The development and implementation of a learning heuristic algorithm for project scheduling problem

Young Je Park
University of Wollongong

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:

This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author.

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.

Unless otherwise indicated, the views expressed in this thesis are those of the author and do not necessarily represent the views of the University of Wollongong.

Recommended Citation

The Development and Implementation of A Learning Heuristic Algorithm for Project Scheduling Problem

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

MASTER (HONS) OF COMMERCE

from

UNIVERSITY OF WOLLONGONG

by

YOUNG JE PARK

Department of Business Systems

1999
Abstract

Scheduling is the cornerstone of any fundamental application of Artificial Intelligence (AI) and Operations Research (OR). In general, scheduling problems are NP-hard. Many methods have been proposed and implemented in the past. Early approaches were only able to solve simplified versions of this problem. For approaching larger and more complicated problems, many heuristic methods were devised to find good solutions or simply feasible solutions.

This research aims at developing an approach to implement an admissible learning heuristic search algorithm for solving resource-constrained project scheduling (RCPS) problems. The algorithm LBA* uses heuristic estimates as the criterion to search through solution space, and is featured with its heuristic learning capability in updating the solution path. This approach is developed using Object-Oriented design technique, and the system is written with C++ language and runs with IBM PC.

The performance of this approach was tested using the commonly accepted 110 benchmark problems designed by Prof Jim Patterson. Although computationally expensive, this approach performed fairly well on a wide variety of problems. Most problems were solved in less than 99 seconds. In addition, this research attempted to identify those factors which are likely incur lengthy computational times. The statistical analysis showed that there is a high predictability that, the performance of our approach deteriorates as problems' characteristics become less related to heuristic estimation.
# TABLE OF CONTENTS

## CHAPTER I.
INTRODUCTION  1

## CHAPTER II.
PREVIOUS LITERATURE REVIEW ON THE RCPS PROBLEMS  5
  2.1 Exact Methods  6
      2.1.1 Integer Linear Programming (ILP) Models  7
      2.1.2 Enumeration Procedures  11
  2.2 Heuristic Methods  18

## CHAPTER III.
HEURISTIC SEARCH METHODS AND THE LEARNING BACKTRACKING A* ALGORITHM  25
  3.1 Brute-Force Searches  26
  3.2 Heuristic Search  30
      3.2.1 Heuristic Search Settings  31
      3.2.2 Heuristic Search Algorithms  32
      3.2.3 The Learning Backtracking A* Algorithm  38

## CHAPTER IV.
SYSTEM DESIGN AND DEVELOPMENT  41
  4.1 Formation of the RCPS Problem  41
  4.2 State Space Representation of the RCPS Problems  43
      Definition of States  44
      State Transition Operator  45
      Transition Costs  45
      An Approach for Estimating Initial Heuristic  46
4.3 System Design and Development 46

4.3.1 Object-Oriented Design 48

4.3.1.1 Class Description 49

4.3.1.2 Relationships among Classes 52

4.3.2 Implementation Description 54

4.3.2.1 Initial Heuristics Evaluation 56

4.3.2.2 Neighbouring States Identification 59

4.3.2.3 Heuristic Learning Test 67

4.3.2.4 Forward Scheduling 69

4.3.2.5 Heuristic Learning and Backtracking 72

CHAPTER V.

SYSTEM PERFORMANCE WITH PATTERTON’S 110 PROBLEMS 73

5.1 Computational Results of Patterson’s 110 Problems 73

5.2 Statistical Analysis 75

CHAPTER VI.

CONCLUSION 83

BIBLIOGRAPHY 86
Artificial Intelligence (AI) and Operations Research (OR) are both concerned with the solutions of scheduling problems. In many years, these two communities have studied extensively to develop and improve techniques to solve the problems. The general definition of the problems is defined by Baker (1974) as the allocation of resources over time to perform a collection of tasks. In general, scheduling problems are known as NP (Non-deterministic Polynomial) -hard. This indicates that there are no known techniques or algorithms for finding optimal solutions in polynomial time. In the early studies, OR techniques focused on the producing optimal solutions. Solutions found by OR techniques are optimal but may require high computational expense when the problem size grows or when additional constraints are added. As early OR approaches experienced difficulties with finding optimal solutions in reasonable time, the effort has turned to developing AI techniques, which approximate optimality while incurring significantly less computational expense.

The problem addressed in this research is the non-preemptive, Resource-Constrained Project Scheduling (RCPS) problem in which resources are renewable on a per period basis. In general, a project consists of a set of activities coordinated by precedence relationships. The objective is to complete the project by the minimum time permitted by the technological precedence relationships. However, in practice activities require resources during progress. This makes the scheduling problem difficult to solve, because the issue of allocating scarce resources among competing
activities must be considered in optimising a specific objective. This problem is
known as the RCPS problem and is also NP-hard.

The main objective of this research is to develop an approach to implement a learning
heuristic search algorithm LBA* (Learning and Backtracking A*) for solving non-
preemptive resources-constrained project scheduling (RCPS) problem. This RCPS
problem has been identified as the most complex and general problem in the field of
scheduling and occurs not only within industrial organisations but also in many
business enterprises.

The detail of this research is presented in five chapters. Chapter II provides a review
of relevant literature on the RCPS problems. This chapter presents some of the major
works in the RCPS problems which include both exact and heuristic methods. It is
clear from the literature that the RCPS problems are the most difficult and complex
problems in the area of scheduling problems.

In Chapter III, various search techniques in the field of AI are investigated. The
learning heuristic search algorithm (LBA*), which is chosen for implementation in
this research, is introduced. Two major directions in the development of the AI
search techniques, Brute-force search and heuristic search, are described in detail.
The development of the LBA* algorithm is described in detail. In essence, LBA*
algorithm was developed by incorporating a backtracking mechanism into a search
technique LRTA*, which was developed by Korf (1990). The major contribution of
LRTA* is the repetitive application of heuristic learning along the search process.
The LBA* algorithm inherits that property from LRTA*, and further utilises a
backtracking operation to review the search path when heuristic value of a state is improved.

In Chapter IV, an approach to implement the LBA* algorithm for solving the RCPS problems is presented. At first, it provides a description of how a RCPS problem is generally formulated. It is widely used that a project can be depicted as an acyclic activity-on-node (AON) graph, and is associated with a duration time and a set of resource requirements of each activity. Then it is described in detail how the RCPS problem can be represented as a state-space problem. This is important because the LBA* algorithm is a real time heuristic search algorithm that searches through the solution space of a state-space problem. Next, it shows how an approach for implementing the LBA* algorithm can be designed using the object-oriented design approach. In this section, the functions of the six classes and their interactive relationships are described. Finally, five main functions of the system are explained using flowcharts and codes.

Chapter V contains a computational evaluation of the procedure developed in chapter IV. Computational results of Patterson's 110 problems are provided, and the results are compared with the result published by Bell and Park (1990). Furthermore, statistical analysis is provided to identify those factors which can assist in identifying those problems for which a longer computational time will likely be required. Three factors are constructed to perform the statistical analysis. These factors include Project Complexity, Heuristic Tightness, and Resource Constrainedness.
In Chapter VI, the concluding remarks are provided, and some limitations of the current research as well as suggestions regarding the future directions of this research are indicated.
Chapter II. Previous Literature Review on the RCPS Problems

One of the major research area in Operational Research (OR) and Management Science (MS) has always been the Project Scheduling under Resource Constraints (RCPS). Since RCPS problem has been extensively studied since the early 1960s, it is a relatively well understood class of problems. A commonly discussed version of the RCPS problem can be a set of starting times for the activities of the project in such a way that all involved precedence and resource constraints are satisfied and the total completion time is minimised.

Numerous papers have been written describing various formulations, scheduling techniques, and optimisation algorithms for the RCPS problem. The progression of these approaches to solving the problem moves along two dimensions, both having to do with the nature of the desired solution. Early work focused on the production of optimal solutions, according to singular objective functions. However, many difficulties were encountered with finding optimal solutions for relatively large-scale problems, hence there has been a transition towards developing methods for obtaining near-optimal solutions, which would approximate optimality while incurring significantly less computational expense. The early approaches, which produce optimal solutions using mathematical programming or other rigorous analytical procedures (Davis, 1973), are called exact methods. In contrast, the approaches, which aim at producing near-optimal solutions using some rule of thumb or heuristics in determining
scheduling priorities among jobs competing for limited resources, are called heuristic or inexact procedures.

The main content of this chapter focuses on discussing some major works of these two major classes for scheduling project activities with limited resources. With regard to the exact methods, existing procedures are divided according to whether they utilise some form of integer linear programming or a variation of some enumerative techniques. With regard to the heuristic procedures, several approaches, which use one or more heuristic rules, will be discussed.

2.1. Exact Methods

Exact methods are known as the methods that are guaranteed to find a solution if it exists, and typically provide some indication if no solution can be found. When the RCPS solutions were first proposed, simple mathematical models were used with exact methods for solving the RCPS problem. Given a problem, the exact methods find the optimal solution every time they are run. However, the methods typically become impractical when faced with large sets of constraints or problems of any significant size. Many researches have actually concluded that solving the RCPS problem using exact methods is not realistic (Kelley 1963, Brand 1964, Norbis and Smith, 1986). For example, in the early years, Kelley (1963) concluded that formulating and solving the RCPS problem from a mathematical point of view is quite difficult, because it lacks explicit criteria for obtaining a solution for the problem and it is difficult to produce solutions using mathematical techniques in a reasonable time. Furthermore, Norbis and Smith (1986) noted that mathematical programming procedures for the RCPS
problems have proved to be unsuccessful in dealing with problems of realistic size because of their NP-Completeness. As a result of unsuccessful attempts using mathematical programming techniques, researchers were focused on enumerative approaches to solve the problem. A number of various forms of enumerative techniques have been developed for producing optimal solutions with reasonable time frames, yet the success in applying these techniques to realistically-sized problems has not been widespread.

In this section, the known optimal solutions procedures are described by dividing them into two major classes; Integer Linear Programming (ILP) models and enumerative procedures.

2.1.1. Integer Linear Programming (ILP) Models

Of all the available techniques in the RCPS, ILP has been one of the most extensively investigated and widely used methods, despite some limitations behind this method. In the early research years on the RCPS, the problem was generally formulated in traditional linear or integer programming form. However, significant simplifications of the problem were required as the formulation of the problem from a mathematical point of view was difficult and involves many binary integer variables. Earlier ILP models were formulated but no attempt was made to implement the models (Burton 1967), because of the complexity of the problem. Wiest (1964) presented the first ILP formulation for the RCPS problem. 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 whether or not a job is being processed for each period. He
pointed out that his approach was not feasible for large projects and even a small project with 55 jobs and 4 resource types would require 6,870 constraint equations and 1,650 variables. Subsequently, Hadley (1964) also proposed an ILP model but again it was not implemented. The objective function he proposed was the minimisation of the total project cost, including the cost of resource usage on regular time and overtime and the cost of changing resource usage level. In his book, he concluded that the formulation he proposed was impractical for any realistic problem as the number of constraints became huge and thus finding a solution was quite impossible.

Brand, Meyer and Shaffer (1964) presented an application of ILP formulation to solve a multi-resource constraint single-project scheduling problem, which was well quoted in many publications. The example problem had 14 jobs, each job requiring either 0 or 1 unit of each of 3 resource types. The computational results showed that only 7 jobs required non-zero amounts of the 3 resources, and 2 of the 7 jobs required more than 1 resource type. As a result of these simplifications of the problem, the problem required only 57 constraint equations and 33 binary variables. A total of 4.9 minutes execution time was required to obtain a solution from the IBM 7094.

The integer programming formulation presented by Pristker, Watters and Wolfe (1969) was the first formulation that was implemented. They proposed a fairly general zero-one linear programming formulation for multi-project scheduling with multi-resource constraints. Their formulation could accommodate for a wide range of real-world situations such as due dates, job splitting, and resource substitutability. Three different objective functions were considered: minimum total throughput time of all projects, minimum make-span, and minimum total delay penalty for all jobs. The results showed
that the number of variables increased very rapidly with problem size, and their formulation could only be used on very small problems. A sample problem, consisting of 8 jobs and requirements of three resource types, required 37 constraint equations and 33 variables, and required only 2.3 seconds on an IBM 7044.

An approach using graph theory was proposed by several authors (Balas 1969, Gorenstein 1972). Balas (1969) utilised the concept of a disjunctive graph for the job shop scheduling problem. In the case that there is just one machine of each type, then two activities i and j, that need the same machine, can not be processed simultaneously. To avoid this possibility, a disjunctive pair of arcs, i-j and j-i, is added to the original graph, creating a disjunctive graph. Taking a feasible set of arcs from each disjunctive pair yields a graph in which the longest path is a feasible solution to the original problem. Balas (1970) later generalised this concept to cater for more than one machine of each type. Some stability conditions were presented to obtain a feasible solution, but the implementation of an algorithm and computational results are not presented. Gorenstein (1972) further developed an algorithm to the works of Balas. Gorenstein's algorithm is based on a maximum flow in a bipartite graph. A feasibility check is used to determine whether the resource constraints could be met by any particular network representation of the project. The author also presented results of 7 test problems. The computational results show a high degree of efficiency in the algorithm for these problems, which involved up to 5 jobs and 8 machines with few activities per job. However, no data on computation time is given.

Fisher (1973) made the first attempt to use a Lagrangian relaxation to an integer programming formulation of the resource-constrained network scheduling problem.
Although the solution of the relaxed problem is generally infeasible for the original problem, it can obtain strong bounds by adjusting the multipliers iteractively and develop a branch and bound formulation that uses these bounds as lower bounds in the solution of the network scheduling problems. Talbot and Patterson (1978) proposed an integer programming formulation that consists of a systematic evaluation of finish times for all possible jobs for each task in the project. Fathoming rules of this formulation are based on the notion of network cuts, which are used to eliminate many partial schedules from explicit consideration. The results show that this formulation pruned partial schedules earlier and consequently used much less computer storage. In IBM 370/168, solution times are ranged from 0.01 to 37.41 CPU time, with a mean of 6.47 seconds.

A more recent research in linear programming procedures was presented by Deckro, et al. (1991). They developed an optimal integer programming formulation for solving multi-project, resource constrained scheduling problems by applying a decomposition algorithm developed by Sweeney and Murphy (1979). Deckro, et al. stated that the use of the decomposition algorithm provides two advantages over a direct optimisation method: (1) the capability of solving large problems, and (2) the option of using the decomposition approach as a heuristic. The Sweeney and Murphy's decomposition procedure starts by breaking down a problem into sub-problems. By decomposing a problem into sub-problems, the generated sub-problems are characterised by all of their constraints being special ordered sets; that is, where exactly one variable must be non-zero in each constraints. The remaining constraints in the master problem serve as coupling constraints in the decomposition procedure. Through this procedure, the number of variables and constraints required in the solution can be reduced.
Optimality tests are used to provide an upper and lower bound for each feasible solution. These bounds can be used as heuristic values, which automatically provide a performance guarantee for the incumbent solution.

