# Abstract

KARNOUB, RAZEK E. **An Exact Bidirectional Approach to the Resource Constrained Project Scheduling Problem.** (Under the direction of Salah E. Elmaghraby)

The aim of this research is to develop a new approach to the Resource Constrained Project Scheduling Problem. Traditionally, most exact approaches to solve the problem have been either Integer Programming approaches or Branch and Bound (BaB) ones. Of the two, BaB procedures have proven to be the more successful computationally. But, while it is quite intuitive to conceive that the root node of a BaB search tree should be the start activity, it is no less conceivable that it be the terminal activity. Indeed, it is conceivable that the search starts from both ends and concludes somewhere in the middle of the ensuing trees. Unfortunately, BaB as a methodology is not amenable to deriving a termination criterion for such a procedure which guarantees optimality. To a large extent, this research can be seen as an attempt at accomplishing just that.

We start with a comprehensive review of the literature related to the problem. We present a new Integer Programming model to describe it together with a 'look-ahead' heuristic procedure which may be used along with it. The main advantage of this procedure is its ability to reflect planning over the short horizon in anticipation of changes to the project in the more distant

future.

Our chief contribution is in the third part of this study which sets up the problem as a Shortest Path Problem in two 'state networks', forward and reverse, where the nodes reflect the precedence feasibility or partial completion of the activities of the project. We develop the conceptual tools to construct the networks and to properly detect a 'path' between their sources from which a makespan optimal schedule could be derived. The theoretical constructs ultimately result in algorithms that solve the problem proceeding forward, in reverse, or bidirectionally. These algorithms have been tested on the J30 benchmark data set of Kolisch, Sprecher and Drexl (1995). Computational results show important advantages of the bidirectional approach but also point out significant avenues for improvement.

# An Exact Bidirectional Approach to the Resource Constrained Project Scheduling Problem

by

Razek E. Karnoub

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Operations Research

Raleigh
2001

Approved by:

| | |
|---|---|
| Dr. Bibhuti B. Bhattacharyya | Dr. Carla D. Savage |
| Dr. Hien T. Tran | Dr. Salah E. Elmaghraby<br>(Chair of Advisory Committee) |

© Razek E. Karnoub

*To my parents,*

*Maha,*

*Elias-Ramzey and Tamara*

# Biography

This preliminary section, required by the graduate school of this university, is generally filled with a glimpse of the author's personal pre-dissertation life story. While this maybe quite interesting, a dissertation biography would probably be more informative if its author were to concentrate on the life of his or her dissertation and the major issues that eventually shaped it. Such issues, in all likelihood, must have been central to an author's life for a significant length of time yet they may not have necessarily found their way into the text itself. They, in fact, would tell the story behind the story and as such could be fairly instructive to other researchers in the same field; especially future dissertation authors.

With this in mind, here is my biography.

In the early stages of this research, my focus was on modeling the Resource Constrained Project Scheduling Problem as an integer programming problem and solving it as such. At that time, I was fascinated by the power of integer programming as a modeling tool. All it needed, I thought, was an efficient solution methodology to match its modeling capability. But with several known integer programming models for the problem, all of which solvable for relatively small instances only, another such model had to be markedly different to stand a chance at solving larger size instances. This led me to a model of the problem as a set of linear mixed

integer diagonal blocks coupled with linear constraints in terms of continuous variables only. But, as it turned out, a procedure to take advantage of such a structure already existed while my research to develop a new exact methodology for it was inconclusive. Subsequently, I took the suggestion of my advisor to come up with a procedure that sort of calculates several moves ahead to pick the best current move. The result was the 'look-ahead' heuristic based on integer programming formulations of the problem.

With that, I put my integer programing attempts on hold and shifted my aim to take advantage of the structure of the problem in a direct way. More precisely, I realized that the problem could be modeled as a shortest path problem between two nodes of a network. The difficulty was that the network might be so large that it becomes impractical to generate. To ease this difficulty, I thought, one could develop rules to fathom a multitude of nodes generated. Additionally, one could apply a shortest path procedure on the network as it is developed rather than after its generation. This potentially allows the procedure to terminate before many possible nodes are actually generated. Further, one could start from both terminal nodes of the network to work out paths from each that meet somewhere in the middle avoiding, in the process, even more inessential nodes. But more generally, one could introduce uniformly directed cutsets into this network, determine the 'shortest paths' between nodes associated with consecutive cutsets and ultimately deduce a shortest path between the network's terminal nodes through the cutsets. Clearly, this would constitute a decomposition, or a multi-directional, approach to the problem; in contrast to the previous approach, more appropriately characterized as bidirectional.

These were the fundamentals of what I hoped would be this dissertation. Among the two approaches, the multi-directional one, I initially thought, was the one with a potential to be

‘original and substantive’ enough for a dissertation topic while the bidirectional approach would be its mere special case. My initial goal for this dissertation, then, was to develop the multi-directional approach.

Naturally at that point, numerous details had yet to be hammered out. But I was confident enough they eventually could be that I turned my attention towards a thorough literature review of the problem. All the while hoping, of course, that these ideas were not already developed. Fortunately for me, they indeed were not.

Following the literature review, I set out to develop the remaining details of the bidirectional procedure. In this process, a fundamental revision of one of my original ideas had to be made. Restricting the nodes of a network that could be generated entailed that some paths originating at the two terminal nodes may actually not meet at a common node. Resolving this issue required that two networks, not just one, be distinguished and that a new concept of intersecting paths be introduced; with all the implications regarding the recognition of path intersection and the optimality of a resulting schedule.

With all the theoretical details of the bidirectional approach finalized, I presented my work to my advisory committee for preliminary approval. I was hoping at the time that, with the small illustrative example furnished, they would not require additional computational validation; that, if anything, they would ask that I develop the multi-directional approach. But sure enough, my committee asked for computational experiments to assess the practical merits, or maybe demerits they thought, of the bidirectional procedure. As a result, development of the multi-directional was put on hold and, instead, considerable effort was expended designing an implementation of its special case; together with the indispensable supporting data structure, inescapable coding and inevitable debugging that ensue. But the task was finally accomplished

and with it this dissertation, however short of its initial goal, at last concluded.

# Acknowledgements

It is a pleasure to express my sincere appreciation to all who have provided me their help and assistance in this effort.

My thanks go to Dr Salah E. Elmaghraby, chairman of my advisory committee, for his numerous comments and suggestions, his advice, his hard work and not the least for his patience.

Thanks also to the members of my committee, Drs B.B. Bhattacharyya, Carla D. Savage and H.T. Tran, for valuable discussions and helpful comments.

Thanks are also due to my office-mates Hao Cheng, S.Ilker Birbil, Shunmin Wang and Burcu Özçam. Their help in some of the finer details of the arduous task of debugging my code has been instrumental.

Last but not least, my thanks go to my parents and my wife for all the support and patience they had to exhibit.

# Contents

**List of Tables** xii

**List of Figures** xiii

**1 Introduction, Review and Critique of Literature** 1

1.1 Problem Definition, Graphic Representation and Schedule Types . . . . . . . . . 2

1.2 Special Cases and NP-Completeness . . . . . . . . . . . . . . . . . . . . . . . . 10

1.3 Extensions and Classification . . . . . . . . . . . . . . . . . . . . . . . . . . 11

1.3.1 Activities . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

1.3.2 Resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

1.3.3 Precedence Relationships . . . . . . . . . . . . . . . . . . . . . . 14

1.3.4 The Objective Function . . . . . . . . . . . . . . . . . . . . . . . 15

1.3.5 Classification Schemes . . . . . . . . . . . . . . . . . . . . . . . 17

1.4 Existing Methodologies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

1.4.1 Lower Bounds . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

1.4.2 Exact Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

1.4.3 Heuristics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

1.5 Complexity Measures and Test Data Sets . . . . . . . . . . . . . . . . . . . . . . . 71
1.5.1 Activity Related Measure . . . . . . . . . . . . . . . . . . . . . . . . . . 73
1.5.2 Precedence Related Measures . . . . . . . . . . . . . . . . . . . . . . . . 74
1.5.3 Resource Related Measures . . . . . . . . . . . . . . . . . . . . . . . . . 79
1.5.4 Test Data Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

**2 Integer Programming Models** **86**

2.1 An RCPSP Integer Program . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
2.2 A PRCPSP Integer Program . . . . . . . . . . . . . . . . . . . . . . . . . . 92
2.3 A Look-Ahead Heuristic . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

**3 A Shortest Path Paradigm** **98**

3.1 Critique of Integer Programming and Branch and Bound Approaches . . . . . . . 99
3.2 Problem Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
3.3 The State Network . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
3.4 Node Reductions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
3.5 Computational Complexity Implications . . . . . . . . . . . . . . . . . . . . . 127
3.6 Node Generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
3.6.1 An Integer Programming Based Scheme . . . . . . . . . . . . . . . . . . . 138
3.6.2 A Branch and Bound Scheme . . . . . . . . . . . . . . . . . . . . . . . . 141
3.7 A Bidirectional Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
3.7.1 Path Completion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
3.7.2 Optimality Condition . . . . . . . . . . . . . . . . . . . . . . . . . . . 177
3.7.3 Outline of an Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 188

3.8 Further Reductions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190

3.9 Implementation Issues and Decomposition . . . . . . . . . . . . . . . . . . . . . . . . 193

3.9.1 Brief Survey of Relevant Shortest Path Algorithms for the Arbitrary Arc Lengths Case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 194

3.9.2 Shortest Path in the Identical Arc Lengths Case . . . . . . . . . . . . . . . 200

3.9.3 Shortest Path Implementations for the Reduced State Networks . . . . . . 201

3.9.4 Decomposition Approaches . . . . . . . . . . . . . . . . . . . . . . . . . . . . 216

3.10 Computational Experience . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 220

**4 Future Research** **230**

**Bibliography** **233**

# List of Tables

3.1 J30 - Time Analysis per Group . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 224

3.2 J30 - Instances for which the Bidirectional Algorithm Performs Slower than Average 225

3.3 J30 - Node Analysis per Group . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 228

3.4 J30 - Instances for which the Bidirectional Algorithm Requires more Nodes than Average . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 229

# List of Figures

1-1 The interdictive graph - AoN representation . . . . . . . . . . . . . . . . . . . . . . 3

1-2 The interdictive graph - AoA representation . . . . . . . . . . . . . . . . . . . . . . 4

1-3 Theorem 4: case n = 2 - AoN representation. . . . . . . . . . . . . . . . . . . . . . 7

1-4 Theorem 4: case n = 3 - AoN representation . . . . . . . . . . . . . . . . . . . . . 7

1-5 Theorem 4: general case - AoN representation . . . . . . . . . . . . . . . . . . . . . 8

3-1 A project to illustrate the state network concept - AoA representation . . . . . . 105

3-2 The state network of the project in Figure 3-1. . . . . . . . . . . . . . . . . . . . . 105

3-3 Counterexample to maximizing resource consumption - single resource, unit duration case. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

3-4 Counterexample to maximizing the number of activities started - single resource, unit duration case. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

3-5 Counterexample to generalizing Corollary 10. . . . . . . . . . . . . . . . . . . . . . 113

3-6 Graphic depiction of a $d_i$-submaximal subset. . . . . . . . . . . . . . . . . . . . . . 115

3-7 The state network of the project instance in Figure 3-1 as reduced by Corollary 15. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118

3-8 The state network of the project instance in Figure 3-1 as reduced by Corollary 15 and Theorem 17. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
3-9 A decision tree to generate all maximal subsets. . . . . . . . . . . . . . . . . . . . . . 142
3-10 The flowcharts for implementing Rules 3 and 4 in Algorithm 1. . . . . . . . . . . 146
3-11 The reverse reduced state network of the project instance in Figure 3-1. . . . . . 161
3-12 The project instance of Example 26 - AoA representation . . . . . . . . . . . . . 178
3-13 The Bidirectional Approach solving the project instance in Figure 3-1. . . . . . . 191
3-14 A partial depiction of the data structure supporting the index function approach. 207
3-15 A partial depiction of the data structure supporting the alphabetic approach. . . 215

# Chapter 1

# Introduction, Review and Critique of Literature

This chapter starts with defining our problem, some of its solution types and special cases. Section 1.2 establishes the complexity of the problem. Section 1.3 extends the basic model and presents a recent classification scheme for these extensions that is compatible with the established classification scheme of a related class of problems. Section 1.4 is devoted to reviewing the major solution methodologies from the literature; namely lower bound, exact and heuristic methodologies. The last part, Section 1.5 discusses factors that render problem instances difficult to solve and known data sets generated to test the effectiveness of developed solution methods.

## 1.1 Problem Definition, Graphic Representation and Schedule Types

The **Resource Constrained Project Scheduling Problem**, RCPSP for short, is the problem of scheduling a set of activities subject to given precedence relationships among the activities and the availability of the resources required for their completion. In this context, an activity is said to precede another if starting the second requires the completion of the first. The objective of the problem is to minimize the project duration under three further assumptions:

- Activity durations are discrete. The time horizon is subdivided into equal periods. Activities start at the beginning of the periods.
- The activities' resource consumption levels are given discrete constants over their durations.
- The resource availability limits are constant throughout the duration of the project.

The concept of precedence relationship described above yields naturally the concept of immediate precedence. An activity is said to immediately precede a second if there is no third activity that precedes the second and is preceded by the first. In this case, we also say that the second activity immediately succeeds the first. This concept makes possible a visual representation of a project a lot simpler than it would otherwise be. Indeed, there are two commonly used methods to represent projects that rely on it.

The first method represents an activity by a node and the immediate precedence relationship by an arc. If activity $i$ immediately precedes activity $j$ then the arc points from node $i$ to node $j$. In this way, if there is more than one activity with no predecessors, we frequently

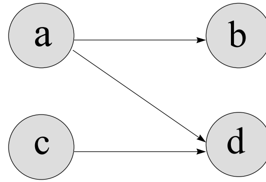


Figure 1-1: The interdictive graph - AoN representation

create a fictitious activity of zero duration that precedes them. Similarly if there is more than one activity with no successors, we create a fictitious activity of zero duration that succeeds them. These two activities are commonly referred to as the **dummy activities**. The resulting graphical representation of a project is called the **Activity on Node** representation, **AoN** for short and, for any given project, can be proven to be unique. Figure 1-1 shows the AoN representation of a project of four activities $a$, $b$, $c$ and $d$ where $a$ precedes $b$ and $d$, $c$ precedes $d$ and where the dummy activities are omitted.

The second method represents an activity by an arc pointing from one node to a second node. The first node stands for the event that marks the start of the activity and the second stands for the event marking its end. If activity $i$ immediately precedes activity $j$ then the end node of $i$ and the start node of $j$ may be one and the same. By convention, only one activity is drawn between any two nodes. This sometimes necessitates the use of dummy nodes. Occasionally a dummy activity of zero duration has to be introduced into such a graphical representation to depict a precedence relationship between two activities that otherwise would not be implied. Such a dummy activity is usually represented by a dashed arc. This representation of the

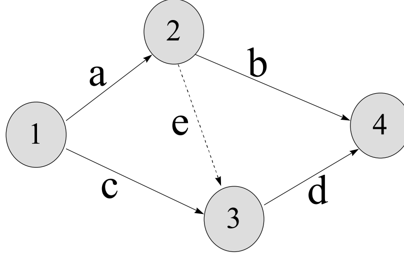


Figure 1-2: The interdictive graph - AoA representation

project is referred to as the **Activity on Arc** representation, **AoA** for short, and is not unique; a project may admit several AoA representations. Figure 1-2 shows the AoA representation of the project in Figure1-1.

The two representations are interchangeable. Given an AoA representation we can easily translate it into an AoN representation. The opposite translation, though, is more difficult due to the non-uniqueness issue which prompts the search for representations with a minimal number of dummy nodes and/or a minimal number of dummy activities. Michael, Kamburowski and Stallmann (1993) present a translation that minimizes the use of dummy activities subject to the minimum number of dummy nodes; this problem is NP-Complete but solvable for medium size projects. Given an AoA representation, Bein, Kamburowski and Stallmann (1992) present an $O(n^{2.5})$ algorithm to minimize the number of its dummy nodes, where $n$ is the total number of nodes. The problem of minimizing the number of dummy activities in an AoA representation is NP-hard (Krishnamoorthy 1979). Both methods of representation are frequently used. One may be preferred to the other depending on the application. The advantage of the AoN representation is that it is unique and more intuitive. But many algorithms, use the AoA

representation rather because in their context it is the simpler representation.

