182</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>25</td>
<td>60</td>
<td>.53</td>
<td>236</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>35</td>
<td>66</td>
<td>.57</td>
<td>276</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>45</td>
<td>70</td>
<td>.60</td>
<td>434</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
<td>15</td>
<td>62</td>
<td>.38</td>
<td>290</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>25</td>
<td>56</td>
<td>.32</td>
<td>272</td>
</tr>
<tr>
<td>11</td>
<td>20</td>
<td>35</td>
<td>148</td>
<td>.74</td>
<td>1310</td>
</tr>
<tr>
<td>12</td>
<td>20</td>
<td>45</td>
<td>430</td>
<td>.90</td>
<td>2794</td>
</tr>
<tr>
<td>13</td>
<td>25</td>
<td>15</td>
<td>114</td>
<td>.57</td>
<td>806</td>
</tr>
<tr>
<td>14</td>
<td>25</td>
<td>25</td>
<td>136</td>
<td>.72</td>
<td>1104</td>
</tr>
<tr>
<td>15</td>
<td>25</td>
<td>35</td>
<td>148</td>
<td>.74</td>
<td>1311</td>
</tr>
<tr>
<td>16</td>
<td>25</td>
<td>45</td>
<td>327</td>
<td>.85</td>
<td>4730</td>
</tr>
<tr>
<td>17</td>
<td>30</td>
<td>15</td>
<td>138</td>
<td>.57</td>
<td>1390</td>
</tr>
<tr>
<td>18</td>
<td>30</td>
<td>25</td>
<td>194</td>
<td>.70</td>
<td>3374</td>
</tr>
<tr>
<td>19</td>
<td>30</td>
<td>35</td>
<td>426</td>
<td>.87</td>
<td>7876</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>45</td>
<td>319</td>
<td>.81</td>
<td>9180</td>
</tr>
</tbody>
</table>
2.3 Learning and Controlled Backtracking A* (LCBA*)

The computational methods for producing solutions to optimisation problems can be classified according to whether they are designed to produce optimal solutions or approximate solutions. The first class of methods can be called "exact" and the second class "approximate". LBA* belongs to the first class and LRTA* belongs to the second. When looking for an approximate solution, in some cases one may require a solution within a distant boundary from the optimal solution and in other cases some solution within special distance is required. In addition, in some cases one has only a small amount of computer resources (CPU time and memory) available and in other cases resources are not tightly constrained.

An important but least investigated class of algorithms referred to as guaranteed accuracy methods produce approximate solutions with costs guaranteed to be within specified brackets of the optimal solutions. When one is faced with a problem so large and complex that its exact solution with regard to available computational resources appears infeasible, the next best option is to seek a solution of known quality. Guaranteed accuracy methods can be interpreted as the combination of exact and approximate algorithms. Usually these methods are very case sensitive and can be thought of as a particular theorem proving method applied to a special problem, providing a feasible solution together with a proof that its cost is within a certain bracket of the optimal solution. For instance, in branch and bound algorithms this proof is reflected in the construction of upperbound and lowerbound subroutines, relying heavily on insight and knowledge about a special problem. Despite branch and bound algorithms which due to the construction of upperbound and lowerbound subroutines are very case sensitive, this class of algorithms has received relatively very little attention.

As stated, the problem with LRTA* is that when it finds a solution it gives no measure as to how far this solution can potentially be from the optimal solution. Although LBA* removes this problem and finds the optimal solution, its application to graphs with a very large number of potential nodes takes an excessive amount of
time. Learning and Controlled Backtracking A* (LCBA*) is an improvement over LBA* which addresses this problem and trades speed with accuracy. Belonging to the class of guaranteed accuracy methods, LCBA* accepts a parameter as input, $p$, and finds a solution within a bracket dependent on $p$ of the optimal solution. The magnitude of this input value, which is called alternatively a threshold or bracket parameter, determines the speed of LCBA*. For instance, if this parameter is very large then LCBA* works like LRTA* and without any guarantee finds a solution without any backtracking. On the other hand, if this parameter is selected as zero then LCBA* works like LBA* and finds the optimal solution by doing all necessary backtracks.

The difference between LCBA* and many other guaranteed accuracy methods is that this method is not problem sensitive and can be applied to any graph-search problem. The reason for the general applicability of LCBA* is that it is based on a very general theorem. This theorem states that the application of LCBA* to any graph-search problem with positive edge cost and non-overestimating heuristic estimates leads to a guaranteed solution within a determined bracket of the optimal solution.

LCBA* accepts a threshold parameter ($p$) as input and, as soon as the total amount of learning exceeds the parameter, it backtracks all the way to the preceding state of the first state which has experienced learning. The only major difference between LBA* and LCBA* is that the backtracking process in LCBA* is controlled and it is not activated by an updating in the heuristic estimate of a state. In LBA*, updating caused the backtracking process to be activated, whereas in LCBA* this updating only causes the value of $p$ to be decreased by the amount of improvement in the updated heuristic estimate. For instance, suppose the heuristic estimate of a state, as the result of a comparison between the compound values of its neighbours, changes from 9 to 12. This situation will cause the process of backtracking to be activated in LBA*, whereas in LCBA* it only causes the value of remaining threshold, which originally was set to $p$, to be decreased by 3, and the backtracking process is activated as soon as the value of remaining threshold takes on a negative value. Notice that the
value of remaining threshold after any updating in the heuristic estimate of a state decreases and this process gradually can change its initial value, p, to a negative value which activates a backtracking process. This process in LBCA* causes the state preceding the first state on the OPEN list which has experienced learning and has changed the current value of p to become the current state. Since the value of p indicates the maximum distance of the optimal solution from the solution found by the algorithm, with any backtracking process in LCBA* the current negative value of remaining threshold is set to p and again the process of decreasing remaining threshold upon the improvement of a heuristic estimate continues. All other parts of LCBA* are the same as those of LBA*. This algorithm can be implemented as follows:

Step 0: Apply a heuristic function to generate a non-overestimating initial heuristic estimate h(x) for the distance of every state x to the goal state. Let p'=p and set b to the initial state. The value of p, as the total threshold, determines the maximum amount by which the final solution may differ from the optimal solution and b is the state to which the algorithm is backtracked. The value of p' measures the remaining threshold, which is updated to p whenever a backtracking happens and is decreased whenever learning occurs.

Step 1: Put the initial state on the OPEN list; and keep p' associated with it on the list

Step 2: Call the top-most state on the OPEN list x and its associated value p'. If it is the goal state stop.

Step 3: Evaluate the compound value of k(x,y)+h(y) for every neighbouring state y of x and find the state with the minimum value. Call this state x'. Break ties arbitrarily. The value k(x,y) represents the positive edge cost from state x to state y.

Step 4: Let L=k(x,x')+h(x')-h(x). If L <=0 and x' is not on the OPEN list then put it on the list, keep p' associated with it and go to step 2.
Step 5: Replace $h(x)$ with $k(x,x') + h(x')$. If $x'$ is already on the OPEN list then backtrack all the way to it, change $p'$ to the value associated with this $x'$ and go back to step 2.

Step 6: If $p = p'$ and $x$ is not the initial state then set $b$ to the preceding state of $x$ on the OPEN list.

Step 7: If $L \leq p'$ then decrease $p'$ by $L$ and put $x'$ on the list and keep $p'$ associated with it on the list and go to step 2.

Step 8: If $x$ is not the initial state then backtrack all the way to $b$ and set $b$ to the initial state.

Step 9: Go to step 2.

As shown above, LCBA* differs from LBA* in the way it delays activating the backtracking process by keeping a record of the remaining threshold, $p'$, in steps 4 and 7. The updating process in LCBA* causes the threshold value to be decreased in step 7 and when it becomes negative, step 8 activates the backtracking process.

A proof for the theorem that the application of LCBA* will lead to finding a solution within a specified range from the optimal solution is presented.

Theorem and Proof

Theorem 2.

For a finite problem space with positive edge cost and non-overestimating initial heuristic values, in which the goal state is reachable from the initial state, the application of LCBA* leads to a solution with accuracy guaranteed within a specified bracket, $p$, of the optimal solution.

Proof:

Let the states on the path found by LCBA* be denoted as:

$x(1), x(2), \ldots, x(n)$ where $x(1)$ is the initial state, $x(n)$ is the goal state and $x(2), x(3), \ldots, x(n-1)$ are the intermediate states.

Let,

$h\{x(i)\} = k\{x(i), x(i+1)\} + h\{x(i+1)\} - L(i)$, for $i=1,2,\ldots,n-1$.  (1)
where $L(i)$ is calculated in step 4 and represents the value by which $h(x(i))$ can be updated.

From step 7 we see that in some cases $L(i)$ in (1) is a positive number and the sum of the positive values of $L(i)$ has $p$ as an upperbound.

(1) can be rearranged as,

$$h(x(i)) - h(x(i+1)) = k(x(i), x(i+1)) - L(i), \text{ for } i=1,2,\ldots,n-1.$$  \hspace{1cm} (2)

By expanding and summing (2) over the state space from $i$ to $n-1$, (2) becomes,

$$h(x(1)) - h(x(n)) = k(x(0), x(1)) + k(x(1), x(2)) + \ldots + k(x(n-1), x(n)) - L(1) - L(2) - \ldots - L(n-1).$$  \hspace{1cm} (3)

Since $x(n)$ is the goal state, with $h(x(n)) = 0$, (3) becomes,

$$h(x(1)) = k(x(1), x(2)) + k(x(2), x(3)) + \ldots + k(x(n-1), x(n)) - \{L(1) + L(2) + \ldots + L(n-1)\}.$$  \hspace{1cm} (4)

Replacing $\{L(1) + L(2) + \ldots + L(n-1)\}$ with $p$, (4) becomes,

$$h(x(1)) \geq k(x(1), x(2)) + k(x(2), x(3)) + \ldots + k(x(n-1), x(n)) - p,$$

which means,

$$h(x(1)) \geq \text{current solution} - p$$  \hspace{1cm} (5)

All heuristic estimates, including $h(x(1))$ which estimates the optimal solution, are kept not-overestimated. Therefore replacing $h(x(1))$ in (5) with the optimal solution the inequality (5) becomes,

optimal solution $\geq$ current solution - $p$,

or,

current solution $\leq$ optimal solution + $p$.

Therefore, the solution found using LCBA* does not exceed the optimal solution by more than $p$ units.

**Examples**

To show how LCBA* works, we implement it with the input threshold value of 2 on a grid problem presented in figure 2.6.
The operations carried out by LCBA* are similar to those of LBA*. It starts by placing state (1,1) as the initial state on the OPEN list. States (1,2) and (2,2) are the second and the third states which join the list. Up to this point there is no difference between the functioning of LBA* and that of LCBA*. Now all two neighbouring states of state (2,2) are evaluated and both result in the compound value of (1+5). Since the heuristic estimate of the current state, (2,2), which now is 4, is less than the compound value of (1+5) then it is updated to (1+5) and two units of improvement occurs. At this point LBA* backtracked to the previous state on the list whereas LCBA* decreases the remaining threshold value (p') which originally was set to the input threshold value (p) by the amount of the improvement made, (2-2=0).

Since the new value of the remaining threshold, 0, is not less than zero, no backtracking occurs and the process continues. The states of (2,1), (3,1), (3,2) and (3,3) are added to the list without any updating in their heuristic estimates.

Now state (3,3) is the current state and both its two neighbouring states are evaluated and both have a compound value of (3+1). Since the heuristic estimate of the current state, (3,3), now is 2, then it is updated to (3+1) and two units of improvement occurs. This causes the current value of the remaining threshold, 0, to be decreased by 2, and because of its negativity the backtracking process is activated.

The backtracking process removes all the states (3,3), (3,2), (3,1), (2,1) and (2,2) from the list and changes the remaining threshold, p', to its original value, 2. Notice that the improvement in the heuristic estimate of the state (2,2) for the first time had
decreased the original value of p from 2 to 0 and that's why the backtracking was done to its preceding state, (2,1), on the OPEN list.

Beginning with state (1,2) as the current state, the states (1,3), (2,3), (3,3), (3,2), (4,2), (4,3) and (4,4) join the list and a guaranteed solution, within bracket 2, is created for the problem. Values of p' associated with these states on the OPEN list are 2,0,0,0,0,0, respectively. Notice that this path is the optimum path as well; however, LCBA* gives no guarantee of its optimality.

After the application of LCBA*, with a threshold value of 2, some heuristic estimates improve and a solution within 2 units of the optimal solution has been found. Figure 2.7 shows updated heuristic estimates and the LCBA* solution.

![Figure 2.7: Updated heuristic estimates and the LCBA* solution.](image)

To show how LCBA* trades speed with accuracy, we now apply it to three problems, determining how the exchange rate between speed and accuracy varies from one problem to another.

In some cases by sacrificing a small amount of accuracy, a very speedy solution is obtained; whereas in other cases by making the algorithm fast, a large amount of accuracy is sacrificed. Figure 2.8 shows a grid problem with its optimal path found by LBA*.
The length of the optimal path is 47 and the number of cells visited during the search is 919. Figure 2.9 shows how these visits have been distributed throughout the state space.

Applying LCBA* with a threshold of 5 leads to the path shown in figure 2.10.
Figure 2.10: A sample grid problem and the path found by LCBA*

The length of the path in this case is 49, which is within 5 units of the optimal solution. The number of cells visited is 167 and figure 2.11 shows how these visits have been distributed throughout the state space.

Figure 2.11: A representation of the number of visits required by LCBA* for every cell.

As shown, all backtracks have been concentrated near the path found by LCBA*.

We now consider another example. Figure 2.12 shows a shortest path problem on a graph with 100 nodes (states) and 300 arcs (operators). The cost of every arc is its...
length on the plate and the heuristic estimate of every state is calculated based on the assumption that there exists an arc connecting that state to the goal state. The middle most left state is the initial state and the middle most right state is the goal state.

![Figure 2.12: A sample randomly-connected graph.](image)

Figure 2.12: A sample randomly-connected graph.

Figure 2.13 shows the optimal path found by LBA*. The length of this optimal path is 1274 and the number of states visited during the search is 1573.

![Figure 2-13: The optimal path found by LBA* for the randomly-connected graph.](image)

Figure 2-13: The optimal path found by LBA* for the randomly-connected graph.

Applying LCBA* with a bracket of 500 leads to the path shown in figure 2.14.
Figure 2.14: A guaranteed-accuracy path using LCBA* for the randomly-connected graph.

The length of the path in this case is 1689, which is within 500 units from the optimal solution.; and the number of states which have been visited is 473. Therefore, in this problem by loosing thirty-nine percent, (500/1274), significance in the solution, speed increased by more than three times (1573/473).

Figure 2.15 shows another shortest path problem on a graph with, again, 100 nodes and 300 arcs. This time the way in which nodes are connected to each other differs from that of the previous problem. In this example nodes are connected to their neighbouring nodes in a manner which typically happens in transportation networks, whereas in the previous problem nodes were connected randomly to each other.

Figure 2.15: A sample neighbour-connected graph.
Figure 2-16 shows the optimal solution obtained by LBA*. The length of this optimal path is 768 and the number of nodes which have been visited is 1375.

![Figure 2.16: The optimal path using LBA* for the neighbour-connected graph.](image)

Applying LCBA* with a bracket of 200 leads to the path shown in figure 2.17.

![Figure 2.17: A guaranteed-accuracy path using LCBA* for the neighbour-connected graph.](image)

The length of the solution path in this case is 844, which is within 200 units of the optimal solution; and the number of nodes which have been visited is 34. Comparing this number, 34, with 1375 reveals that for this problem LCBA* has worked forty times faster than LBA*. However notice that the price paid for this increased speed is near-optimality, within a bracket of 200, sacrificing twenty-six percent, \((200/768)\), significance in accuracy.

Theorem 2 has been proved in a very general manner. In this theorem it has not been specified as to where backtracking has occurred. Therefore, the backtracking can
occur to any other state. This generality allows many other versions of LBCA* to be derived, all based on the same theorem 2. Two of these alternative versions are presented now as VER_1 and VER_2.

**VER_1**
The main point in LCBA* is that when the remaining threshold value \( p' \) takes on a negative value, a backtrack is done to the state on the OPEN list whose associated threshold value is equal to the original threshold value. This indicates that we are aiming to maintain the maximum remaining threshold value \( (p') \). Backtracking to the state with the maximum threshold value is a good choice because it provides a large amount of room for manoeuvring in the selection of the remaining states to connect the current state to the goal state. However, on some occasions it is efficient to set a priority to select a state which is as close as possible to the goal state as well as having a remaining threshold value \( (p') \). In LCBA*, this nearness to the goal state is not considered and the state backtracked to may be quite distant from the goal state.

For instance, suppose that the following states are on the OPEN list: \((1,7,6,8,3,9)\).