2.1.2. Enumeration Procedures

Since early attempts at using ILP to solve the general RCPS problems did not meet expectations, researchers were focused on enumerative approaches for solving the RCPS problems. A number of different approaches have been developed for producing optimal solutions within reasonable time frames, although the success in applying these approaches to practical-sized problems has not been widespread. Approaches based on enumerative procedures are based on the idea of intelligently traversing a developing search space. These approaches start with a solution of some kind (often an optimal solution to a simplified version of the same problem), and then progressively developing, through the tightening of constraints, the best existing solution, until a solution to the desired problem is found. This progression towards the optimal solution involves the generation of a tree structure rooted with the original solution to connect it to all of its feasible, tighter, or more complete, partial solutions. The search is bounded using the process of pruning, where all partial solutions that are probably unable to lead to better quality solutions than the best existing solution are removed from further consideration. The pruning process guarantees that the final solution to the original problem will be optimal. The effectiveness of the enumerative approaches depends on a number of important factors such as the quality of the original root solution, the approach used to determine a new lower bound in the process of pruning all non-optimal solutions, and the approach needed to determine
which solution the sub-problem to expand at any point. The relevant researchers who have experienced with this solution methods will be discussed in turn.

Variations of the enumerative solution methods were first proposed in 1960s (Mueller-Mehrbach 1967, Johnson 1967). Mueller-Mehrbach (1967) presented the first optimal procedure for the single resource-constrained network scheduling problem. His implicit enumeration approach is based on the branch-and-bound form for the job-shop category of network problem. The approach accommodates multiple resources per project, but only one resource type per job is allowed. In August 1967, Johnson (1967) presented another similar approach called BETINA (Bounded Enumeration Technique in Network Analysis). As with the Mueller-Mehrbach method, Johnson’s method is also limited to one type of resource per job. The fundamental principle present in both methods lies in the efficient exploration of the decision tree of the problem, pruning the search along a particular path where a minimum bound for the solution currently being examined exceeds a feasible solution already obtained. The BETINA-program was written in FORTRAN and run on the IBM 360/65, and Johnson reported that computational times tend to rise rapidly with the size of the problem. He concluded that his technique is an unreliable optimisation procedure for most real-world scheduling problem.

Unlike the other two approaches, Davis (1969) devised an enumerative approach for more general cases of the RCPS problem. Davis’s enumerative approach was designed to handle problems involving several resource types per job. Moreover, it allowed job interruptions and variable level of the job requirements of each resource type over the duration of each job. To test the feasibility of the conceptual approach involved, Davis
developed a computer program MARK I on the IBM 7094, and reported that the MARK I program found the optimal solution for 48 problems out of 65 problems, each consisting of about 20 activities and requirements of 3 resource types. For the remaining 17 problems, no optimal solution could be obtained, because the available storage capacity was exceeded before a final solution could be found. For the 17 problems, however, an approximate solution was found which was at least one day shorter than the critical solution.

Patterson (1984) provided an excellent overview of exact procedures for solving multiple resources constrained, single project scheduling problems. In Patterson’s overview, three solution procedures were described and evaluated: the bounded enumeration techniques proposed by Davis and Heidorn (1971), the branch and bound solution approach presented by Stinson et al. (1978), and the implicit enumeration procedure presented by Talbot (1976). Each of these three solution procedures are now discussed.

Davis and Heidorn (1971) presented an enumerative procedure called “bounded enumeration”, which was applicable to the multiple resource-constrained scheduling problem under such assumptions as variable levels of resource requirements and job splitting. The procedure employed techniques originally developed for solving the assembly line balancing problem. The method initially divides each of the activities of the project into a series of unit duration tasks, the number of unit duration tasks created for an activity being equal to the original activity duration. Precedent constraints controls the linking of the activities, and immediate precedent constraints force each sequence of subtasks to be performed together to prevent task splitting.
The next step is to produce a table of feasible subset schedules representing the set of unit duration subtasks that can execute at each time period in the schedule. Resource constraints are ignored at this stage. This table is used to construct a directed graph A-network that represents the progression of precedence-feasible and resource-feasible schedules across time periods. The A-network for problem solution is created by first generating the nodes of the network, each node representing a precedence feasible assignment for some subset of the unit duration tasks. Hence, resource restrictions are originally relaxed in their procedure in generating the tree of partial solutions. Arcs are then added to the network, each arc connecting a precedence and a resource feasible assignment for some subset of the unit duration tasks in the project. The number of states at which nodes are added in their approach is equal to the original critical path length, each level corresponding to a different period in network construction. Dynamic programming is used to determine the shortest route in the generated A-network of partial solutions, resulting in the determination of the minimal schedule length for the resource constrained problem. All possible feasible solution schedules are generated in each time interval and bounded by target duration. If the optimal solution can not be found, then the target duration is incremented by 1 and the same procedure is repeated.

The well-known branch-and-bound (skiptracking) procedure developed by Stinson, Davis and Khumawala (1978) generates the tree by progressively scheduling activities forward from the start of the schedule. Each node is expanded by creating a new node for each possible combination of activities that could be scheduled according to both the precedence and resource constraints. At each point, the start time is increased by the duration of the shortest activity currently in progress. When no more activities
remain to be scheduled, a complete schedule is obtained. Also two pruning rules are incorporated in the procedure to effectively reduce the size of the tree:

(1) dominance pruning and

(2) lower bound pruning.

The first rule was initially developed by Johnson (1967), and can be introduced into the branch-and-bound procedure as follows:

Partial schedule X dominates partial schedule Y if all the following four conditions are met:

1. The unscheduled activities in X are a subset of those in Y.

2. The set of activities currently in process (active set) in X are a subset of those in Y.

3. The projected 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.

4. The current partial schedule time of X is equal to or less than that of Y.

Schräge (1970) further refined this rule and stated that a partial schedule can be pruned, if any activity in a partial schedule can be started earlier without violating either a precedence or resource constraint. This rule was later described as “Left-Shift” rule.

The application of the lower bound pruning is accomplished in one of following three ways:

1. A precedence based lower bound is computed as the earliest time that any unscheduled activity can be started plus the critical path length of the remaining unscheduled activities;
2. A resource based lower bound is computed as the sum of the earliest start
time for an unscheduled activity plus the work-period requirements for the
unscheduled activities divided by the per-period availability of the resources
for the resource yielding the maximum remaining length; and

3. A critical-sequence lower bound is computed by simultaneously considering
both precedence and resource constraints.

The largest of the three bounds is taken as the lower bound for the partial schedule and
is used to eliminate inferior partial schedules from further consideration. The authors
evaluated their procedure using project type problems ranging in size from 23 to 43
activities. A total of 240 problems were solved on an IBM 370/155. Mean solution
time ranged from 0.20 minutes for projects with 23 activities to 5.84 minutes for
projects with 43 activities.

Talbot (1976) developed another implicit enumeration procedure (Backtracking) that
systematically enumerates all possible job finish times for the activities of a project.
This method uses integer variables which requires much less memory over other
procedures. Fathoming rules are used to eliminate candidate schedules that can not
possibly lead to improved schedules. These rules were much stronger than those used
with a general implicit enumeration approach. Talbot (1982) also investigated the
RCPS problem in a non-preemptive case in which the duration of a job is a function of
resource allocation. In the report, he suggested two-stage solution procedure. In the
first stage, the activities and modes have been ordered in accordance with a heuristic
scheduling rule, and in the second stage, the improved solutions have been searched by
implicit enumeration in a backtracking strategy. To test the efficiency of the approach,
the enumerative scheme was programmed on FORTRAN. A total of 100 problems
were tested to evaluate 8 heuristic rules, and the author reported that a MINSLACK (Minimum Slack) heuristic results in less computational time in scheduling for the optimal solution in comparison with other rules tested.

Christofides, Alvarez-valdes, and Tamarit (1987) presented a branch-and-bound algorithm, which is based on the idea of using disjunctive arcs for resolving resource conflicts. In the report, four lower bounds were examined. The first is a simple lower bound based on longest path computations. The second is based on the Linear Programming relaxation with the addition of cutting planes. The third bound is based on a Lagrangean relaxation of the formulation. The last bound involves a problem which is a generalisation of the longest path computation and for which an efficient algorithm is given. The last bound is based on the disjunctive arcs used to model the problem as a graph. The significance of resource constraints in the computational evaluation was also discussed, and concluded that using a ratio of total resource requirements to the total available resources is essential to qualify the significance. They randomly generated 40 problems, each with 25 activities and 3 resources. The problems were solved on a UNIVAC 1100 computer. For loose constraints, the mean solution time was 1.95 seconds, and for tight constraints, the mean solution time was 5.65 seconds.

Demeulemeester and Herroelen (1996) presented a branch and bound procedure for optimally solving the pre-emptive RCPS problems. The procedure is based on a depth-first search strategy in which nodes in the solution tree represent resource and precedence feasible partial schedules. Unlike other approaches, it starts by creating a new project network in which all activities are split into sub-activities, where the
number of these sub-activities is equal to the duration of the original activity. Each of these sub-activities has a duration of one, and resource requirements are equal to those of the corresponding activity. A critical path based lower bound and five dominance rules are also proposed.

Sung and Kim (1997) presented a branch and bound procedure for solving a RCPS problem, where various operating modes are allowed to perform each activity in the project and all activities are non-preemptive. With the objective of minimising the makespan of the project, two lower bound computation procedures are derived for the associated tree search algorithm, where a depth-first search strategy is utilised.

All the works of the authors mentioned in this section dealt with exact methods and similarly the works of researchers using heuristic methods will now be discussed in turn.

2.2. Heuristic Methods

While optimal solution procedures represent well-understood and established problem solving methods, many of the formulations and algorithms for solving the RCPS problems optimally can be infeasible, and implementation of the formulations can be difficult to specify and understand. This is particularly true of enumeration approaches which are often applicable where analytic and iterative procedures can not be found. In addition, the introduction of real-world dynamic complications into large-scale RCPS problems greatly increases the difficulty involved in their solution.
As mentioned previously, recognition of the problems of trying to achieve optimal solutions through exact methods led to a shift in focus towards other methods for obtaining near-optimal, or simply good solutions to the RCPS problems at much less computational effort, in terms of both time and memory. Heuristic methods represent another important approach in addressing the RCPS problems. A heuristic is a rule that specifies how to make a decision given a particular situation. Within the context of the RCPS, heuristics are often referred to as rules or dispatch rules, which schedule those tasks having the earliest possible starting times, or the least available amount of slack time. Using the heuristics, the heuristic approaches often provide the decision-maker simplicity in understanding, hence greatly increasing the chances of implementation. The heuristic approaches operate by applying a heuristic or a collection of heuristics to the set of unsolved sub-problems comprising the RCPS problem to determine the priority of each individual sub-problem. The following works of authors are chronologically discussed.

Kelley (1963) as a pioneer on heuristic methods outlined two single-pass strategies, serial and parallel, that require modest computational effort and can provide useful results. In a serial strategy, a feasible schedule can be constructed by considering the activities in the order of their appearance on such a list and scheduling them one at a time as early as precedence and resource constraints permit. The nodes of a project are numbered so that for each arc the head node will have a larger number than the tail node. For a given set of node numbers, the numbering procedure is generally not unique, since several activities can share the same head-node. The numbering among activities can be obtained at random or by a priority function such as resource requirements, activity duration, total float, weighting factor, or some combination.
Since the construction of a feasible schedule by serial methods is computationally rapid, it would be possible to try several combinations of sorting factors and to select the best schedule among those that are constructed. The parallel methods, on the other hand, construct a feasible schedule by scheduling several activities at once. At any point in time during the construction of a schedule, there exists a set of activities that can be scheduled and whose predecessors are complete. From this schedulable activities set, a preferred subset can be scheduled up to the resource capacities. Hence, a new set of schedulable activities is encountered and the preferred subset can be reconstructed. By using this method, a schedule can be created by proceeding chronologically forward. In general terms, the serial and parallel strategies represent the basic heuristic approaches to the solution of large-scale problems.

Crowston (1968) tested nine different heuristics for 65 projects ranging from 40 to 230 activities with three resource types, and concluded that the Minimum Late Start Time was more effective heuristic than all other heuristics. Cooper (1976) presented a list of 26 priority rules for scheduling order on parallel processors. Thesen (1976) provided a multi-dimensional knapsack algorithm for determining combinations of jobs to schedule at given points in time. Any of the existing priority measures could be used in the knapsack profit, and knapsack weight limits can be used to ensure that resource constraints are respected.

Fendley (1968) presented a heuristic method for scheduling multiple PERT projects with probabilistic activity times. He tested several basic dispatching strategies under a variety of performance measures. His computational results showed that for makespan and tardiness performance measures, the dynamic minimum total float priority
assignment was particularly effective. He outlined that this priority rule produced high utilisation of resources and led to relatively uniform behaviour among the different projects.

Elsayed and Nasr (1986) proposed two heuristics for allocating resources to activities with single resource constraints using critical path methods. Norbis and Smith (1988) presented a multi-objective formulation and corresponding heuristic procedure for dynamic resource constrained scheduling problems. For the dynamic multi-objective scheduling problem, a multi-level, multi-priority schema was used. The method is based on a set of priority rules which consider resource utilisation, network critical path, and job due dates.

Since 1980s, many researches have been focused on multiple heuristic rules. The combination of rules can also be used as a single heuristic rule. Ulusoy and Ozdamar (1989) presented an efficient heuristic procedure, which is based on hybrid heuristics. In this procedure, heuristic rules whose performances are tested against problem characteristics are selected according to their success and popularity in previous usage. A resource conflict set in which a set of activities competes for the same resource at a certain scheduling time-point is created, and priority is given to an activity in the conflict set according to the weighted combination of its resource utilisation ratio and the number of its immediate successors. For any combination of precedence and resource utilisation weights, a priority list is calculated. The scheduling algorithm is applied using this list, which results in a project duration. Thus different weight combinations might result in different project durations. A search procedure is performed to determine the best combination of weights which can result in the least
project durations. The advantage and effectiveness of using this method lies in its priority distribution to activities which enables it to deal with almost all types of problems more successfully than other widely used heuristic rules.