A **schedule** of a project is an assignment of starting times to the activities of the project. Note that the earliest possible start times of the activities can be easily computed via a forward pass that ignores the resource constraints. Similarly, given an upper bound on the project duration, the activities' latest start times can be computed via a backward pass, again ignoring the resource constraints. This yields time intervals during which to schedule the activities to achieve a minimum project makespan. Several types of schedules can be distinguished (cf. Sprecher 1995).

**Definition 1** *Given a schedule, a* ***local left shift*** *of activity* i *is a feasible time reduction of the starting time of* i*, while keeping the starting times of all other activities the same, that can be accomplished in steps of one time unit decrements. A* ***global left shift*** *of* i *is a feasible time reduction of the starting time of* i*, also while keeping all the other start times the same, but that cannot result from a local shift of* i*.*

**Definition 2** *A feasible schedule is said to be* ***semi-active*** *if no local left shift is possible. A feasible schedule is said to be* ***active*** *if no global left shift is possible.*

**Definition 3** *A schedule is said to be* ***non-delay*** *if there is no time period* t *for which an activity is precedence and resource eligible to be started yet is delayed. A* ***delay*** *schedule is, of course, one that does allow delaying an activity beyond time* t *even though it may be eligible to start at* t*.*

**Remark 1** *It follows from the definitions that a non-delay schedule is active and that an active schedule is semi-active. Reportedly, it can also be proven (Klein 2000 p. 165) that at least one*

*optimal solution to the RCPSP is active. Hence, searching the active schedules of an RCPSP for an optimal solution might be more efficient than searching the set of semi-active schedules which itself might be more efficient than searching the whole set of feasible solutions. Unfortunately, the set of non-delay schedules might not contain an optimal solution. Worse still, the optimal non-delay solution can be a very poor approximation of the optimal delay one.*

**Theorem 4** *: For RCPSP, the ratio of the optimal non-delay schedule to its optimal delay one can be arbitrarily large.*

**Proof.** To prove this assertion, we will show that for any integer $n \geq 2$, there exists an RCPSP instance whose ratio of optimal non-delay solution to optimal delay one can be arbitrarily close to $n$.

For $n = 2$ consider the RCPSP instance in Figure 1.3; in its AoN representation. By inspection, the optimal non-delay schedule is: *2, 1 and* $a_2$*, 0,* $a_1$*, end*, of duration $max(\varepsilon, \theta) + \varepsilon + \theta$. In the same way, the optimal delay schedule is: *2, 1, 0,* $a_1$ *and* $a_2$*, end*, of duration $\varepsilon + \varepsilon + \theta$. The ratio of non-delay to delay durations is $\frac{max(\varepsilon,\theta)+\varepsilon+\theta}{2\varepsilon+\theta} \rightarrow 2$ as $\varepsilon \rightarrow 0$.

For $n = 3$, this instance can be generalized but requires two resources. The generalization is in Figure 1.4. Again by inspection, assuming that $\varepsilon < \theta$, the optimal non-delay schedule is: *3, 2 and* $a_3$*, 1 and (4 and* $a_2$*), 0,* $a_1$*, end*, of duration $\theta + (\frac{\varepsilon}{2} + \theta) + \varepsilon + \theta = 3\theta + \frac{3}{2}\varepsilon$. The optimal delay schedule is: *3, 2, 1 and 4, 0,* $a_1$ *and* $a_2$ *and* $a_3$*, end*, of duration $\varepsilon + \varepsilon + \varepsilon + \theta = \theta + 3\varepsilon$. The ratio of non-delay to delay durations is $\frac{3\theta+\frac{3}{2}\varepsilon}{\theta+3\varepsilon} \rightarrow 3$ as $\varepsilon \rightarrow 0$.

The extension of this instance to the case where $n$ activities can be run together is in Figure 1.5. Assuming that $\varepsilon < \theta$, the optimal non-delay schedule is *n, n-1and* $a_n$*, n-2 and (2n-2 and* $a_{n-1}$*),$\cdots$, 2 and (n+2 and* $a_3$*), 1 and (n+1 and* $a_2$*), 0,* $a_1$*, end*, of duration $\theta + (\frac{\varepsilon}{2} + \theta) + \cdots +$

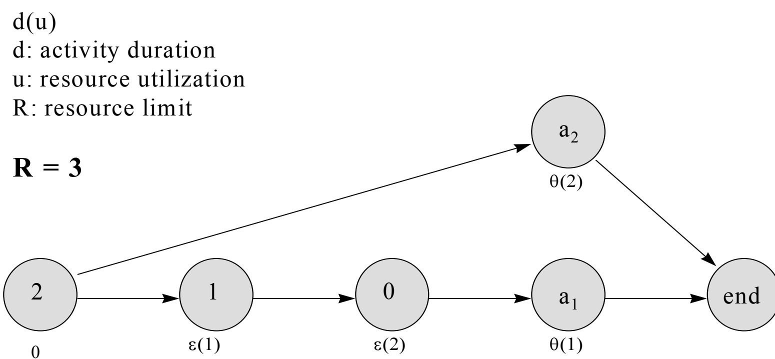


Figure 1-3: Theorem 4: case n = 2 - AoN representation.

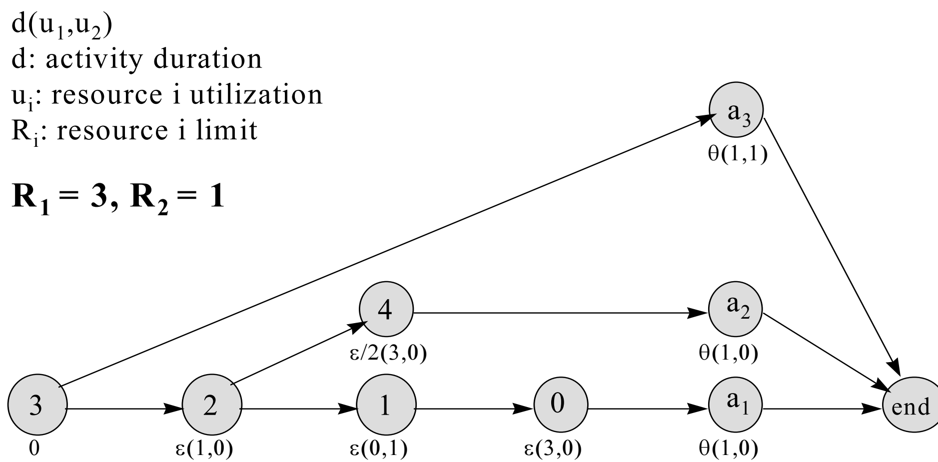


Figure 1-4: Theorem 4: case n = 3 - AoN representation

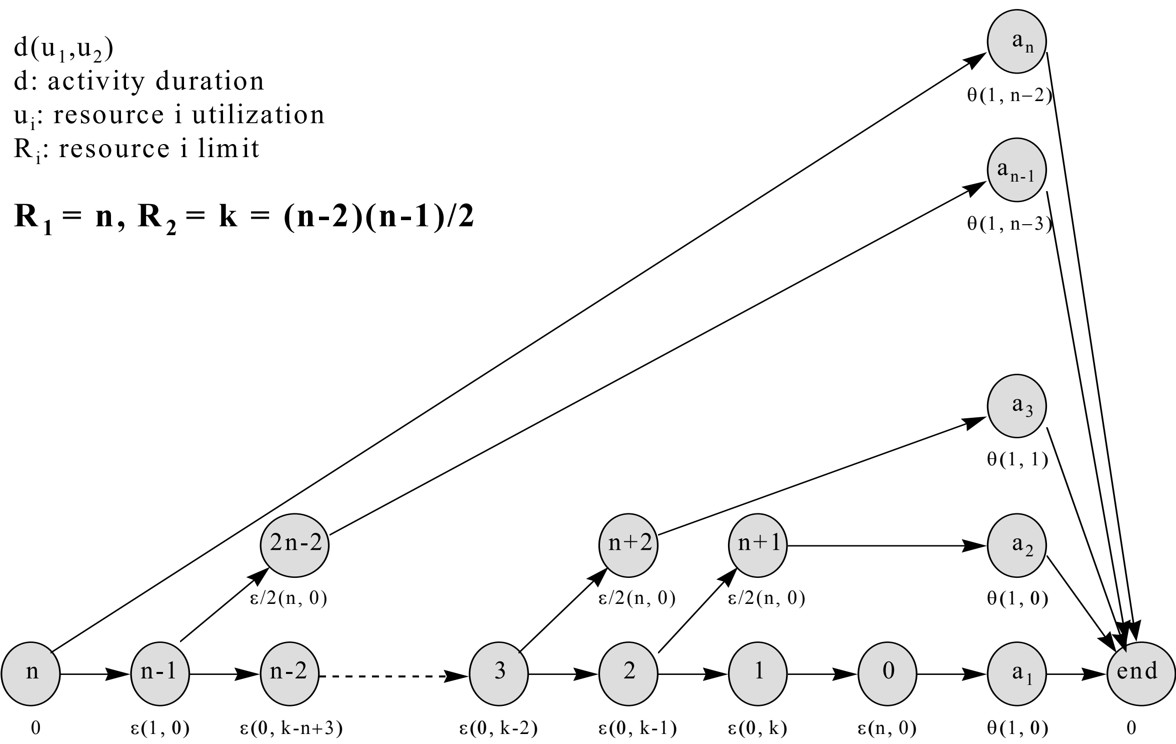


Figure 1-5: Theorem 4: general case - AoN representation

$(\frac{\varepsilon}{2}+\theta)+\varepsilon+\theta = n\frac{\varepsilon}{2}+n\theta.$

The optimal delay schedule is *n, n-1, n-2 and 2n-2,$\cdots$, 2 and n+2, 1 and n+1, 0,* $a_1$ *and* $a_2$ *and* $\cdots$ *and* $a_n$*, end*, of duration $\varepsilon+\varepsilon+\cdots\varepsilon+\theta = n\varepsilon+\theta$. The ratio of optimal non-delay to optimal delay durations is $\frac{n\frac{\varepsilon}{2}+n\theta}{n\varepsilon+\theta} \to n$ as $\varepsilon \to 0$. ■

**Remark 2** *It may seem that by letting* $\varepsilon \to 0$ *in the previous theorem, we have violated the first assumption made about the durations of activities being discrete. This objection, however, is easy to deal with. For one, we could have replaced* $\varepsilon$ *with* $\frac{1}{k}$*, where* $k$ *is a positive integer and let* $k \to \infty$ *instead. Even better, we could have fixed* $\varepsilon$ *at* 1 *and let* $\theta \to \infty$*. The ratios of optimal non-delay to optimal delay durations would still be the same in the limit. What is important is that* $\theta$ *becomes a lot larger than* $\varepsilon$*.*

More than being merely a counter-intuitive mathematical curiosity, this theorem is important for one approach to RCPSP that it unequivocally rules out. Suppose the theorem is false, *i.e.,* there is an upper bound $M$ to the ratios of optimal non-delay to optimal delay solutions of RCPSP instances. It is not too hard to see that the non-delay optimal schedule to RCPSP is a lot easier to obtain than the delay one. Let $d_n^*$ be the optimal non-delay duration of an instance and $d^*$ be its optimal delay one. Suppose we have secured $d_n^*$ and a heuristic solution to the delay case of duration $h$. Then $d_n^*/d^* \leq M$ *i.e.,* $d_n^*/h \cdot h/d^* \leq M$. This says that $h/d^* \leq M \cdot h/d_n^*$. But $1 \leq h/d^*$ as $h$ is only a heuristic duration. It follows that the relative difference of $h$ and $d^*$ is: $(h-d^*)/d^* \leq (M.h/d_n^*) - 1$ which tells whether $h$ is a sufficient approximation to $d^*$ or not. Unfortunately, however, this analysis does not hold. To rectify it, one may think of obtaining upper bounds on the ratio in terms of the number of activities. That may or may not be possible but the values of those bounds would be questionable as the

sequence of instances in the theorem shows that those bounds would have to tend to infinity as the number of activities tends to infinity.

## 1.2 Special Cases and NP-Completeness

Several special cases of RCPSP can be identified. The simplest is the case where resources are in such abundance that they do not constitute any limitation on processing the activities. Here, the forward pass mentioned previously yields a minimum project duration. An optimal feasible schedule is a schedule whose starting times lie in the time intervals derived by the forward and backward passes where the minimum project duration is seen as the project duration estimate of the backward pass. This case is called the **CPM** model.

The opposite of the resource abundance case is the case when the resource limits are one for each resource. A special case of this extreme case is the **Job Shop Problem** (JSP). It is the problem of scheduling $n$ jobs on $m$ machines. Here the resources are the machines, one of each type, and each job may not need to be processed on each machine but the sequence in which it has to be processed on the machines is given. Except for precedences in the sequences given, no other precedences are imposed. As in the case of RCPSP, the objective is to minimize the makespan. Note that JSP itself is a generalization of the **Flowshop Problem** (FP). This, in turn, is the problem of scheduling $n$ jobs on a sequence of $m$ machines. Each job has to go on the $m$ machines in the same sequence. Now, it is well known that FP, for $m \geq 3$, is strongly NP-complete (cf. Pinedo 1995 p. 101)[1] so JSP must be strongly NP-complete too and therefore so must RCPSP.

[1] Pinedo actually states that FP is strongly NP-hard. He attributes the result to Garey, Johnson and Sethi (1976). But Pinedo does not differentiate between NP-complete and NP-hard. We do and that is why we quote him in this way.

We remark that this derivation of the strong NP-completeness of RCPSP is not exactly what usually is reported in the literature; for example in Alvarez-Valdes and Tamarit (1989) and Koloisch and Hartmann (1999) to name a couple. More commonly, the derivation is attributed to a result by Blazewicz *et al* (1983) concerning the strong NP-completeness of JSP with two machines, a single resource, single unit availability and single unit consumptions. We note that neither JSP nor FP in our derivation require resource factors to be incorporated.

## 1.3 Extensions and Classification

A close look at the defining elements of RCPSP reveals numerous possible ways of extending it. Practical experience necessitated research into most of them. These defining elements are, of course, the activities, the resources, the precedence relationships and the objective. Modifying any of their attributes gives rise to a new problem. This section is devoted to presenting those new problems but we also note that one can extend RCPSP by considering several RCPSP's simultaneously. The common link between the different problems would be a set of shared resources. This problem is called the **Multi-Project Scheduling Problem**. The obvious way to solve it is to look at it as a single RCPSP. But this may not be the most efficient way. For an early reference to this problem the reader is referred to Kurtulus and Davis (1982).

### 1.3.1 Activities

The first element we look at is the activities. RCPSP assumes the activity durations are known entities. In practice this may not be the case, activity durations may be random variables following some distribution. Replacing deterministic durations with random variables leads to a significantly different problem that belongs in the field of Stochastic Scheduling (Möhring

1984, 1985). Another way of modifying activity durations is to allow them to vary in accordance with the amount and/or type of resources spent on them. In this model, a finite number of ways to complete each activity is proposed. Each way comes with its own resource requirements. This problem is called the **Multi-Mode RCPSP**. Such a model would have applications in the construction industry, for example, where intensive manual labor can be replaced with machine work for substantial savings in time, albeit at a premium cost. For recent a review of exact methods to this problem, the reader is referred to the paper by Hartmann and Drexl (1998). A special case of this problem is the **Discrete Time Resource Trade-off Problem**, DTRTP, which assumes a single resource. A generalization of it is the case where the set of activities is partitioned into subsets. The choice of a mode for one activity forces the same choice on all the activities of the subset to which it belongs. This problem is called the **Mode Identity Resource Constrained Project Scheduling Problem**, MIRCPSP (Drexl *et al* 1999).