<table>
<thead>
<tr>
<th>Order in list</th>
<th>State Number</th>
<th>Current Heuristic Estimate</th>
<th>Last Improvement Made In the Heuristic Estimate</th>
<th>Remaining Threshold value ((p'))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>40</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>35</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>33</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>10</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>7</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 2.2 shows the heuristic value and the remaining threshold value, \( p' \), for each state on the OPEN list. As shown, the total threshold, \( p \), is 5 which is associated with the initial state in row 1 and the first state on the list whose heuristic estimate has been updated is state 6 in row 3. One unit improvement in the heuristic estimate of this state has caused the value of the remaining threshold to be decreased by 1 to a
value of 4. The next improvement in the heuristic estimates of states has occurred in row 5. In this row two units of improvement in the heuristic estimate of state 3 has occurred and its heuristic estimate has changed, from 10 to 8. This has caused the current value of the remaining threshold, \( p' = 4 \), to be decreased by 2 and to change to 2. Since this value still is positive, no backtracking occurs and the process continues. It is in row 6 that three units of improvement in the heuristic estimate of state 9 causes the current value of \( p' \), 2, to be decreased by 3 and to change to a negative value of -1. On such an occasion because of the negativity of \( p' \) LCBA* backtracks to the state preceding state 6 which for the first time has caused the original value of \( p' \) to be decreased. It then chooses to continue the search process from state 7 in row 2 with \( p' \) equal to 5. Notice that based on theorem 2, LCBA* could have backtracked to any other state on the list rather than state 7. Because of this flexibility let's consider the question of whether there exists any other promising state on the path to which backtracking could have occurred. State 7 to which LCBA* backtracked has the heuristic estimate of 35 with 5 units of remaining threshold, whereas at state 8, the heuristic estimate is 10 and 4 units of threshold are left. If these two states, 7 and 8, are compared, it can be claimed that: 4 units of threshold with the estimate of 10 units of distance left to reach the destination is better than 5 units of threshold with a distance of 35.

To clarify this claim, we use a simple analogy. The remaining threshold value, \( p' \), associated with every state on the OPEN list can be considered as the amount of petrol left to reach a destination, the goal state; and the heuristic value of each state is its approximate distance to this destination. By this analogy when comparing state 7 with 35 units of distance and 5 units of petrol with the state of 4 that has 10 units of distance and 4 units of petrol, it would be better to backtrack to state 4 rather than state 7. VER_1, in contrast to LCBA*, backtracks to the state with the maximum value of \( p'/(h+\epsilon) \) on the OPEN list. In this relation \( p' \) is the remaining threshold associated with a state on the list, \( h \) is the heuristic estimate of the state and \( \epsilon \) is a small value preventing a division by zero error in the case that \( h \) is zero. In terms of
our analogy, a state will be selected from the list where the amount of petrol, \( p' \), per approximate distance to the destination, \( h \), is maximised, instead of maximising just the amount of petrol without any consideration of the distance. In the case where we encounter different states with equal heuristic estimates the tie is broken randomly. Keeping track of such promising states is an advantage because once the trace to such states is lost, it can be very costly to identify them again. Consequently, VER_1 works, in detail, as follows.

**Step 0:** Apply a heuristic function to generate a non-overestimating initial heuristic estimate \( h(x) \) for the distance from every state \( x \) to the goal state. Set \( p' = p \) and \( b = \) initial state. The value of \( p \), as the total threshold, is the maximum amount by which the ultimate solution may vary from the optimal solution and \( b \) is the state to which the algorithm backtracks. The value of \( p' \) is the remaining threshold.

**Step 1:** Put the initial state on the OPEN list and keep \( p' \) associated with it on the list.

**Step 2:** Call the top-most state on the OPEN list \( x \) and its associated value \( p' \). If \( x \) is the goal state stop.

**Step 3:** Evaluate the compound value \( k(x,y)+h(y) \) for every neighbouring state \( y \) of \( x \) and find the state with the minimum value. Call this state \( x' \). Break ties arbitrarily. The value \( k(x,y) \) represents the positive edge cost from state \( x \) to state \( y \) and \( h(y) \) represents the heuristic estimate of state \( y \).

**Step 4:** Let \( L = k(x,x')+h(x')-h(x) \). If \( L \leq 0 \) and \( x' \) is not on the OPEN list then put it on the list and keep \( p' \) associated with it on the list and go to step 2.

**Step 5:** Replace \( h(x) \) with \( k(x,x')+h(x') \). If \( x' \) is already on the OPEN list then backtrack to it, change \( p' \) to the value associated with this \( x' \) and go to step 2.

**Step 6:** If \( L \leq p' \) then decrease \( p' \) by \( L \) and put \( x' \) on the list and keep \( p' \) associated with it and go to step 2.

**Step 7:** If \( x \) is not the initial state then amongst all states that precede \( x \) on the OPEN list find the state with the maximum value of \( p'/(h+\epsilon) \) and call it \( b \) noting that \( \epsilon \) is a small number used to prevent division by zero.

**Step 8:** If \( x \) is not the initial state then backtrack to \( b \).
Step 9: Go to step 2.

VER_1 is similar to LCBA* and the only difference is the way in which the state which is backtrackte to is selected. In LCBA*, this state was the first state on the OPEN list which has decreased the original value of \( p' \) and here based on step 7, this state is selected from OPEN list members to achieve a maximum value of \( p'/(h+\epsilon) \).

We have applied VER_1 to the same three problems shown in figures 2.8, 2.12 and 2.15, and found that VER_1 is better with the second problem and LCBA* is better with the other two. The application of VER_1 to those three problems led to different rates of exchange between speed and accuracy. VER_1, similar to LCBA*, is a general graph-search method. Its generality allows it to be applied to any graph-search problem; however, as with LCBA* the exchange rate between speed and accuracy depends on the structure of a problem. Sometimes by sacrificing a small amount of accuracy it provides a very fast solution and sometimes it is necessary to sacrifice a lot of accuracy to obtain high speed. There is no way to predict this exchange rate for a given problem except by applying the algorithm. Apparently, LCBA* and VER_1 have no real advantages over each other and although they operate differently they are based on the same theorem 2. However, serial application of them to the same problem can decrease the initial bracket.

Serial application means applying them separately to the same problem and then selecting the best result that has been produced. For instance suppose that LCBA* and VER_1 both with an input bracket of 3 have been applied to the same problem and have produced the results of 8 and 7, respectively. By selecting 7 as the minimum value among these results, one can guarantee that the answer is in a bracket of 2 rather than in the original bracket of 3. The reason is that since the solution of 8 is in the original bracket of 3 then the solution of 7 will be in the bracket of \( 3-(8-7) \). This can be formulated as a simple rule that: if both of these methods with the same input bracket of \( p \) are applied separately to the same problem and provide the results \( R_{LCBA*} \) and \( R_{VER_1} \), the final solution for that problem can be considered as,

\[
\text{Min} \{ R_{LCBA*}, R_{VER_1} \},
\]
with the new bracket of,

\[ p = \text{Max}\{R_{\text{LCBA}^*}, R_{\text{VER}_1}\} + \text{Min}\{R_{\text{LCBA}^*}, R_{\text{VER}_1}\}. \]

In other words, the original bracket, \( p \), is updated by the difference between the two results. Therefore, in this way, \( \text{LCBA}^* \) and \( \text{VER}_1 \) can complement each other to produce more precise solutions, with smaller brackets.

\( \text{VER}_2 \)

The precision of solutions to optimisation problems is dependent on computer resources. In some cases an algorithm is required to perform within only a few seconds and in other cases several minutes or even hours may be available. The same is true with memory requirements because in some cases the program is supposed to be run on a computer with only 64K space and in other cases several Megs of space are available. We have observed that in using \( \text{LCBA}^* \) and \( \text{VER}_1 \), the two factors of overflowing storage and running out of time before prescribed accuracy had been reached prevented any solution from being found and this was resolved by increasing the bracket before running the program once again. However, suppose that one is required to find a solution in a certain time with a specified computer. \( \text{VER}_2 \) is a method based on pre-specified computer resources (CPU time and storage) which finds guaranteed accuracy solutions for combinatorial optimisation problems. As mentioned, increasing the threshold value causes precision to be traded with speed. Therefore, in a situation involving time and memory constraints, necessary speed can be obtained by increasing the original threshold value. This method accepts, as input, three parameters including CPU time allowed, the storage available on the machine on which the program is supposed to run, and the initial threshold value. \( \text{VER}_2 \) increases the original threshold at the backtracking times if the progress toward the goal is not satisfactory. The question arising is how, at a certain time, \( \text{VER}_2 \) determines whether the progress toward the goal with available resources is satisfactory or not. In order to answer this question, we consider an example.

Suppose that the maximum time specified for a problem to be solved is 100 time-units and that the storage of the machine on which the program is being run is 1000
storage units. Also, suppose that 20 units of time have passed and the storage left at this time is 500 units. How can one determine that, up to this point, whether the search process has made satisfactory progress or not? Up to this point of time, twenty percent of available time, and fifty percent of available storage have been used. Since running out of either memory or CPU time can prevent the program from running, the maximum of twenty and fifty will be selected as the percentage of resources which have been used. When fifty percent of resources have been used, based on the assumption of a linear relation, fifty percent of the distance toward the goal should have been traversed. Since heuristic values are estimates for these distances then dividing the difference between the heuristic estimate of the current state and that of the initial state by the heuristic estimate of the initial state is a good measure of this percentage of distance. Suppose that the heuristic estimate of the initial state is 40 and the heuristic estimate of the current state is 10. It is obvious that in this case there is no need to increase the threshold value to gain speed. The reason is that roughly \( \frac{40-10}{40} \), seventy-five percent, of distance is traversed with fifty percent of the resources left. However if the heuristic estimate of the current state was 30, it would mean that the progress was not satisfactory and only twenty-five percent of the distance had been traversed, implying that we should increase the specified threshold and trade precision for speed. The following algorithm shows how this method operates.

**Step 0:** Input \( m \), \( t \) and \( p \). The value of \( m \) is the amount of memory available, \( t \) is the CPU time allowed and \( p \) is the initial total threshold value.

**Step 1:** Set the current time, \( t_0 \), to zero and begin to run LCBA* and supervise its running as follows.

**Step 2:** Continue running until either a backtracking or stopping condition occurs. In this case pause running and let \( m_0 \) = the amount of memory left and \( t_0 \) = the current time.

**Step 3:** Let \( r = \max \{ \frac{m_0}{m}, \frac{t_0}{t} \} \).

**Step 4:** Let \( r' = \frac{[ h(\text{initial state}) - h(\text{current state})]}{h(\text{initial state})} \),
Step 5: If $r' < r$ then increase the maximum threshold value, $p'$, by one unit.

Step 6: If LCBA* has been paused because of backtracking, go to step 2 and resume running

Step 7: If a solution is found and still some resources are left, save this solution and after updating $m$ to the amount of the memory left and updating $t$ to the time left and go to step 1

Step 8: If there are no resources left, represent the best solution by $b$ and the worst by $w$. In the case that just one solution has been found, $b$ and $w$ will be equal.

Step 9: Update the original threshold value, $p$, by subtracting $(w-b)$ from it.

Step 10: Print the solution, if any, along with the updated threshold value and stop.

There is a subtle point in steps 8 and 9 of this algorithm that needs to be clarified. Sometimes VER_2 without any increase in the original threshold finds a solution for the problem. In these cases if enough resources are left, steps 8 and 9 are designed to decrease this original threshold by finding other guaranteed solutions. To show this, we consider an example where the maximum threshold is five and three solutions with the values of 40, 41 and 43 have been found. This results in the solution with the value of 40 being selected and the original threshold parameter being decreased (improved) by three units which is the difference between the worst and the best solutions found. Therefore, VER_2 not only increases the original threshold when confronted with limited computer resources but also decreases this parameter when sufficient resources are available. This characteristic was tested as follows.

By introducing 64K as maximum memory available and allowing 1 second as CPU time, the application of VER_2 was tested on each of the three different problems represented in figures 2.8, 2.12, and 2.15 and in each case it led to decreasing the original input thresholds to zero and accordingly finding their optimal solutions. The result was predictable since the number of backtracks involved in finding the optimal solutions in those problems was small. Thus in a very small fraction of time the solution was found and it was optimal. Whenever the brackets were increased, again because of steps 7 and 8, the optimal solutions were found.
This method can be very useful in dealing with search problems that must be solved within a certain amount of time and with a certain amount of available computer memory.
3. APPLICATIONS OF LCBA* TO PROJECT SCHEDULING PROBLEMS UNDER MULTIPLE RESOURCE CONSTRAINTS

3.1 Introduction

In this chapter, we consider the application of LCBA* to the most general problem of scheduling. Scheduling problems arise in many practical circumstances and under a wide variety of conditions. Many are basically optimisation problems having the following form: given a collection of tasks to be scheduled on a particular processing system, subject to various constraints, find a feasible schedule that optimises the value of a given objective function (Garey et al. 1978).

Baker (1974) defines scheduling as the allocation of resources over time to perform a collection of tasks, beginning after the planning phase in which fundamental planning problems are resolved. The planner first identifies the collection of tasks to be carried out and sets limits on the amount of resources available and then the scheduler takes this information and determines how to allocate the available resources to perform the required specific tasks. Planning decisions represent long range commitments whereas scheduling decisions are made originally by considering the restrictions set by these commitments.

This general definition conveys two different meanings. Firstly, scheduling is a decision making function pertinent to the process of determining a schedule. In this sense, many of the practical methods of decision making such as LCBA* can be applied to this field. Secondly, scheduling is a body of theory, being a collection of principles, techniques and logical conclusions that provide insight into the scheduling function. In this sense by successfully applying LCBA* to the most general problem of scheduling, it can be added to this valuable collection of techniques and logical conclusions, providing a useful framework for efficiently performing the scheduling function.

Many scheduling problems, including the one to which we apply LCBA*, are NP, and hence the power of LCBA* in dealing with this large group of problems is clearly demonstrated. Indeed, the scheduling field has become a focal point for the
evaluation of combinatorial procedures and consists of a wide collection of these procedures.

While informal scheduling methods have been used for centuries and are still in use in most daily activities, formal scheduling models appear to have originated during World War I with the advent of the Gantt model. These models were developed to handle the scheduling problems associated with loading cargo onto allied ships. Their use reduced the ship turn-around time by about half.

Today's critical path methods, as successors to Gantt charts, continue to be amongst the most widely used scheduling tools. Formal mathematical models of scheduling problems started to appear in the literature in the mid fifties, some forty years after Gantt's work. Since that time the interest in these problems has increased significantly.

Numerous articles that have appeared in operations research, industrial engineering and recently artificial intelligence journals are evidence of that interest. In the scheduling area, there are many specialised techniques that work particularly well for a given problem but are not necessarily transformable and applicable to other problems. The literature is full of such techniques.

The reason for this inapplicability stems from two facts: (1) due to the complexity of problems, practical algorithms do not exist for finding optimal solutions and this is why approximate algorithms are used where in some cases these algorithms, which may find a solution for a particular problem in milliseconds can take a very long time to find the solution for another problem; (2) some problems in the scheduling area are special cases of other problems in this area and hence solutions to particular problems are not always applicable to more general problems. For these reasons we have applied LCBA* to the most general problem in scheduling and we now discuss the formulation of this general problem.

Scheduling begins with the translation of decision-making goals into an explicit objective function and decision-making restrictions into explicit mathematical constraints. These constraints describe the system, including the kinds of tasks and
conditions of their processing, the type of processors and their number, and all other properties necessary to specify feasible schedules. The objective function assigns a "value" to each feasible schedule and can consist of all costs in the system that depend on scheduling decisions. However, as Baker (1974) states in practice such costs are often difficult to measure.

Three types of decision-making goals are used in scheduling: efficient utilisation of resources, rapid responses to demands and close conformance to prescribed deadlines. Two kinds of feasibility constraints are commonly found in these problems. First, there are limits on the capacity of available resources; and second, there are technological restrictions on the order in which tasks can be performed. A solution to a scheduling problem is any feasible resolution of these two types of constraints.

Resource constraint project scheduling is a scheduling problem that contains both resource and technological constraints. Having two groups of constraints gives a generality to this problem in the scheduling area that can encompass many other problems in this field. As stated by Slowinski, Roman and Weglarz (1990), a resource-constrained project scheduling problem subsumes job shop, flow shop, assembly line balancing and related problems which cover a wide range of scheduling problems. Baker (1974) states that the difficulties present in those simpler problems are superimposed in resource-constraint project scheduling and any method capable of tackling this problem will potentially be able to tackle those simpler problems.