Ulusoy and Ozdamar (1994) also discussed a heuristic approach for doubly-constrained project scheduling problems. With this method, an activity is permitted to operate in one of its modes, each of which represents the trade off between different choices of resource requirement types and operating durations. A Local Constraint Based Analysis (LCBA) where the selection of activities and their respective modes is made locally at every decision point is used. The authors emphasised that the approach can be utilised in a dynamic environment where resource absenteeism, activity duration changes and readjustment of the project network configuration are accounted for. The procedure has been tested on a set of 95 problem instances with 20 to 57 activities, one to six renewable and one non-renewable resource types. The constraint-based approach produced an average increase over the precedence-based lower bound of 59%. The average computational time requirement ranged from 20 to 25 seconds on an IBM 70/386 PS/2 computer.

Unlike most of previous researches on heuristic procedures for solving RCPS problems, Boctor (1990) presented multi-heuristic procedures employing both parallel rules and serial rules. He suggested that using a pre-selected combination of heuristic rules is necessary to obtain the best solution. Based on 66 projects of different degree of complexity, the performance of 13 sequencing rules were evaluated in order to identify the most efficient heuristic rules. From these results, the best combinations of heuristic rules were determined. In the report, it was shown that a combination of
three heuristics had a relatively high probability of producing the best and even the optimum solution. These probabilities were estimated to be as high as 85% for the best solution among 13 methods used, and 75% for the optimum solution. Furthermore, Boctor (1993, 1996) developed two heuristic solution procedures for the RCPS problem with renewable resource types only. The first heuristic proposed in Boctor (1993) is a single-pass approach which employs a parallel scheduling scheme. In the procedure, an activity can be in the decision set if all its predecessors are completed and it can be started in at least one of its modes at the current schedule time. Activities are selected from the decision set in the order given by the MSLK (Minimum Slack) priority rule. A chosen activity is then scheduled in the mode with shortest duration. In Boctor (1996)'s procedure, all possible activity-mode combinations which can be started at the schedule time are evaluated by applying a lower bound on the increase of the makespan. On his own set of 240 test problems, with 50 and 100 activities and up to four renewable resource types, Boctor reported an average percentage deviation from the precedence-based lower bound of 36.8% for the single-pass procedure presented in 1993, of 34.4% for the heuristic presented in 1996.

Li and Willis (1992) presented a new heuristic procedure, which is based on a serial iterative method, for solving RCPS problems. The main feature of this procedure is that a project is scheduled forwards and backwards iteratively until a better schedule results, or no further improvement in the project duration can be obtained. In the forward schedule, starting times of activities are based on the earliest starting time and conversely, in the backward schedule, starting times of activities are based on the latest starting time. The procedure is then scheduled forward and backwards iteratively until no improvement can be found.
Slowinski et. al. (1994) presented a decision support system for a multi-objective RCPS problem which combines three different heuristic solution strategies: a single pass approach, a multi-pass approach, and simulated annealing. The core feature of all solution procedures is a precedence-feasible activity list obtained by one of twelve priority rules. The single pass approach deterministically selects the next activity on the list and schedules it in the shortest resource-feasible mode at the earliest period possible. On the other hand, the multi-pass approach randomly selects one of the next precedence-feasible activities on the list for scheduling. Finally, the simulated annealing uses the activity list to represent a solution. The objective function is then calculated by applying the single-pass approach. A new neighbour is obtained by interchanging the position of two activities which are not precedence-related. Since a focus of the approach was on the DSS, no computational experiments are reported.
Chapter III. Heuristic Search Methods and the Learning Backtracking A* Algorithm

The concept of search plays an important but distinct role in the Artificial Intelligence (AI) and Operations Research (OR). One common feature is that both disciplines strive to find effective and efficient means of utilising computers to assist in solving problems. However, these two fields approach their tasks from different perspectives (Pearl, 1984). Operation researchers view problem solving as a split-and-prune process, where the entire set of potential solutions is identified and repetitively trimmed down the potential solutions to obtain a final solution. AI researchers, on the other hand, view problem solving as a process of generate-and-test, where possible problem solutions are created and subsequently checked for their acceptability. Thus, search has become one of the major issues in AI and OR whenever the system, through the lack of knowledge, is faced with a choice from a number of alternatives. Numerous researches revealed that all problem solving activities can be viewed as search processes (Nilsson 1971; Pearl 1984). Although this idea is commonly accepted in the AI field, where problem solving employs search in the generate-and-test context, OR techniques can also be interpreted as search processes. For instance, general branch-and-bound techniques for solving scheduling problems can be viewed as search processes that employ search in the split-and-prune context. Kumar and Kanal (1988) have pointed out the close relationship between search, especially heuristic search and the branch-and-bound techniques.
The development of the search methods moves along two dimensions: Brute-force search and Heuristic search methods. Where the search space is relatively small, brute-force search methods can be used to explore the whole search space. A brute-force search can be defined as a search technique not requiring any a priori domain-specific information concerning the possible solution region of the state space, but using state transformation operators along the solution path (Popovic and Bhatkar, 1994). The heuristic search methods, on the other hand, apply heuristic knowledge, gained from hands-on experience, to determine what might be promising lines of developments.

This chapter briefly reviews several brute-force search algorithms, namely Depth-First search, Breadth-First search, and Depth-first Iterative-Deepening (DFID), and then provides a descriptive discussion of the heuristic search methods. With regard to heuristic search methods, some of the most commonly quoted heuristic search methods are described, including an in-depth coverage of the Learning Backtracking A* Algorithm (LBA*), which is chosen for implementation in this research.

3.1. Brute-force Searches

Brute-force search is a common term for a series of systematic searches such as Depth-First search, Breadth-First search, and Depth-First Iterative Deepening (DFID). The brute-force search guarantees a high efficiency due to the systematic search component which is incorporated for selection of the next states to be considered along the solution path (Popovic and Bhatkar, 1994).
In Depth-First search, priority is given to nodes at deeper levels of the search graph (Pearl, 1984). That is, when a state is examined, all of its children and their descendants are examined before any of its siblings. Depth-First search goes deeper into the search space whenever this is possible. The obvious advantage of Depth-First search is that it eliminates the need for constant backtracking procedures, and thus reduces the bookkeeping of nodes generated. However, such a search method can be dangerous in that the system may unnecessarily spend a long or infinite time exploring a hopeless path (Thornton and Du Boulay, 1992). Therefore, many programmers and researchers developed bounded depth first search method which is a variation on depth first search. This depth bound sets a limit on the depth which is deep enough to ensure that a solution will be found, but must also be shallow enough to avoid too much unnecessary computation. The detailed procedure of Depth-First search is given at Figure 3.1.

```
Begin
    OPEN = [Start]
    CLOSED = []
    While OPEN not [], do
        Begin
            Remove the top-most state from OPEN, call it X.
            If X = goal then return Success.
            Else begin
                Generate successors of X.
                Put X on CLOSED.
                Eliminate Successors of X on OPEN or CLOSED.
                Put remaining successors on top end of OPEN.
            End
        End
    End
Return failure.
End
```

Figure 3.1. Procedure for Depth-First Search (Luger and Stubblefield, 1993)
Breadth-First search, as opposed to depth first, expands a search tree level by level. That is, it assigns a high priority to nodes at the shallower levels of the search graph, progressively exploring all nodes of a given depth (Pearl, 1984). The advantage of this method is that it is guaranteed to find the shortest path from the root to the goal node. Breadth-First search, however, requires redundant operations. In a program, movement from node to node involves performing some operation on the parent node. To return to the parent node, that operation must be repeated. Furthermore, Breadth-first must retain in storage the entire portion of the graph that it explores. The need for the redundant operations and the large storage requirements is the main reason that this method is rarely adopted by human problem solvers (Pearl, 1984). The procedure for Breadth-First search is given at Figure 3.2.

```
Begin
  OPEN = [Start]
  CLOSED = []
  While OPEN not [], do
      Begin
          Remove top-most state from OPEN, call it X.
          If X = goal then return success.
          Else begin
              Generate successors of X.
              Put X on CLOSED.
              Eliminate successors of X on OPEN or CLOSED.
              Put remaining successors on bottom end of OPEN.
          End
      End
  Return failure.
End
```

Figure 3.2. Procedure for Breath-First Search (Luger and Stubblefield, 1993)

These two methods can be applied to the problems represented in either a state-space or problem reduction representation. 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”.

Depth First Iterative Deepening (DFID) is a search procedure that combines the Depth-First and Breadth-First search in order to soften the disadvantages of both search methods (Popovic and Bhatkar, 1994). The first use of DFID is in Slate and Alkin’s chess 4.5 program (Slate and Alkin, 1977). The main idea of DFID is to use depth-first search but place increasing depth bounds on it, starting with one and increasing as far as necessary. If the solution is found, the algorithm terminates. Otherwise, the depth bound is increased by one and again a complete depth-first search with the new depth is performed. At this stage, the algorithm does not take the results of the previous search into account. This ensures that the solution found is the optimal solution path. This is because, with each iteration, another level of the tree is generated for the first time. Thus, once a solution is found, it is a shortest solution path. This seems to be a very time consuming algorithm, since DFID performs unnecessary computation before reaching the goal depth. However, Korf (1987) showed that this unnecessary computation does not affect the asymptotic growth of the computational time for search procedure. The reason is that almost all the work is done at the deepest level of the search. Thus, the extra work in the shallower levels does not affect the asymptotic time complexity.
It should be noted that although all the search strategies described here can produce satisfactory results with some problems, whether they are applied in state-space representation or problem-reduction representation, their failings are easy to see. The computer is forced to travel blindly along every possible path and to stop at predetermined depths, or breadths before moving on. Thus, this is true that all brute-force search algorithms are shown to have worst-case exponential time complexity (Luger and Stubblefield, 1993). The only search approaches that reduce this complexity employ rule of thumb that estimate which paths that are likely to yield the correct solution. The search techniques that employ this rule of thumb, or heuristic, are called heuristic search methods. By sacrificing the requirement of optimal solution, the heuristic search methods may arrive at good solutions to many problem instances most of the time.

3.2. Heuristic search

The concept of heuristic search as an aid to problem solving was first introduced by George Polya (1945). He defined heuristic as the study of the methods and rules of discovery and invention. Since the first appearance of heuristic search in 1945, various definitions have been announced by many AI and OR researchers with some of these definitions covering all the control methods employed in search problem (Firebaugh and Morris, 1988). The term heuristics are defined as criteria, a collection of methods, principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal (Pearl, 1984). In terms of states space search, heuristics are formalised as rules for choosing those branches in a state space that are most likely to lead to an acceptable problem
solution. Heuristic search methods are particularly useful in two basic situations:

1. Where a problem may not have an exact solution because of inherent ambiguities in the problem statement or available data.

2. Where a problem may have an exact solution, but none of the problem solution methods is feasible to solve the given problem.

3.2.1. Heuristic Search Settings

The essential components of the search direction mechanism consist of three elements: 1) a set of states, 2) a collection of operators, and 3) control strategy. A state is a configuration that could be reached using a sequence of legal moves or decisions, starting from an initial configuration. Given a current status of the search; i.e., state in the search space, operators are transformations that map one state to another. These operators are often formulated as production rules with the condition or premise of the rule corresponding to the current search location and the action or decision corresponding to one or more immediately reachable locations in the search space. Finally, the control strategy is the top level mechanism for determining which operators to apply next during the process of searching for a solution to a problem. This is particularly important since often more than one operator will have its possible move in the current state. Even without a great deal of thought, it is obvious that how such decisions are made will have crucial impact on how quickly, and even whether, a problem is eventually solved. Heuristics can be used both in the implementation of operators and as elements of the control strategy.
3.2.2. Heuristic Search Algorithms

The heuristic search algorithms shown in every text includes search concepts like hill-climbing, best-first search, branch-and-bound search, A*, and Learning-Real-Time-A* (LRTA*). Hill-climbing is the simplest and the most direct heuristic search procedure based on local optimisation (Pearl, 1984). The logic behind this search method is quite straightforward, and, combines the depth-first search and a local evaluation function to determine the best state toward the goal state. In terms of graph-search representation, its strategies expand the current node, evaluate its children, and select the best child for further expansion; neither its siblings nor its parent are retained.

Many AI researchers report problems with hill-climbing strategies. According to Lugar and Stubblefield (1993), an erroneous heuristic can result in an infinite search that contains no solution. Because it keeps no history, the strategy can not recover from these failures. Furthermore, hill-climbing can only deliver local solutions to the problem by finding the local maxima (Popovic and Bhatkar, 1994). As discussed, if it reaches a state that has a better evaluation than any other successor states, the search stops. However, there is no guarantee that this state is a goal, but just a local maximum, in which case, the strategy fails to find a solution. That is, performance might well improve in a limited setting, but because of the shape of the entire space, it may never reach the overall best solution. Despite of its limitations, hill-climbing can be useful when a highly informative evaluation function is available to keep away from local maxima (Pearl, 1984). Generally, however, heuristic search requires a more informed strategy. This is provided by best-first search.
The best known informed heuristic search procedure is best-first search with its specialised versions of branch and bound, and A*. Unlike hill-climbing, best-first search is based on a global evaluation function. The aim of best-first search is to find solution state by considering as few states as possible (Luger and Stubblefield, 1993). At each iteration, best-first search selects the most promising state out of all states considered so far. Therefore, best-first search is seen as a generalisation of breadth-first search since it considers the state with the closest estimated distance to the goal as the best state (Popovic and Bhatkar, 1994). Like depth-first and breadth-first search methods, it uses lists to maintain states: OPEN to keep track of the current path of the state and CLOSED to record states already visited. OPEN contains nodes that have been generated and have had the heuristic function applied to them, but which have not yet been examined. This list is a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function. CLOSED consists of the nodes that have already been expanded and examined during the search and are not currently under consideration for expansion. Therefore, this search concept requires an exhaustive bookkeeping and continuous maintenance of a list of states of the possible solution path. Figure 3.3 shows the general best-first OR graph search algorithm, which is based on Luger and Stubblefield (1993).