Now, in certain situations, such as large software development projects, for example, activities can be interrupted during their processing and restarted later with little or no cost. A schedule that permits such action is called a **preemptive** schedule. The corresponding problem is called **Preemptive-Resume RCPSP**. The problem where the preemption cost is not negligible is also of interest, it is called **Preemptive-Restart RCPSP**. We will have a closer look at Preemptive-Resume RCPSP later. It is also conceivable to impose release dates or due dates, or both, on the activities. In the latter case, the activity is said to possess a **time window**. Situations also exist where the activities have specific times, called **time schedule**, at which they can start. Note that time windows can be incorporated within the context of Generalized Precedence Relationships which are discussed later in this section.

One last twist on the activities' attributes is the **State Preserving Activities** case of

RCPSP. A state preserving activity is one that has its start time determined by the activities preceding it and its finish time determined by the ones succeeding it. This might be useful to modeling limited buffer space in between two workstations that could lead to blocking. Reportedly (Klein 2000 p.101), some scheduling software incorporate this concept yet no scientific research has been published to treat it.

### 1.3.2 Resources

The second defining element of basic RCPSP are the resources. One of the requirements in the definition of the problem is that resource limits be constant throughout the project duration. This type of resources is referred to as **renewable resources**; labor and machines are two examples. But in the context of the multi-modal case other types of resources could be envisaged as well. Resource availabilities can be constrained for the duration of the project not on a per period basis as for the classic RCPSP. This type of resources is referred to as **non-renewable resources**. An example would be materials needed by the activities of the project. And, of course, resources can be limited on a per period basis as well as on the global basis. Such resources are called **doubly constrained resources**; an example of those are budgets. A fourth type of resources in the multi-modal case are called **partially renewable resources**. They are resources limited globally as well as for subsets of time periods as opposed to, simply, time periods. Budgets could also be an example of such resources. This type has been recently introduced by Böttcher *et al* (1999). Still a fifth type of resources that are relevant in many project are the **continuous resources**. Energy is one good example. This resource type can be approximated to be discrete and treated as such or, with additional assumptions, they can be modeled as continuous. In the latter case, the new model would be significantly different.

Examples are the papers by Weglarz (Slowinski and Weglarz 1989) and the paper by Icmeli and Rom (1996).

Another important attribute of resources could be their **time dependence**. A major cause of this is the temporary unavailability of resources due to sharing them among different projects. But there are others such as the scheduled maintenance of machines which results in their temporary unavailability. Time dependent resource availabilities are incorporated within the context of Generalized Precedence Relationships.

### 1.3.3 Precedence Relationships

The third defining element of RCPSP is the precedence structure. In this model, "activity $i$ precedes activity $j$" says that $i$ has to be completely finished before $j$ can start. Although this may be the appropriate relationship to model in many situations, it is not suitable for others. For example, activity $j$ may be eligible to start after the start of $i$ by a given time lag, possibly before $i$ finishes. This relationship between $i$ and $j$ is referred to as a **start-to-start** relationship, SS for short. It could also be that $j$ needs to be finished a given time lag after the start of $i$. This relationship is called **start-to-finish**, SF. In the same way, we can define **finish-to-finish**, FF, and **finish-to-start**, FS, relationships. These new types of precedences give rise to the **Generalized RCPSP**, or **GRCPSP** for short. This model can accommodate activity release and due dates as well as time varying resource limits. For a recent extensive discussion of this problem the reader is referred to Klein (2000). Note that the FS precedence with zero time lag is the precedence relationship of the basic problem.

### 1.3.4 The Objective Function

Last but not least, the fourth defining element of RCPSP is the objective function. The classic objective has been the minimization of the project duration, or **makespan**, denoted by $C_{max}$. But this objective can be modified in many ways; each leading to a new perspective on the schedule sought. One way to modify the objective, borrowed from Scheduling Theory, is to impose due dates on the activities. One objective that could be considered, then, is the **maximum lateness**, $\mathrm{L}_{\mathrm{max}}$, which calls for minimizing the maximum due date violation. A second objective is minimizing the **tardiness** objective, $\sum\limits_{j} T_j$, where $T_j = \max(C_j - d_j, 0)$, with $d_j$ as the due date of activity $j$ and $C_j$ is its completion time. Effectively, it minimizes the average job tardiness of the feasible schedules. Naturally this can be extended to the **weighted tardiness** objective, $\sum_j w_j T_j$, where the $w$'s are positive weights. A third and fourth objectives, in this context, could very well be the **total completion time,** $\sum_j C_j$, which minimizes the average activity completion time, or the **weighted completion time,** $\sum_j w_j C_j$.

Another objective modification consists of imposing a deadline on completing the project under which the problem may be infeasible due to lack of resources. To complete the project, additional resources would need to be enlisted in certain time periods at additional costs, of course. The question becomes how to meet the project deadline in minimal cost. This is known as the **Time Constrained Project Scheduling Problem**, TCPSP. Kolisch (1995) discusses a heuristic to solve it.

Another objective that sometimes needs to be considered in practice is the **net present value** objective. This situation arises when a contractor working on a project is paid not at the beginning nor at the end of the project but when completing each activity, or subsets thereof. If

payment is the same irrespective of when it is made, then it is in the interest of the contractor to realize it as soon as possible, so as not to loose some of it to inflation, for example. Additionally, the contractor may want to fund other activities of the project or even redirect the payments towards other projects. If $r$ is the inflation rate and $p_j$ is the payment for completing activity $j$, then $j$ is worth $p_j.(1-r)^{C_j}$ upon completion.. It is natural for the contractor, then, to seek the maximization of $\sum_j p_j.(1-r)^{C_j}$ subject to the precedence and resource constraints[2]. This problem is called the **Resource Constrained Net Present Value Problem**, RCNPV. It has its own variations and extensions; such as to include generalized precedence relations. For a recent paper on this topic the reader is referred to De Reyck and Herroelen (1998).

Still other objectives than $C_{max}$ are the resource leveling objective and the resource investment one. In the **Resource Levelling Problem**, RLP, the aim is to limit the fluctuations in the resources usage in the hope of reducing the costs associated with the fluctuations. The idea is to obtain the minimum project duration and then possibly reshuffle the activities within the minimum project duration to level the resources usage. Bandelloni *et al* (1994) deal with this problem. The **Resource Investment Problem**, on the other hand, is the problem of determining the resource requirements of a project given the project duration. It arises in the definition phase of the project rather than in the planning phase. Per period-unit costs are associated with the resources, $c_r$, and an estimate of the makespan is determined. The objective is to minimize the total cost $\sum_r c_r a_r$ where $a_r$ denotes the per period availability of resource $r$. Demeulemeester (1995) studies this problem.

A radically different approach to project scheduling has been studied by Icmeli and Rom (1998). The authors conducted a survey of project managers nationwide and across different

[2] A simple modification takes care of the payments upon completion of subsets of activities case.

industries. The objective of the survey was to determine the most important project characteristics that managers focus on. The result of the survey was that quality was, by far, the most sought after objective. Quality, they state, is manifested when the amount of rework that goes into activities is kept to a minimum. To deal with this issue the authors define a significantly different model from RCPSP or its variants mentioned above. The model turns out to be a lot easier to solve than RCPSP (Icmeli and Rom 1997).

Note that all the objectives mentioned so far have been single criterion objectives, at least on the surface. It is conceivable that in some situation one may want to explicitly consider two or more of these objectives either by combining them into one global objective or by approaching them in sequence as in the Goal Programming approach. We also note that one can combine characteristics of the extensions mentioned so far to come up with more variations and extensions. For example, suppose a project requires a single resource and that the activities are multi-modal; the more resource allocated to an activity the shorter its duration and the higher its cost. Assume also that the resource is non-renewable. One may be interested, in this case, in scheduling the activities to minimize the project duration or minimize the project cost or even generate the time/cost trade-off curve. This problem is called the **Discrete Time Cost Trade-off Problem**, DTCTP, and has been extensively studied since the early days of Project Management. For a treatment of this problem, the reader is referred to Elmaghraby (1977, Chapter 2).

### 1.3.5 Classification Schemes

The multitude of extensions and special cases that RCPSP possesses has prompted a search for a notation scheme, different from the alphabet soup we have used here, that concisely and

precisely describes any RCPSP variant. It is important that this notation scheme be compatible with the notation scheme that has been adopted for machine scheduling because many problems of the latter are special cases of RCPSP variants. Two such schema have recently been proposed by Herroelen *et al.* (1999) and Brucker *et al.* (1999). A problem description, in both schemes, consists of three fields. The first field is reserved to describe the resource environment, the second field describes the activity characteristics, which for both teams includes the precedence structure, and the last field states the objective function being considered. But whereas the Herroelen *et al* scheme tries to emulate the notation scheme in Scheduling Theory, it is not compatible with that scheme. Additionally, it is elaborate enough so that one would often have to consult the definitions of its parameters to decipher their meanings. The scheme by Brucker *et al*, on the other hand, is more compatible with that of Scheduling Theory and allows the practitioner to be less dependent on frequent consultations of the definitions. According to this latter scheme RCPSP is $PS\,|prec|\,C_{\max}$ , where 'PS' stands for Project Scheduling and 'prec' stands for the existence of precedence relationships. It remains to be seen which notation scheme will be adopted by the project scheduling community, if at all.

## 1.4 Existing Methodologies

As with any NP-Complete problem, the size of general instances that could be solved to optimality in reasonable time is limited. And of course, as with any other NP-Complete problem, it is not easy to determine cut off sizes below which every instance is solvable in reasonable time and/or above which every instance is guaranteed to require unreasonable computing time. In fact, even if such cut off's were determined for the known pool of exact methods and current

hardware technology, they are bound to eventually change with the advent of new methods and hardware. We shall see in this section that with the current state of affairs, many instances with 30 activities have been solved by a variety of exact methods and that many instances with 60 activities are so far intractable.

For cases where an instance size is clearly too large for current exact methods, heuristic approaches are the only way to reach a feasible solution. No bound on the quality of the solution can be given, however, as reportedly "no polynomial time approximation algorithm with a performance guarantee less than $n^{\varepsilon}$ " can be given unless $P = NP$ (Möhring *et al* 1999)[3]. Here $n$ represents the size of an instance and $\varepsilon > 0$ is an arbitrary error parameter. To gauge the quality of a particular heuristic solution, though, lower bounds on the optimal value of an instance furnish what perhaps is the only available test. If a lower bound is judged close enough to a heuristic solution then that solution may rightly be deemed acceptable. Else, if the difference is seen inadmissible then a better lower bound needs to be obtained and/or the heuristic solution is poor.

In this section we start with a review of some lower bounds. We then move to a review and critique of what presently are, or until quite recently were, viewed as promising exact methods. We end the section with a discussion of heuristics proposed for the problem.

### 1.4.1 Lower Bounds

We have already mentioned one significant reason for studying lower bound methods. Another important one is their use within the most common exact methods for solving RCPSP: Branch

[3]Möhring *et al* reach this conclusion by noting that the same applies to the Node Coloring Problem which, they remark, is a special case of RCPSP according to a result by Schäffter (1997).

and Bound methods. In such methods, a comparison of the lower bound at a node of the search tree with the current or, sometimes, local upper bound on the optimal solution is needed to decide whether to fathom the corresponding node or not. Additionally for many depth first search methods, the lower bounds at the nodes are also used to select the node from which the search is to proceed from. Namely, the node with the lowest lower bound amongst the descendents of the current one.

Examples of lower bounds in the literature abound. They generally depend on relaxing one or more characteristic of the problem. Some are the solution, or sometimes a solution, to a relaxed integer programming formulation that may omit several constraint sets. Others are derived by simply ignoring imposed limitations such as resource limits or precedences. As such, different lower bounds may have different qualities in terms of how close they can be expected to get to an optimal solution. Often, the better the expected quality of the bound, the more computationally expensive it may be. But that may not always be the case. Naturally, the choice of the one to use in a particular situation should depend on the situation itself. If the lower bound is to be used infrequently, such as for comparison with a heuristic solution or at the root node of a search tree for example, one might want to select a 'good quality' bound eventhough it may be computationally costly. If a lower bound is to be repeatedly used within a procedure, on the other hand, a balance between the quality of the bound and its computational burden needs to be sought. In this case, the choice is generally made only after trying several lower bounds within the procedure and comparing the performance of the resulting algorithms.

An extensive survey of existing lower bounds can be found in Klein's book (Klein 2000). It comprises several lower bounds developed by Klein and Scholl together with a computational study of the quality of most of the bounds detailed. In what follows we briefly examine seven of

what we think are the more important bounds in the literature; in terms of either effectiveness, widespread use within leading exact methods or just simplicity. Some of these will be more elaborated on with their corresponding exact method in the following subsection.

#### The CPM Bound

This is one of the simplest lower bounds for RCPSP, both conceptually and computationally. It consists of ignoring the resource limitations and applying the previously mentioned forward pass to obtain the early finish time for the project. This constitutes a lower bound on the optimal project duration. It is not a terribly effective bound but it is computationally cheap. It was used within a revised version of what was perceived as the foremost RCPSP exact procedure of its time, the Demeulemeester and Herroelen procedure of 1992 (cf. Demeulemeester and Herroelen 1997).

#### A Capacity Bound

This is another simple bound derived by considering, for each resource, the cumulative resource requirement of the project's activities versus the per period availability of the resource. The lower bound is the maximum of the corresponding ratios over the different resources. We are not aware of any competitive exact method that uses this bound. But it could, as we shall shortly see, be part of a more involved lower bound strategy, namely destructive improvement.

#### A Critical Sequence Lower Bound

This is the bound used by Stinson *et al* (1978) in the authors' branch and bound method. It was also adopted by Demeulemeester and Herroelen in their 1992 procedure. An extended version

of it was later developed for the same procedure but both were eventually dropped in favor of the simpler and, for that procedure, provenly more effective CPM bound (Demeulemeester and Herroelen 1997). We expand on it in our review of their approach.

### A Truncated LP Relaxation Based Bound

This is a bound based on the relaxed integer programming formulation of RCPSP in Mingozzi *et al* (1998). In this formulation, all constraint sets but one are eliminated and the remaining one transformed into an inequality set. The resulting linear program can be generated and solved for projects with up to 30 activities (Brucker *et al* 1998). For larger size projects, with 60 or 90 activities, its columns become too numerous to generate. But Mingozzi and his collaborators did not report solving it directly for such size problems. Instead, they turned to its dual and came up with the next bound we review, the weighted node packing bound. Brucker and his collaborators (Brucker *et al* 1998), on the other hand, made several attempts at a direct solution before they found one they were convinced was the best to use with their procedure (Brucker and Knust 1999). This happened to be a hybrid of column generation and the destructive improvement strategy which we will discuss shortly. Mathematically, it is one of the more sophisticated bounds that we encountered. It is also the best in terms of deviation from the optimum. Not too surprisingly though, its computational burden is one of the largest as well.

### A Weighted Node Packing Bound

This is the bound used by Mingozzi and his collaborators in their branch and bound procedure (Mingozzi *et al* 1998). The authors derived it by considering an integer formulation of RCPSP

and going through a sequence of truncated, relaxed, dualized and integerized versions of it. We cover this approach in our review of the procedure but note that it can also be derived more simply by the following argument in Klein (2000).

Basically, if two activities cannot run concurrently due to precedence relationships or resource requirements then they must be scheduled sequentially. Hence, a subset of the activities that pairwise cannot be concurrent must have its total of the activities' durations as a lower bound on the optimal duration of the project. Constructing a graph to model the situation where each vertex matches an activity, each vertex weight matches the duration of its corresponding activity and each edge corresponds to the feasibility that the activities matching its adjacent vertices be concurrent, one realizes that the goal must be to find the independent subset of the graph with maximum total weight[4]. This is exactly the Weighted Node Packing Problem that Mingozzi and his colleagues arrived at.