The Resource Constrained Project Scheduling (RCPS) problem may be stated as follows: A finite set of jobs are given, each requiring a fixed integer duration and a fixed amount of one or more different resource types, and jobs are subject to a set of precedence relations which specify permissible job orderings. Jobs may not be interrupted once started, and there are specified fixed limits on the availability of each resource type. The objective is to minimise the project duration.

This problem occurs not only within industrial organisations but also in variety of applications such as computer programs competing for processor resources, patients
competing for services of medical facilities, the optimisation of multi product chemical processes, radioisotope production in nuclear reactors and tyre testing policies (Norbis and Smith 1986).

3.2 Historical Approaches

Because of the generality of the RCPS problem and its many and varied applications, a great deal of research has focused on it and many procedures have been proposed to solve this problem. These procedures can be categorised into two major groups on the basis of their distinctly different approaches.

The first category, and by far the smallest, includes procedures designed to produce the optimal schedule using approaches mainly based on linear programming and implicit enumeration. The second category of heuristic approaches, in contrast to the first, is very large and includes many procedures which are designed to produce good rather than optimal solutions. A tremendous amount of effort has gone into the investigation and creation of elaborate heuristic-based scheduling models and these procedures enjoy a very prominent role in solving RCPS problems.

We begin by examining exact (optimal) procedures and then we turn our attention to heuristic approaches. Next we consider the relationships between the characteristics of the RCPS problem and approaches to its solution involving characteristics such as: size (number of activities), general appearance (short or fat, long and thin), tightness of constraints, duration of the project, the number of critical paths, and other relevant factors which potentially determine when and how well a solution method will work. Despite the wealth of research on the RCPS problem we note that little attention has been paid to considering the relationships between these characteristics of the problem and the choice of an appropriate solution approach.

3.2.1 Exact methods

The development of exact methods or optimal procedures has progressed very slowly. This stems from the fact that the formulation of the RCPS problem from a mathematical point of view is difficult and involves many binary integer variables. Restrictions on the sequence in which jobs may be performed interacts with resource
requirements and availabilities thereby creating an uncountable solution set. The following claim by Kelly (1963) after three decades is still correct.

"...In view of the difficulties involved, there does not appear to be any direct (i.e. exact) approach for formulating and solving this problem."

It was not only Kelley who reached such a pessimistic conclusion. The Du Pont company who devised CPM, also, stated that (Davis 73):

"Linear programming, Dynamic programming, self-correcting and combinatorial approaches were investigated and discarded because of the complexity of the problem."

These statements about exact approaches were made very soon after the development of PERT and CPM. However, after three decades, this pessimism is still predominant among researchers in this area. For instance, Norbis and Smith (1986) state that optimal procedures for this problem have proved to be unsuccessful in dealing with problems of practical size because of their NP-completeness. Ulusoy and Ozdama (1989) even go further and predict that:

"The optimisation techniques remain computationally impractical for most real-world problems since even modest-sized projects have an enormous number of possible schedules."

However, in spite of all these pessimistic conclusions, these procedures, under special circumstances can solve problems with tens of activities. We will describe briefly the known exact methods in the two groups linear programming and enumerative techniques.
3.2.1.1 Linear programming

The first linear program for RCPS was presented by Wiest (1963). His approach was an adaptation of Bowman's formulation of the job-shop problem (Bowman 1959). Bowman's formulation used 0-1 variables to indicate for each period over a scheduling horizon whether or not a job is being processed. This formulation does not explicitly extend to solving multiresource constrained problems, although such an extension could be made. No attempt was made to implement this formulation; Wiest merely presented it and pointed out that this approach was infeasible for large projects. He noted that for a project with 55 jobs and 4 resource types, it would require some 6870 constraint equations and 1650 binary variables. He concluded that:

"The use of linear programming and a 7090 computer for such problems would be somewhat akin to using a bulldozer to move a pebble."

In spite of this negative conclusion by Wiest, and some of his predecessors, other researchers continued to pursue the possible use of linear programming procedures. Gorenstien (1972) treated the problem as a disjunctive graph. The perspective advantage of this approach is the elimination of the need to consider individual time periods over the program horizon and a feasibility check determines whether the resource constraints can be met by any particular network representation of the project. He developed an algorithm employing a maximum-flow computation as a check for feasibility with respect to available resources. The first implemented integer programming formulation of the RCPS problem was proposed by Pristker, Alan and Watters (1969) and for the first time the application of linear programming to projects involving up to 8 jobs and 3 resource types was demonstrated. This formulation was expanded by Baker (1974) for the case that an activity may simultaneously require several units of more than one resource. The result is fairly general and reasonably efficient in its use of decision variables. An example presented in the original paper
with 8 activities and 3 limited resources required only a few seconds to be solved. However a small increase, in the size of the problem, made it unmanageable.

Talbot and Patterson (1978) developed an efficient integer programming algorithm based on systematic enumeration of "all possible job finish times" for each activity of the project. We classify this algorithm as an enumeration technique and will discuss it later. Christofides, Nicos, Valdes and Taramit (1987) present a linear program for the RCPS problem which is capable of solving some problems with up to 25 jobs in acceptable computer time. However they state that even for problems with up to 15 activities which are tightly resource constrained this formulation is impractical and too expensive to be used. Therefore the structure of a problem has a significant influence on the efficiency of a procedure.

Deckro, Wincogsky and Herbert (1991) developed the use of a decomposition algorithm in solving the linear programming approach to RCPS problems. To carry out this decomposition, a project is broken to several sub-projects. Sometimes, these sub-projects are easily identified and the whole project is then referred to as a multiproject. Decro et al. (1991) state that the decomposition approach offers two distinct advantages over a direct optimisation approach: (1) the ability to solve realistically large problems, and (2) the option of using the decomposition approach as a heuristic. By decomposing a project into sub-projects, the generated sub-problems are characterised by nearly all of their constraints being special ordered sets where exactly one variable must be non-zero in each constraint. The bulk of the remaining constraints are located in the master problem and serve as coupling constraints in the decomposition procedure. This approach provides a performance measure on all feasible solutions, allowing the user to decide if further calculation expense is warranted in view of the absolute potential gain in the incumbent solution value. They highlight two major advantages of this approach by an example. First, an overall problem of 880 zero-one variables was analysed by solving a series of smaller problems. The largest subproblem contained 160 variables, while the largest master problem contained 110 variables when computations were completed. Second, once a
feasible solution is stabilised for the master problem, an absolute range can be determined for the value of the incumbent solution. An incumbent solution is a solution which up to a certain point in the process of solving the problem is the best solution. This gives the user an opportunity to terminate the computation if it is decided that the potential gains do not justify further computational expense. Given the imprecision in time estimates present in most actual operational settings, coupled with the normally burdensome amount of time required by integer programs to establish optimality, this stopping feature offers analysts a valid alternative to grinding out an optimal solution. This procedure is not suitable for the general RCPS problem and is of use only in situations where a project can be decomposed to at least two other sub-projects. This research indicates how the development of specialised algorithms for particular problems can take advantage of the special structure of that problem, thus offering a promising effective solution procedure. The efficiency of this method depends on the number of sub-projects into which the main problem is decomposed.

The above analytical techniques are useful in formulating the problem in a rigorous manner, they also supply theoretical insight into the development of other approaches and provide optimal solutions for small problems. However, their computational requirements for even moderate sized problems are prohibitive.

3.2.1.2 Enumerative Techniques

Allocation of scarce resources to the individual activities in a project to achieve minimum completion time involves resolving conflicts whenever the concurrent demand for resources by the competing activities of a project exceeds resource availabilities. Whenever these resource-conflict-resolution decisions arise, a schedule is sought such that the resulting increase in the project is the minimum that can be achieved with the given resource availability. Enumeration techniques systematically search the set of possible conflict resolving decisions in such a way that not all possibilities need be considered individually. Enumeration techniques may differ in the manner in which they address two important factors. The first factor is the manner
in which a tree representing partial schedules is generated and saved; and the second factor is the means used to identify and discard inferior partial solutions. Various authors refer to this method as bounded enumeration, branch and bound, and implicit enumeration.

The term "enumeration" is used to describe a general technique for solving sequencing and other combinatorial problems. As implied by the name, the technique involves enumerating and searching the potential set of solutions for a given problem. Terms like "bounded", "branch and bound" and "implicit" emphasise the fact that the enumerative procedure involves enumerating over only a portion of that total set of solutions.

The rules defining the search procedure are developed in such a way that only those solutions which are improvements over an existing solution are considered and that the final solution is optimal. This approach had not been applied to the RCPS problem until 1967. In February of that year, Merbach (1967) presented an enumeration technique for RCPS which was reported at the first INTERNET conference. Nearly six months after devising the first enumeration technique for RCPS, a second technique was developed by Johnson (1969). Although these two schemes were different in detail, they were both limited to one type of resource per job. Davis (1969) devised an enumerative approach for solving RCPS which unlike the other two, could handle problems involving several resource types per job called "multi-resource" problems. To test the feasibility of the conceptual approach involved, a computer program ("MARK1") was written for the IBM 7094 and Davis reported that:

"Comparisons showed a remarkable advantage for the MARK1 program. Problems which required as long as 12 minutes via linear programming were solved in 2 seconds with the MARK1 program".
One might expect that since this procedure was about 360 times faster than integer programs then it could handle RCPS problems with hundreds of jobs. However as with all other combinatorial problems, an increase in the speed of an algorithm is of little significance compared to the size of the typical problems to be handled. A series of 65 artificially-created test problems that were solved with MARK1 were limited to between 20 and 30 original jobs, with each job requiring various amounts of three different resource types. Davis reported that:

"The initial tests of procedure indicated that computer memory storage, and not computation time, would be the primary operating constraint on the IBM 7094. That is, in attempting to solve problems of only 20 jobs, the amount of available storage capacity was sometimes exceeded before a final solution was obtained. The procedure as developed in its current form is obviously not suitable for computing optimal solutions to realistic sized networks. It could, however, be applied to portions of larger networks."

Work on enumeration techniques for the RCPS problem eagerly continued. Patterson (1984) as a pioneer and authority on these techniques considered three of them typical and superior to others, namely the bounded enumeration procedure presented by Davis and Heidorn (1971), the implicit enumeration procedure presented by Talbot (1978) and the branch and bound procedure presented by Stinson (1978). We now discuss each of these three procedures.

Stinson's Branch and Bound Procedure.

This solution procedure consists of creating nodes in a tree which in the RCPS problem characterise certain "partial schedules". These partial schedules represent scheduling decisions for some subset of the total number of activities in the network. The partial schedules are always feasible with respect to both precedence and resource constraints. The procedure develops the tree from some given partial
schedule by branching to a new family of partial schedules. Each member of the new family has in common with the others all scheduling decisions made previously by a common ancestral node and is unique from the others in that it includes one new decision about the scheduling of one or more activities previously unscheduled. In each branching operation, the procedure creates only as many new partial schedules as there are feasible combinations of activities that may enter the schedule at some point in time, including in some cases scheduling the null set. This branch and bound tree offers a framework upon which all possible feasible schedules can be evaluated, thereby determining the best schedule.

Without pruning away portions of this tree, this is quite impractical even for modest sized problems since the number of complete schedules rapidly becomes very large. Therefore, it is necessary whenever possible to prune away portions of the tree, preferably in its earlier stages of the development. The pruning of a node is justified if it can be established that further branching cannot lead to a better complete schedule than other complete schedules that are either already known to exist or could be developed in further branching from some other nodes. In this procedure, pruning operations are done in two ways: dominance pruning and lowerbound pruning.

Dominance pruning is introduced into this branch and bound algorithm in two ways. The first involves a comparison process. It is said that partial schedule $X$ dominates partial schedule $Y$ if all the following four conditions are met:

- the unscheduled activities in $X$ are a subset of those in $Y$,
- the set of activities currently in progress (active set) in $X$ are a subset of those in $Y$,
- the project completion time of each activity in the active set of $X$ is equal to or less than that of the same activity in the active set of $Y$,
- the current partial schedule time of $X$ is equal to or less than that of $Y$.

The second class of dominance pruning is much more efficient to implement. It states that if there is any activity already assigned a start time in some partial schedule which can be left-shifted to an earlier start time without violating either a precedence or resource constraint then the partial schedule is dominated. This dominance
condition was established by Scharge (1969) nine years before the development of this procedure. These two pruning operations are both classified as dominance pruning.

Lowerbound pruning involves three pruning operations and their associated lowerbounds: "precedence based", "resource-based" and "critical sequence". In "precedence based" lowerbound, the resource constraints are ignored and a critical path for the unscheduled activities of a partial solution is computed. The completion time of this path constitutes a lowerbound on the completion time of any partial schedule branched from the partial schedule. If this lowerbound completion time is not less than the time of completion for some other known complete schedule (which is called the current upperbound) it will be pruned away.

On the contrary, in "resource-based" lowerbound, the precedence-constraints are ignored. In effect, there are some number of man-days requirements for unscheduled activities which when divided by the number of men available per day will yield a minimum number of days for which the schedule must actively continue. This idea is easily generalised to multi-resource constraints by finding the maximum value among different bounds based on the consideration of individual single resource constraints.

"Critical sequence" lowerbound simultaneously takes into account both precedence and resource constraints. At first a critical path for the unscheduled activities in the network is computed and resources are given to activities on the critical path. Each activity not on the critical path is tested to see by how much we can delay this critical completion time. Operationally, all these three lowerbounds are computed and the largest of the three is taken to be the lowerbound for a partial solution.

Since a skiptracking scheme builds the tree uniformly downward and therefore does not concentrate branching activities on the terminal areas of trees, the generation of upperbounds is not automatic as in the case of backtracking. For this reason, some criteria in this procedure are introduced so that a potentially good partial schedule is recognised as the tree develops. This good partial schedule then is set as a point of departure from which we can proceed directly downward in hope of finding an
improved upperbound solution. This is accomplished by selecting a new search origin when the first node is selected which has a lowerbound greater than that of any node previously branched from. With this as new search origin, the procedure proceeds directly downward to a new upperbound solution. In this procedure like any other branch and bound procedure, a node is selected for branching based on a special priority. This selection is important, because to some degree, pruning is enhanced through the rules of selection of nodes. Based on the node selection rule, the branch and bound tree can be developed in quite contrasting fashions.

One selection rule, for example, might be to branch next from the last node created in the immediately preceding branching operation. Using this rule the tree tends to grow within restricted areas. Before moving to other areas of the tree all the areas previously explored will have been pruned away except for one complete schedule. The ease of structuring partial schedules by using pointer links, and memory efficiency are the main advantages of this method.

However, there are two primary disadvantages: computation times tend to be longer and the use of dominance rules, which rely on comparisons of nodes held in storage, is not possible. The resulting priorities for these node selection rules are: select the node having the largest number of activities with assigned start times, select the node with highest partial schedule time, or select the node with highest lowerbound. In contrast to backtracking, this form of tree development is known as skiptracking. With skiptracking, the tree tends to grow in breadth and moves downward in a more or less uniform fashion. In this scheme the additional power of dominance pruning through comparison of open nodes is possible.

Two node selection rules which result in skiptracking are: select the node having the least lowerbound, or select the node which has the least total accumulated resource idleness up to the current partial schedule time. All priorities selected in this procedure result in skiptracking which is in contrast to backtracking. Since the use of single node selection for large scale problems like RCPS, particularly when
skiptracking is employed, is of limited effectiveness, in this procedure vector node selection rules have been considered.

The reason that a single node selection rule is of limited effectiveness is that there could exist within the tree hundreds of nodes all with identical lowerbounds. In effect, a decision vector is a series of tie-breaking rules for selecting the next node from which to branch. If no ties exist for the first attribute in the vector, of course, the other attributes are ignored.

Vector node selection or "decision vector" can enhance performance in two ways: first, dominance by comparison is improved because potentially better partial schedules are developed earlier; and second, lowerbound pruning is improved if the decision vector can lead to optimal or near optimal upperbound solutions as early as possible.

Six different vectors are considered, each comprising of four different priorities. The first selector element in four of these six vectors is the lowerbound priority (LB\(_n\)) obtained as the largest of the three individual lowerbounds previously discussed. In two of these vectors, this priority is not the first selector and it has been replaced by the total accumulated resource idleness priority as the first selector. Other selectors have been chosen from the following eight priorities:

1. Critical sequence lowerbound (LBC\(_n\))
2. Precedence based lowerbound (LBP\(_n\))
3. Resource based lowerbound (LBR\(_n\))
4. Current partial schedule time (t\(_n\))
5. Number of activities with assigned start time (NS\(_n\))
6. Number of activities with assigned finish time (NFn)
7. Total number of immediate followers of activities in the active set (TF\(_n\))
8. Total accumulated idleness (In)

This procedure has been programmed and among different possible decision vectors, the one represented as (LB\(_n\), t\(_n\), ln, NFn) has performed almost uniformly better than all other vectors. All problems were generated randomly and ranged in size from 23 to
43 activities with the number of resource classes ranging from 1 to 6. Fifty-seven out of 60 problems with 35 activities were solved in reasonable time. However, essentially unpredictable variance in the computation time from problem to problem was observed even for problems with the same general size and complexity. For 60 problems with 35 activities this variance has been reported as 19.17 minutes (Stinson 1978).