Branch and bound search is a heuristic search method that determines the shortest possible path of the entire incomplete path and takes it as a boundary for the further expansion (Popovic and Bhatkar, 1994). Expansion is continued until the complete path is found that is shorter than any other incomplete path. Branch and bound search method is an optimal search method with minimal computational efforts required.
Begin  
OPEN = [Start]  
CLOSED = []  
While OPEN not [], do  
Begin  
Remove the leftmost state from OPEN, call it X.  
If X = goal then return the path from Start to X.  
Else begin  
Generate children of X.  
For each child of X do.  
Case  
The child is not on OPEN or CLOSED.  
Begin  
Assign the child a heuristic value.  
Add the child to OPEN.  
End  
The child is already on OPEN.  
If the child was reached by a shorter path  
then give the state on OPEN the shorter path.  
The child is already on CLOSED.  
If the child was reached by a shorter path then  
Begin  
Remove the state from CLOSED.  
Add the child to OPEN.  
End  
End  
Put X on CLOSED.  
Re-order states on OPEN by heuristic merit.  
End  
Return failure.  
End

Figure 3.3. Procedure for Best-First search of OR graphs

The A* algorithm is the most commonly used heuristic search algorithm. It was first described in Hart, Nilsson and Raphael (1968). From then on much researches regarding the effective use of heuristic functions had been made on this algorithm. The A* algorithm is an enhanced version of a branch-and-bound search algorithm (Popovic and Bhatkar, 1994) in which the cost of a node is calculated as \( f(n) = g(n) + h(n) \), where \( g(n) \) is the cost of the path from the initial node to node \( n \), and \( h(n) \) is the heuristic estimate of a path from node \( n \) to a goal node. A well known property of A* and a distinct advantage over other methods is that if the search is properly organised
and the heuristic evaluation function $h(n)$ never overestimates the actual cost to the goal, then the first feasible solution found is guaranteed to be optimal (Powley, Ferguson, and Korf, 1993).

A general description of A* algorithm in Winston (1984) is as follows:

1. Form a one-element queue consisting of the root node

   Until the queue is empty or the goal has been reached, determine if the first node in the queue is the goal state

   a. If the first node is a goal node, stop; the goal has been reached.

   b. Otherwise, remove the first node from the queue, add the first node’s children to the queue, and sort the entire queue by estimated remaining distance.

3. If the goal has been reached, Success; otherwise failure.

Despite this advantage, its applicability is limited by its exponential memory requirement (Korf, 1993), since the A* algorithm stores all nodes in the open and closed lists. Pearl (1984) also showed that if the heuristic used by A* exhibits even constant relative error, then the number of nodes generated by the algorithm increases exponentially with solution cost. Thus an A* algorithm, for problems of practical sizes, will eventually exhaust the available memory long before an appreciable amount of time is used.

Although the large memory requirement of the A* algorithm can be seen as its serious limitation, it can be considerably overcome by modification of the search algorithm to iterative-deepening A* (IDA*) algorithm. IDA* combines the use of a heuristic
evaluation function and a modification of the iterative deepening. It was the first algorithm applied to find optimal solutions to the Fifteen Puzzle. The best version of IDA* was presented by Korf (1985). It is based on a depth-first search method which only stores the current path from the root to the current node. IDA* uses a depth-first search technique, and in each iteration a branch is pruned when the cost of a node \( f(n) \) exceeds the threshold for that iteration. Generally, the value of the threshold starts from the heuristic value of the initial state, and for each iteration it is set to the minimal cost value of all nodes that exceeded the threshold on the previous iteration.

In terms of space requirement by IDA*, since IDA*, at any stage, is based on a depth-first search, its space requirement is only linear in the solution depth (Korf, 1993). Furthermore, since it does not maintain open and closed lists, its implementation is shorter and easier than is by A*. However, its major drawback is that all nodes expanded in one iteration should also be expanded in all subsequent iterations, and since this is a cost of temporarily pruning off branches, IDA* expands more nodes than best-first search method like A*. Therefore, if A* expands \( N \) nodes, then IDA*, in the worst case, expands \( O(N^2) \) nodes (Sarkar, et al., 1991). This analysis has been done by Vampy, Kumar and Korf (1991).

The attempt to reduce the exponential running time of IDA* had led to the development of the Learning Real Time Algorithm (Korf, 1990), abbreviated as LRTA*. LRTA* is a real-time admissible heuristic search algorithm. This algorithm differs from the previous two in that, it adapts a limited search horizon before making a decision move, and the heuristic estimate of visited states may be improved as the result of heuristic learning along the search process. The search horizon for selecting a state to expand the search path consists of only the neighbouring states of the front state. Hence, from a
front state x, this algorithm finds a neighbouring state y with the min \( \{ k(x,y) + y(h) \} \) as the new front state, where \( k(x,y) \) is the positive edge cost between x and y. Before making the subsequent expansion from state y, this algorithm compares this min \( \{ k(x,y) + y(h) \} \) with \( h(x) \) to determine if \( h(x) \) can be improved - heuristic learning. This heuristic learning is based on the rationale that the further away a state is from the goal state, the larger its heuristic estimate should be. If the former is greater than the latter, then \( h(x) \) can be improved to this min \( \{ k(x,y) + y(h) \} \) and still remains as non-overestimating: new \( h(x) = \min \{ k(x,y) + y(h) \} \). With the assumption of non-overestimating initial heuristic estimates and positive edge costs between states, the repetitive applications of the algorithm to a given problem will continue to improve the heuristic estimates of visited states, and eventually find an optimal solution. This algorithm presents the obvious advantages in both space complexity and time complexity over the previous two algorithms. However, a limitation is that there is no guarantee of an optimal solution in any single solution trial, and there is no guideline to indicate how many problem solving trials are needed to find an optimal solution.

The procedures of the LRTA* algorithm can be described as follow:

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. Expand the path to a neighbour state 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 neighbour states, if \( h(x) < \{ h(x') + k(x, x') \} \).
The valuable contribution of this algorithm to search techniques is the learning capability of the heuristic estimates during the process of problem solving. This improved heuristic estimates can be used in the following searches until the algorithm reaches the goal state. Although this algorithm does not guarantee of finding optimal or near-optimal solutions in any single problem solving process, the repeated and the improved search will eventually leads to the optimum solution of the problem.

3.2.3. The Learning backtracking A* Algorithms

The LBA* algorithm (Reza, 1995) works with the same assumptions as LRTA* in that the initial heuristic estimate of a state to the goal state is a lower bound estimate. At a front state x, like LRTA*, this algorithm identifies the neighbouring state y with the minimum \( k(x, y) + h(y) \). It then applies the same rationale to decide if \( h(x) \) can be improved as the result of heuristic learning. This algorithm differs from LRTA* in that when \( h(x) \) is improved, it initiates a review process to examine how the new \( h(x) \) affects the heuristic estimates of the earlier states on the path and the path itself. This review process is basically a backtracking operation, which uses the heuristic learning as the control mechanism to determine how far back the backtracking operation should be applied.

When the heuristic estimate of a front state x is improved, its implications are two folds. Firstly, it casts the doubt if this state should still remain on the path, because the improvement of its heuristic may change its former status as the state with the minimum \( k(x-1, x) + h(x) \) for its proceeding state x-1. Secondly, by applying the
rationale of the heuristic learning, the improvement of \( h(x) \) may also lead to the improvement of \( h(x-1) \). In order to deal with these two implications, the review process begins by backtracking to its previous state \( x-1 \), and conducts the same heuristic learning test to see if the new \( h(x) \) can lead to the improvement of \( h(x-1) \). Should \( h(x-1) \) be improved as a result, then the algorithm backtracks further to state \( x-2 \). In this way the review process examines the states of the path one by one in the reverse order, and stops at the state whose heuristic remains unchanged after the heuristic learning test, or at the root state if it backtracks all the way back to the root state. The algorithm then resumes the search for expansion from this state. Along the process, the search path itself changes following the changes of the front state; each backtracking is equivalent to removing one state from the path. As a result, the search path is fully updated including the heuristic estimates of its states and the path itself every time following a heuristic learning. When the forward search resumes, the subsequent path that will be developed before the next heuristic learning will be one of the best paths with the known heuristic at that time. Hence, when the goal state is reached, the path is an optimal path. This algorithm can be implemented in the following manner, where \( k(x,y) \) representing the positive edge cost from state \( x \) to a neighbouring state \( y \).

Step 0: Apply a heuristic function to generate non-overestimating initial heuristic estimate \( h(x) \) for every state \( x \) to the goal state, and continue.

Step 1: Put the root state on the backtrack list called PATH, and continue.

Step 2: Call the top-most state of the PATH list \( x \). If \( x \) is the goal state, stop; otherwise continue.

Step 3: If \( x \) is a dead-end state, replace its \( h(x) \) with a very large number, remove \( x \).
Step 4: Evaluate $k(x, y) + h(y)$ for all neighbouring state $y$ of $x$, and find the state with the minimum value; break ties randomly. Call this state $x'$, and continue.

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

Step 6: Replace $h(x)$ with $\{ k(x, x') + h(x') \}$, and continue.

Step 7: If $x$ is the root state, go to step 2; otherwise, remove $x$ from PATH list and go to step 2.
The main purpose of this chapter is to develop an approach to implement the LBA* algorithm for solving the multiple resource constraints project scheduling (RCPS) problem. This chapter is comprised of three major sections. Section 1 introduces the generally used formulation of the RCPS problems. In section 2, states, state transition operators and state transition costs are defined, which can help facilitate the implementation of the algorithm to solve the RCPS problem. Finally, section 3 describes the object-oriented approach in implementing the algorithm as well as major functions of the approach.

4.1. Formation of the RCPS problem

The RCPS problem in this research can be depicted as an acyclic network as shown in Figure 4.1. Activities are represented by integer-labeled nodes, such that the label of a node is always greater than the labels of all its immediate predecessor nodes. Arcs represent precedence relations between activities. Unique start and end dummy activities, which have zero duration and require no resources, are appended to the network. Several assumptions need to be made prior to solving the RCPS problems. Underlying assumptions in this research can be identified as follows:

1. Integer period of processing time for each activity must be deterministic.
2. Each activity requires a constant level of resource usage for each resource type.
3. Resources are not allowed to be either shared or depleted.
4. Resources are assumed to be renewable. Thus resources are used and constrained on a period-by-period basis.

5. Activities are not to be interrupted (pre-empted) during execution.

6. The level of availability of each resource type is constant throughout the project schedule.

Given these assumptions, the task of constructing a suitable network requires four major types of input; a detailed list of the individual activities, a specification of their precedence relations, resource requirements, and total resource availability. Table 4.1 shows a detailed list of the individual activities with their precedence relations for the problem in Figure 4.1.

<table>
<thead>
<tr>
<th>Act. No.</th>
<th>Res 1</th>
<th>Res 2</th>
<th>Res 3</th>
<th>Duration</th>
<th>Suc 1</th>
<th>Suc 2</th>
<th>Suc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1. List of the activities for the problem in Figure 4.1

In table 4.1, the first column indicates the activity number, and the next three columns “Res N” indicate the quantity of resources required in each period by the activity. The fifth column “Duration” indicates the processing time required by the activity, and the
next four columns “Suc N” indicate immediate successors of the activity. Figure 4.1 shows a model of the activity on nodes (AON) network for the data form shown in Table 4.1.

Figure 4.1. AON network of an examplary problem instance

4.2. State Space Representation of the RCPS Problems

As stated earlier in the introduction, the main principle of the LBA* algorithm is to search through the solution space of a state-space problem. Hence, prior to developing an approach to implement the algorithm to the RCPS problems, it is necessary to represent the problem as a state-space search problem. Furthermore, as depicted in the network model shown at Figure 4.1, the two dummy nodes, start and end, represent the initial position and the goal position of a problem. With these two

Resource availability : 2 1 2

Figure 4.1. AON network of an examplary problem instance

4.2. State Space Representation of the RCPS Problems

As stated earlier in the introduction, the main principle of the LBA* algorithm is to search through the solution space of a state-space problem. Hence, prior to developing an approach to implement the algorithm to the RCPS problems, it is necessary to represent the problem as a state-space search problem. Furthermore, as depicted in the network model shown at Figure 4.1, the two dummy nodes, start and end, represent the initial position and the goal position of a problem. With these two
given positions, the RCPS problems can be viewed as the process of solving a problem for which the initial position is given and a goal position specified. The problem solution then includes the determination of a set of permitted operational steps that enables the achievement of the desired goal position. Thus, in terms of the state-space representation, solving the RCPS problem can be viewed as a process of finding a goal state from the given initial state through applying a set of state transforming operators that gradually transforms the initial state into the goal state. Therefore, in the following, we need to first define states, operators, and states transition costs.

**Definition of States**

The objective of scheduling a project under resource constraints is to determine a set of starting times for the activities of the project in such a way that the precedent constraints and the resource constraints are satisfied and, the total completion time of the schedule can be minimised. Hence, the process of finding a feasible schedule for a RCPS problem can be seen as a series of decision makings of allocating available resources to as many activities as possible without violating the precedent constraints. As activities are scheduled, they occupy a certain amount of resources. However, as activities are completed, the previously occupied resources by the activities are released and restored. Hence, in that sense, the status of a RCPS problem changes according to the changes in resource availability levels. Therefore, we define a state as a partially completed schedule, which consists of all activities of a project in three sets: completed, in-progress, and unscheduled sets. The completed set consists of all activities, which have been scheduled and completed at the time of consideration. The in-progress set consists of two sub-sets of activities. The first subset includes activities
which have already been scheduled, but not yet completed. The other set includes activities which are newly scheduled at the time of consideration. The unscheduled set comprises all activities which have never been scheduled at the time of consideration. Thus, the initial state is an empty schedule where all activities are in the unscheduled set, and both the completed and the in-progress sets are empty, while the final solution state is a state where all activities are in the completed set.

**State Transition Operator**

From the previous definition of states, it is obvious that the changes in the status of an activity result in changes in available resource levels. Upon the completion of an activity, the amount of resources which was previously occupied is released, and the available resource levels may be sufficient for next activity scheduling. Thus, upon the completion of an activity and subject to the satisfactory compliance of the precedent constraints, the state transition operators can be defined as follows:

- Move the completed activities from the in-progress set to the completed set.
- Move the unscheduled activities whose precedence constraints are satisfied from the unscheduled set to the in-progress set while resource constraints are not violated.
- Do not schedule any new activity.

**Transition Costs**

It is clear that a scheduling decision is the result of a completion from activities in the in-progress activity set. Hence, the transition costs associated with the transition from
one state to another state can be defined as the time interval between two successive activity completion events.

**An Approach for Estimating Initial Heuristic**

It has been shown that the accuracy of an initial heuristic estimate greatly affects the efficiency of any heuristic search method, since the closer the initial estimates are to their true values, the less the number of updating is required to reach their true values. The approach we utilised in this research to obtain the non-overestimating initial heuristic estimate is to ignore the resource constraint of a network. It is well understood that the solution of such a simplified problem will be a lower bound to its original problem. In this way, the initial heuristic estimate of a state can be computed as the longest remaining path from the remaining activities of the given state to the end activity. It can be expressed in the following format:

\[
\text{Initial heuristic estimate } h(x) = \text{The longest path of the in-progress activity to the end activity}
\]

4.3. **System Design and Development**

In this section, we explain the idea of implementing the LBA* algorithm for solving the RCPS problems. Firstly, we describe the design using the object-oriented approach, which consists of six classes. In this section, the detailed functions of these six classes and their interactive relationships are explained. We then describe the method for implementing the LBA* algorithm, which can be broken down into five main modules. These are Initial Heuristic Evaluation, Neighbouring States Identification, Heuristic
Leaning Test, Heuristic Learning and Backtracking, and Forward Searching. Figure 4.2 illustrates the approach with these functions. Each individual function is explained in detail in section 4.3.2.