Since the problem is NP-Complete, the authors solve it heuristically to obtain a relatively good and not too expensive bound. Demeulemeester and Herroelen (1997) used the same model but a different heuristic as the lower bound of their updated procedure. Sprecher used still a third heuristic for his (Sprecher 2000). Furthermore, Klein and Scholl have developed one extended and another generalized versions of this same bound that reportedly are more computationally expensive but tighter (Klein 2000).

[4] A set of vertices of a graph is said to be independent if no two vertices in the set are adjacent (*i.e.* connected with an edge).

**The Destructive Improvement Strategy**

This is a general strategy that can be adapted to deduce a lower bound, or even an optimal solution, for many types of problems not just RCPSP. The basic idea is as follows. Starting with an arbitrary lower bound $LB$, examine the bound extensively to detect whether it might be infeasible. If infeasible, the bound can be increased by at least one and the process restarted. Otherwise if infeasibility cannot be proven, the lower bound has to be accepted. Needless to say, the effectiveness of this strategy strongly relies on the effectiveness of the infeasibility test, or tests, employed.

Klein and Scholl's adaptation of this strategy to RCPSP (Klein 2000) involves a two-phase infeasibility test. The first phase consists of applying so called reduction techniques with the aim of deducing sharp time windows for the activities of the project. It is quite possible that while deducing these time windows, an infeasibility of the lower bound is detected. But if not, the test moves to its second phase. Here, the time windows are used in the application of several simple lower bound techniques, such as any or all of the previously mentioned ones albeit possibly with some modification. If a lower bound is detected that is larger than the latest one used in the reduction techniques then we know that this latter one, *i.e.,* the bound used in the reduction techniques, is infeasible and so must be increased to the newly deduced one[5]. If not then the same simple lower bound techniques are applied to judiciously chosen time intervals in a further effort to detect infeasibility[6].

To illustrate what those **reduction techniques** are about, we present one based on the

[5] Klein doesn't quite make this conclusion but we think this step is in the spirit of the same strategy.

[6] In principle, the bounds should be applied to every time interval in the project's estimated span but since the number of those subintervals is large, the authors limit them to the early start/latest finish times of the activities. Klein mentions sharper choices of intervals are also possible.

notion of **core time** of an activity. Given an estimate of a project's duration, this is the interval of latest start/earliest finish times of the activity. This interval is empty when the earliest finish time is less than the latest start time but otherwise its length never exceeds the duration of the activity[7]. This says that an activity is active in its core time, if one exists. Now, with the existing core times, the resource requirements are checked against their limits. If the limits are violated then the lower bound is infeasible and so has to be increased. If they are not, each activity is tested for scheduling within its time window taking into account the resource requirements of the other activities within their core times only. If, in this case, it so happens that an activity cannot be scheduled within its time window then the lower bound is infeasible. But if it happens that an activity can be started at only one possible time then this starting time is fixed and the time windows of the other activities are adjusted accordingly. This may result in new time cores in which case the process is repeated. If, on the other hand, one of the activities cannot be scheduled within its time window then the lower bound is again infeasible and has to be increased.

Now, the Klein and Scholl's destructive improvement lower bound for RCPSP in its full version includes two more reduction techniques and eight simple lower bounding techniques, four of which are all the previously mentioned ones except for the truncated LP relaxation bound. Clearly then, we have only covered its essence here. Computational results by the authors show that the full version is one of the best lower bounds available for RCPSP, in terms of deviation from the optimum, though it is not as strong as the Brucker and Knust hybrid bound and requires the longer running time of the two on the J60 instances (Klein 2000)[8].

[7]If the core time of an activity has length larger than its duration then the earliest start of the activity is larger than its latest start which is absurde.

[8]We report on this data set in a latter section.

More importantly, computational experiments also show that 'lite' versions of the full one run a lot faster and for only a small reduction in bound quality. Evidently then, the choice of which version of this bound to use with which procedure can only be procedure dependent.

**A Lagrangian Relaxation Bound**

This is a bound by Möhring *et al* (1999) based on an integer programming formulation of the similar problem with temporal constraints instead of simple precedences. If $i$ and $j$ are two of a project's activities, a temporal constraint between $i$ and $j$ takes the form: $S_j \geq S_i + d_{ij}$, where $S_k$ is the start time of an activity $k$ and $d_{ij}$ is a time lag that could be negative. Note that if $d_{ij}$ is the duration of activity $i$ then the temporal constraint reduces to the regular precedence relationship. Consider a project of $n+2$ activities including the two dummy ones. Let $T$ be an estimate of the project duration. For all $j = 0, \ldots, n+1$ and $t = 0, \ldots, T$, let $x_{jt} = 1$ if activity $j$ starts at time $t$ and $x_{jt} = 0$ otherwise. Let $H$ be the set of pairs of activities $i$ and $j$ with temporal constrains among them and let $d_{ij}$ denote the corresponding time lags. Let $r_{jk}$ be the requirement of activity $j$ from resource $k$ and let $R_k$ be the availability of that resource. Finally, let $p_j$ be the duration of activity $j$. The problem can be formulated as:

$$\begin{aligned}
(P) \quad &: \quad Min \sum_t t x_{n+1,t} \\
s.t. \quad \sum_t x_{jt} &= 1, \forall\ j, \\
\sum_{s=t}^{T} x_{is} + \sum_{s=0}^{t+d_{ij}-1} x_{js} &\leq 1, \forall (i,j) \in H \quad and\ \forall\ t, \\
\sum_j r_{jk} \left( \sum_{s=t-p_j+1}^{t} x_{js} \right) &\leq R_k, \forall\ k, t, \\
x_{jt} &\in \{0,1\},\ \forall\ j, t.
\end{aligned}$$

The first set of constraints insures that an activity is started only once. The second insures that if $(i, j) \in H$ and $i$ starts on or after time $t$ then activity $j$ cannot start by time $t + d_{ij} - 1$, *i.e.*, it must start on or after time $t + d_{ij}$. Since this is particularly true if activity $i$ starts at time $t$, the time lag between $i$ and $j$ is preserved. The third set of constraints clearly insures that the resource limits are preserved. We note here that the restriction of this formulation to RCPSP, *i.e.,* the case where $d_{ij} = p_i$ for all $(i, j) \in H$, has much in common with one of the first mathematical formulations of RCPSP; namely that of Pritsker *et al* (1969). The difference between the two is the second set of constraints which in the formulation here has $T + 1$ as many constraints and so is stronger when relaxations of the integer programs are considered.

To obtain their lower bound, Möhring and his collaborators propose a Lagrangian relaxation of the above model in which the third set of constraints is dualized. As with any Lagrangian relaxation procedure, if $\lambda \geq 0$ is the multiplier used to dualize the constraints then the minimum value of the Lagrangian, for fixed $\lambda$, is a lower bound on the optimal objective value of the original problem. Therefore, to obtain the best lower bound on that optimal value, the aim should be to find the $\lambda \geq 0$ that maximizes the lower bound. Usually, and this is what the researchers have done, the maximization is done using subgradient optimization which utilizes the optimal $x^*(\lambda)$ from the Lagrangian minimization step. Now, other researchers have previously used this same approach; namely Christofedes, Alvarez-Valdez and Tamarit (1987)[9]. But whereas Christofedes and his collaborators have used a Branch and Bound procedure to obtain the optimal $x^*(\lambda)$, Möhring and his collaborators remark that it has subsequently been shown that LP relaxation of the Lagrangian has an integral solution anyway and, in a further push, transform it into a Minimum-Cut problem for an even faster solution.

[9] The reader interested in this paper should also consult Demeulemeester *et al* (1994).

The resulting lower bound has been compared to the Mingozzi *et al* node packing bound and to the Brucker and Knust column generation/destructive improvement bound on the J60, J90 and J120 data sets which we discuss in a latter section. The authors report that their lower bound is tighter than the first but that it is not as sharp as the second, though it is much faster.

### 1.4.2 Exact Methods

Indubitably, the most compelling and most obvious reason for developing exact methods for any problem is to obtain proven optimal solutions for it, to be certain that once such a solution is secured no other possible solution out there can be any better. Naturally, when several exact methods for a problem are available, the question of which of those is the more efficient is swiftly raised. Efficiency, here, could be in terms of computer execution speed and/or required memory space but, regardless, for NP-Complete problems it ultimately translates into the largest size for which an instance is solvable on a given computer with a given configuration and in a given amount of time. Consequently, a second reason to develop an exact method for an NP-Complete problem is to attempt to expand the size of what generally is recognized as a solvable instance in reasonable time. Still, a third important reason is to provide a direct means to assess the quality of lower bounds and heuristics on small size instances in an effort to gauge their quality on larger size instances where exact methods are simply impractical.

Over the past forty years or so that RCPSP has been studied, numerous exact methods were proposed for its solution. The vast majority of these can be classified either as integer programming methods or as enumerative methods. Historically, Integer Programming was first to be tried. The study by Pritsker *et al* (1969) was among the first to present an integer programming formulation for the problem. But the subsequent lack of success of such attempts at yielding

the solutions sought has prompted the research into techniques of the second class (Patterson 1984). Within the class of enumerative methods, several approaches could be distinguished among which are Dynamic Programming, Implicit Enumeration, Bounded Enumeration and Branch and Bound. Among these, Branch and Bound methods were eventually recognized as the most successful. But as we shall see, success here is only relative as the size of instances that can generally be expected to be solved within reasonable time remains rather small.

This subsection is devoted to the review and critique of what we came across in the literature as the more successful exact procedures. They happen to be all of the Branch and Bound variety. Each of these is based on a completely different perspective of how a solution to the problem should be derived. The first family we examine, the DH family, has until recently been recognized as the family of the most efficient exact methods for RCPSP. The second procedure we shall be interested in, BBLB3, outperforms the first on certain types of problems but more importantly has generated the previously mentioned popular LB3 bound. The third, BKST, has prompted the creation of a heuristic and the best lower bound available for the problem. The fourth, GSA, employs an extensive array of dominance rules. And last but not least, the last two procedures, Progress and Scatter, are not only very competitive computationally but also implement new concepts for Branch and Bound that could very well be employed for other types of problems.

We have intentionally left out, from this review, methods that eventhough were interesting from a theoretical point of view, were not competitive computationally. Noteworthy among these are the bounded enumeration procedure of Davis and Heidorn (1971), the implicit enumeration procedure of Talbot and Patterson (1978), the branch and bound procedure of Stinson *et al* (1978), the artificial intelligence A* search method of Bell and Park (1990), the parallel

procedure of Simpson and Patterson (1993) and the recent fractional cutting plane algorithm of Sankaran *et al* (1999). Two interesting studies, from a mathematical point of view, that we have also left out are one by Möhring and Radermacher (1989) and another by Alvarez-Valdés and Tamarit (1993). The first discusses, among other concepts, the issue of project decomposition. The second discusses the facets of the feasible polyhedron of a disjunctive integer programming formulation of the problem and introduces cutting planes to help in its solution.

### The DH Procedures

These are three RCPSP procedures that Demeulemeester and Herroelen have introduced starting in 1992. Each successive procedure is based on its predecessor and is meant to be a significant improvement over it. All are branch-and-bound (BaB) based. In the following, we describe the first, which the authors refer to as **DH**, and then point out the changes that make up the second and third improvements. The first procedure was later adapted for the preemptive RCPSP (the resume variant). We briefly describe that one last in this subsection.

The nodes of the BaB tree of the DH procedure correspond to **partial schedules** $PS_m$ at times $m = 1, 2, \cdots$ where temporary finish times are assigned to a subset of the activities of the project. The partial schedules are precedence and resource feasible and are built up by adding subsets of eligible activities at time instants corresponding to the completion of temporarily scheduled activities. The search strategy is the depth first strategy.

Branching in DH is carried out when, due to resource limitations, not all precedence-feasible activities can be started simultaneously. Three theorems were developed to limit the number of possible branches. The first asserts that if an activity is precedence-feasible, at time $m$, but cannot be scheduled with any other unscheduled activity due to the resource restrictions then

there exists an optimal schedule in which this activity starts at time $m$. The second theorem states that if at time $m$ no activity is in progress, that if there is an activity $i$ that can be concurrently scheduled only with activity $j$ at any time $m' \geq m$ without violating the resource constraints and that if, further, $i$ is longer in duration than $j$, then there exists an optimal continuation of the partial schedule in which both activities start at time $m$. In the third theorem, the researchers resolve resource conflicts by considering so called **minimal delaying alternatives** only. As the name might suggest, a delaying alternative for $PS_m$ is a subset of previously scheduled activities at time $m$ the delay of which resolves a resource conflict and allows other of the precedence feasible activities at time $m$ to be concurrently processed. A delaying alternative is minimal if it does not contain any other delaying alternative.

We note that the authors do not shed any light on how the minimal delaying alternatives are to be obtained. This is not a trivial concern as the brute force method to generate them is to separately test all the subsets of the precedence feasible activities, corresponding to $PS_m$, to check whether they qualify as minimal delaying alternatives or not. If the subset of eligible activities is large, this method would, obviously, have to be the last resort to generate these alternatives as the number of subsets to be tested is exponential even if the number of such alternatives is not. We will address this issue in a lot more detail in Chapter 3.

We mentioned earlier that the search strategy of DH is a depth first strategy. This calls for a rule to determine the order in which minimal delaying alternatives are to be explored. In this regard, the researchers have adopted the "**Critical Sequence Lower Bound**" of Stinson *et al* (1978), denoted by **LBS**. This is implemented in the following way. If $E_m$ is the set of eligible activities at time $m$ and if $D_q$ is a minimal delaying alternative then a set of extra precedences is added which ensures that the activities in $D_q$ are preceded by the earliest finishing activity

among the ones in $E_m$ and the ones in progress at time $m$. The new partial schedule, $PS'$, is now $PS_m + E_m - D_q$. A critical path, for the modified network, of duration $z$ and the set $NC$ of activities $i$ not on this path, *i.e.* non-critical, and not in $PS'$ are both determined. Further, using the earliest start times $es_i$ and the latest finish times $lf_i$ for $i \in NC$ and using the resource usage configuration of the critical path, the longest time, $e_i$, during which $i$ can be scheduled uninterrupted in the interval $[es_i, lf_i]$ is determined. It is clear now that the project duration has to be extended by at least $d_i - e_i$, where $d_i$ is the duration of activity $i$, to allow $i$ to be processed. Whence, the least amount of time the project should be extended by, or the critical sequence lower bound, is: $L_q = \max_{i \in NC}\{z + d_i - e_i\}$. Branching, now, proceeds with respect to the minimal delaying alternative with the smallest $L_q$, ties broken arbitrarily. Incidentally, the authors use this smallest $L_q$ to determine the lower bound at a node of the tree as the maximum of this $L_q$ and the lower bound at the parent node.

Demeulemeester and Herroelen make use of two dominance rules to prune the search tree. The first is a **Left-Shift Dominance Rule** proposed by Schrage (1970) and intended to prune partial schedules that contain activities that can be left shifted without violating the resource limits. It is supposed to yield a semi-active optimal schedule. However, as Sprecher and Drexl (1999) point out, the procedure can indeed yield no semi-active schedules. This is not to say that the dominance rule is useless, though. It just may not be implemented as effectively as it was meant to be. The second dominance rule is based on the concept of cutset at time $m$, denoted by $\mathcal{C}_m$. This is defined as the set of all unscheduled activities at time $m$, the predecessor of which all belong to $PS_m$. The researchers state and prove a theorem related to cutsets that says: if $k < m$ and if cutset $\mathcal{C}_m$ contains the same activities as a cutset $\mathcal{C}_k$ that was explored in another path of the tree; and if, further, all of the activities of $PS_k$ that are in progress at

time $k$ finished by the maximum of time $m$ and the finish times of the corresponding activity in $PS_m$, then $PS_m$ is dominated. This is referred to as the **Cutset Dominance Rule**. In retrospect, this second dominance rule is a very important contribution in the procedure as it was proven to significantly reduce the solution time for relatively hard instances.