*Davis and Heidorn's Bounded Enumeration Procedure.*

This procedure uses techniques originally developed for solving the assembly line balancing problem. Assembly line balancing is the process of assigning work elements to assembly stations in such a manner that each station has an equivalent amount of work. The goal is to minimise the number of stations to complete the required quantity of assemblies in a given period of time. In this approach the formulation of the problem for computational solution is facilitated by representing each original job of the project as a series of unit-duration tasks. To observe the constraint that jobs once begun may not be interrupted before completion, it is specified that each task in a series representing the original job must immediately follow its predecessor, if one exists. The precedence constraint is represented by <, whereas immediate precedence is represented by ≤. Therefore T1<T2 means that in order to begin activity T2 the activity T1 must be completed, whereas T1≤T2 means that after completing activity T1 the activity T2 will begin. To describe the model, the following definitions are required.

- \( T_j \) represents task \( j \), \( j=1,2,3,... \)
- \( A \) is the set of all tasks in the project ( \( T_1, T_2, \ldots \) ),
- \( Z_i \) is the set of tasks assigned on day \( i \),
- \( N \) is project duration,
- \( r'(T_j) = (r''_1j, r''_2j, \ldots, r''_mj) \), where \( r''_{kj} \) is defined as the number of units of resource \( k \) required by \( T_j \), \( k=1,2,\ldots,m \).
Let the resource requirements associated with each subset of tasks $Z_i$ be denoted by the vector,

$$R_d = \sum_{T_j \in d} r'(T_j) = (r_{1i}, r_{2i}, \ldots, r_{mi}),$$

where $r_{kj}$ denotes the units of resource $k$ required for the completion of all tasks in $Z_i$. and,

$$r_{ki} = \sum_{T_j \in Z_i} r''_{kj}.$$

Also let the specified amount of each resource available be denoted by the resource limits vector:

$$R_L = (l_1, l_2, \ldots, l_m),$$

where $l_k$ = number of units of resource type $k$ available on each day of the project duration. Now using the notation above the model can be represented as follows.

\[
\begin{align*}
\text{Minimize} \quad & N \\
\text{Subject to} & \\
\bigcup_{i=1}^{n} Z_i = A, \\
Z_i \cap Z_j = \emptyset, & \text{for all } i \neq j, \\
R_d \leq R_u, \\
\text{If } T_w < T_q & \text{ and } T_w \in Z_i, T_q \in Z_j \text{ then } i \leq j, \\
\text{If } T_w \in Z_i & \text{ and } T_w \leq T_q \text{ then } T_q \in Z_{i+1}.
\end{align*}
\]

This formulation is the project scheduling counterpart of the assembly line balancing model.

In this procedure, the above formulation is transformed into a problem of finding the shortest route between two specified nodes of a finite directed network. In this network called an A-network, nodes represent a subset of tasks, and arcs connect subsets which can be completed on adjacent days. The minimisation of the project duration is accomplished by finding a path from start to finish in the A-network which contains a minimum number of arcs. To discuss the operation of this
procedure, the term "feasible subset" needs to be defined. A subset of A, $S_j=(T_1, T_2, ..., T_m)$, is said to be feasible if $T_j \in S_j$ and $T_i < T_j$ imply that $T_i \in S_j$. Thus, a feasible subset is one where if it contains a given task, then it also contains all predecessors of that task. Now, let $S_j, j = 0, 1, ..., r$, be the entire collection of feasible subsets, with $S_0 = \emptyset$, the empty set, and $S_r = A$, the set of all tasks in the project. An immediate follower of a feasible subset $S_j$ is defined as a task which is an immediate successor of at least one of the tasks in $S_j$ and is not preceded by any task not in $S_j$.

By using this immediate follower and beginning with the empty subset, the feasible subsets are generated. This method of generating feasible subsets has the properties that every feasible subset is generated and that no feasible subsets are duplicated.

Two properties of this procedure are:

1. a task appears for first time in feasible subsets generated in stage $K$, where $K$ is the task early start time (EST),

2. the number of stages at which nodes are added to existing feasible subsets is equal to the number of time periods in the longest path through the task precedence network (original critical path length).

Starting with the empty set, $S_0$, directed arcs are drawn between nodes if precedence and resource constraints are satisfied, until the final node $S_r$ is reached. No arcs enter $S_0$ and none leave $S_r$; and the result is a directed network from $S_0$ to $S_r$ on the set of all feasible subsets. A directed arc $S_i S_j$ is created from the feasible subset $S_i$ to subset $S_j$ if and only if the three conditions given below are satisfied.

First, in order to observe the precedence constraints of the original problem, it is required that $S_i$ is a proper subset of $S_j$ and all predecessor tasks of a given task in the subset of $S_j$ are contained in the subset $S_i$.

Second, to observe the constraint that jobs in the original network be performed continuously once started, additional "must follow" constraints are imposed on the task sequence (i.e. if $T_s \in S_j$ and $T_r \leq T_s$ then $T_r \in S_i$)
Finally, it is required that both $R_{Sj}$ and $R_{Si} \leq R_L$, where $R_L$ is the resource limits vector previously defined. In the original paper, Davis and Hiedorn (1971), this condition has been stated as $R_{Sj} - R_{Si} \leq R_L$ which is an error.

Arcs constructed by observing the above criteria represent the assignment of tasks on particular schedule days. That is, the $r^{th}$ arc $S_jS_j$ in a path from $\emptyset$ to $A$, in the A-network, corresponds to the assignment of tasks $(S_j - S_j)$ on the $r^{th}$ day of the original problem. From the restrictions set on the arcs in the A-network, it is clear that arcs can exit only between feasible subsets in the same and adjacent stages.

Two different criteria namely "Precedence-based elimination" and "resource based elimination" are used to improve the procedure by reducing the total number of feasible subsets generated, and still producing an optimal solution if one exits. The basic idea for this reduction is to select a "target" duration ($n'$) for the project and then, during the performance of the algorithm, discard those feasible subsets which (a) could not possibly appear on a path through the network of length $n'$ or less, and (b) could not guarantee any other feasible subsets which would do so.

Given the target duration, two independent sets of criteria based on precedence and resource constraints are determined and then applied to each feasible subset when it is generated. This procedure has been programmed for the IBM 7094 computer and was designed to handle projects involving no more than 25 jobs and a maximum of 105 unit-duration tasks with 3 different resource types per job. The main operating constraint of the procedure is the number of feasible subsets which may be stored at each stage. Therefore, if at some stage the allowable number of feasible subsets is exceeded, an optimal solution cannot be guaranteed. When this occurs, the program selects the most promising feasible subsets from the previous stage and then continues.

For testing this procedure, 65 artificially created problems have been attempted each containing 30 original jobs with no more than 90 unit-duration tasks and involving 3 different resource types per job. Optimal solutions were found for 48 of them, and for the remaining 17 an approximate solution was found. Notice that in this procedure
every job is divided into unit-duration tasks and that is why a constraint is set on the number of these unit-duration tasks. Therefore, if the duration of one activity increases then the number of unit-duration tasks and subsequently the number of feasible subsets in its corresponding A-network will increase. This will cause catastrophic results in projects with long duration tasks, and that is why the test problems were selected so that the total duration of jobs do not exceed 90 days. This means that in the case of 30 jobs in a project, the average duration of each job is only three days. Therefore, The main drawback of this procedure is that the number of nodes generated in the A-network is dependent on the duration of the jobs in the project.

*Talbot's implicit enumeration procedure.*

This solution procedure can be considered as a combination of integer programming and enumeration methods. The reason it can be considered as an enumeration technique is that it consists of a systematic enumeration of all possible job finish times for each task in the project. To discuss this enumerative procedure, we introduce the following notation.

- $B_j$ is the finish time for job $j$ in the current best complete schedule,
- $CP$ is the critical path completion time for the project,
- $ES_j$, $LS_j$, $EF_j$, $LF_j$ are respectively the critical path early start time, late start time, early finish time, late finish time for job $j$, $j=1,2,\ldots,N$,
- $f_j$ is an integer variable representing the "current" finish time for job $j$,
- $d_j$ is the duration of job $j$,
- $HP$ is a known competition time for the project. This is usually determined by using a priority dispatch scheduling rule,
- $K$ is the number of resource types,
- $k$ represents a specific resource. $k=1,\ldots,K$,
- $N$ is the total number of jobs,
- $P_j$ is the set of all immediate predecessors of job $j$,
P is the set of all immediate predecessors,

\( R_{kt}^* \) is the total amount of resource \( k \) available in a period of time \( t \),

\( R_{kt} \) is the amount of resource \( k \) remaining in period \( t \). Initialised to \( R_{kt}^* \),

\( r_{jk} \) is the amount of resource \( k \) required by job \( j \) each time job \( j \) is active,

\( S_t \) is the set of all jobs active in time period \( t \),

\( u_j \) is the latest possible finish time for job \( j \) and \( u_j = LF_j + (HP - LF_N) - 1 \), \( u_j \) is updated as improved solutions are obtained.

Before the procedure begins, jobs are numbered based on their precedence order, such that if \( i \in P_j \) then \( i < j \). If a heuristic solution, \( HP \), is not known for the completion time of the project, it is set as follows:

\[
HP = \sum_{j=1}^{N} d_j.
\]

The upperbounds \( u_j \) are then obtained by calculating the critical path late finish times for each job (\( LF_j \)) and setting \( u_j = LF_j - 1 + (HP - LF_N) \). Also, initially, \( B_N \) is set equal to \( HP \). The algorithm begins by assigning job 1 to its earliest completion time, \( f_1 = d_1 \), and subtracting \( r_{1k} \) from \( R_{kt} \) for \( t = 1, \ldots, d_1 \) and \( k = 1, \ldots, K \). In this procedure resource restrictions are maintained thorough the use of two compact matrices of \( r_{jk} \) and \( R_{kt} \).

\( R_{kt} \) is initialised to the amount of resource \( k \) available in time period \( t \), \( R_{kt}^* \). When a job \( j \) is assigned a resource and completion time \( t^* \), \( f_j \) is set equal to \( t^* \) and \( r_{jk} \) units are removed from \( R_{kt} \) for \( k = 1, \ldots, K \) and \( t = (t^* - d_j + 1), \ldots, t^* \). Next, job 2 is assigned to its earliest feasible completion time. To find the earliest start time of job 2 the value of \( t^* = \max \{ f_j \mid j \in P_2 \} \) is calculated. This value is the current finish time of all predecessors of job 2.

In this procedure, precedence relationships are maintained by selecting the lowest numbered job \( j \) that has not been assigned a feasible completion time. The job numbering rule described insures that a job is considered for assignment only if all its predecessors have been assigned. Thus, the earliest starting time \( t \) for job \( j \) is obtained from \( t = \max \{ f_j \mid j \in P_j \} + 1 \). From time period \( t^* + 1 \) to \( u_2 \) the resources remaining matrix
is searched for the earliest interval $d_2$ periods long where $R_{kt} \geq r_{2k}$, for $k=1,\ldots,K$. If this interval is from $(t'-d_2+1)$ to $t'$, then $f_2$ is set equal to $t'$ and $r_{2k}$ is subtracted from $R_{kt}$ for $t=(t'-d_2+1),\ldots,t'$ and $k=1,\ldots,K$. This assignment process continues for jobs $3,4,\ldots,N$ until either job $N$ is assigned a completion time or some job $j^* \leq N$ cannot be assigned due to resource infeasibility. In the case where job $N$ is assigned a completion time $f_N$, an improved solution to the problem has been found. This solution is $H=B_N-f_N$ units shorter than the incumbent best solution $B_N$. The improved solution is stored in vector $B$; i.e. $B_j$ is set equal to $f_j$ for $j=1,\ldots,N$ and upperbounds $u_j$ are each reduced by $H$ units. But, in the case that a job $j^*$ cannot be assigned a resource completion time less than or equal to $u_{j^*}$, backtracking occurs. During the backtracking process we try to reassign job $j^*-1$ to the earliest feasible completion time greater than $f_{j^*-1}$. If this is possible then the assignment process continues with job $j^*$. If it is not possible to reassign $j^*-1$ either because of resource infeasibility or because $f_{j^*-1}=u_{j^*-1}$, then backtracking proceeds to job $j^*-2$. Optimality is assured when backtracking proceeds past job one.

What has been described is the main core of this procedure which guarantees an optimal solution to the RCPS problem. This main core is improved through employing a fathoming technique which significantly reduces solution times by identifying, early in the enumeration procedure, partial schedules that cannot possibly improve the solution.

This fathoming technique introduces a method called "network cut". A network cut is an integer time period $c$, between 1 and $HP$, which identifies when schedule elimination rules can be applied. It is said that a job $j$ is cut by $c$ if $ES_j \leq c < u_j$. An integer time period $t^*$ qualifies as a cut if the following two conditions hold: first, there exists a job $j^*$ with $ES_{j^*}=t^*+1$, and second there does not exist a job $j > j^*$ such that $ES_j < ES_{j^*}$.

To describe this schedule elimination method, good partial schedules and inferior partial schedules are defined. A partial schedule is good if when compared to a saved partial schedule it is found that it can improve (reduce the duration of) the schedule
for the entire project. Considering a partial schedule as an ordered list of completion times for a set of jobs, a good partial schedule is a list that may permit the algorithm to assign feasible completion times to jobs not on the list in a manner which results in a reduced duration schedule for the entire project. In contrast, an inferior partial schedule is any partial solution that cannot improve the existing schedule. During the process, if one schedule is found inferior then it will be discarded and if it is good then it is saved as a criteria to improve the discarding process. This removes from explicit consideration many inferior schedules, resulting in a reduction in solution times. The total number of good partial schedules generated is very large and it is undesirable to save all these schedules for two reasons: computational efficiency would suffer, and considerably more computer storage would be required. Therefore only some good partial schedules are selectively saved for each network cut. In solving 50 test problems, Talbot (1978) has found that to store the 100 most recent good partial solutions for each cut c is the most practical way to proceed.

Talbot has tested this procedure on 50 RCPS problems. These problems consisted of either 22 or 28 jobs, with each job in a project requiring up to 3 different resource types. The results show that the procedure with cuts works three times faster than without cuts.

Comparing the procedures.

Patterson (1984), with the assistance and cooperation of Davis, Stinson and Talbot in providing the initial versions of their procedures, presents a comparison of these three approaches. One other approach which he considered and then later abandoned consisted of the use of a problem formulator (matrix generator) and a general purpose solution procedure (MPSX/MIP/370) for integer programming solutions.

He assembled one hundred and ten test problems. The number of activities included in these test problems varied between 7 and 50 with the number of resource types between 1 and 3 and the majority of projects consisted of activities which required the use of each of the of three different resource types for their performance. Each of the three above procedures was programmed in FORTRAN V for use on an Amdahl
470/V8 computer and was written to accept problems in a common data format to facilitate evaluation. Further, each solution procedure was redimensioned to accept problems with similar characteristics. This was done in order to compare approaches on similar problem types and with similar limitations. The result were as follows.

From the point of view of computer memory, Talbot's procedure was the best. This procedure knows precisely and in advance of problem solution the amount of storage required to solve a given problem. This is because the order in which the partial solutions are considered is known in advance of problem solution. Patterson considers this as a strength of this approach. The storage requirement using Davis' and Stinson's procedure, on the other hand, is determined during the execution of the algorithm, and cannot be predetermined. This storage requirement is a function of the number of nodes in the solution tree that are stored and is often an indeterminable function of the problem characteristics. Davis' procedure roughly consumed three times more storage than Stinson's. Davis' procedure was very cleverly written to use auxiliary storage when primary memory is exceeded in storing the solution tree and required pointers. This enhancement gives virtually an unlimited storage capacity for the problem solution tree, albeit at an increase in computation time.

From the speed point of view, on average, Stinson's procedure outperformed the other procedures. This procedure solved all test problems within an imposed time limit of five minutes per problem, whereas Davis' procedure solved 96 and Talbot's procedure solved 97 of the problems. Furthermore, Stinson's procedure solved 76 of the 110 test problems in less computation time than the other procedures.

Finally, it was shown that in those instances where the predicted number of precedence feasible subsets surviving elimination is small, Davis's procedure is likely to produce the optimal solution in the minimum amount of computation time.