![Diagram showing the components of the approach of implementing LBA* algorithm](image)

Figure 4.2. Components of the approach of implementing LBA* algorithm

In order to discuss the implementation of the algorithm in detail, it is useful to consider solving the RCPS problems as a search process by establishing a state space search tree containing nodes corresponding to the states. The nodes of the tree are linked
together by arcs that correspond to the operators. Each node (state) characterises a partial schedule in which start times have been assigned, and represents scheduling decisions for some subset of the jobs in the project. Feasible decisions can only be made in the case where all the predecessors are scheduled and the required resources are available. The search starts from the initial state, the root of the tree, at starting time 0. This initial state is an empty schedule where none of the activities have yet been scheduled. As the tree is expanded from some given intermediate node, a new set of partial schedules is created. Each member of this new set corresponds with its parent in all scheduling decisions made previously.

4.3.1. Object-Oriented Design

We applied the object-oriented approach to design the system. As shown in Figure 4.3, this system consists of six classes which comprise of one main class and five supportive classes. The class Evaluation-Stage, the main class of the system, conducts the actual scheduling operation upon user input. This class is supported by five classes: Activity, State, Resource, Combination and List. The class Activity monitors and controls overall properties of each activity such as the remaining duration times of each activity and the progress status of each activity. The class State conducts computation of heuristic estimates of states. The class Resource controls levels of the resource availability and monitors the resource requirements. The class Combination constructs neighbouring states of a front state. Finally, The class List monitors a list of states as well as their activity numbers for state identification purpose. In this section, detailed functions of each class and their relationships are provided.
4.3.1.1. Class Description

The main functions of the class Evaluation Stage are to:

- interact with the classes Activity, State, Resource and Combination,
- conduct the actual scheduling operation upon user input from the class Activity,
- initiate the scheduling process by searching for new activities that can be scheduled at the time of consideration,
- conduct a precedent and resource constraints check to facilitate a new schedulable activities identification process,
- facilitate implementation of the left-shift pruning rule by pruning particular states that can be commenced earlier without violating either a precedent or resource constraint,
- decide if the heuristic learning and backtracking processes are to take place or be continued into a further branching process from a given state selected,
- conduct heuristic learning and initiate the backtracking process for a state whose heuristic estimate can be improved.

The main functions of the class Activity are to:

- accept a specific project scheduling problem instance into the system, which consists of the list of the activity numbers, quantity of resources required in each period by the activity, duration time required by the activity, immediate succeeding activity numbers, and total resource availability throughout the project,
- monitor and update the remaining duration time of the activity through the progress of the search, and
monitor and update the progress status of the activity (unscheduled, in-progress, and completed), when one or more activities are completed or when one or more activities become schedulable activities in the time of consideration.

The main functions of the class State are to:

- compute heuristic estimates of neighbouring states,
- determine activity numbers which make up a state,
• compute the longest duration time of unscheduled activity to facilitate computation of heuristic estimates of states,

• compute the longest duration time of in-progress activity to facilitate computation of heuristic estimates of states, and

• locate activity numbers and determine their remaining duration time for the activities whose remaining duration time are the minimum among in-progress activities of the state.

The main functions of the class Resource are to:

• control number of resource types and levels of the resource availability for the entire project, and

• monitor the resource requirements upon request of a resource feasibility check from the class Evaluation.

The main function of the class combination is to:

• construct any combination of eligible activities identified in the class Evaluation-Stage, which then become a child state as long as both the resource availability constraint, and the precedence constraint, are not violated.

The class List:

• is a parameterised class which declares two concrete list objects, a list of states and a list of integers (activity numbers),
• generates a list of states in cases of constructing neighbouring states, identifying best states whose heuristic estimates have previously been updated, or identifying states which have been pruned from further consideration, and

• generates a list of activity numbers in cases of identifying preceding or succeeding activity numbers, or identifying activity set of states.

4.3.1.2. Relationships among classes

As shown in Figure 4.3, the relationships among the six classes can be represented via either Using, Aggregation (has) or Instantiation relationships.

• Using Relationships

In the object-oriented design process, "Using" relationships among classes parallel the peer-to-peer links among the corresponding instances of these classes. Typically, a "Using" relationship manifests itself via the implementation of some operation that declares a local object of the used class. There are three cases where their relationships can be represented via "Using" relationships. The first case is the relationship between the class Evaluation-Stage and the class Combination. The function "Neighbouring States Identification" of the class Evaluation-Stage uses the class Combination. When one or more schedulable activities have been identified, this function uses the attributes of the class Combination to facilitate constructing possible neighbouring states. The second case is the relationship between the class Evaluation-Stage and the class Resource. The Evaluation-Stage class uses the class Resource when a resource feasibility check is required, which is to identify new schedulable activities in the member function Schedulable Activities Identification. Finally, the
relationship between the class Evaluation-Stage and the class Activity is also represented as the Using relationship. The Evaluation-Stage class uses the class Activity throughout the scheduling process of the main class. The attributes of the class Activity are accessed by the class Evaluation-Stage, which are needed during the process of searching and scheduling of activities. For instance, when the function “Schedulable Activities Identification” is activated, this function uses the attributes of the class Activity to make a decision on whether the given activities are allowed to be scheduled at the time of consideration. In addition, when precedent relationships are to be checked by this function, the attribute of the class Activity (Activity number) is used to identify whether preceding activities of the given activity have all been completed.

- Aggregation (Has) Relationships

Typically, aggregation relationships between classes represent a whole/part hierarchy, with the ability to navigate from the whole to its parts (attributes). In the proposed approach, as shown in Figure 4.3, the class Evaluation-Stage denotes the whole and an instance of the class State is one of its parts. Similarly, the class Activity denotes the whole and an instance of the class Resource is one of its parts. Furthermore, the class Evaluation-Stage has instances of the State List and Activity number List, which are derived from the parameterised class List. Within the class Evaluation-Stage, some attributes of the class State, State List and Activity number List are declared as the private attributes of the class Evaluation-Stage. In addition, the class Activity declares an attribute of the class Resource (Resource Requirement) to access information of resource requirements of certain activities.
• Instantiation Relationships

Instantiation relationships between classes denote relationships via parameterised class. With a parameterised class, appending or retrieving objects via a template argument can be easily performed. A parameterised class cannot have instances unless its instantiation has been done. Hence, within the parameterised class List, the class T (type of list) is declared as a template argument. Using this, two concrete list objects, State List and Activity numbers List are declared. Although they are both derived from the same parameterised class List, they have distinctly different objectives. The class State List is to assist in generating a list of neighbouring states, updated states, or pruned state, and the class Activity numbers List is to assist in generating a list of activity numbers which belong to a state.

4.3.2. Implementation Description

This approach is implemented through five main modules, as shown in Figure 4.2. These five main modules are Initial Heuristic Evaluation, Neighbouring States Identification, Heuristic Learning Test, Heuristic Learning and Backtracking, and Forward Searching. Starting from the Initial Heuristic Evaluation, these modules are executed in sequence in order to solve a given RCPS problem. We use the example network problem in Figure 4.4 to demonstrate the approach of implementation. This example will be used for explanation of the following sections. The complete scheduling process is given in Table 4.2. In the table, the three sets of a state (completed, in-progress, and unscheduled) is given in the ordered 3-tuple inside a bracket. N represents the number of heuristic learning and T is the time for next state selection.
Figure 4.4. Examplary Project Model

Table 4.2. The Complete Scheduling Process for the Example Problem

<table>
<thead>
<tr>
<th>N</th>
<th>T</th>
<th>Front state &amp; total heuristic</th>
<th>Child states &amp; total heuristic</th>
<th>Heuristic Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(0,123456) 6</td>
<td>(0,123456) 6 (0,2,13456) 8' (0,12,3456) 6* (01,2,3456) 7*</td>
<td>Nil</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>(0,123456) 6</td>
<td>(0,123456) 6* (0,2,13456) 8' (0,12,3456) 7</td>
<td>Nil</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(0,123456) 6</td>
<td>(0,123456) 6* (0,1,23456) 8* (0,1,23456) 8</td>
<td>Nil</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>(0,123456) 6</td>
<td>(0,1,23456) 8 (0,2,13456) 8' (0,12,3456) 7*</td>
<td>(0,123456) 7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>(0,123456) 7</td>
<td>(0,1,23456) 8 (0,2,13456) 8' (0,12,3456) 7*</td>
<td>Nil</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>(0,123456) 7</td>
<td>(0,1,23456) 7* (01,2,3456) 9</td>
<td>Nil</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>(01,2,3456) 7</td>
<td>(012,3,456) 9</td>
<td>Nil</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>(012,4,356) 7</td>
<td>(0124,3,56) 9' (0124,5,36) 9'</td>
<td>Nil</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>(0124,35,6) 7</td>
<td>(01243,5,6) 7*</td>
<td>Nil</td>
</tr>
</tbody>
</table>

* indicates the selected state
† indicates the pruned state using Left-Shift rule
In the table, at N=0 and T=0, the front state is (0,123456) and its total heuristic is the
longest path to completion: max{6,4}=6. Its child states of possible scheduling
combinations are (0,1,23456), (0,2,13456), (0,12,3456), and their respective total
heuristic are: max{[1+5], [6]}=6, max{[2+6], [4]}=8, and max{[1+5], [6]}=6. The
minimum of the three is 6 of (0,12,3456), which is same as the heuristic of
(0,123456). The state (0,12,3456) is selected as the new front state. Since there is no
learning occurs, the scheduling process continues at T=1. At T=1, activity 1 is
completed, but no new activity can be scheduled due to resource constraint. With
activity 2 in progressing, its heuristic estimate is: max{[1+5], [3]}=6+1=7, which is 1
unit greater than 6, therefore the heuristic of (0,12,3456) is updated from 6 to 7, and
the front state backtracks to the previous front state (0,123456). Then, at N=1 and
T=0, the state (0,1,23456) is selected as the new front state without heuristic learning,
and its next activity completion happens at T=1. At T=1, either activity 2, 3 or 4 can
be scheduled, and their respective heuristic are: max{[2+5], [4]}=7+1=8 for state
(01,2,3456), max{[2+5], [2]}=7+1=8 for state (01,3,2456), and max{[3+4],
[5]}=7+1=8 for state (01,4,2356). The total heuristic estimates of the three states are
8, hence one state is chosen randomly. In this case, the state (01,2,3456) is selected,
and the heuristic is 8, which is 2 units greater than 6. Therefore, heuristic of
(0,1,23456) is updated from 6 to 8 and the front state backtracks to (0,123456). This
process continues with the same manner, and a solution of 7 is found at N=3 and T=7.

4.3.2.1. Initial Heuristic Evaluation

The main purpose of this module is to compute a non-overestimating initial heuristic
estimate for a state to the goal state. As already explained in section 4.1, the approach
we utilised in this research to obtain initial non-overestimating heuristic estimates is to ignore the resource constraint. If the resource constraint is ignored, the conventional CPM technique can be applied to compute the initial heuristic of a state as the duration of the longest remaining path from the state to completion.

Figure 4.5 shows the flowchart for implementing this module using the recursive definition. As shown the function ‘Initial Heuristic Evaluation’ is a recursive function. It starts with a given activity and its original duration as input, and searches for the immediately succeeding activities of the given activity, and adds up the durations of the activities one at a time. At each step it calls the function itself until it reaches the end activity, and the initial heuristic estimates are returned. For the example network model shown in Figure 4.4, the initial heuristic estimate for the starting dummy activity 0 is 6, that is the longest duration from the activity a0 to the end activity a6 (a0->a1->a4->a5->a6). This is also the initial heuristic estimate of the root state (0,123456). The C++ codes for implementing this module are given in the Figure 4.6.
Given Activity A & its original duration

Find list B (immediate successors of A)

Return Original Duration

Is B empty?

No

Initialise MaxH = 0
Initialise TempH = 0

Set Longest Duration = MaxH

Last Activity of B?

No

Set TempActy to activity no. of B

TempH = Original Duration + CalcInitHeuristics(TempActy)

Is TempH > MaxH?

No

Yes

MaxH = TempH

Figure 4.5. Flowchart for function CalcInitHeuristics
int CSchedule::CalcInitHeuristic( YActivity& yActy )
{
    {
        const YListEx<UINT,UINT>& rActyNums = yActy.GetNextActyNums();
        if ( rActyNums.GetCount() )
        {
            int MaxH = 0;
            int TempH = 0;
            POSITION Pos = rActyNums.GetHeadPosition();
            While (Pos)
            {
                yActivity& rTmpActy = gActivities[rActyNums.GetNext(Pos)];
                rTmpActy.AddPrevActyNum(yActy.GetActyNum());
                TempH = yActy.GetOrglDur() +
                        CalcInitHeuristics(rTmpActy);
                If (TempH > MaxH)
                MaxH = TempH;
            }
        }
    yActy.SetLongestDur(MaxH);
}
else
return yActy.GetOrglDur();
}
return yActy.GetLongestDur();

Figure 4.6. Codes for Initial Heuristic Evaluation

4.3.2.2. Neighbouring States Identification

The main purpose of this module, for a given state, is 1) to identify any new schedulable activities from the neighbouring activities, 2) to construct neighbouring states for these schedulable activities, 3) to identify the states, which have previously been updated or have previously been pruned from further consideration, and 4) to identify the states whose activity can be left-shifted at the time of consideration. Accordingly, this module consists of the following 4 functions: Schedulable Activities Identification, States Construction, Updated States & Pruned States Identification, and Left-shiftable States Identification. A flowchart for these four functions is given at Figure 4.7.
Figure 4.7. Simplified flowchart for function Neighbouring States Identification

**Schedulable Activities Identification**

The purpose of this function, for a given front state, is to identify any new schedulable activities from the unscheduled activities set. The LBA* algorithm, at any stage, locates the next activity for expansion by selecting an activity from the neighbouring activities of its front activity – the last activity of the search path. In order to identify new schedulable activities, the following two conditions should be satisfied:
1) An activity is a candidate for scheduling if all of its preceding activities have already been scheduled (Precedent Relations Check),

2) An activity is a candidate for scheduling if its resource requirement levels are less than or equal to the maximum resource availability of the entire project problem (Resource feasibility Check).