The second DH procedure, which Demeulemeester and Herroelen refer to as **DH1**, appeared in the literature in 1997. Its introduction was, at least partly, prompted by the emergence of a new benchmark data set for RCPSP, J30, and the new Mingozzi *et al* (1998) procedure; both of which we will report on later. It turned out that DH could not solve 52 of the 480 instances of J30 in reasonable time while the Mingozzi procedure could. DH1 was meant to rectify this. Its implementation for the newly introduced 32-bit Windows operating system took advantage of several special features of that operating system. This allowed, among other things, a more efficient implementation of the cutset dominance rule. The researchers experimented also with another lower bound that Mingozzi *et al* came up with. They found their implementation of the new bound, denoted **LB3**, more efficient to use. Furthermore, they experimented with the amount of allotted addressable computer memory the size of which they found can greatly affect computational time as the cutset dominance rule requires large computer memory to implement. Additionally, they examined two more search strategies, Best First and a hybrid Best First-Depth First, and concluded that Depth-First is the preferable one[10]. Computational results have shown that the new procedure solves 479 of the 480 J30 problems in reasonable time. In our view, among all the changes that the new DH procedure introduced, the more fundamental one is the use of the newer lower bound due to Mingozzi *et al*.

[10]Here, Best First refers to selecting the node of the BaB tree with the least lower bound as opposed to the node with the least lower bound among the descendents of the last node generated, as in Depth First. The hybrid refers to using Depth First in branching and Best First in backtracking.

To the best of our knowledge, the third DH procedure has not been published yet but has been mentioned in the Herroelen *et al* review paper (1998). The researchers indicate that the main changes there include a new lower bound, a new scheduling rule to replace the one in the first two theorems, a new version of the cutset dominance rule, and the preprocessing of instance data.

An adaptation of the first procedure, DH, to the preemptive RCPSP was published by the authors in 1996. The important change needed to use DH directly was splitting the activities into subactivities of unit durations with the same resource requirements. Each activity of the original network is now a chain of subactivities. Note that the only activities needed to have zero duration are the start and end ones as an AoN representation avoids the use of other dummy activities. The pruning and dominance rules of the non-preemptive case carry over to the preemptive one. In the same context, the authors consider also the case where the resource limits are variable. This necessitates a change in the BaB algorithm as a semi-active optimal schedule need not exist anymore and the adapted first theorem no longer holds. But these are the main changes required. In their computational tests, the researchers experience a 33 fold increase in the computational time required to solve the Patterson set of problem for a mere 0.78% decrease in project length. A 12 fold increase of computational time was experienced on the Simpson data set, which is the same as the Patterson set but with variable resources, for a 2.88% improvement in project length. Finally, we point out that DH was also adapted for GRCPSP, the generalized RCPSP (Demeulemeester and Herroelen 1997 b) mentioned in section 1.3.3.

**BBLB3**

This is a BaB procedure developed by Mingozzi *et al* (1998) based on a new integer programming formulation of the problem. The BB in the acronym stands for "branch and bound" and the LB3 for the bound that we mentioned previously in connection with DH1. We summarize it here and suggest the original Mingozzi *et al* paper for the reader interested in all the details.

Let $X = \{1, \ldots, n\}$ be the set of activities of a project where activities 1 and $n$ are the start and end activities, respectively, and let $X' = X \backslash \{1, n\}$. A subset $R \subset X'$ is said to be feasible if the activities of $R$ are resource compatible and no precedence relationship exists between any two of its activities. A solution to the problem, of completion time $t^*$, is a sequence $S$ of feasible subsets of activities in progress at times $t$, $S = \left(R_{l_1}, R_{l_2}, \ldots, R_{l_{t^*}}\right)$, that preserves the precedence relationships and does not manifest activity preemptions. Let $\mathcal{R}$ be the index set of all feasible subsets and let $\mathcal{R}_i$ be the index subset of $\mathcal{R}$ corresponding to the feasible subsets containing activity $i$. Let $y_{lt} \in \{0,1\}$ be 1 if and only if all the activities of $R_l$ are in execution at time period $t$ and let $\xi_{it} \in \{0,1\}$ be 1 if and only if activity $i$ starts at time period $t$. Let $d_i$ be the duration of activity $i$ and $es_i$ and $ls_i$ be its earliest and latest start times under the CPM model. Let $H = \{(i,j) \in X \times X : \ i \ precedes \ j\}$ and let $T_{\max}$ be an upper bound on the project duration. The Mingozzi *et al* integer programming formulation of RCPSP is:

$$(P) \quad : \quad Min \quad z_P = \sum_{t=es_n}^{ls_n} t\xi_{nt}$$

$$\begin{aligned} s.t. \quad \sum_{l \in \mathcal{R}_i} \sum_{t=es_i}^{ls_i} y_{lt} &= d_i, \quad \forall\, i \in X', \\ \sum_{l \in \mathcal{R}} y_{lt} &\leq 1, \quad \forall\, t = 1, \ldots, T_{\max}, \end{aligned}$$

$$
\begin{aligned}
\xi_{it} &\geq \sum_{l \in \mathcal{R}_i} y_{lt} - \sum_{l \in \mathcal{R}_i} y_{l,t-1}, \quad \forall\, i \in X', \; t = es_i\,, \ldots, ls_i, \\
\sum_{t=es_i}^{ls_i} \xi_{it} &= 1, \quad \forall\, i \in X, \\
\sum_{t=es_i}^{ls_i} t\xi_{jt} - \sum_{t=es_i}^{ls_i} t\xi_{it} &\geq d_i, \quad \forall\, (i,j) \in H, \\
y_{lt} &\in \{0,1\}\,, \quad \forall\, l \in \mathcal{R}, \; t = 1\,, \ldots, T_{\max}, \\
\xi_{it} &\in \{0,1\}\,, \quad \forall\, i \in X, \; t = es_i\,, \ldots, ls_i.
\end{aligned}
$$

The objective specifies the minimization of the start time of activity $n$; which is the same as the completion time of the project. The first set of constraints insures that feasible subsets containing activity $i$ are in progress exactly $d_i$ time periods, for any $i \in X'$. The second set insures that at any time period $t$, only one feasible subset is in progress. The third set insures that $\xi_{it}$ is 1 if $i$ is in progress at time period $t$ but not time period $t-1$, *i.e.,* if $i$ starts at time period $t$. The fourth set makes sure that $i$ starts only once and the last set insures that the precedence constraints are respected. Together, the third and fourth set guarantee that no activity preemption is allowed.

Notice that a feasible subset can be in progress at different times. The length of time it is in progress is given by $\sum_{t=1}^{T_{\max}} y_{lt}$. Since by the second constraints' set at most one feasible subset is in progress during any time period, the project completion time can be given by $\sum_{l \in \mathcal{R}} \sum_{t=1}^{T_{\max}} y_{lt}$. This says that an alternative objective function can be: $Min \quad z_P = \sum_{l \in \mathcal{R}} \sum_{t=1}^{T_{\max}} y_{lt}$. Notice also that the integer program requires a number of variables in the order of the number of feasible subsets which, potentially, is exponential in the number of activities. Consequently, it cannot be solved for large or even medium sized problem instances. This prompted Mingozzi and his colleagues to exploit $(P)$ by obtaining lower bounds on the completion time for use in

a BaB procedure.

The five bounds they derived are all based on deleting all but the first set of constraints of $(P)$. This allows replacing $\sum_{t=1}^{T_{\max}} y_{lt}$ with a variable $x_l$ to get the following linear program:

$$
\begin{aligned}
(RP1) \quad &: \quad Min \quad z_{RP1} = \sum_{l \in \mathcal{R}} x_t \\
s.t. \quad \sum_{l \in \mathcal{R}_i} x_l &= d_i, \quad \forall\, i \in X^{'}, \\
x_l &\geq 0, \quad \forall\, l \in \mathcal{R}.
\end{aligned}
$$

The optimal solution to $(RP1)$ is clearly a lower bound on the optimal value of $(P)$. It is denoted by LB1.

A second bound that the researchers came up with is by replacing the equality constraints of $(RP1)$ with corresponding inequalities of the " $\geq$ " variety. They prove that with this relaxation there is no need to consider feasible subsets that are contained in other feasible subsets; which significantly reduces the number of subsets that need to be accounted for. Of course, the new linear program, $(RP2)$, has its optimal solution bounded from above by LB1 and so is itself another lower bound to the original problem $(P)$. They denote it by LB2 and show that $LB2 \geq LB0$, the CPM lower bound.

Now, let $\mathcal{M}$ denote the reduced feasible subsets index set. The dual of $(RP2)$ is:

$$
(DRP2) \quad : \quad Max \quad z_{DRP2} = \sum_{i \in X'} d_i u_i
$$

$$
\begin{aligned}
s.t. \quad \sum_{i \in \mathcal{R}_i} u_i &\leq 1, \quad \forall\, l \in \mathcal{M}, \\
u_i &\geq 0, \quad \forall\, i \in X'.
\end{aligned}
$$

By Linear Programming Duality, any feasible solution to $(DRP2)$ is a lower bound to LB2. This inspired Mingozzi and his colleagues to use heuristic procedures to solve $(DRP2)$ and thus deduce more lower bounds for $(P)$. Three more lower bounds were derived this way by exactly and heuristically solving an integerized $(DRP2)$ known as the **Set Packing Problem**:

$$
(IDRP2) \quad : \quad Max \quad z_{IDRP2} = \sum_{i \in X'} d_i u_i
$$

$$
\begin{aligned}
s.t. \quad \sum_{i \in \mathcal{R}_i} u_i &\leq 1, \quad \forall\, l \in \mathcal{M}, \\
u_i &\in \{0,1\}, \quad \forall\, i \in X'.
\end{aligned}
$$

But it turns out that the Set Packing Problem is equivalent to the **Weighted Node Packing Problem (**Nemhauser and Trotter 1975**)**. The correspondence is as follows. Let $G = \left(X', E\right)$ be the graph where an edge $(i, j) \in E$ *iff* $i$ and $j$ are resource compatible and do not precede each other; in other words, $i$ and $j \in R_l$ for some $l \in \mathcal{M}$. Let $d_i$ be the weight associated with node $i$. The problem is to find an independent subset of $G$ with maximum total weight. In mathematical terms, the problem is:

$$
(WNP) \quad : \quad Max \quad z_{WNP} = \sum_{i \in X'} d_i u_i
$$

$$
s.t. \quad u_i + u_j \leq 1, \quad \forall\, (i, j) \in E,
$$

$$u_i \in \{0,1\}, \quad \forall\, i \in X^{'}.$$

Now the effort is concentrated on $(WNP)$. Its optimal solution, denoted by LBP, provides one lower bound for $(P)$. Another lower bound, denoted with LBX, is furnished by a heuristic solution to $(WNP)$ due to Xue (1994). The trouble with LBX, though, is that sometimes it is smaller than LB0; which renders it unreliable. Still a third lower bound, LB3, can be derived by means of one more heuristic algorithm to $(WNP)$. Let $L = \left(i_1, i_2, \ldots, i_k, i_{k+1}, \ldots, i_{n'}\right)$ be a sequence of the vertices of $G$ s.t. $i_1, i_2, \ldots, i_k$ form a longest path in the CPM model and $i_{k+1}, \ldots, i_{n'}$ are arranged by non-decreasing order of vertex degree. The algorithm generates $n^{'}$ independent subsets that contain a different vertex each and chooses the one subset with the maximum total weight.

Each node of the search tree for the BaB procedure corresponds to a sequence of feasible subsets that preserves the precedence constraints and does not exhibit activity interruptions. Branching proceeds from the node with the largest number of distinct activities in its corresponding sequence. It consists of simply adding to the sequence a feasible subset that maintains the sequence feasibility. Since the number of feasible subsets that could be added may be quite large, three dominance rules are introduced. Let $\alpha$ be a node of the tree and $t$ the execution time of its partial schedule. Let $R_{l'}$ and $R_{l''}$ be two feasible subsets that could feasibly be added to the partial schedule to form descendent nodes $\beta^{'}$and $\beta^{''}$ of $\alpha$, respectively. Compute the execution time of $R_{l''}$ as $\tau^{''} = \min_{i \in R_{l''}} \{s_i(\beta^{''}) + d_i\} - t$, where $s_i(\gamma)$ is the starting time of activity $i$ when an arbitrary node $\gamma$ is continued. The authors prove that if $R_{l'} \subset R_{l''}$ and $d_i \leq \tau^{''}$, $\forall i \in R_{l'} \backslash R_{l''}$ then $\beta^{''}$ cannot lead to a better solution than the best one obtainable from $\beta^{'}$. This is their first dominance rule which, parenthetically, they fail to mention is an

immediate consequence of the global version of the second rule they use; namely, the (local) Left Shift Dominance rule applied in the DH procedures and several others[11]. The third dominance rule they employ is the Cutset Dominance rule first proposed by Demeulemeester and Herroelen (1992) and used in DH and its successors as well.

Mingozzi and his colleagues ran experiments to compare their bounds to one another and to LB0 and the Stinson bound, LBS. The data sets they used are the 110 Patterson instances and a set, denoted **KSD**, of 23 groups of 10 instances each chosen from two sets: J30 and a 200 instance set generated using ProGen[12]. Each group amongst these 23 was selected because either DH or BBLB3 required 40 or more seconds on the average to solve its instances. All the bounds they generated performed well on these two sets, though better on the Patterson set than on KSD. In general, the bounds were closest to the optimal solution in the following order: LB1, LB2, LBP, LB3, LBX, LBS and LB0. The difference between LB1 and LB2 was not significant. The same goes for the difference between LBP, LB3 and LBX and between LBS and LB0. LB1 and LB2 are on the average within 2.5% of the optimum on the Patterson set and within 6.2% on KSD. LB3, on the other hand, is on the average within 6.2% of the optimum on the Patterson set and 9.4% on KSD.

In view of the good performance of LB3 on the two data sets and its relatively small computational cost, the researchers chose it as the lower bound of their BaB procedure. LB3 attracted, even, the attention of other researchers; Demulemeester, Herroelen and Sprecher for example. Variations of it were subsequently adopted in their procedures as well. Mingozzi and his collaborators run experiments comparing their algorithm with DH. They discovered

[11]We will see shortly that Sprecher (2000) uses a local as well as a global version of this same rule in his GSA algorithm.

[12]We report on Progen and all of these sets in a latter section.

that BBLB3 dominates DH on problems where LB3 is greater than LBS but not when the two bounds are close[13]. This, it turned out, demonstrated the superiority of DH as BBLB3's success seems to rely on LB3; which, consequently, suggests incorporating LB3 within DH just as Demeulemeester and Herroelen have done. Besides this comparison, the authors evaluated the importance of the dominance rules by running BBLB3 with each of them missing. They observed, using KSD, that the absence of any of the three rules adversely affects the average solution time but that this is particularly true for the Cutset Dominance rule as its absence leads to a many folds increase in the average solution time. Incidentally in these results, the difference between the absence of the first and second rules is not too large.

**BKST**

This is a BaB procedure developed by Brucker, Knust, Schoo and Thiele (1998) based on the concept of schedule schemes. Its acronym stands for the first letters of its authors' names. Unlike other exact methods, this procedure aims to reach an optimal solution by fixing all possible concurrence and immediate precedence relationships among the activities of a project. We present its digest here and suggest the Brucker and Knust (1999) paper as a more detailed version of the original article, at least in a couple of aspects.