*Other Procedures.*

Christofides et al. (1987) present another branch and bound algorithm for the RCPS problem which is based on a completely different approach. It is based on the idea of
using disjunctive arcs for resolving conflicts that are created whenever sets of activities have to be scheduled whose total resource requirements exceed the resource availabilities in some periods. This approach is based on the original idea of Balas (1969) for solving the simpler problem of job shop scheduling. It is clear that in a job shop problem in which only one machine of each type exists, two activities, i and j, that need the same machine cannot be processed simultaneously. To avoid this possibility a disjunctive pair of arcs \{ (i,j), (j,i) \} is added to the original graph, creating a disjunctive graph. Taking a feasible selection S which includes exactly one arc from each pair, together with the set H of fixed arcs, the longest path in the resulting graph is a feasible solution for the original problem. The problem is then to select that S which produces a graph whose longest path is minimised.

Christofides' algorithm is based upon branch and bound during which a feasible schedule is built. At each node of the tree, we try to put in progress all the unscheduled activities satisfying the precedence constraints. The only times to be considered are those at which one or more activities finish. At every one of these times, the candidate activities for inclusion are those which are not in the partial schedule and their predecessors have been completed. These activities can start at that time if resource constraints permit. The candidates are ordered by increasing L(i), the length of the longest path from activity i to the end of project. Based on this order, a candidate goes into the schedule and starts at that time if its resource requirements do not exceed the available resources left by the activities already in progress.

In a case where the candidate cannot start, there is a conflict that produces new branches. These branches describe ways of resolving this conflict by deciding which activities are to be delayed. One way is to delay the activity i which caused the conflict. In order to determine the other resolutions, the idea of "alternative" is introduced. A set A of activities in progress is an alternative to a candidate i if i could start if the activities of A were not in progress. The delay of activities in A allows the progress of i and causes the conflict to be resolved. The delay of a candidate or an activity is introduced by adding arcs that force some activities to wait until the
completion of some others. The best way of delaying a candidate is to choose as the initial node of the added arcs either the candidate or some other activity in progress belonging to A. If an arc \((i,j)\) is added in order to delay \(j\), a lowerbound for the total completion time of the project will be: \(LB_0 = t_i + d_i + L(j)\), that is, the finishing time of \(i\) plus the length of the longest path from \(j\) to \(N\). This bound, based only on precedence constraints, is useful for problems in which the resource constraints are not very tight and it is easily computed.

A depth-first strategy is followed in the tree search. The algorithm backtracks when a schedule is completed or a branch is fathomed by the lowerbound. On backtracking, the added arcs corresponding to the last alternative are removed, and new arcs for the next alternative at the same level are added. If there is no alternative left at that level, we move to the previous level. When the algorithm gets to level zero, the process is finished. There is a special case that needs to be considered. If at a certain time the set of activities in progress is empty and a candidate cannot progress simultaneously with any other unscheduled activity, then this "non-sharing" candidate can be put in progress without considering the alternatives. It is clear that all other means of resolving such a conflict cannot lead to a better solution. The Christofide's procedure is summarised as follows:

**Step 1 (Start).** Let \(T\) be an upperbound on the total completion time. For every activity \(i\), calculate \(L(i)\), the length of the longest path from activity \(i\) to the end of the project. Set \(p = 0\) (level of branching). Put the initial dummy activity 1 in progress: \(t_1 = 0; \ Q = \{1\}\) (partial schedule); \(S = \{1\}\) (activities in progress).

**Step 2.** Set \(m = \min\{t_j + d_j; i \in S\}\). For all \(j \in S\) such that \(t_j + d_j = m\), let \(S = S - \{j\}\), \(C = \emptyset\) (set of candidates); and \(N = \emptyset\) (activities which cause conflicts).

**Step 3 (Construction of C).** For each \(i \in Q\), if for all \((j,i) \in H\) (\(H\) is the set of pairs of activities with precedence constraints), \(t_j + d_j \leq m\), let \(C = C \cup \{i\}\). If \(C = \emptyset\), go to step 2. Otherwise, order \(C\) by decreasing \(L(i)\). If \(S = \emptyset\), go to step 11. Otherwise,
Step 4 (Test of Candidates). For each $i \in C$, if $\sum_{j \in S} r_{jk} + r_k > b_k$ ($r_{jk}$ is amount of resource $k$ required by activity $i$ and $b_k$ is the total availability of resource $k$) for some $k$, then $N = N \cup \{i\}$. Otherwise, $Q = Q \cup \{i\}$, $S = S \cup \{i\}$, and $t_i = m$. If the last scheduled activity is $n$, the schedule is completed: Set $T = \min\{T, t_n + d_n\}$ and go to step 10.

Step 5 If $N = \emptyset$, go to step 2. Otherwise,

Step 6 (Construction of $\alpha(i)$, the set of alternatives to $i$). Select $i \in N$ and $N = N \setminus \{i\}$. Form the set,

$$\alpha(i) = \{A : S \setminus \sum_{j \in S} r_{jk} - \sum_{k \in A} r_{jk} \leq b_k \text{ for all } k\}.$$

Step 7 (Identification of the best way of delaying the candidate and every alternative).

For the candidate $i$, find $a^*$ such that $t_a + d_a = \min\{t_a + d_a : a \in A \subset \alpha(i)\}$. For each $A \subset \alpha(i)$, find $a^* \in A$, such that $t_a + d_a = \min\{t_a + d_a : a \in A \subset \{\alpha(i) \setminus A\}\}$. If $t_a + d_a < m + d_i$, let $b(A) = a^*$ to delay $A$ in backtracking. If $m + d_i \leq t_a + d_a$, keep $b(A) = i$.

Step 8 (Branching). Set $p = p + 1$, $H'(p) = \{(a^*, i)\}$, $B(p) = \alpha(i)$ (Set branches at level $p$).

Step 9 (Lowerbound). Set $LB_0 = \max\{t_j + d_j + L(i) : (j, i) \in H'(p)\}$. If $LB_0 > T$, go to step 10. Otherwise, go to step 5.

Step 10 (Backtracking). $H = H \setminus H'(p)$, $H'(p) = \emptyset$, If $B(p) = \emptyset$, set $p = p - 1$. If $p = 0$, STOP. otherwise, select $A \subset B(p)$. Let $B(p) = B(p) \setminus \{A\}$, $H'(p) = \{(b(A), j) : j \in A\}$, $H = H \cup H'(p)$, $Q = Q \setminus A$, and $S = S \setminus A$. Go to step 9.

Step 11 (Non-sharing candidate). For each $i \in C$, if, for all $j \in Q$, $r_{jk} + r_{jk} > b_k$ for some $k$ then $Q = Q \cup \{i\}$, $S = S \cup \{i\}$, $t_i = m$, and go to step 2. Otherwise, go to step 4.

This algorithm has been tested on 40 generated problems with 25 activities and three different types of resources (Christofides et al. 1987). The first twenty problems were loosely constrained and the next twenty were tightly constrained. The duration of activities was in the range 1 to 9 and resource requirements in the range 0 to 6. The availability of each resource was 6 units and in the graph of precedence constraints the ratio between arcs and nodes varied between 1 and 3. The problems were run on
the Univac 100. All of the first twenty problems were solved within the limit of 60 CPU seconds. Among the second twenty problems, which were tightly constraints, 6 could not be solved within the limit of 60 CPU seconds. With the help of three other lowerbounds all 40 problems were solved within the specified time limit.

Bell and Park (1990) employed another successful approach to solve the RCPS problem. This approach, like the previous approach, repairs resource conflicts rather than constructing detailed schedules by dispatching activities. Approaches which construct detailed schedules search through a space of partial schedules. Here, a partial schedule means a schedule which satisfies all precedence constraints and assigns an earliest start time to all activities, but it is "partial" because it may violate one or more resource constraints. The only constraints on each state of this state space are precedence constraints, and a set of precedence constraints has an associated minimum makespan. Because of slack in scheduling noncritical activities, there can be many schedules corresponding to a given precedence network which achieve this minimum makespan.

Since there is no loss in scheduling each activity at its earliest possible time, Bell and Park consider only the single schedule which assigns the unique early start time to each activity. Thus a state is a set of precedence constraints and a state's schedule is the associated early start time schedule. Resource violating sets (RVS) of activities whose concurrent execution would violate resource constraints are identified. For any state, an RVS of activities is defined such that: all activities in the RVS are executed concurrently under the state's schedule and the usage of some resource i by these activities exceeds the availability, R_j.

Child states are then generated by adding in turn each additional precedence constraint which will guarantee that all activities in the RVS can no longer be executed simultaneously. A leaf node of this search tree is a state with enough added precedence constraints to ensure that the associated schedule does not violate any resource constraints. When resource violations are detected in numerous time
intervals, the earliest time interval, \((t, t+1)\), at which a resource violation occurs is selected and repaired. Thus, as in existing approaches, it proceeds from time 0 forward. When a state \(S\) with precedence network \(P\) is found to contain a resource violating set \(V\) of activities in some time interval, \((t,t+1)\), a disjunctive constraint is identified in order to repair this violation. This constraint requires the addition of exactly one new arc to \(P\). Thus, one child state is generated for each disjunctive constraints giving new states \(S_1, S_2, \ldots\) with augmented precedence networks \(P_1, P_2, \ldots\).

Bell and Park found that adding the strongest possible constraints leads to an algorithm with slower performance in the test problems and their final algorithm involved adding weaker constraints. In fact, in the final version of this algorithm, if children of \(S\) are generated to correct the resource violating set \(V\) in time \((t,t+1)\), the same violating set \(V\) may reappear in some state which is a descendant of \(S\). In this case, \(V\) will appear in some interval \((t^*, t^*+1)\) with \(t^*>t\). The strategy of adding weaker constraints appears better in practice because of a lower branching factor and an improved ability to apply pruning rules. Since a parent state is transformed into a child state by the addition of exactly one arc to the network, the search process is a sequential process of identifying new disjunctive constraints and deciding on the order of investigation of generated child states. States which have not been investigated are assigned a heuristic value by the state evaluation function, being put into a priority queue for further investigation. The state evaluation function of every state is its makespan, which is the project completion time ignoring resource constraints, and it never overestimates the objective function. Therefore, the first feasible state investigated is guaranteed to be optimal.

At each step of this search process the most promising state which has been generated and not further investigated is selected. If the selected state has no resource conflict, then an optimal schedule is found and the search process terminates successfully. Otherwise, the selected state will be expanded by discovering and using a disjunctive constraint to generate the state's successors. Newly generated states are then added to
the set of states which have not yet been expanded. Again, the most promising state is selected and the process continues.

As can be seen, this algorithm is based on the famous A* search technique that is described in most introductory artificial intelligence texts. The smallest time $t$ such that a resource violating set of activities are in progress is called RVST ("resource violating set time"). Knowing RVST, activities can be classified into two sets: the scheduled set, which finish at or before RVST and the unscheduled set, which finish after RVST. This unscheduled set may contain activities which start before RVST and continue beyond it. When two or more states have the same makespan, tie-breaking is required in selecting the most promising state. Among states with the lowest makespan, a state with the highest RVST is selected. A high RVST indicates that a state may be "close" to feasible; and it is hoped that correcting remaining resource violations may require smaller increases in the makespan. If a tie remains after applying this rule, a state is selected arbitrarily. Increasing RVST by at least 1 unit of time requires that all violating sets in (RVST, RVST+1) be corrected. However, only "minimal resource violating sets" (MRVS), which have no proper resource violating subsets, are considered in sequencing decisions.

There are $m(m-1)$ choices of arcs to add to break up a MRVS of cardinality $m$. This creates a very high branching factor even for a moderate value of $m$. Two pruning rules are employed. The first one is Schrage's left-shiftable which prunes a state whenever an activity in the scheduled set of that state can be left-shifted. The second pruning rule involves the comparison of two states, $S$ and $S'$, to conclude that the best feasible scheduled derivable from the first, $S$, must be at least as good as the best feasible schedules derivable from the second, $S'$. State $S$ weakly dominates state $S'$ if:

1. $S$ and $S'$ have the same scheduled set of activities $A_S$ (and thus the same unscheduled set of activities $A_U$),
2. the precedence network of $S$ and $S'$ have the same subsets of arcs connecting pairs of activities in $A_U$. 

(3) the schedule from S has start times for activities in A_u which are less than or equal to the corresponding start times in the schedule from S'.

The general principle of adding the strongest possible constraints to a particular state in creating its children often leads to a search tree with many states in S and S' satisfying (1) but violating (2) because particular arcs have been added connecting pairs of activities in A_u in S (S') which are not present in S'(S). On the other hand, by applying the approach of adding weaker constraints, condition (2) proves to be superfluous because the only arcs present connecting pairs of activities in A_u will be those present in the original precedence network (the root state of the search tree).

In adding a weakly constrained arc (a,b), a dummy activity z is added with duration d_z = t_a + d_a. Activity z requires no resources and can be assigned start time 0. An arc is added from the dummy activity z to b. The immediate impact of this change to the network is to increase the scheduled start time of b to t_a + d_a. The further impacts of adding this arc on the start time of the rest of activities, continues as in the strongly constrained approach. However, there is no arc added to the network connecting any pair of unscheduled activities. That is why condition (2) never needs to be checked.

Using the weakly constrained approach also allows for a lower branching factor in some instances. With the first violating set at RVST, both strong and weak approaches push the scheduled start times of certain activities to RVST+1 or higher until no violating set remains in (RVST,RVST+1). With the weakly constrained approach there is no need to consider all possible ways of doing this. In this case, for a violating set V with m members, we need consider only m possible modifications to the network. For each activity b in V, an arc (a,b) is added where "a" is an arbitrary member of the subset V-{b} of V with earliest finish time. Thus activity start times are pushed as little as necessary into the future to remove violating sets in (RVST, RVST+1). The branching factor in this operation is thus reduced from m(m-1) for the strongly constrained approach to m for the weakly constrained approach.

With strong constraints, any violating set eliminated by an arc addition can never reoccur in a descendant state. Nevertheless, the weaker approach allows for
significant pruning which makes up for the occasional reoccurrence of a violating set. The algorithm (for either the strong or weak approach) has been described by following procedure.

PROCEDURE A*-Schedule (Initial-State)

1. Scheduling-Done = NIL
2. Upper-Bond = Big Number
3. OPEN = List (Initial-State)
4. While Scheduling-Done = NIL Do
   5. Current-State = First (OPEN)
   6. OPEN = Rest (OPEN)
   7. IF MRVSs(Current-State) = NIL Then
      8. Report-Optimal-Schedule( Current-State)
      9. Scheduling-Done = T
   ELSE
      10. OPEN = Merge (Descendants(Current-State, Current-State)

The OPEN list is constructed and initialised to contain a single element, the initial state. OPEN is an ordered list of states which have been identified but not yet examined. The Rest list is composed of all but the first element of List. The "Descendants" procedure generates new states to add to OPEN by creating and returning a list of new states. This procedure is called recursively and has two parameters which are: the states to examine immediately and the root state of the search tree. The "Merge" procedure inserts newly found states into an appropriate position on OPEN so that OPEN will remain ordered. The A* procedure will terminate before expanding states with makespans which are greater than or equal to an upperbound value. The use of an upperbound saves memory but does not reduce the number of states expanded.

A hash table is chosen for implementing the setlike data structures used in this procedure. A hash table comprises a set of entries, which map a key to a value. For OPEN, makespan acts as the key and states are listed as values for each key. In
implementing the dominance pruning rule, for a n-activity problem, a key will be an n-component bit vector, called the scheduled-vector. Each index in this bit vector represents an activity and the value 0 and 1 represent unscheduled and scheduled, respectively. When a comparison for dominance pruning takes place, only states with the same scheduled-vector are accessed.

The procedure has been tested on 110 randomly-generated problems with network structures and resource requirements similar to those encountered in practice (Bell and Park 1990). One hundred of these problems had between 8 and 27 activities and 10 of these problems had 51 activities. The results show that this approach performs very well for problems where resource-constraints are not tight, regardless of the size of problems. For one of the problems with 22 activities, which was highly resource-constrained, it took more than 20 minutes to solve on an Apple Macintosh Plus. However, one of the problems with 51 activities only took 2 minutes to solve. The computational results also show that both pruning rules cut down the search process effectively, with the state-dominance rule outperforming the left-shift rule.

To date, these procedures are the main successful approaches to finding an exact solution to the RCPS problem. Now we shift our attention to heuristic approaches.

3.2.2 Heuristic Methods

Because of the relative lack of success of optimisation procedures with the RCPS problem, many researchers have resorted to heuristic approaches. Here, by heuristic we mean a method which creates "good" solutions rather than optimal solutions. Perhaps the term "non-exact" is more suitable than heuristic for describing these approaches. However, in accordance with the literature we will use the term "heuristic".