From the root state of the example problem shown at Figure 4.4, the new schedulable activities are activities a1 and a2. As soon as their preceding activity a0 is completed, both a1 and a2 can be considered as the new schedulable activities, because their resource requirements are below the maximum resource levels \{5 5 3\}. The two conditions in checking precedent relations and resource feasibility of this function are implemented as follows:

```cpp
BOOL YEvaluationStage::PrecedentRelationsCheck(const YActivity& yActy)
{
    const YListEx<UINT, UINT>& yPreActyNs = yActy.GetPrevActyNums();
    POSITION Pos = yPreActyNs.GetHeadPosition();
    While (Pos)
    {
        int ildx = yPreActyNs.GetNext(Pos);
        if (!gActivities[ildx].IsCompleted())
            return FALSE;
    }
    return TRUE;
}

BOOL YEvaluationStage::ResourceFeasibilityCheck(const YListEx<UINT, UINT>& yActySet) const
{
    YResource yRes;
    POSITION Pos = yActySet.GetHeadPosition();
    While (Pos)
    {
        YRes += gActivities[yActySet.GetNext(Pos)].GetResRqmnt();
    }
    Return (yRes <= gyMaxResLevel);
}
```

Figure 4.8. Codes for checking precedent relations and resource feasibility
The function ‘PrecedentRelationsCheck’ checks whether the preceding activities \((y\text{PreActyNs})\) of a given activity have all been completed. If they are in the completed activity set, it returns TRUE. The function ‘Resource Feasibility Check’ compares the resource requirements of a given activity with the available resource for the project \((gy\text{MaxResLevel})\).

**States Construction**

Once the new schedulable activities have been identified for a given state, the next task is to construct neighbouring states, which includes the identification of any feasible combination of the schedulable activities. It should be noted that the in-progress activity set consists of activities which have already been scheduled, but not yet completed, as well as activities which are newly scheduled at the time of consideration. Furthermore, since the new schedulable activities have already satisfied the precedent constraint, only the resource constraint has to be considered. The function for the resource feasibility check is already shown at Figure 4.8. Therefore, as long as the resource levels of any combinations of schedulable activities are feasible, scheduling decisions, possibly with the partially completed activities, can be made. For example, with the root state as the front state, new schedulable activities are \(a1\) and \(a2\). Since there are no partially completed activities, three new feasible scheduling decisions can be made: scheduling \(a1\), scheduling \(a2\), and scheduling both \(a1\) and \(a2\) simultaneously. Note in this case, the scheduling of both \(a1\) and \(a2\) simultaneously is resource feasible, since the sum of the resource requirement for both activities \(\{2,4,2\}\) can be met with the total resource available \(\{5,5,3\}\).
Updated States and Pruned States Identification

The purpose of this function is to 1) identify those states whose heuristic estimates have previously been updated, and 2) identify those states which have previously been pruned from further branching. With the former, the system knows which states have been updated before and uses their updated estimates for calculation. With the later, the system can avoid selecting pruned states.

As shown at Figure 4.7, this function is initiated after all neighbouring states of the front state have been identified. Before proceeding to applying the activity selection criterion, it is necessary to check whether any of these states have been visited and their heuristic estimates updated in the previous scheduling decision. For example, as can be seen in Table 4.2, at N=0 T=1, the front state is (0,12,3456) with its total heuristic 6. Its child state is (01,2,3456), and the heuristic estimate is 7. Since the heuristic estimate of the child state is 1 unit greater than 6, the heuristic estimate of (0,12,3456) is updated from 6 to 7. Hence, at the next scheduling process, at N=1 T=0, when the state (0,12,3456) is evaluated, its updated heuristic estimate 7 is used in the scheduling process.

Left-Shiftable States Identification

In order to improve the search efficiency those states, which have previously been considered as non-productive states and therefore have been pruned from further branching, should also be identified along the search process. These states can be determined as having any activity in the in-progress set which can be started earlier without violating either precedent or resource constraints. Any states satisfying this condition will not improve current solution any further, and can be pruned from further
branching. This pruning rule was developed by Schrage (1970), and is referred to as the “left shift rule”. This pruning rule is effective in reducing the solution space. This rule not only eliminates unnecessary search paths, but also it improves resources usage. For the purpose of conducting the backtracking process, in this study, we always maintain the list for front states and their heuristic estimates as well as left-shifted states.

In order to identify left-shiftable states among given neighbouring states, the following two conditions should be satisfied.

1. A given partial schedule, not yet including activity $i$, can be left-shifted, if it is resource feasible when activity $i$ is included in the partial schedule, and

2. A given partial schedule, not yet including activity $i$, can be left-shifted, if the remaining duration of activity $i$ is less than or equal to the remaining durations of the other activities in the partial schedule.

Figure 4.9 and 4.10 shows how the state $(0,2,13456)$ can be pruned from further branching. The former shows the schedule before activity 1 is left-shifted, and the later shows the schedule after activity 1 is left-shifted. The shadowed box indicates after a1 is left-shifted. In this state, a1 can be left-shifted without violating either a precedence or resource constraint. Furthermore, after a1 is left-shifted, resource usage of resource 2 and 3 is twice effective compared to before a1 is left-shifted. Figure 4.11 is the flowchart for carrying out the Left-Shiftable States Identification. Figure 4.12 shows one of the methods of implementing the ‘Left-Shiftable States Identification’ in C++ language.
Figure 4.9. Resource allocation before a1 is left-shifted

Figure 4.10. Resource allocation after a1 is left-shifted
Figure 4.11. Flowchart for function Left-Shiftable States Identification
BOOL YEvaluationStage::MustCutThisState(const YState& yState) const {
    BOOL bMustCut = FALSE;
    UIList ActySetOfNew = yState.GetActySet();
    UIList ActySetOfNotState = m_yNewActyNums - ActySetOfNew;
    POSITION Pos1 = ActySetOfNotState.GetHeadPosition();
    while (Pos1)
    {
        bMustCut = FALSE;
        UINT uiActyNum = ActySetOfNotState.GetNext(Pos1);
        if (IsValidResource(ActySetOfNew + uiActyNum))
        {
            bMustCut = TRUE;
            POSITION Pos2 = ActySetOfNew.GetHeadPosition();
            while (Pos2)
            {
                UINT RemDur1 = gActivities[uiActyNum].GetRemDur();
                UINT RemDur2 = Activities[ActySetOfNew.GetNext(Pos2)].GetRemDur();
                if (RemDur1 > RemDur2)
                    bMustCut = FALSE;
            }
            if(bMustCut)
                return bMustCut;
        }
    }
    return bMustCut;
}

Figure 4.12. Codes for implementing Left-Shift Pruning Rule.

4.3.2.3. Heuristic Learning Test

Having finished identifying neighbouring states of the front state, the next task of the algorithm is to 1) calculate heuristic estimates of every neighbouring state, 2) identify best state from them, and 3) compare the heuristic estimate of the selected best state with that of the front state to determine heuristic learning. The heuristic Learning Test involves two main functions: Heuristic Estimates Computation, and BestState Identification.
The non-overestimating heuristic estimate of a given state is computed as the longest remaining path to completion, while the resource constraint is ignored. It can be expressed in the following format:

\[
\text{Max} \{ \left[ \text{minimum completion time of the current in-progress activity} + \text{the longest path from unscheduled activity to the goal activity} \right], \left[ \text{the longest path from the current in-progress activity to the goal activity} \right] \}
\]

This function is implemented in the approach as the following:

```cpp
void YState::CalcHeurEstmt(const UINT cuiProgTime) {
    UINT uiMinCompTime = GetMinCompTime();
    UINT uiLongestPathInUnscheduled = GetLongestDurInUnscheduled();
    UINT uiLongestPathInProg = GetLongestDurInProgress();
    m_uiHeurEstmt = ::Max((uiMinCompTime + uiLongestPathInUnscheduled), uiLongestPathInProg) + cuiProgTime;
}
```

Figure 4.13. Codes for calculating heuristic estimates of states

In this function, three variables \((\text{uiMinCompTime, uiLongestPathInUnscheduled, uiLongestPathInProgress})\) are used to compute a non-overestimating heuristic estimate of the corresponding state. Two heuristic estimates, \((\text{uiMinCompTime + uiLongestPathInUnscheduled})\) and \((\text{uiLongestPathInProgress})\), are computed and the largest of these two estimates is selected as the heuristic estimate of that given state. For the problem shown at Figure 4.4, with the root state \((0,123456)\) as a front state, heuristic estimates of three scheduling decisions, scheduling \((a1)\), \((a2)\), and \((a1 \text{ and } a2)\), can be computed as follows:

- Heuristic estimate of scheduling \(a1\): \(m\text{_uiHeurEstmt} = \text{Max} \{[1+5], [6]\} = 6\)
- Heuristic estimate of scheduling \(a2\): \(m\text{_uiHeurEstmt} = \text{Max} \{[2+6], [4]\} = 8\)
BestState Identification

The main purpose of this function is to identify the best state for expanding the search path, a state with a minimum heuristic estimate from the neighbouring states. As shown in Figure 4.14, this is done in two parts. It initially searches for those states whose heuristic values have been updated, and selects the one with the minimum value. Then, from those states whose heuristics have not been updated, the system finds the one with the minimum value. The best state is the smaller one of the two. Codes for this function is also given in Figure 4.15.

At this point, one neighbouring state with the minimum heuristic estimate is selected as the best state of the current evaluation stage. Hence, with this current best state, the next task is to make a decision on whether the scheduling process should be continued further or it should update the search path through backtracking procedures.

In the next part, we will discuss about the process of deciding if one of the two scheduling decisions should be carried out following the Heuristic Learning Test.

4.3.2.4. Forward Scheduling

When the heuristic estimate of the selected best state is not greater than the heuristic estimate of the front state, there will be no heuristic learning, and the current best state becomes the new front state and the search continues for next state expansion.
Figure 4.14. Flowchart for function Select Best State
YState& YEvaluationStage::SelectBestState(void) {
    YState *pyBestState, *pyBestStateFromBestStateList;
    pyBestState = pyBestStateFromBestStateList = NULL;
    BOOL IsFromBestState = TRUE;
    POSITION Pos = m_BestStates.GetHeadPosition();
    if (Pos)
        pyBestStateFromBestStateList = &(m_BestStates.GetNext(Pos));
    while (Pos)
    {
        YState *pyState = &(m_yEvalStates.GetNext(Pos));
        if (*pyState < *pyBestStateFromBestStateList)
            pyBestStateFromBestStateList = pyState;
    }

    POSITION BestPos, PrePos;
    PrePos = BestPos = Pos = m_yEvalStates.GetHeadPosition();
    pyBestState = pyBestStateFromBestStateList;
    if (!pyBestState)
        pyBestState = &(m_yEvalStates.GetNext(Pos));
    while (Pos)
    {
        PrePos = Pos;
        YState *pyState = &(m_yEvalStates.GetNext(Pos));
        if (*pyState < *pyBestState)
        {
            BestPos = PrePos;
            pyBestState = pyState;
        }
    }

    if (pyBestState != pyBestStateFromBestStateList)
    {
        POSITION NewBestPos = m_BestStates.AddTail(*pyBestState);
        pyBestState = &(m_BestStates.GetAt(NewBestPos));
        m_yEvalStates.RemoveAt(BestPos);
    }
    return *pyBestState;
}

Figure 4.15. Codes for the function Select Best State

For the system to advance to the newly selected state, the clock time must advance to
effect the next activity completion. The following must be made:

1. the remaining longest durations to completion for those in progress activities are
   reduced accordingly by the remaining duration of the newly completed activities,
   and

2. the remaining durations of those in progress activities are reduced accordingly.
4.3.2.5. Heuristic Learning and Backtracking

The process of Heuristic Learning and Backtracking is initiated when the heuristic estimate of the selected best state is greater than the heuristic estimate of the front state. In this case, (1) the heuristic estimate of the front state is updated to the value of the selected best state, and (2) the system backtracks to the previous state to see if its heuristic estimate can be improved as well. The way in which the heuristic estimate of the front state is adjusted ensures that the newly updated heuristic value raises the lower bound and still remains as non-overestimating. As the search progresses, the heuristic estimates of states on the final path will finally converge to their actual values.

As soon as the process of updating heuristic estimate of the front state is completed, the backtracking process is followed. For the system to backtrack to the previous state, clock time must retreat to the point when the previous state was selected. Furthermore, properties of activities such as the status, duration times and longest duration time must be reverted as they were at previous scheduling time. Depending on the original estimate of the initial state, this process of heuristic learning and backtracking may retreat all the way back to the initial state as often as necessary in order to update its estimate before the final solution is found.
Chapter V. System Performance with Patterson’s 110 problems

This chapter provides a computational evaluation of the procedure developed in the previous chapter. In section 1, the computational results with the Patterson’s 110 problems are provided, and a comparison of computational times with the search approach developed by Bell and Park (1990) is also presented. In section 2, an attempt is made to identify those factors which can assist in identifying those problems, which are likely to require lengthy computation times to find a solution. In order to identity those factors, this study used Pearson correlation analysis to measure the degree of relationship between the dependent variable and the three independent variables, and the regression analysis to determine statistical relationship between the dependent variable and the three independent variables.

5.1. Computational results of Patterson’s 110 problems

The test problem set used in this experiment consists of 110 problem instances, which was developed by Patterson (1984) and has been used as a standard test set by many researchers in this area. The problems in this test set have activity number ranging from 7 to 51, and most problems have over 22 activities. Each problem has fixed multiple-unit requirements of three different resource types, except 7 problems. Of the 7 problems, problems 7, 8, 9 and 14 require only 1 resource, and problem 10, 11 and 15 require 2 different resources.
For this experiment, computations were performed on a Pentium PC and the system was developed using C++ language. The detailed results are shown in Table 5.6, at the end of this chapter. The procedure found an optimal solution for every problem, except in the case of 2 problems: problem 72 and 77. The execution of these two problems were terminated, because they are likely to exceed the time limit of the study. Table 5.1 shows the summary of the CPU requirement for the 110 problems.

As shown in the table, CPU times for solving the 108 problems range from 0 to 5672 seconds. Our approach was able to solve most of the problems in less than 9 seconds. Among the 110 problems, 71 problems were solved in less than 9 seconds and 100 problems were solved in less than 100 seconds, while 104 problems were solved in less than 399 seconds. However, there are 4 particular problems which required more than 1000 seconds to solve, and two problems took longer than 5999.