In the notation of the authors, let $V$ be the set of activities of a project and let $A = \{(i,j) : i,j \in V,\ i \neq j\}$. A schedule scheme is a quadruplet $(C, D, N, F)$ of pairwise disjoint subsets of $A$ defined as follows. $C$ represents the set of conjunctions, *i.e.,* precedence relationships, of the project where $(i,j) \in C$ is denoted by $i \rightarrow j$. $D$ represents the set of disjunctions,

[13] To reach this conclusion, they had to use to another set of 450 ProGen instances where LB3 and LBS were close as for KSD this was far from being the case.

denoted by $i - j$ , which are undecided conjunctions in the sense that $i - j$ if one of the two conjunctions $i \rightarrow j$ and $j \rightarrow i$ holds. $N$ represents the set of parallelity relationships: $(i, j) \in N$, denoted by $i \parallel j$, if activities $i$ and $j$ are concurrent for at least one time period. $F$ represents the set of flexibility relationships, *i.e.,* the undetermined ones; $(i, j) \in F$, denoted by $\grave{\imath} \sim j$, if $(i, j)$, $(j, i) \notin C$ and $(i, j) \notin D \cup N$. Now, given a schedule scheme $(C, D, N, \emptyset)$ of a project, its disjunctions can be fixed to yield a schedule scheme $(C^{'}, \emptyset, N, \emptyset)$. This new schedule scheme is called a transitive orientation of the first if $C^{'}$ is transitive in that whenever $i \rightarrow j$ and $j \rightarrow i$, we have $i \rightarrow k$. Clearly, transitive orientations are what we would like to deal with since the non transitive ones yield logical inconsistencies. Note that with a transitive $C^{'}$, it is simple to construct an early start schedule that disregards the parallelity relations. Of course, such a schedule may be infeasible due to violations of the resource limits.

At this point, two observations can be made which lie at the core of the BKST procedure. The first, which according to the authors is due to results of Golumbic (1980) and Mohring (1985), states that the earliest start schedules corresponding to transitive orientations of a schedule scheme have the same makespan regardless of the transitive orientations. The second, which is due to the authors themselves, states that if $\mathcal{S}_{es}(C^{'})$ is the earliest start schedule of an arbitrary transitive orientation, $(C^{'}, \emptyset, N, \emptyset)$, of a schedule scheme $(C, D, N, \emptyset)$ then either $\mathcal{S}_{es}(C^{'})$ is feasible or else no way of fixing the disjunctions in $D$ can yield a feasible solution.

In light of these results, nodes in the BKST search tree correspond to schedule schemes $(C, D, N, F)$. The root of the tree is the schedule scheme $(C_0, D_0, \emptyset, F_0)$ where $C_0$ is the set of conjunctions in the project's definition, $D_0$ is the set of disjunctions induced by the resource limits, *i.e.,* $(i, j) \in D_0$ if and only if $i$ and $j$ are resource incompatible, and $F_0$ is the set of pairs $(i, j)$ where $(i, j), (j, i) \notin C_0 \cup D_0$. Branching consists of choosing a flexibility $i \sim j \in F$ and

transforming it into a disjunction, $i - j$, or a parallelity relationship, $i \parallel j$. Hence, the tree is binary. The search strategy is a depth first strategy where the choice of flexibility relationship to branch on is based on lower bound computations for each alternative; the relationship corresponding to the maximum sum of lower bound approximations for its two descendents is chosen. A leaf of the tree is reached when $F = \emptyset$; at which time a feasible schedule is simple to find if one exists. This, in brief, is the outline of the procedure. But there remains three issues intimately connected to its computational performance that still need to be discussed: (1) what the authors refer to as concepts of "immediate selection", (2) the lower bound and (3) the upper bounds.

Recall that the common way to fathom nodes in a general BaB procedure is by computing lower bounds to compare with the best solution or best upper bound at hand. Immediate selection criteria are means to help fathom nodes right after or even before they are generated. The idea is the same as with dominance rules we have seen so far in connection with BBLB3 and the DH procedures and which we will encounter as part of the rest of the procedures we are reviewing as well. In fact, the authors classify dominance criteria as one of two classes of immediate selection criteria. The other class is one for feasibility criteria.

As the name might suggest, feasibility criteria are criteria whose purpose is to point out an inconsistency in a potential descendent of a schedule scheme and therefore, given that the tree is binary, consider the other descendent for branching. Two types of feasibility criteria are employed by the authors. The first is based on constructing a sort of "distance" matrix $(l_{ij})$ for the activities in a schedule scheme that satisfies some transitivity property. An entry $l_{ij}$ of this matrix can be interpreted as the minimum delay that may elapse between the start of activity

$j$ and the start of activity $i$:

$$l_{ij} = \begin{cases} 0, \text{ if } i = j \\ d_i, \text{ if } i \to j \\ -(d_j - 1), \text{ if } i \parallel j \\ d_i - (UB - 1), \text{ otherwise.} \end{cases}$$

where $d_i$ is the duration of activity $i$ and $UB$ is the current best upper bound. Due to the transitivity property, the transitive closure of the matrix, $(\bar{l}_{ij})$, is computed. Based on that, two conclusions are drawn: (1) if $\bar{l}_{ij} > d_i$ then $i \to j$ can be fixed and (2) no feasible schedule exists if $\bar{l}_{ii} > 0$ for some $i$. The second type of feasibility criteria used is based on symmetric triples. A triple of activities is said to be symmetric if $k \parallel i$ and $k \parallel j$ hold but $i$, $j$ and $k$ together are resource incompatible. Hence in this case one can conclude that $i - j$. The authors propose to find all symmetric triples in a schedule scheme. Six rules are then introduced to fix conjunctions or parallelity relationships within the scheme, one of which we present as an example. Suppose $h \parallel i$ is given and that $j$, $k$ and $h$ cannot be concurrent. Then $h - j$. Suppose further that $i \to j$ then $h \to j$ can be fixed. If $j \to i$ were to hold instead, then $j \to h$ could have been fixed. The five remaining rules are variations on the same theme. We note here that eventhough fixing disjunctions at the leaf of a tree can be done arbitrarily, fixing them before leaves are reached might be helpful to settle some flexibility relationships and point out inconsistencies; thus, control the spread of the tree.

The authors classify two additional rules under the dominance criteria category. The first one notes that if $i - j$ is a disjunction of a schedule scheme whose corresponding entry in the associated distance matrix satisfies $l_{ij} \geq -(d_j - 1)$ then $i$ starts before $j$ finishes and so $i \to j$.

Also, if for activities $i$ and $j$ both conditions $l_{ij} \geq -(d_j - 1)$ and $l_{ji} \geq -(d_i - 1)$ hold then $i$ starts before $j$ finishes and, similarly, $j$ starts before $i$ finishes. Therefore, in this case, $i \parallel j$. The second dominance criterion, more precisely process, is based on the concepts of heads and tails of activities. The former is a lower bound on the start time of the activity. The latter is a lower bound on the time span between the completion of the activity and the optimal project duration. Their importance lies in helping settle more disjunctions through the first feasibility and the first dominance criteria.. To make the most of them, though, good estimates of their values are needed.. To this end, the first step is to obtain their estimates; which is accomplished by means of recursion formulas. Next, a graph associated with the schedule scheme and whose vertices represent the activities and edges represent the conjunctions and disjunctions of the scheme is constructed. The graph is partitioned into cliques on which a jobshop procedure is applied to improve the heads and tails estimates. These estimates are retransformed into distances which are then further used to help settle more disjunctions. Now the graph is reconstructed, repartitioned into cliques, better estimates are deduced and more disjunctions are fixed. This process is repeated until no more disjunctions could be settled.

For a lower bound, the authors found the previously mentioned relaxation based LB2 bound of Mingozzi *et al* (1998) as the best one for their procedure. Recall that the columns of the associated linear program, $(RP2)$, correspond to reduced feasible subsets, *i.e.,* resource compatible subsets with no precedence relationships among their activities that are not contained within other such subsets. In their 1998 paper, the authors generated the numerous reduced feasible subsets and solved $(RP2)$ directly despite the large size of the problem. That was for the J30 instances of 30 activities each, which we will be reporting on. For larger size problems, the number of reduced feasible subsets is too large to generate. For problems up to 100 activ-

ities, they resorted to starting Simplex with a fixed number of columns and augmenting them with a fixed number of reduced feasible subsets at each iteration. Subsequently, Brucker and Knust (1999) reported that, for problems with 60 or 90 activities, instead of the direct way to solve ($RP2$) they tried few fairly elaborate column generation techniques some of which, they mention, did not lead to good bounds. They finally settled on a bound based on the Klein and Scholl's destructive improvement technique and on column generation; no mention is made of how this bound fairs compared to the seemingly much simpler LB3. Using the optimal solution and slack variable for the lower bound at a node, approximations for the lower bounds of the two descendent nodes to the current one are obtained. Their sum is used to determine the next node to be selected in the depth first strategy.

Similarly to many BaB procedures, upper bounds are used in conjunction with lower bounds to help fathom nodes of the search tree. Whenever the lower bound on the project duration at a node is computed and found to be less than the best upper bound at hand, an upper bound on the project duration is computed at the same node. If an improvement in the bound is found, the upper bound is updated. Two heuristics were constructed in this regard. The first is a heuristic based on the "parallelity components" of the project and the precedences among them. The second is a Tabu Search heuristic based on the schedule scheme concept but which also incorporates the first heuristic. It is rather computationally expensive to execute, though, and hence is run at the root node only.

Computational experiments with BKST revealed that 425 of the 480 instances of J30 could be solved in one hour. For J60, whose 480 instances have 60 activities each, 326 instances could be solved in the same amount of time on the same machine. No comparisons with other RCPSP procedures were reported by the authors. We note that despite the relatively

many mathematical refinements of this procedure, we will see, in comparisons with subsequent procedures, that its computational performance does not match its mathematical sophistication. In fact, this is the slowest running procedure we are reviewing for RCPSP.

### GSA

This is a BaB procedure developed by Sprecher (1997) based on his related work for the multi-mode RCPSP (Sprecher 1994 and Sprecher and Drexl 1998). Its roots lie in Talbot's implicit enumeration procedure (Talbot and Patterson 1978) and the precedence tree scheme of Patterson *et al* (1989) . The acronym stands for "Generalized Sequence Algorithm". We briefly summarize it here and refer the reader to the original paper by the author for its full details.

Each node of the BaB search tree corresponds to scheduling one activity of the project. The root node corresponds to the start activity of the project and the leaves to its finish activity. Each activity $j$ has latest start and latest finish time bounds associated with it, $LS_j$ and $LF_j$ respectively, which are updated during the course of the algorithm. The $i^{th}$ **level** of the tree is the $i^{th}$ node along a path from the root node to one of its leaves. At each level, the activities considered for scheduling are the ones whose predecessors have already been scheduled. If $j$ is considered for scheduling on level $i$ then the start time of $j$, $ST_j$, is the lowest feasible start time that (1) is not less that the start time of the latest scheduled activity (2) does not violate the resource constraints and (3) does not exceed $LS_j$. If scheduling $j$ according to these conditions is not feasible then backtracking to the previous level is performed where the same conditions are tested on the next eligible activity at that level. When a leaf of the tree is reached, a bound on the project makespan is found. If this bound, $B$, is an improvement over the existing one then $LS_j$ and $LF_j$ are updated with $LS_j-(LF_n-B+1)$ and $LF_j-(LF_n-B+1)$, respectively; for all

activities $j$; $n$ being the finish activity of the project. In such a case, backtracking is performed to level $i$ defined by $i = min\{k : CT_{g_k} > LF_{g_k}\} - 1$, where $g_k$ is the activity scheduled at level $k$ and $CT_{g_k}$ is its completion time. The algorithm terminates when backtracking reaches level 0. Note that the ordering of the eligible activities at the different levels affects the search. The procedure assumes that the activities are tested for scheduling according to increasing labels. But clearly, these labels can be rearranged using any priority rule.

For a lower bound, the procedure uses yet another variant of LB3; the Mingozzi *et al* (1998) bound. But to further control the branches of the search tree, seven more "bounding rules" are employed. We present them in the order they are applied.

The first bounding rule is the **Contraction Rule** which sets up an upper bound on the starting time of an activity $g_{i+1}$ considered for scheduling at some $(i+1)^{st}$ level of the search tree. Essentially, the starting time of $g_{i+1}$ cannot exceed the minimum of the completion times of the activities previously scheduled at that same level. As such, the bound is automatically updated during the course of the algorithm. The second bounding rule is the **Local Left Shift Rule** which insures that partial schedules with possible local left shifts are not continued.

The third rule is the **Extended and Simplified Single Enumeration Rule** which rules out continuing the enumeration of the same partial schedules induced by different paths of the search tree. It is based on the observation that if two sequences are the same up to level $i-1$ but are continued with $g_i$, $g_{i+1}$ and $g_{i+1}, g_i$, respectively, where $g_{i+1} < g_i$ and $ST_{g_{i+1}} = ST_{g_i}$, then the first sequence is dominated and subsequently discontinued.

The fourth bounding rule is the **Set Based Dominance Rule** derived from the realization that if two search paths have the same activities but in different orders and if an activity considered for scheduling in both sequences has its starting time, in a continuation of the

first sequence, at least as large as the maximum of completion times of the activities of the second sequence then the first sequence is dominated by the second and hence discontinued. Sprecher notes that this is a weaker rule than the DH Cutset Dominance Rule but is simpler to implement.

The fifth rule is the **Non-Optimality Rule** which can be invoked in certain situations to rule out continuing particular sequences. Say $S$ is a sequence of $i$ activities and $g_{i+1}$ is one activity being considered to continue $S$. Let $g^{\max}$ be the activity of $S$ that finishes last and $\Delta_{g^{\max}}$ the maximum duration that $g^{\max}$ could have had and still be left shiftable. Let $\Delta^{\max}$ be the difference between the largest, *i.e.,* $CT_{g^{\max}}$, and the second largest completion times of the activities of $S$. Suppose that $ST_{g_{i+1}} = CT_{g^{\max}}$ and $\Delta^{\max} > d_{g^{\max}} - \Delta_{g^{\max}}$ then $g_{i+1}$ cannot be an optimal continuation of $S$.

The sixth rule is the **Simple Permutation Rule**. Let $S = [g_1, g_2, ..., g_i]$ be a sequence and $g_{i+1}$ an activity considered for continuing $S$. Let $g_k$ be an activity with $k \leq i$ and $CT_{g_k} \leq ST_{g_{i+1}}$ and $g_l$ an activity such that $l < k$ and $g_l > g_k$. Assume that $k$ and $l$ are interchangeable in away that $g_k$ starts at $ST_{g_l}$ and $g_l$ finishes at $CT_{g_k}$. Then continuations of the sequence $[g_1, g_2, ..., g_i, g_{i+1}]$ are dominated by a previously evaluated sequence. Finally, the seventh bounding rule that the procedure employs is the **Extended Global Left Shift Rule**. Its essence is to reduce the search to active schedules only.

Sprecher notes that some of those rules, as presented, cannot be implemented by themselves while still guarantying the optimality of a solution. He compared versions of the procedure that contain some of the bounding rules to the one containing all of them. His conclusion was that the full version, GSA, is computationally the faster one and so is preferable. Using J30, the

480 instances test set we report on in 1.5.4, he compared his procedure to DH1[14] and observed that DH1 requires a large random access memory space (RAM) while the RAM requirements of GSA were modest in comparison. He also concluded that with 24 MB of RAM available DH1 is slightly faster. Sprecher also compared his procedure to BKST on the J60 dataset. In roughly equivalent time spans, given that the machines the procedures were tested on were different, GSA solves more instances of this dataset than BKST and a lot faster.

#### Progress and Scatter

These are two BaB procedures developed by Klein and Scholl for GRCPSP (Klein 2000). Since RCPSP is a special case of GRCPSP, the procedures apply to it as well. We will see that whereas Progress might be construed as a standard BaB procedure in that it sequentially explores the branches of the search tree, Scatter, on the other hand, incorporates a diversification strategy in the sense that it may interrupt its search at any node of the tree and pick it up from another unrelated and possibly distant node. In what follows, we describe our adaptations of these two procedures to RCPSP and note that their designers' implementations do not take advantage of the relative simplicity of RCPSP compared to GRCPSP. Rather, the authors treat RCPSP instances as general GRCPSP instances.

As in the DH procedures, the nodes of the **Progress** search tree correspond to partial schedules, $PS$, at decision time points $t$. For GRCPSP, the determination of the set of eligible activities to be scheduled at time $t$, denoted $E(PS, t)$, proceeds according to a rule incorporating a "modified" forward pass. Restricted to RCPSP, this rule determines the eligible activities as

[14] Sprecher did not have an executable version of DH1 so he adjusted his computational times by a factor of 2.7 to mimic the computational enviroment reported in Demeulemeester and Herroelen (1997).

those that are precedence feasible at decision point $t$. In this special case, if time $t$ is a decision point then the next decision point $t'$ is the earliest time at which an active activity at time $t$ is completed. Of course, time 0 is the first decision point.