"Heuristics" are schemes for assigning an activity properties used in making activity sequencing decisions which are required for the resolution of resource conflicts. Heuristic-based procedures for the RCPS problem are the only practical means of obtaining solutions to problems of the type commonly found in business. In describing exact approaches it was mentioned that, heuristics also are employed in
connection with certain optimisation procedures to produce a base for selecting promising branches. There is a vast literature on heuristic methods. We make special mention of Olaguibel and Goerlich (1990) and Tsubakitani (1991) which include surveys as well as extensive reference lists. Our intention here is not to give a general recipe for constructing heuristic methods nor do we give an exhaustive comparison of many existing heuristic procedures. Instead we systematise and classify heuristic methods used to solve the RCPS problem. Kelly (1963) categorised heuristic approaches to the RCPS problem into two categories of "serial" and "parallel" and this categorisation is still valid today. For methods in both of these categories, once a partial solution is created, it is used as a basis for the final solution and a job is never rescheduled.

The serial methods begin by numbering the nodes of a project so that for each arc the head node will have a larger number than the tail node. This numbering scheme has the property that if activities are sorted by number, no activity will appear before any of its predecessors. Any such order is called a "topological" order. A serial schedule then is created by considering the activities in one topological order and scheduling them one at the time as early as precedence and resource constraints permit. This topological order is generally not unique. Many priority criteria such as resource requirements, activity durations and the total float can be used for building a topological order to define a different serial algorithm.

Parallel methods, on the other hand, construct a feasible schedule by proceeding forward in time. At any point in time during the construction, a set of feasible activities is determined that can be scheduled according to their precedence constraints. This set is sorted by a priority rule and activities are scheduled in that order up to resource capacities. Like serial methods, every priority rule defines a new heuristic. If activities receive the same priority value regardless of the state of the schedule, the method is called "static parallel" and if this depends on the state of schedule, the method is called a "dynamic schedule". Most algorithms proposed in the literature are parallel algorithms because they work much better than serial
Olguibel and Goerlich (1990) present the following scheme that covers all parallel algorithms.

**Step 1 (Initialisation)** Set PS (Partial Schedule) = ELG (eligible set, the set of activities with all their predecessors completed)=AVL (available set, the set of activities that can be scheduled)=ACT (set of activities in progress)=∅. Set t=0; and RU_k (resource of type k in use)=0, for all values of k.

**Step 2 (Construction of ELG)** Set ELG={ j / (i,j)∈H, t_j+d_j<t} where H is the set of precedence constraints. If ELG=∅, go to step 7.

**Step 3 (Construction of AVL)** Set AVL={ j∈ELG / r_{ik}≤R_k-R_{uk}, for all k} where r_{ik} is amount of resource k required by activity i and R_k is total availability of resource k. If AVL=∅, go to step 6.

**Step 4 (Priority rule)** Order AVL according to the chosen priority rule.

**Step 5 (Construction of the schedule)**

5.1. Take the next j∈AVL. If r_{jk}≤R_{uk}, for all k, go to step 5.2 else go to step 6.

5.2. Set ACT=ACT∪{i},t_j=t,PS=PS∪{t_j}, RU_k=RU_k+r_{jk}, for all k. Go to 5.1.

**Step 6 (Determining a new time t)** Find j so that t_j+d_j =min { t_i+d_i, i∈ACT}. Set t=t_j+d_j, ACT=ACT-{j / t_j+d_j=t }, RU_k=RU_k-r_{jk}, for all j so that t_j+d_j=t; and for all k. Go to Step 2.

**Step 7 (Stop)** All activities have been scheduled.

Different serial algorithms only differ in step 4 where a priority rule is determined. Two broadly different categories of heuristics have been used as priority rules: heuristics incorporating some measure of time and precedence, such as: job slack, job duration, and heuristics incorporating some measure of resource usage. There are hundreds of these heuristics and the following are some which are currently used.

SIO (Shortest Imminent Operation). This rule assigns higher priority to the operations with shorter duration.

LIO (Longest Imminent Operation). This rule assigns higher priority to the operations with longest duration.
MIS (Most Immediate Successors). This rule picks first the activity on which most other jobs are waiting so that they can be started,

LIS (Least Immediate Successors). This rule picks first the activity on which the least number of other jobs are waiting so that they can be started,

LNRJ (Least Non Related Jobs). This rule selects first the activity that has the largest total number of predecessors and successors,

EFT (Earliest Finish Time). This rule gives priorities to the activities with the earliest finish time,

EST (Earliest Start Time). Priority is given to activities with the earliest start time,

LFT (Latest Finish Time). Priority is given to activities with latest finish time,

MINSLK (Minimum Slack). This rule gives priority to the activity with minimum slack,

GRD (Greatest Resource Demand). This rule assigns priority on the basis of the total resource-unit requirements of all types. Higher priorities are given to activities with greatest resource demands,

GRU (Greatest Resource Utilisation). This rule gives priority to the combination of activities which results in maximum resource utilisation in each scheduling interval (i.e. minimum idle resources). The rule is implemented by solving zero-one integer programming problems,

RANDOM. This rule assigns priority among competing activities on a random basis.

None of these heuristic rules performs consistently well on all test problems. However, researchers have reported that the MINSLK rule, which bases activity priority on activity slack, performs better on average than others (Davis and Patterson 1973; Olaguibe and Goerlich 1990).

Khatab and Choobineh (1992) assessed statistically the ability of various heuristic rules in order to determine which are most likely to develop improved schedules. In their research a large group of these priority rules are compared with each other. A set of projects consisting of 110 networks is used, each project having a single scarce resource type. For each network the available resource level is varied, in increments
of 2 units, from their minimum required resource level for completing the project to a maximum level beyond which no reduction in project duration is possible. Like all other research, this confirmed that none of heuristic rules performs consistently well on all test problems. They show that when a resource is scarce, priority rules which are a function of resource requirements utilise the resource better than those rules which are not a function of resource requirements. However, when available resource levels are increased, the priority rules that put less emphasis on resource requirements can produce better results.

A combination of heuristic rules can also be used as an individual heuristic rule. For example, Jain and Harad (1985) propose a heuristic algorithm in which three different types of penalty are considered, resource penalty, predecessor penalty and slack penalty where the combined effect of the total penalty for each activity is considered as the sum of these three penalties. One of the efficient methods based on hybrid heuristics was presented by Ulusoyle and Ozdamar (1989). In this method, the priority is the weighted combination of its individual resource utilisation ratio and the number of its immediate successors. For any combination of precedence and resource utilisation weights, a priority list is obtained. Thus different weight combinations might result in different project durations. A search needs to be conducted to determine the combination of weights resulting in the least project duration. The general performance of this procedure has been found to be quite satisfactory in the sense that its priority distribution to activities enables it to cope with almost all types of problems more successfully than other widely-used heuristics. In this procedure priorities are assigned for each weight combination at the beginning of each iteration and are not changed during the iteration.

Bector (1990) conducted research regarding the combination of heuristic rules. In order to identify the most efficient combinations of rules, he proceeded as follows. First, based on 66 projects with different degrees of complexity the performance of 13 priorities are evaluated. Then based on these results, the best combinations of rules are determined. He shows a combination of three heuristics has a relatively high
probability of giving the best and even the optimal solution. By the best solution he 
means the best among the 13 solutions given by the 13 tested heuristics.

Not only can heuristic rules be combined together but also they can be used in multi 
levels. Mario and Macgregor (1986) present a two level heuristic method for the 
RCPS problem. This heuristic is based on a combination of priority rules where 
utilisation of resources by operations and the critical path of operations are taken into 
account. These two criteria, i.e., operations on the critical path and resource 
utilisation are combined in a two level priority scheme. At level 1, operations are 
placed into sets according to the criticality of the path where they occur. At this level 
priorities are static. Level 2 corresponds to prioritizing dynamically those operations 
in each set based on resource utilisation. One important point about this procedure is 
that every resource has a weight based on its scarcity. This means that between 
different resource types those types of resources which are more scarce have higher 
multipliers. The parameter $w_k$ which represents the weight on the resources of type k 
is calculated as follows:

$$w_k = \frac{1}{H \cdot R_k} \sum_{j=1}^{n} c_j \cdot r_{jk}$$

In this relation, H is the duration of the project calculated on the basis of standard 
CPM calculations, $R_k$ is the availability of resource k, $d_j$ represents the duration of 
activity j, and $r_{jk}$ is the amount of resource k required by job j. Mario and Macgregor 
claim that from a theoretical point of view this heuristic approach shares many 
properties that are recognised in the theory of scheduling as characteristic of good 
heuristic approaches.

Heuristic procedures can be multipass procedures. For example, Li and Willis (1992) 
present a serial iterative procedure in which scheduling is carried out iteratively until 
a better schedule results, or further scheduling is stopped when no further 
improvement to the project duration can be obtained. In this serial method, a project 
is first scheduled forwards to yield a forward schedule. The project duration of the 
schedule is then used as the starting point for the backward schedule. In the forward
schedule, activities are started as early as possible but in the backward schedule activities are started as late as possible subject to meeting the project target date. The procedure is then scheduled forwards and backwards iteratively until no improvement can be found. Within the iterative stage of the method, the heuristic rules for compiling the activity loading lists are as follows:

A) Rules set for the construction of the iterative forward schedules;

A-1) Activities are sorted in ascending order by their start times on the immediately preceding backward schedule;

A-2) Ties are broken by the activities orders on the activity loading list for the last forward schedule construction process,

B) Rules set for the construction of the iterative backward schedule;

B-1) Activities are stored in ascending order by their finish times on the immediately preceding forward schedule,

B-2) Ties are broken by the activities orders on the activity loading list for the last backward construction process.

These heuristic rules incorporate the properties of the previous schedule into a current schedule and subsequently improve the current schedule if the properties of the two schedules are complementary to each other.

Determining which heuristic produces shortest-duration project schedules is difficult and there is little basis for making a choice among different procedures. Actually there exists somewhat conflicting evidence in regard to heuristics which resolve the conflicts that develop between the resources demanded by an activity and those available. These inconsistencies are due to the nature of the data examined and different projects having different network structures and resource requirements.

The effect of network structure on the efficiency of procedures

An apparently common disadvantage of many exact solutions to the RCPS problem is the extreme variation in computation time experienced from problem to problem. For heuristic procedures this disadvantage appears in the extreme variation of solutions from optimal solutions. Davis (1975) showed that solutions of two similar 27-job
networks with the same network and resource requirements and only different durations for each job could differ by as much as a factor of nine. This illustrates that even for seemingly similar networks the computation time can vary extremely.

A large number of studies have been made of the effect of time and resource characteristics on different procedures. For example: Patterson (1984) used networks with different characteristics to compare three exact methods, Ulusoy and Oztamgar (1989) chose four network/resource characteristics and investigated for their effects on heuristic performances, and Khattab and Choobineh (1990) used different combinations of resource levels and networks to investigate the effects of the characteristics of test problems on the efficiency of procedures.

In studies conducted in this area, usually parameters are described in three general classes: measures which characterise the size, shape and precedence structure of the network, measures which indicate time characteristics and measures which characterise resource demands and availability. Measures such as "size" (number of activities), general appearance (e.g. "short and fat" or "long and thin"), number of resources, limitations of resources and critical path duration are usually reported for every test problem. These provide good descriptions of the problems but they are of little help in answering other important questions about changes in the characteristics of a given network. Variations in either the characteristics of the network (precedence relations, resource requirement and/or duration of activities) or resource constraints drastically affect the efficiency of procedures. A procedure that behaves well in producing a solution for a special problem can be extremely inefficient when a small change is made to the characteristics of that problem. Despite the fact that the idea of a network summary measure is not new and there has been little research on this subject, up to this point there are no criteria which isolate the factors responsible for this unpredictability in the behaviour of procedures. This is, to a considerable extent, due to the combinatorial nature of the RCPS problem. Generally, the performance of every procedure is a function of the characteristics of every individual problem and
neither the important variables of this function nor its form has been specified for any procedure. This is a difficult and, as yet, unexplored research area.
3.3 The Application of LCBA*

In this section, we will describe how LCBA* can be applied to solve the RCPS problem. As stated earlier, the main feature of this algorithm is to search through the states of a state-space problem, update heuristic estimates of states and backtrack when necessary. In the language of states and operators, LCBA* finds a solution of a problem as a sequence of operators that transform an initial state into the goal state within a specified threshold. In order to apply the algorithm to the RCPS problem, at first this problem must be stated as a state-space problem, with states and operators, where an operator transforms one state into another state. Starting with the initial state, we apply a series of operators to reach the goal state. Associated with each operator is a cost, and hence any series of operators that transform the initial state to the goal state has an accumulated cost.

Finding a feasible schedule for a RCPS problem can be considered as a series of decisions about scheduling one or more jobs. The process of decision making begins by deciding on the starting dummy activity 0 as the beginning of a project and proceeds whenever resource availability and precedence constraints together permit the starting of one or more other jobs. This process continues until all jobs are scheduled. In this process, we only make new scheduling decisions at points in time when at least one activity is completed. The reason is that the level of the resource pool does not change except when some resources are released by the completion of an activity.

In setting up a state-space formulation of the problem, we must first determine the states of the problem. In a RCPS problem, states can be described as partial schedules. A partial schedule is a schedule in which some of the activities have already started. Operators associated with each state are all the possible decisions about scheduling other jobs with respect to both precedence and resource constraints. In a general state-space problem, the cost of any operator can depend on the states we transform to and from. However in this problem, this cost only depends on the state transformed from and is equal to the minimum time remaining for an incomplete
activity to be completed. Therefore associated with each operator is a cost that determines the minimum time left for one of the incomplete jobs of the state transformed from to be accomplished.

The initial state is an empty schedule where none of the jobs has started yet and the goal state is a complete schedule within a bracket of the optimal completion time.

To discuss the idea in detail, it is useful to consider the space of states reachable from the initial state as a tree containing nodes corresponding to the states. The nodes of this tree are linked together by arcs that correspond to the operators. The root of this tree, the initial state, is an empty schedule and the intermediate nodes are the partial schedules. In this tree a leaf (terminal node) is a complete schedule in which all activities are scheduled and none of them is left in the unscheduled set. The goal is to reach from the empty schedule, the root of this tree, to one of the complete schedules (leaf nodes) provided that the completion time of that schedule is within the threshold range of the optimal solution. This threshold value can either be set to zero and in this case the optimal solution will be found, or be set to any other non-negative number.

Figure 3.1 shows some parts of the space of states for a simple project comprising 5 activities or jobs. At the root of this tree stands an empty schedule, {}, and all five activities of the project are placed in an unscheduled set, (j1, j2, j3, j4, j5). One of the leaf nodes of this tree has been represented. In this leaf node, like all others, all 5 jobs have been scheduled and the set of unscheduled jobs is empty. Also, some of the intermediate nodes of this tree have been shown, each characterising a "partial schedule" and representing scheduling decisions for some subset of the jobs in the project. All these partial schedules are feasible in the sense that all their predecessors are completed and the required resources are available. As this tree is expanded from some given intermediate node (partial schedule), a new set of partial schedules is created. Each member of this new set has in common with its parent all scheduling decisions made previously. The only difference between any partial schedule and its parent is that it includes one new decision about the scheduling of one or more jobs that have not yet been scheduled. This decision is made when at least one of the
incomplete jobs of a partial schedule is completed and consequently some resources are released.

Figure 3.1: The space of states for a project comprising 5 jobs.

Now we discuss the way in which heuristic estimates associated with states are found. Since each state is a set of some incomplete jobs, it can be considered as a new project. What makes the minimum duration of a project difficult to find is the simultaneous existence of two sets of precedence and resource constraints. By ignoring resource constraints, The RCPS problem becomes a simple problem whose optimal solution is less than or equal to that of the original problem. Incomplete jobs in each state can be differentiated into the three sets of "newly started", "partly completed" and "unscheduled" jobs. These three sets together can be considered as a new project. Notice that the two sets of "newly started" and "partly completed" jobs together comprise the set of "in progress" jobs. Now, the only modification required for this new project is to update the duration of the activities which are in the "partly
completed" set. The reason is that these activities have been in progress for some time and the time remaining to complete each of them is less than what has been stated in the original project. By ignoring the resource constraints, this new project can be scheduled and its minimum completion time can be considered as a non-overestimating heuristic estimate of the corresponding state. This heuristic estimate is improved by considering the fact that none of the tasks in the "unscheduled" set can be started unless at least one of the tasks in the "in progress" set is completed. It is clear that, the more accurate non-overestimating heuristic estimates of states are, the more efficiently LCBA* performs. Therefore, for every state two non-overestimating heuristic estimates are calculated and the maximum estimate is selected. The first heuristic estimate is calculated by ignoring resource-constraints, and the second is calculated by ignoring precedence-constraints. When a precedence-constraint is ignored, there are some resource requirements needed for incomplete activities. These resource requirements are multiplied by their duration and after being added together the result is divided by the amount of that resource available per day, yielding a minimum number of days for which the project continues. In the case of multiple-resource constraints, this idea is generalised by finding the maximum value among different types of resources. Again, this heuristic estimate is improved by considering the fact that none of the unscheduled tasks can start unless at least one of the tasks of the "in progress" set is completed. Based on ignoring either resource or precedence constraints, two heuristic estimates are computed and the largest of these two estimates is selected.