<table>
<thead>
<tr>
<th>CPU time (sec.)</th>
<th>Number solved</th>
<th>Cumulative</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-9</td>
<td>71</td>
<td>71</td>
</tr>
<tr>
<td>10-99</td>
<td>29</td>
<td>100</td>
</tr>
<tr>
<td>100-199</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>200-299</td>
<td>2</td>
<td>103</td>
</tr>
<tr>
<td>300-399</td>
<td>1</td>
<td>104</td>
</tr>
<tr>
<td>400-999</td>
<td>0</td>
<td>104</td>
</tr>
<tr>
<td>1000-1999</td>
<td>1</td>
<td>105</td>
</tr>
<tr>
<td>2000-2999</td>
<td>1</td>
<td>106</td>
</tr>
<tr>
<td>3000-3999</td>
<td>1</td>
<td>107</td>
</tr>
<tr>
<td>4000-4999</td>
<td>0</td>
<td>107</td>
</tr>
<tr>
<td>5000-5999</td>
<td>1</td>
<td>108</td>
</tr>
<tr>
<td>Terminated</td>
<td>2</td>
<td>110</td>
</tr>
</tbody>
</table>

Table 5.1. Number of problems solved in each time range for the 110 test problems

Although, due to different computation facilities, it may not be appropriate to make a direct comparison with other researches on the absolute terms, we believe the direct comparisons may still serve useful purposes on relative terms. In the following, we
present the direct comparison of computational times with the results by Bell and Park (1990) was made. Their problem solving approach combined the A* search algorithm with an activity selection scheme. The identical 110 test problems were tested on an Apple Macintosh Plus using a version of “Common Lisp”. Table 5.2 shows a comparison of the computational times. For the 110 test problems, the mean computational CPU time of this study is 133.7 seconds, and the standard deviation 675.6. The mean computational CPU time of Bell and Park’s approach is 340.6 seconds, with a standard deviation of 558.5. Thus, on the average, as far as computational times are concerned, this study appears to require a smaller mean but a larger standard deviation. The standard deviation may even become larger if the two terminated problems are included.

<table>
<thead>
<tr>
<th></th>
<th>Our approach</th>
<th>Bell &amp; Park’s Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean CPU time (sec)</td>
<td>133.7</td>
<td>340.6</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>675.6</td>
<td>558.5</td>
</tr>
</tbody>
</table>

Table 5.2. Comparison of Computational Times

5.2. Statistical analysis

Given the variability in computational times reported for the 110 test problems, where 71 problems were solved in less than 9 seconds, 29 problems between 10 and 99 seconds, 4 problems between 100 and 999, another 4 problems between 1000 and 5999, and 2 problems requiring longer than 5999, this study has attempted to identify those factors which may distinguish one group from the other. Through these factors, one may be able to identify the types of problems, which are likely to incur lengthy computational times.
In this part, the statistical analysis using SPSS package was performed to try to identify those factors. Based on previous studies, three factors, CMPLX (Project Complexity), HEUR (Heuristic Tightness) and CONSTR\textsubscript{k} (Resource Constrainedness), which are defined in Table 5.3 were considered. The Pearson correlation analysis was first applied to measure the degree of correlation between the three factors and computation time. This analysis was used to find out the relationship that may exist between several variables. Further, a linear regression analysis was performed to identify which factors, among the three, are likely to have an effect on the amount of computational time.

<table>
<thead>
<tr>
<th>CMPLX</th>
<th>Project complexity = number of arcs / number of activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEUR</td>
<td>Heuristic tightness = GH – IH</td>
</tr>
<tr>
<td></td>
<td>GH: Goal Heuristic</td>
</tr>
<tr>
<td></td>
<td>IH: Initial Heuristic</td>
</tr>
<tr>
<td>CONSTR\textsubscript{k}</td>
<td>Resource Constrainedness = DMND\textsubscript{k} / R\textsubscript{k}</td>
</tr>
<tr>
<td></td>
<td>DMND\textsubscript{k} = Average Quantity of Resource k</td>
</tr>
<tr>
<td></td>
<td>Demanded when required by an activity =</td>
</tr>
</tbody>
</table>
|       | \[
|       | \sum_{j=1}^{N} r_{jk} / \sum_{j=1}^{N} \{ 1 \text{ if } r_{jk} > 0 \} \text{ for } k = 1, 2, ..., k \]
|       | R\textsubscript{k}: Availability of resource k in an entire project |

Table 5.3. Definition of the Factors used in identifying solution difficulty

The results of the Pearson correlation analysis, which measure the degree of linear relationship between the dependent variable (Computational time) and the three
independent variables (CMPLX, HEUR, and CONSTR$_k$), are given in Table 5.4. As shown in the table, all three independent variables have some degree of linear relationships with the dependent variables. Among the three independent variables, the highest correlation was 0.58 with HEUR. This clearly shows that the heuristic tightness factor may be more significant in determining the computation time requirement than the other two variables. This indicates that an increase in the value of the variable HEUR is likely to lead to an increase of the Computational time, and vice versa. Therefore, if one can improve the heuristic estimation to reduce the difference between the initial estimate and the final solution, there may be a higher possibility to reduce the computation time in finding an optimal solution.

<table>
<thead>
<tr>
<th></th>
<th>CMPLX</th>
<th>HEUR</th>
<th>CONSTR$_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational</td>
<td>0.35</td>
<td>0.58</td>
<td>0.28</td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4. Results of Pearson Correlation analysis

The above analysis shows the existence between the dependent variable and the three independent variables. Next, we attempted to find a statistical relationship that is useful in forecasting the dependent variable. In order to further verify the above results, a regression analysis was undertaken to determine a statistical relationship between the dependent variable and the three independent variables. The result of the analysis is shown at Table 5.5. As indicated in the column ‘Beta’ in the Table, which measures the average relationship between variables in statistical terms, HEUR had the highest value 0.48 among the three independent variables. This value is nearly twice as the next highest value 0.28 of variable CMPLX, and the variable CONSTR$_k$ has the lowest 0.14. This again indicates that HEUR has the highest predictability in
In terms of computational time requirement.

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>S.D</th>
<th>Beta</th>
<th>t</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPLX</td>
<td>1.55</td>
<td>0.32</td>
<td>0.28</td>
<td>3.566</td>
<td>.01</td>
</tr>
<tr>
<td>HEUR</td>
<td>4.25</td>
<td>4.21</td>
<td>0.48</td>
<td>5.904</td>
<td>.01</td>
</tr>
<tr>
<td>CONSTR_{k}</td>
<td>0.33</td>
<td>8.07E-02</td>
<td>0.14</td>
<td>1.723</td>
<td>N.S*</td>
</tr>
</tbody>
</table>

*: Not Significant

Table 5.5. Summarised regression analysis
Table 5.6. Computational Results for Patterson’s 110 test problems