In contrast with DH and its derivatives, all subsets $E^* \subset E(PS, t)$ may be considered for branching. The sequence in which these subsets are considered is important. Klein and Scholl propose to generate them in the following way. Suppose that the eligible activities are ordered in some way. Starting with the lowest ordered one, activities are sequentially placed on a stack as long as the total resource requirements of these activities and the ones already active do not exceed the resource limits. This is the first subset. To obtain the second, the top activity on the stack is removed and replaced with the next fitting one from the ordered eligible set. If resources still permit, sequentially adding activities to the stack continues as long as a resource limit is not breached. If no such resource compatible activity exists, the top activity is simply removed. In both cases, the stack activities constitute an eligible subset for branching. When carried to completion, this process enumerates all resource feasible eligible subsets including the empty set.

The branching strategy of Progress is a depth first strategy that is a hybrid of two strategies: the depth first strategy with complete branching, **DFSB**, and the depth first strategy organized as a so called laser search, **DFSL**. The difference between the two is that while DFSB generates all the descendents of a given node and, within the depth first search, explores them in non-decreasing lower bound values, DFSL on the other hand chooses the descendent node to explore without generating the others. By contrast, the **local lower bound method**, **LLBM**, the hybrid that Klein and Scholl propose, implicitly aggregates the nodes with the same lower bound values into classes and, again within depth first, generates these nodes class by class.

This strategy is supposed to combine the advantage of DFSL, which yields a feasible solution relatively quickly, and the advantage of DFSB which presumably directs the search towards the most promising parts of the tree first. It is based on a computationally inexpensive lower bound for the nodes constructed in the following way.

Initially, for each activity $j$, a lower bound on the time remaining to complete the project from the activity's start time is obtained. This bound is computed as: $\Psi_j = LF_n - LS_j$, where $LF_j$ and $LS_j$ are, respectively, an estimate of the latest finish time of the end activity and the latest start time of $j$ secured in a backward pass. Then at the node of the search tree corresponding to a partial schedule $PS$ and decision time $t$, a local lower bound on the project completion time is: $LLB = t + \max\{\Psi_j : j \in E(PS,t)\}$. Let $E^* \subset E(PS,t)$ be considered for branching and let $t^{'}_{\min}$ be the smallest possible next decision point among all next decision points corresponding to feasible continuations of $PS$, *i.e.,* $t^{'}_{\min}$ is the minimum completion time of both all the active activities at time $t$ and all the eligible activities if started at time $t$. Then a local lower bound on the project completion time is: $LLB^{'} = \max\left\{LLB, t^{'}_{\min} + \max\{\Psi_j : j \notin J(PS) \cup E^*\}\right\}$, where $J(PS)$ is the index set of the activities of $PS$. Recall that the aim, at this point, is to aggregate the eligible subsets with the same lower bounds into classes that could then be considered by non decreasing order of the lower bounds. The obvious way to accomplish this is to generate all the eligible subsets and compute their $LLB$'s à la DFSB. The other way is to order the activities according to non-decreasing $\Psi_j$ and apply the eligible subset generation scheme discussed earlier. In this way, whenever an activity $j$ with $t^{'}_{\min} + \Psi_j > LLB$ is removed from the top of the stack, $LLB$ is replaced by $t^{'}_{\min} + \Psi_j$ and a new class is created. In this way, the classes are arranged in non-decreasing order and do not need to be all generated before a choice on the eligible subset to branch on is made.

We note that generating the classes in this way is possible due to using a lower bound on the next decision points, $t'_{\min}$, instead of the actual next decision points which are subsets $E^*$dependent. The drawback here is that when the difference between the smallest completion time of the active activities at time $t$ and the smallest completion time of an eligible activity started at time $t$ are significant then $LLB'$ will be a significantly lower local lower bound on the project completion time than it could be. This is the case also when the difference between the largest and smallest durations of activities in $E(PS, t)$ is significant and $PS$ is continued with a subset $E^*$ containing only activities having the significantly larger durations. Now, the effect of this shortcoming in the local lower bound is to include subsets $E^*$ in classes that precede the classes that their larger possible local lower bounds warrant. This runs counter to the spirit of DFSB and so, at least, does not fully capture the advantage of DFSB. It may have to be the price to pay for the compromise, though.

Besides computing $LLB$ based on a backward pass, the authors list three other lower bounds they experimented with. The first is a capacity bound computed as the maximum of $r$ lower bounds to the project duration, where $r$ is the number of resources. Each bound corresponds to a resource and is calculated as the ratio of the total activities' requirement of the resource over the resource limit. The second and third bounds are based on identifying pairs and triplets of incompatible resources and are, therefore, computationally expensive. The authors' computational experiments on GRCPSP data sets show that the version of Progress with the first lower bound, backward pass, is either the fastest or second fastest among the four versions. No experiment results were provided for RCPSP data sets; though the same conclusion probably does hold.

As in most exact RCPSP algorithms, dominance and reduction rules are used to curb the

size of the search tree. Klein and Scholl use five of them. The first rule is based on the idea of an activity core time. Given a partial schedule $PS$ at a decision time point $t$ and an upper bound on a project's duration, $UB$, let $EF_j(PS)$ be the earliest finish time of activity $j$ computed by forward pass based on $PS$ and let $LS_j(UB)$ be the latest start time of activity $j$ computed in a backward pass with a project duration of $UB$. The **core time**[15] of activity $j$ is defined as the interval $[LS_j(UB), EF_j(PS)]$, whenever $LS_j(UB) < EF_j(PS)$. The **Core Time Rule** says that if $UB - 1$ is an upper bound on the project duration then the total resource requirements of the activities in the corresponding core time intervals, based on $UB - 1$, must not exceed the availabilities of the resources. If this rule is not satisfied then continuations of $PS$ with an upper bound of $UB - 1$ need not considered.

The second and third dominance rules used in Progress are what Klein and Scholl call "**Semi-active**" and **"Active Schedule Rules"** which are, respectively, the same as the Local and Global Left- Shift Rules mentioned previously. The fourth dominance rule the authors use, called **Suppression Rule**, is based on the idea of potential dominance between two activities. It is adapted from similar concepts in Scheduling Theory and defined as follows. Activity $j$ dominates activity $h$ if: (1) its duration is at least as large as that of $h$, (2) its resource requirements, for each resource, are at least as large as that of $h$ and (3) its set of immediate successors contains that of $h$. The rule fathoms a node corresponding to a partial schedule $PS$ at a decision time point $t$ if an activity $h \in PS$ can be replaced by an activity $j \notin PS$ such that $j$ dominates $h$ and finishes by time $t$.

The fifth rule is termed the **Extended Schedule Storing Rule**. It relies on the partial schedule and decision time point dependent concept of latest feasible starting time of activities,

[15]Recall that this concept was also discussed in connection with the destructive improvement strategy.

$LFS_j(PS)$. To obtain it, a modified backward pass is run with $ES_j(PS)$, the earliest starting times of activities $j$ given the partial schedule $PS$, as the starting times of activities $j \notin J(PS)$. If $LSB_j(PS)$ is the latest starting time for $j \in J(PS)$ when computed this way then $LFS_j(PS)$ is defined as $\max\{LSB_j(PS), t - d_j\}$ for all $j \in J(PS)$, where $t$ is as before and $d_j$ is the duration of activity $j$. The Extended Schedule Storing Rule says that a partial schedule $PS$ is dominated by another previously stored partial schedule $\overline{PS}$ with $J(\overline{PS}) \supset J(PS)$ if $LFS_j(\overline{PS}) \leq LFS_j(PS)$ for all $j \in J(PS)$ and $LFS_j(\overline{PS}) + d_j \leq t$ for all $j \in J(\overline{PS}) - J(PS)$. The rationale is that this condition is sufficient for the unscheduled activities in a continuation of $\overline{PS}$ to start earlier than they would in a continuation of $PS$ and hence is sufficient for the project to finish earlier. Note that as in the case of the DH Cutset Dominance Rule, and to a lesser extent the GSA Set Based Dominance Rule, the implementation of this rule requires large memory storage space. To reduce the storage space required, whenever a non-dominated partial schedule is added, the partial schedules it dominates are deleted. Moreover, the authors observe, all subsets of $J$ containing $J(PS)$ need to be enumerated to find the stored partial schedules $\overline{PS}$ dominating $PS$. Since this is prohibitive, the enumeration is restricted to partial schedules $\overline{PS}$ extending $PS$ by no more than a pre-specified number of activities.

Klein and Scholl report that their procedure is faster than GDH, the DH procedure adapted to GRCPSP, but report no comparisons of Progress to previously existing RCPSP procedures.

Note that the effectiveness of Progress heavily depends on finding good partial schedules to explore at the beginning of the search process. Such partial schedules are more likely to lead to good solutions that provide tight upper bounds and valuable dominance information to help fathom the branches of the search tree. However, as the authors note, a dependence on early good solutions for effectiveness can be pointed out in any traditional depth first BaB procedure

and is ultimately due to the rigidity of the search strategies in those procedures. In particular, these strategies do not allow for interrupting the search in unpromising parts of the tree in order to pick it up from other more seemingly favorable ones. This, exactly, is the weakness of traditional Depth First BaB that Klein and Shcoll's **Scattered Branch and Bound** (Klein 2000) is designed to redress. The idea is to build on top of a traditional BaB procedure a mechanism whereby, analogously to Simulated Annealing and Tabu Search, breaking off the search in one part of the tree and selecting another to continue it from are made possible. All the elements of the underlying BaB procedure are then kept except for the one selecting the node to branch from.

Following the above discussion, the basic ingredients of a scattered BaB procedure, besides an already established BaB algorithm, are a breaking off rule and a node selection rule. For their implementation to be effective, though, prior information on the search tree itself is required. Namely, information to help decide when is a relatively good time to break off the search and a pool of candidate nodes to continue it from. To this end, the search tree is subdivided into regions by so called **cut solutions**. Each of these matches a feasible solution constructed by building up partial schedules whose corresponding branches and nodes can be marked on the search tree. Each region created this way is bordered by two cut solutions, one on each side, and is meant to be searched by itself. In general, the branching scheme determines the ordering of the cut solutions and hence each region's borders. In the case of Progress, for example, it is the eligible subset ordering scheme that plays that role. By convention, a cut solution belongs to the region to its right. Note that their could be nodes inside a given region that are descendents of nodes on the cut solution of the adjacent region to the right.

As far as RCPSP is concerned, and likewise for GRCPSP in fact, Klein and Scholl's Scattered

BaB procedure, dubbed **Scatter,** is implemented in broad terms as follows. At the beginning, the left most cut solution is determined. Given parameters $\mu$, $\nu$ and $\gamma$, with $\mu > \nu$, $\mu$ cut solutions are created among which the $\nu$ best ones are selected. For each of the corresponding regions of these, $\gamma$ nodes are developed mainly to gather control information for the selection rule. Starting with the last developed node of a selected region, the underlying BaB procedure Progress is run until the region has been completely searched or the break-off condition has been satisfied. When either of these conditions is met, selecting another region to pursue the search in is made. Progress is continued from the last developed node in that region. The process ends when all regions have been completely searched; at which time the best solution obtained so far is the optimal one. One factor that may appear to complicate the complete search of a region is the search along its boundaries. Klein and Scholl do not address this issue but we think the way to efficiently handle it is to generate all the nodes on either side of the boundary that have immediate predecessors on that boundary but only develop the ones inside the region itself. With that, there remains two issues to be discussed in order to complete the outline: the **break off rule** and the **selection rule**.

One of the simplest criteria to use in deciding a possible break off is a comparison of the number of branched nodes in a region to the number of fathomed nodes. If more nodes are branched than fathomed then it may be assumed that more branching may lead to a better solution than the one at hand and so a break off is postponed. On the other hand, if more nodes are fathomed than branched then chances are that the region may not contain a new incumbent solution. Of course, attention needs to be paid to the case when a break off decision could be made based on too few nodes explored. The same goes for the case when a region is entered repeatedly which suggests proportionately more exploring than the simple comparison

warrants. To put this in mathematical terms, three measures are recorded for each region $i$ of the search tree: the number of branched nodes in the region, $nb_i$, the number of fathomed nodes, $nf_i$, and the number of times the region has been visited, $it_i$. Given a predefined parameter $\delta$, to check whether to break off the search or not the inequalities $nb_i + nf_i > \delta \cdot it_i$ and $nf_i > nb_i$ are tested. If they are both satisfied a break off is decided. Note that each time region $i$ is reentered $nb_i$ and $nf_i$ are set to zero. Note also that the $\gamma$ nodes developed in each region at the beginning of the search allow the inequalities to be tested while still avoiding the many breakoffs that otherwise would take place at the early stage of the procedure's execution.

Once a region to resume the search is selected, the node to continue Progress from should naturally be the last one developed in that region. Associated with that node is, of course, a partial schedule. Therefore, selecting a region is, in fact, selecting a partial schedule. It follows that a non random selection rule for Scatter should necessarily rely on a comparison of the last partial schedules developed in each region. The best performing rule that Klein and Scholl discovered in that regard, and probably even counting the random selection rule in, is one based on the total slack of jobs. Recall that the **total slack** of an activity, in case of unlimited resources, is the maximum time by which the activity can be delayed without delaying the project. Although this concept looses much of its significance when resources become limited, an argument may be made that activities with large total slack when resources are unlimited stand a better chance of being started within their CPM model earliest start time/ latest start time intervals when resources become tighter. This motivates comparing partial schedules based on the sum of total activity slacks that they generate. Formally, given an estimate of the project

duration, $\overline{T}$, and a partial schedule $PS$, the total slack of an activity $j$ is defined as:

$$TSL_j(PS) = \begin{cases} LS_j(\overline{T}) - SS_j(PS), \text{ for } j \in J(PS) \\ LS_j(\overline{T}) - ES_j(PS), \text{ for } j \notin J(PS) \end{cases}$$

where $LS_j(\overline{T})$ is the latest starting time of activity given a project duration of $\overline{T}$, $SS_j(PS)$ is the scheduled starting time of $j$ under $PS$ and $ES_j(PS)$ is the earliest starting time of $j$ under $PS$. The measure that Klein and Scholl propose to compare partial schedules is now: $SSL(PS) = \sum_j TSL_j(PS)$. Each time a break off occurs, the $SSL$ value of the last developed node is computed. Normally, the $SSL$ value of a region is, by definition, the $SSL$ value of the last developed node. But for the $SSL$ value of a node to be a better indicator for its region, its corresponding partial schedule should be relatively advanced. Therefore, the $SSL$ value of a region is only updated when the corresponding node is a low level node. By convention, the level of that node needs to be at least half the maximal number of levels for the cut solutions. With that, region $i$ is selected to continue the search if its corresponding $SSL$ value is the largest among the $\nu$ regions.

Now, we would like to interrupt our discussion of RCPSP to make a point regarding Scatter and GRCPSP. Recall that Klein and Scholl's real interest lies rather with GRCPSP which Progress and Scattered BaB were designed for in the first place. But as Klein himself notes in his book (Klein 2000, p.95), finding a feasible solution to this problem is an NP-Complete problem in itself. Certainly, then, finding several is even more time consuming. Clearly, this appears to cast doubts on the general applicability of the new procedure to GRCPSP. As we see it, the way to resolve this issue in the general case is to not insist that the "cut solutions" match feasible solutions as Klein himself does in his book (Klein 2000, p.243). This modification,

however, renders the selection rule inappropriate to use as some partial schedules it would be based upon might be infeasible. In fact, this modification would probably render any selection rule based on partial schedules and/or lower bounds inappropriate and for the same reason. The only selection rule we can think of using in that case is the random one; which, curiously enough, is more in the spirit of Simulated Annealing and Tabu Search.

To computationally evaluate Scatter, the authors compared it to Progress on several GRCPSP test data sets. Their results show that Scatter does run faster than Progress but that the difference is not "convincingly" large enough. They, then, extracted from those data sets the "hard instances" and analyzed the performance of the procedures on these only. Now the difference became more noticeable which lead to their conclusion that Scatter outperforms Progress specially on the hard instances.