The branching process takes place from the last node created in the tree and backtracking occurs whenever total learning exceeds the threshold value. Among the children of a node, we select the child that has the minimum heuristic estimate. In LCBA*, the transformation cost $k(x,x')$ plus the heuristic estimate $h(x)$ determine which node is selected, but as stated earlier the transformation cost in the RCPS problem depends only on the state transformed from and is consequently ignored so that the selection is based on only heuristic estimates.
A node from the tree can be pruned away if it can be established that further branching from it cannot lead to a better complete schedule than other potential complete schedules where a potential complete schedule is a complete schedule that is either already known or can be developed from some other intermediate nodes. LCBA* can easily be equipped with a built-in pruning rule based on an incumbent solution which is the best solution up to a certain point in the solution process. In this case it prunes away any state that could potentially lead to an equal or worse solution compared to the incumbent solution. In the RCPS problem, if a state includes any job already scheduled which can be left-shifted without violating either resource or precedence constraints, this state can be pruned away. As noted previously, this is a pruning rule which is used by all enumerative algorithms for the RCPS problem. We also adopt this pruning rule as the built-in pruning rule of LCBA*.

We use a sample project to explain the application of the algorithm. Figure 3.2 is a representation of the order in which four jobs in this simple project must be done. In this network the nodes indicate jobs, and the links indicate precedence relationships between the jobs. The two dummy activities of "0" and "5" define the "beginning" and the "end" of the project respectively. Figure 3.3 shows the corresponding tree for the state space formulation of the problem. At the root of this tree there is state 0, representing the only possible decision at time zero. As stated, job 0 is a dummy activity that shows the beginning of the project and based on the project network no job can start before this dummy activity is completed. The duration of this activity is 0 and therefore at time zero there are two possible scheduling decisions to make. The first decision is to start job 1 and the second is to start job 2. Notice that because of the constraint on the resources, it is not possible to start both of these two jobs simultaneously.

Although figure 3.3 shows the entire state space of the problem, the algorithm only generates a portion of these states. The states generated by the algorithm (with threshold value set to zero) are shown in figure 3.5. Notice that states 3, 4, 5 and 6 in this figure correspond to states 6, 7, 8 and 18 in figure 3.3.
In determining possible operators applicable to a state, all feasible combinations of activities that can enter the corresponding partial schedule are considered. Therefore, even in the case where both activities can start together, the scheduling of either of them alone is not ignored. For instance, at state 7 it is possible to take together both jobs 3 and 4 as state 17, but states 15 and 16 only schedule one of these jobs by itself. The left-shifted rule can prune away some of the states such as state 11. In this state, job 3 can be left-shifted to time 20 without violating either a precedence or resource constraint. Figure 3.4 shows how this job can be left-shifted.

The generating process begins by generating state 0. In this state, only the dummy activity 0 starts and all other activities are in the "unscheduled" set. The heuristic estimate associated with this state is 40 which is the duration of the entire project without resource constraints. After beginning this dummy activity, the two activities
1 and 2 can start. State 1 corresponds to the decision about starting activity 1 and state 2 to starting activity 2. Concurrent demand, \((7+6)\), for the resource by these two competing activities 1 and 2 exceeds resource availability, 10, causing them not to be considered together.

Figure 3.3: A corresponding tree for a state space formulation of the sample project.
Therefore, only two operators are defined in association with state 0. The first operator transfers state 0 to state 1 and the second operator transfers it to state 2. As previously stated, in this state-space formulation, the cost of any operator depends only on the state transformed from and not on the state transformed to.

The cost of either of these operators, \( k(0,1) \) or \( k(0,2) \), is zero. This cost always indicates the time left for one of the jobs in the "in progress" set of the state transformed from to be accomplished. The associated heuristic estimate for state 1 is 60 and for state 2 is 47.

Based on what has been said it is roughly clear how these heuristic estimates are calculated and the complete detail will be provided later in this chapter. Among these two states the one with the minimum heuristic estimate, state 2, is selected for transforming from.

**Figure 3.4:** Comparison of resource levels when job 3 is left-shifted.
Figure 3.5: The states generated by the algorithm from the corresponding state space tree.

Since the cost of transforming from state 0 to state 2, 0, added to the heuristic estimate of state 2, 47, is seven units greater than heuristic estimate of state 0, 40, the heuristic estimate of state 0 is improved by 7 units and changes to 47. Since the threshold value, zero, is less than the accumulated learning, which up to this point is
7 units, backtracking should occur. However, as stated in the description of the algorithm, no backtracking occurs from the root of the state space tree.

Now state 2, where job 2 was scheduled to start at time zero, will be transformed from by selecting the best applicable operator. The algorithm first determines the time when the next set of scheduling decisions can potentially be made. This new time is the start time of state 2 plus the minimum time left for an activity in the "in progress" set of this state to be completed. In state 2 there is activity 2 which is in the "newly started" set and there is no activity in the "partly completed" set. Activity 2 takes 15 days to complete, and thus the time when the next set of scheduling decisions can be made is the addition of this value, 15, to the start time of state 2, 0, which results in 15. The result, 15, as the start time of all states transformed to from state 2 is considered. Three new feasible decisions now appear, scheduling activity 4 by itself (state 3), scheduling activity 1 by itself (state 4) and scheduling both activities 1 and 4 together (state 5). Notice that the scheduling of both jobs 1 and 4 simultaneously is resource feasible (because 1 + 7 ≤ 10). The cost of transforming state 2 to any of states 3, 4 or 5, k(2,3), k(2,4) or k(2,5), is 15, the time left for at least one of the jobs in the "in progress" set of this state to be completed. Heuristic estimates associated with these three newly generated states are 57 for state 3, 45 for state 4 and 32 for state 5. Among these three states, state 5, with the minimum heuristic estimate, is selected. Since the cost of transforming from state 2 plus the heuristic estimate of state 2 , (15 + 32), does not exceed the heuristic estimate of state 2, 47, neither any learning nor any backtracking occurs.

Now, by the selection of the best applicable operator, the algorithm transforms state 5, where the jobs 4 and 1 were scheduled to start at time 15. Again, it first determines the time of the occurrence of the next set of scheduling decisions. In state 5 there are two activities 4 and 1 which are newly started and no activity is in the "partly scheduled" set. Activity 4 takes 25 days and activity 1 takes 20 days to be completed, and thus the minimum completion time is 20. When this value, 20, is added to the start time of state 5, 15, the value of 35 is obtained which is the start time of all states
transformed to from state 5. To transform state 5, there exists only one new feasible decision and that is the scheduling of activity 3 (state 6). Notice that state 6 corresponds to a a leaf node and that the heuristic estimate of this state is 12. The reason is that at the start time of this state, activity 4 was in the "partly completed" set and 20 of its 25 days have passed with only 5 days left and that the maximum of 12 which is duration of activity 3, which started at the start time of this leaf state, and 5 is 12. Since for state 6, the cost of transformation plus the heuristic estimate, (20+12), does not exceed the heuristic estimate of state 5 which is 32, neither any learning nor any backtracking occurs. State 6 corresponds to a complete schedule and no further transformation is possible and since the threshold value has been selected as zero then the optimal solution has been found.

Now, at the end of this explanation of the general steps of the algorithm applied to the sample project, there is a need to further examine in detail the way in which heuristic estimates are calculated. To facilitate the calculation of the heuristic estimates of states, before the procedure begins resource constraint are ignored and the length of the longest path from each individual activity to the end of the project is found. Figure 3.6 shows these values for every individual activity in the sample project.

Every state consists of a set of activities that are not yet started whereas all their predecessors are completed. Among these activities, the activity with the maximum "longest path to the end of the project" is found and this longest path, L, is increased by the time left for the first activity in the "in progress" set to be completed, C. Then the result, C+L, is compared with the maximum longest path (modified for activities in the "partly completed" set) for activities in the "in progress" set, M, and the maximum, F, of C+L and M is selected as the heuristic estimate based on ignoring resource constraints. This is done because none of the unscheduled tasks can start unless at least one of the tasks in the "in progress" set is completed.

The idea of calculating a heuristic estimate based on ignoring precedence constraints is to measure the total requirements for a resource divided by the availability of that resource. The relevance of this measure is enhanced by the fact that none of the
unscheduled tasks can start unless at least one of the tasks in the "in progress" set is completed. For multiple resources, the whole idea of determining heuristic estimates based on ignoring precedence constraints is generalised by finding the maximum value of such measures across all the different types of resources involved.

To find heuristic estimates based on ignoring precedence-constraints, at first calculate $C_1$ the total resource requirements for all activities in the "in progress" set, and after dividing by resource availabilities, $R$, the result, $(C_1/R)$, is compared with the time left for at least one activity in the "in progress" set to be completed, $C$, and the maximum of them, $T_1$, is selected, i.e. $T_1 = \text{Max}\{C_1/R, C\}$. In the case where $T_1 = C$, the improvement discussed is possible and in the case when $C$ is less than or equal to the completion time of activities already begun, this improvement is not possible. The reason is that in the latter case these are enough activities to utilise available

---

Figure 3.6: The longest path for each individual activity in the sample project.
resources before completion of at least one of the tasks in the "in progress" set. Next the remaining resource requirements for times after T1 are calculated, including the resource requirements for unscheduled activities. This value C2 divided by the resource availabilities, C2/R, when added to T1 gives a heuristic estimate, E, which is based on ignoring precedence constraints.

The following calculations show how a heuristic estimate for state 5 in figure 3.5 is calculated.

Calculations of a heuristic estimate, F, based on ignoring resource constraints:

\[
L = \text{Max} \{12\} = 12
\]
\[
C = \text{Min} \{25, 20\} = 20
\]
\[
C + L = 12 + 20 = 32
\]
\[
M = \text{Max} \{20, 25\} = 25
\]
\[
F = \text{Max} \{C+L, M\} = \text{Max} \{32, 25\}
\]

Calculations of a heuristic estimate, E, based on ignoring precedence constraints:

\[
C_1 = 20 \times 7 + 25 \times 1 = 165
\]
\[
R = 10
\]
\[
C = 20
\]
\[
T_1 = \text{Max} \{165 / 10, 20\} = 20
\]
\[
C_2 = 12 \times 2 = 24
\]
\[
E = T_1 + C_2 / 10 = 20 + 24 / 10 = 22.4
\]

Consequently, the larger value between F and E, which is 32, is selected as the heuristic estimate of state 5.

Figure 3.7 shows that how jobs have been scheduled based on the optimal solution. As it can be seen, the optimal solution is 47 days, which is the improved heuristic estimate of the root of the state space tree.
Figure 3.7: Resource requirements of the optimal schedule obtained by the application of LCBA* to the sample project.
4. THE APPLICATION OF LCBA* TO A BENCHMARK PROBLEM

The purpose of this chapter is to demonstrate the efficiency of LCBA* when it is applied to a benchmark RCPS problem and to gain a detailed insight into the state selection and heuristic updating operation.

We have chosen the problem from Brand, Mayer and Shaffer (1964) which has become one of the benchmark test problems for many algorithms. Brand et al. (1964) used Gomory-Wade code and solved it in 4.9 minutes on the IBM 7094. Davis (1969) solved this problem with the well known MARK-I program in 1.5 minutes. Other researchers such as Patterson (1984) and Bector (1990) have also solved this problem using different algorithms and different computers. They have not presented their CPU time due to the fact that the improvement of computer technology over the period would have rendered such a comparison meaningless. Patterson (1984) considered Davis' algorithm amongst three most successful approaches to the RCPS problem. This problem is presented in figure 4.1. It has 14 jobs, including the two null jobs of "beginning" and "ending", and three limited resources.

At first glance, it is expected that thousands of different decisions have to be examined for the optimum to be found, implying that the manual application of the algorithm to such a problem is impossible. However, it will be shown that LCBA* only tests a very small proportion of these possible decisions, and finds the optimal solution in only a few iterations.

Table 4.1 shows that LCBA* solves the problem using only 25 backtracks. Since every backtrack tests only one path, the question is why the algorithm only needs to examine a very small number of paths among the thousands of possible paths to find the optimal solution. To facilitate understanding, figure 4.2 presents the time duration of the longest path associated with each job. As stated, the longest path of each job plays a key role in calculating heuristic estimates of each state and the longest path of each job will be important in interpreting table 4.1.
Each row in table 4.1 is associated with one backtrack and each column is associated with the starting time of the activities shown. One very important point about this table is that any cell represents one state and that all the characteristics of this state can be found along its row in the cells to left of this state. Consequently, the terms "cell" and "state" can be used alternatively.

Each cell (state) is the parent of the first completed cell on its right and the child of the first completed cell on its left. Associated with each cell is a heuristic estimate which is calculated by means of the value in the second column of the row where the cell is placed. This column, originally, shows the heuristic estimate of the null-schedule. However, a slight modification is needed to calculate the heuristic estimate of each cell based on it. This modification is to subtract the column number of each cell (which represents time) from this heuristic estimate.

For instance in row 1, the heuristic estimate associated with the cell 8 (the state that starts jobs e, d and f at time 8) is 28 which is obtained by subtracting 8, its column
number, from 36, the heuristic estimate of the null-schedule for that row. The reason is that all the cells of one row except for the two last cells are related to a forwarding process. In the forwarding process, when the threshold is set to zero, no change in the length of the project's completion time is tolerated and since the length of the project is the heuristic estimate of each cell plus its starting time (column number) then this value for all cells in the same row remains constant. In other words, the sum of the starting point (column number) and the time left for the completion of the project (heuristic estimate) of each cell in a row are alike.

For the purpose of emphasising the changes that occur to the heuristic estimate of the null-schedule, nothing is written in the second column until such a change occurs. Thus when a cell in this second column is blank, its value is the same as in the previous row.

Figure 4.2: The longest path of each individual job in the benchmark project.
Table 4.1: The states visited by the algorithm to find the optimal schedule of the benchmark problem.

| n | h | 0 | 7 | 8 | 11 | 12 | 13 | 15 | 19 | 20 | 23 | 29 | 38 | 39 | learn |
|---|---|---|---|---|----|----|----|----|----|----|----|----|----|------|
| 1 | 36 | bac | - | edf | - | i | g | * | - | (3) |
| 2 |    |    |    |    |    |    |    |    |    |    |    |    |    |      |
| 3 |    |    |    |    |    |    |    |    |    |    |    |    |    | (2)  |
| 4 |    |    |    |    |    |    |    |    |    |    |    |    |    | (4)  |
| 5 |    |    |    |    |    |    |    |    |    |    |    |    |    | (2)  |
| 6 |    |    |    |    |    |    |    |    |    |    |    |    |    | (2)  |
| 7 |    |    |    |    |    |    |    |    |    |    |    |    |    | (2)  |
| 8 |    |    |    |    |    |    |    |    |    |    |    |    |    | (8)  |
| 9 |    |    |    |    |    |    |    |    |    |    |    |    |    | (8)  |
|10 |    |    |    |    |    |    |    |    |    |    |    |    |    | (2)  |
|11 |    |    |    |    |    |    |    |    |    |    |    |    |    | (2)  |
|12 |    |    |    |    |    |    |    |    |    |    |    |    |    |      |
|13 | 38 | bac | - | edf | - | - | jg | * | - | (2) |
|14 |    |    |    |    |    |    |    |    |    |    |    |    |    | (2)  |
|15 |    |    |    |    |    |    |    |    |    |    |    |    |    | (2)  |
|16 |    |    |    |    |    |    |    |    |    |    |    |    |    | (5)  |
|17 |    |    |    |    |    |    |    |    |    |    |    |    |    | (5)  |
|18 |    |    |    |    |    |    |    |    |    |    |    |    |    |      |
|19 |    |    |    |    |    |    |    |    |    |    |    |    |    | (1)  |
|20 |    |    |    |    |    |    |    |    |    |    |    |    |    |      |
|21 |    |    |    |    |    |    |    |    |    |    |    |    |    |      |
|22 |    |    |    |    |    |    |    |    |    |    |    |    |    | (1)  |
|23 |    |    |    |    |    |    |    |    |    |    |    |    |    |      |
|24 | 39 | bac | - | edf | - | i | g | - | j | h | * | - | (6) |
|25 |    |    |    |    |    |    |    |    |    |    |    |    |    |      |

In this table, the sign "-" has been used to indicate that at that time no job is starting and the sign "*" indicates that although the jobs following it are the best to start, no forwarding to start these jobs occurs and, instead, backtracking is done. We call these cells which begin with *, interrupting cells (states), because they interrupt the process of forwarding. Actually, an interrupting cell indicates that its parent has an underestimating heuristic estimate and can experience learning. Therefore information after * is only used to indicate the best possible decision, which, despite being the best decision, will increase the project's length. The amount of this increase (learning) appears in parenthesis in the right terminal cell of each row. In other words, this
amount in parenthesis shows the change in the project's duration if the last decision, marked with *, had been taken and since this decision (with minimum heuristic estimate) is the best among all other available decisions then it causes the heuristic estimate associated with its parent state (previous cell) to be increased. For instance, in row 1, at the very beginning steps of the algorithm, after starting job g at time 13 the best available decision increases the project's length by 3 units. Therefore the heuristic estimate associated with the state, in row 1 at time 13 (cell 13) is increased by 3 units of time and backtracking occurs to its previous (parent) state, cell 12, which started job i at time 12. As stated, every backtrack changes the row number; therefore, after the occurrence of this backtracking, row 2 begins with cell 12. In this row again, three units of learning occur, backtracking this time to cell 11. The reason is that there is no other available decision at time 13 except starting job g that was examined in row 1 and would have caused 3 units of increase in the project's length.