<table>
<thead>
<tr>
<th>Problem No.</th>
<th>No. of Activities</th>
<th>No. of Arcs</th>
<th>Resource</th>
<th>Initial Heuristic</th>
<th>Optimal Solution</th>
<th># of Backtracks</th>
<th>CPU Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>20</td>
<td>2 1 2</td>
<td>18</td>
<td>19</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>5 5 3</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>14</td>
<td>6 7 6</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>35</td>
<td>15 20 20</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
<td>35</td>
<td>13 15 15</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
<td>35</td>
<td>13 13 13</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
<td>5 0 0</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>11</td>
<td>4 0 0</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
<td>29</td>
<td>8 0 0</td>
<td>20</td>
<td>20</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>11</td>
<td>4 3 0</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>11</td>
<td>3 3 0</td>
<td>14</td>
<td>18</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>23</td>
<td>31</td>
<td>11 5 8</td>
<td>10</td>
<td>13</td>
<td>37448</td>
<td>206</td>
</tr>
<tr>
<td>13</td>
<td>22</td>
<td>30</td>
<td>10 10 10</td>
<td>13</td>
<td>20</td>
<td>127164</td>
<td>2282</td>
</tr>
<tr>
<td>14</td>
<td>35</td>
<td>55</td>
<td>10 0 0</td>
<td>41</td>
<td>43</td>
<td>107533</td>
<td>263</td>
</tr>
<tr>
<td>15</td>
<td>35</td>
<td>51</td>
<td>10 10 0</td>
<td>43</td>
<td>43</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>22</td>
<td>34</td>
<td>10 10 10</td>
<td>30</td>
<td>32</td>
<td>1416</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>22</td>
<td>35</td>
<td>10 10 10</td>
<td>22</td>
<td>29</td>
<td>15783</td>
<td>21</td>
</tr>
<tr>
<td>18</td>
<td>22</td>
<td>34</td>
<td>7 10 10</td>
<td>30</td>
<td>41</td>
<td>8624</td>
<td>11</td>
</tr>
<tr>
<td>19</td>
<td>22</td>
<td>35</td>
<td>10 10 8</td>
<td>22</td>
<td>31</td>
<td>45677</td>
<td>72</td>
</tr>
<tr>
<td>20</td>
<td>22</td>
<td>32</td>
<td>10 10 10</td>
<td>37</td>
<td>37</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>32</td>
<td>10 6 10</td>
<td>37</td>
<td>48</td>
<td>22685</td>
<td>25</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>34</td>
<td>10 10 10</td>
<td>31</td>
<td>36</td>
<td>2739</td>
<td>4</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>33</td>
<td>7 10 10</td>
<td>20</td>
<td>32</td>
<td>6560</td>
<td>11</td>
</tr>
<tr>
<td>24</td>
<td>22</td>
<td>35</td>
<td>10 10 10</td>
<td>37</td>
<td>40</td>
<td>5178</td>
<td>6</td>
</tr>
<tr>
<td>25</td>
<td>22</td>
<td>35</td>
<td>10 10 10</td>
<td>31</td>
<td>33</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>22</td>
<td>35</td>
<td>6 10 10</td>
<td>31</td>
<td>43</td>
<td>2834</td>
<td>4</td>
</tr>
<tr>
<td>27</td>
<td>22</td>
<td>33</td>
<td>10 10 10</td>
<td>32</td>
<td>36</td>
<td>888</td>
<td>2</td>
</tr>
<tr>
<td>28</td>
<td>22</td>
<td>34</td>
<td>7 10 10</td>
<td>32</td>
<td>43</td>
<td>2575</td>
<td>4</td>
</tr>
<tr>
<td>29</td>
<td>22</td>
<td>35</td>
<td>10 10 10</td>
<td>22</td>
<td>29</td>
<td>7798</td>
<td>16</td>
</tr>
<tr>
<td>30</td>
<td>22</td>
<td>33</td>
<td>10 10 10</td>
<td>26</td>
<td>32</td>
<td>20665</td>
<td>26</td>
</tr>
<tr>
<td>Pro. No.</td>
<td>No. of Activities</td>
<td>No. of Arcs</td>
<td>Resource</td>
<td>Initial Heuristic</td>
<td>Optimal Solution</td>
<td># of Backtracks</td>
<td>CPU Time (sec)</td>
</tr>
<tr>
<td>---------</td>
<td>------------------</td>
<td>-------------</td>
<td>----------</td>
<td>-------------------</td>
<td>------------------</td>
<td>----------------</td>
<td>---------------</td>
</tr>
<tr>
<td>31</td>
<td>22</td>
<td>34</td>
<td>10 8 10</td>
<td>26</td>
<td>35</td>
<td>14955</td>
<td>19</td>
</tr>
<tr>
<td>32</td>
<td>22</td>
<td>32</td>
<td>10 10 10</td>
<td>22</td>
<td>22</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>33</td>
<td>22</td>
<td>32</td>
<td>7 10 10</td>
<td>22</td>
<td>31</td>
<td>23488</td>
<td>30</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>32</td>
<td>10 7 10</td>
<td>22</td>
<td>30</td>
<td>35335</td>
<td>44</td>
</tr>
<tr>
<td>35</td>
<td>22</td>
<td>35</td>
<td>10 10 10</td>
<td>27</td>
<td>31</td>
<td>3900</td>
<td>6</td>
</tr>
<tr>
<td>36</td>
<td>22</td>
<td>33</td>
<td>10 10 10</td>
<td>32</td>
<td>33</td>
<td>455</td>
<td>1</td>
</tr>
<tr>
<td>37</td>
<td>22</td>
<td>34</td>
<td>10 10 10</td>
<td>22</td>
<td>28</td>
<td>5702</td>
<td>8</td>
</tr>
<tr>
<td>38</td>
<td>22</td>
<td>34</td>
<td>8 10 10</td>
<td>22</td>
<td>30</td>
<td>4389</td>
<td>5</td>
</tr>
<tr>
<td>39</td>
<td>22</td>
<td>33</td>
<td>10 10 10</td>
<td>30</td>
<td>31</td>
<td>201</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>22</td>
<td>34</td>
<td>10 10 10</td>
<td>27</td>
<td>31</td>
<td>5171</td>
<td>8</td>
</tr>
<tr>
<td>41</td>
<td>22</td>
<td>34</td>
<td>10 10 10</td>
<td>33</td>
<td>36</td>
<td>5067</td>
<td>7</td>
</tr>
<tr>
<td>42</td>
<td>22</td>
<td>36</td>
<td>10 10 10</td>
<td>28</td>
<td>28</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>43</td>
<td>22</td>
<td>36</td>
<td>10 10 6</td>
<td>28</td>
<td>41</td>
<td>222598</td>
<td>1088</td>
</tr>
<tr>
<td>44</td>
<td>22</td>
<td>34</td>
<td>10 10 10</td>
<td>31</td>
<td>31</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>45</td>
<td>22</td>
<td>34</td>
<td>7 10 10</td>
<td>31</td>
<td>39</td>
<td>14303</td>
<td>19</td>
</tr>
<tr>
<td>46</td>
<td>22</td>
<td>33</td>
<td>10 10 10</td>
<td>25</td>
<td>33</td>
<td>646951</td>
<td>124</td>
</tr>
<tr>
<td>47</td>
<td>22</td>
<td>33</td>
<td>8 10 10</td>
<td>25</td>
<td>35</td>
<td>56192</td>
<td>86</td>
</tr>
<tr>
<td>48</td>
<td>22</td>
<td>24</td>
<td>10 10 10</td>
<td>18</td>
<td>23</td>
<td>3213</td>
<td>5</td>
</tr>
<tr>
<td>49</td>
<td>22</td>
<td>24</td>
<td>15 15 15</td>
<td>18</td>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>22</td>
<td>32</td>
<td>10 10 10</td>
<td>21</td>
<td>25</td>
<td>1136</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>22</td>
<td>32</td>
<td>10 10 10</td>
<td>18</td>
<td>25</td>
<td>29783</td>
<td>40</td>
</tr>
<tr>
<td>52</td>
<td>22</td>
<td>40</td>
<td>10 10 10</td>
<td>22</td>
<td>27</td>
<td>12266</td>
<td>16</td>
</tr>
<tr>
<td>53</td>
<td>22</td>
<td>64</td>
<td>10 10 10</td>
<td>24</td>
<td>28</td>
<td>2850</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>22</td>
<td>68</td>
<td>6 6 6</td>
<td>27</td>
<td>50</td>
<td>720999</td>
<td>5672</td>
</tr>
<tr>
<td>55</td>
<td>22</td>
<td>68</td>
<td>10 10 10</td>
<td>27</td>
<td>29</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>56</td>
<td>22</td>
<td>68</td>
<td>15 15 15</td>
<td>27</td>
<td>27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>57</td>
<td>22</td>
<td>32</td>
<td>15 15 15</td>
<td>21</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>58</td>
<td>27</td>
<td>43</td>
<td>10 10 10</td>
<td>30</td>
<td>35</td>
<td>6959</td>
<td>12</td>
</tr>
<tr>
<td>59</td>
<td>27</td>
<td>40</td>
<td>10 10 10</td>
<td>29</td>
<td>31</td>
<td>1160</td>
<td>2</td>
</tr>
<tr>
<td>60</td>
<td>27</td>
<td>43</td>
<td>10 10 10</td>
<td>35</td>
<td>39</td>
<td>790</td>
<td>2</td>
</tr>
<tr>
<td>61</td>
<td>27</td>
<td>42</td>
<td>10 10 10</td>
<td>32</td>
<td>36</td>
<td>840</td>
<td>1</td>
</tr>
<tr>
<td>62</td>
<td>27</td>
<td>40</td>
<td>10 10 10</td>
<td>30</td>
<td>37</td>
<td>24002</td>
<td>41</td>
</tr>
<tr>
<td>Pro. No.</td>
<td>No. of Activities</td>
<td>No. of Arcs</td>
<td>Resource</td>
<td>Initial Heuristic</td>
<td>Optimal Solution</td>
<td># of Backtracks</td>
<td>CPU Time (sec)</td>
</tr>
<tr>
<td>---------</td>
<td>------------------</td>
<td>-------------</td>
<td>----------</td>
<td>-------------------</td>
<td>------------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>63</td>
<td>27</td>
<td>42</td>
<td>10 10 10</td>
<td>34</td>
<td>40</td>
<td>9429</td>
<td>15</td>
</tr>
<tr>
<td>64</td>
<td>27</td>
<td>40</td>
<td>10 10 10</td>
<td>33</td>
<td>37</td>
<td>14017</td>
<td>22</td>
</tr>
<tr>
<td>65</td>
<td>27</td>
<td>41</td>
<td>10 10 10</td>
<td>37</td>
<td>40</td>
<td>2751</td>
<td>4</td>
</tr>
<tr>
<td>66</td>
<td>27</td>
<td>43</td>
<td>10 10 10</td>
<td>32</td>
<td>38</td>
<td>98</td>
<td>0</td>
</tr>
<tr>
<td>67</td>
<td>27</td>
<td>42</td>
<td>10 10 10</td>
<td>22</td>
<td>27</td>
<td>12341</td>
<td>32</td>
</tr>
<tr>
<td>68</td>
<td>27</td>
<td>40</td>
<td>10 10 10</td>
<td>39</td>
<td>41</td>
<td>361</td>
<td>0</td>
</tr>
<tr>
<td>69</td>
<td>27</td>
<td>40</td>
<td>10 10 10</td>
<td>26</td>
<td>30</td>
<td>5237</td>
<td>9</td>
</tr>
<tr>
<td>70</td>
<td>27</td>
<td>39</td>
<td>10 10 10</td>
<td>30</td>
<td>31</td>
<td>1259</td>
<td>3</td>
</tr>
<tr>
<td>71</td>
<td>27</td>
<td>41</td>
<td>10 10 10</td>
<td>29</td>
<td>32</td>
<td>1666</td>
<td>3</td>
</tr>
<tr>
<td>72</td>
<td>27</td>
<td>39</td>
<td>10 10 7</td>
<td>30</td>
<td>terminate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>27</td>
<td>41</td>
<td>10 8 10</td>
<td>29</td>
<td>36</td>
<td>6223</td>
<td>10</td>
</tr>
<tr>
<td>74</td>
<td>27</td>
<td>40</td>
<td>10 10 10</td>
<td>29</td>
<td>30</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>75</td>
<td>27</td>
<td>40</td>
<td>10 10 7</td>
<td>29</td>
<td>34</td>
<td>8011</td>
<td>13</td>
</tr>
<tr>
<td>76</td>
<td>27</td>
<td>40</td>
<td>10 7 10</td>
<td>29</td>
<td>43</td>
<td>29269</td>
<td>36</td>
</tr>
<tr>
<td>77</td>
<td>27</td>
<td>41</td>
<td>6 6 6</td>
<td>31</td>
<td>terminate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>27</td>
<td>41</td>
<td>7 7 7</td>
<td>31</td>
<td>53</td>
<td>680190</td>
<td>3455</td>
</tr>
<tr>
<td>79</td>
<td>27</td>
<td>41</td>
<td>8 8 8</td>
<td>31</td>
<td>45</td>
<td>20696</td>
<td>29</td>
</tr>
<tr>
<td>80</td>
<td>27</td>
<td>41</td>
<td>9 9 9</td>
<td>31</td>
<td>38</td>
<td>2428</td>
<td>3</td>
</tr>
<tr>
<td>81</td>
<td>27</td>
<td>41</td>
<td>10 10 10</td>
<td>31</td>
<td>36</td>
<td>2784</td>
<td>4</td>
</tr>
<tr>
<td>82</td>
<td>27</td>
<td>41</td>
<td>11 11 11</td>
<td>31</td>
<td>34</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>83</td>
<td>27</td>
<td>41</td>
<td>12 12 12</td>
<td>31</td>
<td>34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>84</td>
<td>27</td>
<td>41</td>
<td>13 13 13</td>
<td>31</td>
<td>33</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>85</td>
<td>27</td>
<td>41</td>
<td>14 14 14</td>
<td>31</td>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>86</td>
<td>27</td>
<td>41</td>
<td>15 15 15</td>
<td>31</td>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>87</td>
<td>27</td>
<td>42</td>
<td>10 10 10</td>
<td>23</td>
<td>29</td>
<td>25505</td>
<td>44</td>
</tr>
<tr>
<td>88</td>
<td>27</td>
<td>44</td>
<td>10 10 10</td>
<td>36</td>
<td>40</td>
<td>3273</td>
<td>5</td>
</tr>
<tr>
<td>89</td>
<td>27</td>
<td>39</td>
<td>10 10 10</td>
<td>30</td>
<td>31</td>
<td>234</td>
<td>0</td>
</tr>
<tr>
<td>90</td>
<td>27</td>
<td>37</td>
<td>10 10 10</td>
<td>34</td>
<td>39</td>
<td>23287</td>
<td>41</td>
</tr>
<tr>
<td>91</td>
<td>27</td>
<td>38</td>
<td>10 10 10</td>
<td>33</td>
<td>35</td>
<td>1235</td>
<td>2</td>
</tr>
<tr>
<td>92</td>
<td>27</td>
<td>36</td>
<td>10 10 10</td>
<td>26</td>
<td>28</td>
<td>3699</td>
<td>6</td>
</tr>
<tr>
<td>93</td>
<td>27</td>
<td>36</td>
<td>10 10 10</td>
<td>25</td>
<td>26</td>
<td>1581</td>
<td>3</td>
</tr>
<tr>
<td>94</td>
<td>27</td>
<td>38</td>
<td>10 10 10</td>
<td>35</td>
<td>36</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>Pro. No.</td>
<td>No. of Activities</td>
<td>No. of Arcs</td>
<td>Resource</td>
<td>Initial Heuristic</td>
<td>Optimal Solution</td>
<td># of Backtracks</td>
<td>CPU Time (sec)</td>
</tr>
<tr>
<td>---------</td>
<td>------------------</td>
<td>-------------</td>
<td>----------</td>
<td>------------------</td>
<td>------------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>95</td>
<td>27</td>
<td>39</td>
<td>10 10 10</td>
<td>29</td>
<td>33</td>
<td>1156</td>
<td>2</td>
</tr>
<tr>
<td>96</td>
<td>27</td>
<td>36</td>
<td>10 10 10</td>
<td>23</td>
<td>26</td>
<td>1347</td>
<td>2</td>
</tr>
<tr>
<td>97</td>
<td>27</td>
<td>34</td>
<td>10 10 10</td>
<td>26</td>
<td>30</td>
<td>662</td>
<td>1</td>
</tr>
<tr>
<td>98</td>
<td>27</td>
<td>38</td>
<td>10 10 10</td>
<td>35</td>
<td>41</td>
<td>19067</td>
<td>30</td>
</tr>
<tr>
<td>99</td>
<td>27</td>
<td>39</td>
<td>10 10 10</td>
<td>31</td>
<td>37</td>
<td>20473</td>
<td>33</td>
</tr>
<tr>
<td>100</td>
<td>27</td>
<td>37</td>
<td>10 10 10</td>
<td>29</td>
<td>33</td>
<td>103877</td>
<td>313</td>
</tr>
<tr>
<td>101</td>
<td>51</td>
<td>85</td>
<td>10 12 10</td>
<td>71</td>
<td>75</td>
<td>28667</td>
<td>60</td>
</tr>
<tr>
<td>102</td>
<td>51</td>
<td>81</td>
<td>10 12 12</td>
<td>83</td>
<td>83</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>103</td>
<td>51</td>
<td>87</td>
<td>14 14 12</td>
<td>56</td>
<td>56</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>104</td>
<td>51</td>
<td>81</td>
<td>12 12 10</td>
<td>78</td>
<td>79</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>105</td>
<td>51</td>
<td>86</td>
<td>10 10 10</td>
<td>76</td>
<td>76</td>
<td>166</td>
<td>0</td>
</tr>
<tr>
<td>106</td>
<td>51</td>
<td>78</td>
<td>10 10 10</td>
<td>58</td>
<td>60</td>
<td>1834</td>
<td>4</td>
</tr>
<tr>
<td>107</td>
<td>51</td>
<td>77</td>
<td>10 12 10</td>
<td>78</td>
<td>78</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>108</td>
<td>51</td>
<td>76</td>
<td>10 10 10</td>
<td>60</td>
<td>61</td>
<td>2585</td>
<td>5</td>
</tr>
<tr>
<td>109</td>
<td>51</td>
<td>80</td>
<td>12 12 10</td>
<td>57</td>
<td>60</td>
<td>23810</td>
<td>47</td>
</tr>
<tr>
<td>110</td>
<td>51</td>
<td>70</td>
<td>10 12 10</td>
<td>50</td>
<td>50</td>
<td>183</td>
<td>0</td>
</tr>
</tbody>
</table>
Chapter VI. Conclusion

This research has focused on the design and development of a solution approach for implementing the heuristic learning algorithm LBA* to solve the Resource-Constrained Project Scheduling (RCPS) problem, which is a practical combinatorial search problem. The RCPS problem represents one of the most general and complex problems to solve in the field of scheduling. The approach we take in this research involves the transformation of search graph into state space representation. We base on the dynamic nature of the resource availability to define the states, state transition operators, and the state transition cost for the approach.

Conventional wisdom in the field of RCPS is that the RCPS problems exhibit such a richness and variety that no single solution method is sufficient. In the review of the literature on the RCPS problems, we describe a number of approaches that have been proposed for the RCPS problems. This literature survey reveals that even though OR and AI researchers have developed numerous sophisticated solution methods and techniques to overcome the complex combinatorial nature of the RCPS problems, there still is no promising method which guarantees accurate solutions as well as the computational practicality.

Our emphasis in this research was to design and develop a solution approach in such a way that the advantages of the LBA* algorithm are fully utilised. The unique features of this algorithm are the consideration of the effect of the heuristic learning during the search process, which employs the backtracking procedure to update the search path
and improve heuristic estimates for future state selection. The non-overestimating condition of the algorithm has led us to utilise the heuristic learning for the updating of the previously selected states, which is to ensure the lowerbound estimates as required. Along the search process, in order to improve the performance of the approach, we implemented a pruning rule (left-shift rule), that prunes away those unproductive branches, and effectively reduces the search space of a problem and computational times.

To evaluate the performance of this approach, we tested the widely accepted 110 problem instances developed by Patterson. The results show that this approach ability to solve most of the problem instances within reasonable times; with 71 problems solved within 9 seconds CPU time. However, there are few problems which required quite lengthy computation times, and we were not able to complete 2 problems due to excessive computation times. For the purpose of direction comparison of the result, the computational times were compared with the results published by Bell and Park (1990). Because of a fundamental difference on computation facilities, this comparison should serve only as a reference. Overall statistics appear to show that our approach requires less computational time, smaller mean, but the performance of our approach degrades as problems become more tightly resource-constrained, larger standard deviation. Furthermore, we have attempted to identify various factors which may lead to longer computational times as observed in our test. Among the various factors used in describing the problem characteristics, three factors, CMPLX (Project Complexity), HEUR (Heuristic Tightness) and CONSTR (Resource Constrainedness) are evaluated. The statistical analysis reveals that there is a higher predictability that, the performance of our approach degrades as problems become more heavily
heuristic-tightened. This could mean that the initial heuristic estimate is much underestimating from the true cost, and hence the algorithm has to search through a much wider solution space in order to find a solution.

In terms of computer resources requirement, as most backtracking-based algorithms have demonstrated, this approach has suffered from the limitation of computer resources. The solution space requirement by some problems is enormous. This is due to the fact that all the nodes generated by the search process as well as their revised heuristic estimates have to be saved for future references, hence the memory requirements could grow beyond what can be supported. Therefore, it is important for future research to devise strategies which have the ability to control growth of the search space. The heuristic estimation method could play an important role in setting the range of the solution space in the early stage, because the closer the initial estimate to the true cost the less search will be required. The branch pruning methods are another important aspect of the algorithm, which can gradually reduce the solution space by recognising those unproductive branches and eliminate them from further consideration. In this research, we have applied the well-known left-shift rule, which is one of the effective pruning rules currently available. Although, it is very effective, we think that the implementation of other strategies may improve the efficiency. One approach is the "state dominance rule", which was used in the Bell and Park’s approach. Another approach is to trade solution quality with improved search efforts, hence memory requirement and computation times, by specifying a non-zero heuristic learning threshold, which would not initiate the backtracking process until the accumulated heuristic learning has exceeded the specified threshold.
Bibliography