The next sign (notation) which has been used in this table is "cut", indicating that the state is a dead-end and therefore can be cut and backtracked from. The algorithm determines a dead-end state if there is one job which resource availability will allow to be started and finished by the time of the next decision. This is the rule which we discussed in solving the previous simple problem referred to as the "left shift rule". For instance at row 12 the decision to start jobs b and a, "ba", at time 0 (column 0) is fruitless because by the time of the next decision which is 7, considering resource availability, job c could have been started and finished. Therefore, this decision is ignored and the state associated with it is cut and backtracked from.

Notations in this table have been selected so that by referring to figures 4.1 and 4.2, the reader will be able to follow the entire process of the algorithm that leads to finding and guaranteeing the optimal solution. While following the operation of the algorithm in the table, notice that the sign of * in each row can be considered as the termination sign, stating that the information to the right side of it is used only as extra information and that the backtracking process occurs from the cell before * to its previous cell.
Now we interpret all entries in row 1 of this table which is associated with the operation of the algorithm in its first steps:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The heuristic estimate of the root (null_schedule) is in the column h (the second column), and is 36, being the longest path associated with the job of "o" (starting-job) in figure 4.2. Jobs b, a and c ("bac") are selected to start at time 0. At the time of the next decision, 7, when job c completes, no job, ".-.", can start. Then at time 8, when job b completes, e, d and f ("edf") start. At time 11, again no job, ".-.", can be selected to start. Then at time 12, the job i is selected and starts. and at time 13 the job g starts. Up to this point of time, 13, the heuristic estimate associated with each cell has shown no sign of increase in the length of the project, and hence neither any learning nor any backtracking has happened. Now, at time 15, we encounter the sign *, meaning that the decision in this cell, if taken, increases the length of the project by at least 3 units, and this number is recorded in the last column of row 1. This interrupting cell means starting no job ".-" at time 15. Since this is the best, and actually the only, available decision to make at time 15, and considering the fact that the threshold value is set zero, this increase is not tolerated. After improving the heuristic estimate of cell 13 in row 1, a backtrack to cell 12, which started job i, occurs and row 2 begins. All remaining 24 rows can be interpreted in the same manner.

Having finished describing the notations and operations used in the table, we address the question as to how the algorithm manages to test only such a small number of paths in order to reach and guarantee an optimal solution.

The first interrupting cell appears at 15, causing its parent (cell 13) to experience 3 units of learning and to backtrack to cell 12. As column 12 in rows 1 and 2 indicates, the algorithm considers starting job i at time 12, which has the minimum heuristic estimate among all other possible decisions at time 12. However, it is found that this decision will increase the project's length by at least 3 units. Therefore learning occurs and the heuristic estimate of cell 12 in row 2 increase by 3 units, causing a backtrack to cell 11 in row 2 and a move to row 3. Then in row 3, at time 12, starting
h, instead of i, is considered and again, it is found that it is impossible to start this job at this time without increasing the length of the project by 2 units of time. Then at rows 4 and 5, starting no job at time 12 is considered, and it is found that it is not possible to escape increasing the length of the project by 2 units. Therefore, all possible decisions at time 12 have been tried and all of them have resulted in at least a two unit increase in the length of the project. That is why at row 6 it is not possible to start activities e, d and f ("edf") at time 8 without increasing the project's length by at least two units of time. This fact is reflected in increasing the heuristic estimate of cell 8 in row 7 by two units of time. As it can be seen, in only a few iterations very valuable information was obtained (learned). Seven other backtracks only are needed to find that it is not possible to complete the project in 36 units of time; and this information is obtained at row 13 where an initial heuristic estimate associated with the null schedule changes from 36 to 38, experiencing 2 units of learning. Then only are few iterations are required to find that the length of the project is at least 39 and this appears at row 24. In this row, after establishing the fact that the minimum completion time of the project is 39, searching has been completed up to time 20 (cell 20), without encountering any interrupting state. Then, at the time of the next decision, 23, an interrupting state is encountered. The way in which the algorithm deals with this interrupting situation in row 24 reveals an interesting point which is described in detail as follows.

Information in column 23 of row 24 states that if job h starts at time 19, the project's length is increased by 6 units of time. The reason for this increase is that if job h starts at time 19, it occupies the only one available unit of resource 2, and hence job k which needs one unit of resource 2 cannot start at time 23, when job j which is its only uncompleted predecessor completes. This shifts the starting time of the job k, to at least, the time of the next decision which is 29. Since the longest path to the end of the project associated with this job is 16, it changes the project's length to 29+16 which is 6 units greater than the current estimate which is 39. Therefore, in row 25 the next alternative decision is to start no activity at time 20. At the time of the next
decision, 23, when job j completes, because of the availability of one unit of resource 2, job k can start. Then at time 29, when job k completes, two jobs 1 and h which are the only unscheduled jobs are scheduled. Job 1 completes at time 38 and job h completes at time 39 which marks the end of the project.

A glance at this table reveals that the algorithm works in accordance with the manner in which we would naturally think about this problem. It tries to select the best path until it reaches some difficulty. By difficulty we mean the appearance of an interrupting state, which, although it represents the best decision, increases the heuristic estimate of its previous state (parent). In this case after reflecting on the difficulty the parent learns and the algorithm finds new promising states and goes forward, using all previous and initial information until the goal state is reached.

Table 4.1, also, indicates that learning in some cases activates a chain of other learning which is analogous to the way the human mind operates. Actually the distribution of interrupting states in determining this chain of learning plays a key role. In addition, if a heuristic estimate severely under-estimates then these chains of learning will occur more often. For instance, because of a relatively significant heuristic estimate in row 24, 6 units of learning did not initiate a chain of learning. These two factors: the distribution of interrupting states and the degree of under-estimation by the heuristic estimates, totally determine the efficiency of the algorithm in dealing with a problem.

Table 4.1 also reveals that the application of the algorithm with a threshold of 3, instead of 0, leads to the same solution, 39, using only one backtrack. The reason for this interesting fact is quite clear: Firstly, in rows 1 and 24, columns 0 to 13 are similar. Secondly, when the threshold is set at 3, the accumulation of 3 units of learning does not cause any backtrack to occur. Therefore row 24 will become the first row, if the threshold is 3. Only when job h starts, have 9 units of learning accumulated (3 units for the selection of job g at time 13 and 6 units from the selection of job h at time 20) and backtracking applies to cell 13 which for the first time experienced 3 units of learning rather than cell 20 which because of starting job
h experienced the next 6 units of learning. Then row 25 will be the second and the last row in the table.

The observation that increasing the threshold value causes the number of backtracks to decrease is not something new and was shown in previous chapters. However, the ratio of the decrease (1/24) and maintaining the precision of the final solution were not predictable.

As seen in this example, finding the optimal solution might be very easy and it can even be obtained using only one backtrack. What is important is to guarantee the optimality of the obtained solution. The algorithm guarantees an optimal or a near-optimal solution will be found through the setting used for the threshold value which determines a bracket around the optimal solution within which the solution will lie. This is done in a very efficient manner since it only examines promising paths which are characterised by learning values which often activate learning chains thus producing insights into the decisions which are required as searching continues.

When the threshold value is set as a larger number, interrupting states lose their power of interruption to some extent in proportion to the magnitude of the threshold value. As well, the distribution of the interrupting states effects the behaviour of the algorithm since the closer they are to each other, the more backtracking is needed to learn their precise heuristic estimates.

This algorithm has demonstrated its potential power on this benchmark problem. Any approach which can produce a solution, to this 14-job project, guaranteed to be within 3 units of optimal by testing only two paths and finding the optimal solution with as few as 24 backtracks is of significant importance.
5. CONCLUSION

Despite the effort by OR and AI researchers to develop control methods to overcome the explosive nature of combinatorial search techniques, there are still many problems to be solved. The emphasis of OR on optimal methods and AI on goal seeking has resulted in a promising class of algorithms which generate guaranteed accuracy solutions. This class of algorithms has received very little attention in the literature.

We modified an AI algorithm, LRTA*, and developed it into an OR algorithm called LBA*. Then using these two algorithms as two extreme points of an AI-OR continuum we merged them to produce an integrated problem solving technique called LCBA*. An input parameter in LCBA* determines where, in this continuum between AI and OR, LCBA* is operating. It was shown that the selection of this parameter can be done automatically based on available computer resources.

The fact that LCBA* is based on a very general theorem enables it to be applied to any graph-search problem. The correctness of LCBA* was proved in that very general theorem and demonstrated through applications to many graphs involving three different characteristics.

To examine the performance and the generality of LCBA*, we applied it to a practical combinatorial search problem, RCPS. This type of problem is amongst the most general and hardest problems to solve in the field of scheduling. In the literature survey we described a number of approaches that have been proposed for this problem.

Skiptracking and backtracking algorithms presented for this problem had their special disadvantages. Skiptracking schemes tended to build the tree uniformly downward, and hence did not concentrate branching processes on the terminal areas of the tree. Therefore, these schemes did not generate incumbent solutions and a user did not have any opportunity to terminate the computation if it was decided that the potential gains did not justify further computational expense. These schemes also suffered from redundancy of calculation and the need for high memory for keeping partial schedules as independent data. Whenever a partial schedule was selected, there was
no trace of any information to help the new calculations. The reason simply was that there was no connection between the sequence of partial schedules produced one after another in skiptracking.

In backtracking schemes, to some extent, the above problems were solved. For instance, in these schemes, the ease of structuring partial schedules by creating memory links between every node and its parent was of significant advantage. At each node, when searching forward or backward, consideration was given to either its child or parent node. This arrangement of nodes for expansion helped speed up all calculations by tracing the links between the parent and the child nodes and reduced the memory used to store partial schedules. It was not necessary to store scheduling decisions that a node had in common with its parent and only new additional scheduling decisions for nodes were stored and this linkage between a node with its parent made it possible for a node to use all previous scheduling decisions. However, the backtracking schemes had other disadvantages. In these schemes, the tree tended to grow within a restricted area and before moving to other areas of the tree all previously explored nodes should have been pruned away. Once a node was selected to be expanded, all its children should have been expanded before any other node was selected.

The application of LCBA* to the RCPS problem appeared to offer considerable advantages over previous approaches both in terms of complexity and the size of problems which could be solved. In LCBA*, the process of learning better heuristic estimates prevented growing a tree within a restricted area and together with backtracking methods focused on the entire problem. As a result, this learning process, albeit at the price of computer storage requirements, removed the disadvantage of the large computation time associated with all backtracking algorithms. In brief, LCBA* by utilising a learning mechanism on the one hand and activating a backtracking process on the other offered a power in resolving conflicts in selecting jobs that was present in other approaches.
The general conflict resolution power of LCBA* in selecting states is emphasised by the fact that the selection of the RCPS problem was only based on its generality and NP-completeness and that LCBA* can be applied to any other graph-search problem. Since the root of conflicts in combinatorial problems lies in the necessity of checking all possible options, the way in which LCBA* tolerated near-optimal options and traded speed with accuracy were its main characteristics in dealing with conflict resolution.

We believe LCBA* is an important development in integrating features from both AI and OR fields. It signals how this integration can be beneficial, thus encouraging other researchers in these two fields to use each other's experience in search efficiency.

We manually applied LCBA* to one practical RCPS problem, emphasising its efficiency in conflict resolution. We believe other researchers can apply LCBA* to other various state-space search problems to examine closely how accuracy can be traded with speed. The phase-transition concept does not leave any doubt that in different problems different trading efficiencies may exist and that there are many search problems that have very promising rates. For instance, suppose the optimal solution of a problem is 42 and among millions of different decisions there are only tens of them leading to 42, whereas there are thousands of decisions leading to 43. In this case the application of LCBA* with threshold value of 1 will transform the problem from a difficult phase to a simple one by sacrificing 1 unit of accuracy.

It is suggested that for solving an RCPS problem a computer program based on LCBA* be developed. The idea of keeping information about states in this program should be the same as the tabling concept used in table 4.1 and such a program will put a stop to a dead-end that for decades has made this problem intractable. It is recommended that this program be written in such a way that, based on available computer resources, the threshold parameter is determined and traded with speed when needed.
We also consider that LCBA* can be applied to other combinatorial problems of a different nature. In state-space search problems, for which LCBA* was originally designed, a sequence of operators to connect the initial state to the goal is sought whereas in problem reduction representation, finding a strategy is required. Particular problems in this area for the application of LCBA* include: theorem proving, diagnosing and even explaining or justifying results. The reason for identifying these problem areas is that, in these problems, the problem solver has a good chance to learn from experience. For instance in a theorem proving problem, the problem solver learns that applying certain rules to some facts is promising and can use this experience during the search process. In order to apply LCBA* in the area of problem reduction representation two major modifications are necessary. The first modification is related to the way it learns and the second one is related to its backtracking routine. These two parts can be modified so that instead of a path a tree will result from the application of the technique to a problem. These modifications should be made so that the major ingredients of learning and backtracking at appropriate times continue to be central features.
REFERENCES


Bell, Collin E. and Park, Kwangho (1990), Solving Resource-constrained project scheduling problems by A* search, Naval Research Logistics, 37, number 1: 280-318.


Kim, Sang O. (1989), Heuristic Famework For the Resource Constrained Multi-
Project Scheduling Problem, *Computers & Operations Research*, 16, number 6: 541-
556.

Kingston J. H. (1990), *Algorithms and Data Structure: Design Correctness Analysis*,
Sydney: Addison-Wesley.

Number 1: 97-100.

Boston: Pittman Advanced Publishing Program.


Li, K. Y. and Willis, R. J. (1992), An Iterative Scheduling Technique for Resource-


Nowicki, Eugeniusz and Zdrazalka, Stanislaw (1987), *Project Scheduling with Multiple, Doubly Constrained Resources*, Archiwum Automatyki i Telemechaniki, 32, number 1:80-93.


AMENDMENT

Following phrases should be added to the thesis.

Page 6 Line 3: **Declarative vs Procedural Representation**

Page 10 Line 11: **State Space vs Problem Reduction Representation**

Page 45 Line 20: (Only are those children of x are evaluated which are not currently in OPEN list)

Page 48 Line 3: This grid Problem can be converted into a simple shortest path problem and be solved in $O(n^2)$ time, however, our purpose of applying LBA* to it is only to demonstrate how the algorithm works.

Page 52 Line 4: Notice that step 4 prevents the creation of any loop because it only evaluates the children which are not currently in OPEN list. This will guarantee that the worst case never exceeds n.s figure.

Page 78 Line 16: As was stated, our algorithm can be considered as an integration between exact and heuristic methods.

Page 134 line 20: Notice that if at time 15, no activity starts the unscheduled activities take 20 days to be completed and the first decision about scheduling some of these activities is made at time 19. This increases project duration to $(20+19)$ which shows 3 units increase in the current project length which is 36, being considered as 3 units of learning.

Page 137 Line 24: Although Davis has not mentioned how many paths have been examined by his method, it is clear that 1.5 minutes of CPU time indicates examining at least hundreds of paths. LCBA* is the first method by which this benchmark problem has been solved with hand calculation. Bell and Park (1990) have also tried to do so but they have had to simplify the benchmark problem by changing resource requirements of some of the activities as well as their durations. We also solved their simplified problem by examining only 8 paths and showed that it was much simpler than the original problem. As well as this benchmark problem, we have solved a simple problem which has been used by Demeulemeester Eric and Herroelen Willy (1992) as a numerical example to demonstrate how their procedure works. This problem has only 9 activities and 1 limited resource. Our algorithm found and guaranteed the optimal solution of their problem by examining only one path, whereas their procedure had examined 14 paths to do so.

Page 141 Line 17: We suggest that the limitations of this research which are the lack of implementation of a computer program and generalising its application to problem reduction representation be the subject of further research.