Scatter was also compared to DH1 and GSA despite the overhead it carries to deal with GRCPSP. The test data sets used are J30 and J60, both of which are covered in 1.5.4. Since the authors had no codes for DH1 and GSA and since they were using computers with different speeds than the ones the two procedures were tested on, they resorted to multipliers to adjust the computational results reported for earlier procedures. They do acknowledge that the multipliers's use allows only a very rough comparison of the CPU-times, however. Their adjusted computational times for the DH1 computer shows that Scatter's average CPU-time on the J30 dataset is 24.08 seconds, that of DH1 is 33.71 seconds while that of GSA is 115.42 seconds. The conclusion drawn is that Scatter outperforms DH and GSA. But we contend that the picture is more complicated than this. The hardest instance for DH1 in the J30 dataset requires 9515 seconds of CPU-time[16]. If this instance is taken out of the CPU-time average for DH1, then

[16] Actually, the figure should be 10261.8 seconds. Klein's 9515 figure is for the DH1 version with LB0. But

that average drops to 13.92 seconds. If Scatter is treated in the same way, in other words its hardest instance of 2886 seconds taken out of the average, then its average CPU-time drops to 18.11 seconds. Suddenly now, DH1 seems a lot more competitive with Scatter than it used to. Another point to be made is that DH1 is reportedly outperformed by DH2 (Herroelen *et al* 1998). Using a multiplier, just as the authors themselves have done although we have strong reservations about it, DH2 places well ahead of Scatter on the speed scale.

A second comparison with GSA on the J60 data set was also performed. The authors mention that the computers the procedures were executed on were nearly identical machines except for the operating system. None of the two procedures solved all problems within half an hour time limit. But Scatter was able to solve more of the 480 problems in that time than GSA; 385 versus 364. Also the average CPU-time of Scatter was faster, 396.66 versus 472.69 for GSA. Moreover, the authors note that 296 of the 480 instances of J60 have for optimal durations their critical path lower bounds. This says that Scatter solved 97 of the 184 harder instances to optimality within half an hour, versus 68 for GSA. All of this suggests, if indeed the different operating systems do not prejudice the results, that GSA is a lot more competitive with Scatter on J60 than on J30 but that nevertheless Scatter, on the two data sets, has the edge. A comparison with BKST on the same dataset but using different hardware and software was also performed. Scatter was the better performer in a significant way even when adjusting for the different hardware and operating systems used.

overall, the best performing version of DH1 is with the LB3 bound, as the reader may recall, and its this version's performance that Klein and Scholl report on in the 33.71 figure. Klein undoubtedly meant to report 10261.8 seconds instead. In any case we are willing to work with his numbers.

### 1.4.3 Heuristics

As we have just seen in our review of exact procedures, none of the methods can solve instances of RCPSP with 60 activities on a routine basis. In fact there are instances of 60 activities that no method has yet been able to solve[17]. This, also as we have mentioned previously, provides a strong motivation to study heuristic approaches for obtaining "good" feasible solutions. As a matter of fact, Alvarez-Valdés and Tamarit (1989) claim there were "literally hundreds of heuristic procedures" in existence at the time of writing their heuristics review. Indeed in that same review, they themselves evaluated no less than 33 such heuristic, seven of which they had introduced in that same review. But another use of heuristics is actually within exact procedures as opposed to instead of them. One example of this is the BKST procedure we have reviewed. There, two heuristics one fast and the other fairly involved were used to obtain upper bounds on the optimal solution. These upper bounds, recall, have been used in conjunction with the lower bounds to fathom nodes of the search tree.

With such abundance of heuristics, it is imperative to have a mean to evaluate and compare their performances. Recall here that to evaluate a particular heuristic solution one may compare it to a lower bound. If the difference is small enough, the heuristic solution should be acceptable. But otherwise, the test is inconclusive. Recall also that, unless $P = NP$ , the existence of a polynomial time heuristic with a performance guarantee is ruled out. This, then, seems to leave little room for an effective technique to test the quality of a specific solution when neither an exact procedure nor a close lower bound are available. The only way to somewhat

[17] For an instantaneous update on this, the reader is referred to the Kolisch and Sprecher RCPSP web site: ftp://ftp.bwl.uni-kiel.de/pub/operations-research/psplib/j60lb.sm As of 12/14/01 many instances with 60 activities had different lower and upper bounds. We will have more to say about the instances posted on that site in the following section.

remedy the situation is to, instead of expecting performance guarantees, look for expected performance. This means estimating the expected performance of a heuristic by observing its average performance on "carefully" assembled test data sets. This raises a whole different set of questions regarding the characteristics of appropriate test data sets and how to build them. We leave discussing these concerns for the following section. But for now we remark that examining heuristics in this way has been and still is the prevailing manner to assess their quality.

Over the past four decades that RCPSP has been studied, many reviews of heuristics have been undertaken. Two notable and recent ones we have relied on are the review by Kolisch and Hartmann (1999) and the one by Klein (2000). Among the less recent surveys and studies we have encountered are the one by Thomas and Salhi (1997), the general review by Özdamar and Ulusoy (1995), the previously mentioned review by Alvarez-Valdés and Tamarit (1989) and the study of some heuristics by Cooper (1976). Still, many more surveys and studies of heuristics, specially older ones, are out there for the interested reader to find.

Following Kolisch and Hartmann (1999), the multitude of heuristic procedures proposed for RCPSP since its inception can be subdivided into three classes. The first one is based on an ordering of the list of activities of a project according to some priority rule and subsequently considering the activities for scheduling in that same order. The second class, called the Meta Heuristics class, is one for more involved heuristics in which, starting with a given schedule, or pool of schedules, the algorithm iteratively moves from one schedule, or pool, to another that may not always be better in order to avoid being trapped in a "locally optimal" schedule. The third class is one for heuristics that do not clearly belong to any of the previous two and contains rather a variety of fundamentally different approaches to generating a good feasible solution. In the remainder of this section, we briefly discuss each of these classes.

**Priority Rule Based Heuristics**

The main idea behind these heuristics is to assign priorities to the activities of a project that creates a **priority list**. Thereafter, an activity is only considered for scheduling when its preceding activities on the list have already been considered, though not necessarily scheduled. Such a priority list can be **static** in the sense that it is generated at the start of the algorithm and followed through without modifications. Or it can be **dynamic**; meaning, it is updated during the course of the algorithm. Examples of simple static priority rules are the shortest processing time rule (SPT), the most total successors rule (MTS), the latest starting time rule (LST) and the minimum slack rule (MINSLK) to name just a few. Of these, LST and MINSLK can be implemented dynamically as well. An example of a significantly more elaborate rule is the one proposed by Thomas and Salhi (1997) based on another previously proposed rule that is itself a composite of other rules.

Once a priority rule is decided, two methods for constructing a feasible schedule can be distinguished, each defining its own subclass. These are the **serial method** and the **parallel method**. The first schedules the activities of the project one by one. Each time an activity is scheduled, the eligible set of activities is determined and ordered according to the priority list. The activity in that eligible set with the highest priority is then considered for scheduling at the earliest possible time. Now, the parallel scheduling method, too, determines and orders the eligible set according to the priority list. But, in contrast with the serial method, it selects the activities to schedule from the ordered eligible set first according to highest priority and second according to resource availability. The selected activities are then scheduled together not one at a time as in the serial method. In other words, the eligible set is not necessarily determined and

reordered after scheduling each single activity but only after the selected subset of the eligible activities is scheduled.

Within each of the serial and parallel scheduling modes, a choice as to the scheduling direction can be made. The obvious way of implementing the methods is forward, commonly referred to as a **forward pass**. But it is also conceivable to reverse the precedence relationships of a project and implement any one of the two methods in the usual way. This is referred to as the **backward pass**. All it requires in addition to the forward pass mechanism is a feasible estimate of the project's duration. Heuristics also exist in which forward and backward passes are alternatively carried out and subsequently interlocked. Such priority based rules are termed **bidirectional**. They may have the same priority rule in each direction or different ones. They may adopt one scheduling mode in one direction and the other in its opposite. Heuristics exist also that perform several of these passes and output the best result. They are known as **multi-pass** heuristics. Actually, some heuristics select the activities to schedule based on probabilities which are themselves based on the priorities assigned. Based on these probabilities, several passes of the scheduling scheme are executed from which the best solution is selected. Within the priority rule based heuristics they are referred to as **sampling methods** and comprise several subtypes too.

A more evolved type of sampling methods are the **adaptive sampling methods.** There, during execution, the probabilities are modified and the priority rule and scheduling method are changed according to instance characteristics[18] and the number of executed passes (Kolisch and Drexl 1996). Schirmer (2000) pushes this idea further by employing **case based reasoning**. Essentially in this method, given several of the promising sampling methods and several proven

[18] Instance characterics are to be reviewed in the following section.

significant instance characteristics, the aim is to subdivide the multi-dimensional range space of the characteristics into regions each of which is assigned the heuristic that statistically preforms best on instances with characteristics belonging to it. Thereafter, given any instance to solve heuristically, the instance characteristics are computed, the region they belong to is determined and the corresponding heuristic is executed. In some regions of the range space of characteristics, it was discovered that the best sampling method to choose depends on the number of passes to be executed. In effect there is certain number of passes beyond which one sampling method is best and below which another method is best. Schirmer's case based reasoning algorithm incorporates this information too.

We point out that it is well known that the serial scheduling method produces active schedules (Kolisch 1996). Now, recall that the set of active schedules of a project reportedly contains at least one optimal solution. This says that priority rule based heuristics implemented using the serial scheduling method may actually find the optimal solution, though no proof of the optimality of such a solution comes with it. On the other hand, it is also well known that the parallel scheduling method is only sure to produce non-delay schedules (Kolisch 1996). Since the set of non-delay schedules does not necessarily contain an optimal solution, a heuristic implemented this way is less likely to come across the optimal solution. In fact, recalling theorem 4, the optimal non-delay duration can be arbitrarily large compared to the delay one. It follows then, in principle, that of the two, the serial method has the higher potential to reach an optimal solution while the parallel method has the higher potential to be misleading. Remarkably, this has not been confirmed by computational experiments run by Klein (2000). His experiments on the J30 test data set, which we report on in the following section, with about two dozens heuristics show that more optimal solutions are found using heuristics implemented with the parallel

method than with the serial method. This is true whether the heuristics are forward pass based or backward pass based. It is specially true for bidirectional implementations, though, which incidentally outperform the other two implementations[19].

**Meta Heuristics**

These are heuristic strategies for optimization problems whose essence is a mechanism whereby the set of feasible solutions is explored in a way that is unlikely to lead to a local optimum or cycle between different solutions. Of the several meta-heuristics for optimization problems that exist, three were tried for RCPSP: Simulated Annealing, Tabu Search and Genetic Algorithms.

Simulated Annealing is a random local search technique that examines the feasible solution space iteratively one point at a time. Starting with a feasible solution, the algorithm randomly chooses the next feasible solution to examine from amongst the neighbors of the current one. The next solution chosen is a worst solution than the current one with a probability that depends on the number of solutions examined up to that point. This probability is supposed to decrease to 0 as the number of examined solutions increases. The idea is that occasional deterioration in the quality of the solution forces the search in new neighborhoods thereby avoiding local optima. The basic requirements for an adaptation of this technique to solve RCPSP are: (1) a representation of a schedule[20], (2) a means to translate the representation into a schedule and (3) a neighborhood definition. Different researchers have adopted different ideas for each of these requirements. An example of several heuristics that were proposed in literature along these lines and reportedly one of the best heuristics for RCPSP (Kolisch and Hartmann 1999)

[19] No execution times need to be discussed here beacuse they are negligeable for all heuristics and their different implementations; as indeed they should be.

[20] Simulated Annealing as well as Tabu Search and Genetic Algorithms do not work with schedules themselves. They work on representations of shedules to ease the implementation of needed operations.

is the algorithm of Bouleimen and Lecocq (1998).

Tabu Search is a mainly local search technique that also examines the feasible solution space one point at a time. In addition to the three basic requirements of Simulated Annealing it also requires (4) a list of solutions and/or move attributes of predefined length, called tabu list, that the algorithm is not allowed to choose and, often, (5) an aspiration criterion. Starting with a good feasible solution, the algorithm tests the best solution in the current neighborhood in the following way. If this solution or any of its attributes are on the tabu list and the aspiration criterion is not satisfied the solution is disregarded and the next best solution is tested. If the solution passes the test then it is selected, the solution and/or some of its attributes placed on the tabu list and another iteration is started. If the tabu list exceeds its predefined length, the oldest item on the list is removed. The iterations are continued until a stopping criterion is satisfied. Sometimes means of concentrating the search on seemingly promising areas of the search space are implemented. Means of driving the search into new regions also exist. Both can be implemented by recording attributes of visited solutions and making an intensification or diversification decision based on the frequencies of these attributes. Note that the existence of the tabu list reduces the likelihood of cycling. Several adaptations of Tabu Search to RCPSP exist in the literature. We have already mentioned one of them: the Brucker and Knust (1999) Tabu Search heuristic used as an upper bound within the author's exact procedure. Another Tabu Search heuristic for RCPSP is RETAPS by Klein (2000).

In contrast with Simulated Annealing and Tabu Search, Genetic Algorithms is neither a local search method nor does it examine the search space one point at a time. It examines, rather, a pool of solutions simultaneously and is modeled along the lines of Darwin's natural selection. In addition to the first two requirements of the other two heuristic strategies, *i.e.,*

a representation and a translation mechanism, it requires ($3'$) a crossover operator which is a means of constructing a new solution from two existing ones and ($4'$) a mutation operator which creates a new solution from an existing by modifying one part of the corresponding solution representation.. A basic genetic algorithm goes as follows. Starting with a pool of feasible solutions, individual solutions are crossed over (mated) with other solutions to produce new solutions. These new solutions are then evaluated for fitness; makespan for RCPSP. Based on their fitness, solutions from the increased pool are selected to number the same as the initial pool. The rest are discarded. Some of the selected ones are further arbitrarily chosen to undergo mutations. The result is a new pool of solutions to go through the same process until a stopping criterion, usually the number of iterations, is satisfied. Several genetic algorithms for RCPSP have been proposed. We mention in this regard the study by Hartmann (1998) and the one by Kohlmogren *et al* (1999) which investigates two strategies to parallelize genetic algorithms and uses RCPSP as one test problem. Hartmann's genetic algorithm for RCPSP is reportedly, along with Bouleimen and Lecocq's simulated annealing heuristic, one of the best heuristics for RCPSP (Kolisch and Hartmann 1999).

**Other Heuristics**

As mentioned previously, this is a class of fundamentally different heuristic approaches to RCPSP. One subclass in this category, for example, is for Branch and Bound based heuristics. Here, at least two ways of turning an exact BaB procedure into a heuristic are possible. For one, the exact procedure need not be run to completion. It can be interrupted after a fixed time duration to return the best solution at hand. Demeulemeester and Herroelen (1997) have studied such a truncated version of their DH1 procedure and discovered, on the J30 data set,

that executing their procedure for a small amount of time yields results that are close to the corresponding optimal solutions. Sprecher (2000) also studied the truncated version of his GSA algorithm. He determined that such a version with 256 KB of RAM available achieves approximately the same results as DH1 with 24 MB of RAM. Since memory requirements increase exponentially with the number of activities, he argues, memory will become the critical resource for large instances. Hence, relatively to truncated DH1, truncated GSA is well suited to handle large instances.

Alvarez-Valdés and Tamarit (1989) propose another way of turning an exact BaB procedure into a heuristic. They suggest to select the most promising alternative for branching and to disregard the backtracking feature of the procedure. In this way, once a leaf of the search tree is reached, the heuristic solution is at hand. They derive three heuristics along those lines.

A second subclass of heuristics is based on the concept of disjunctive arcs and the idea of incompatible subsets activities. These subsets are minimal subsets of activities which are resource incompatible and pairwise unrelated by precedence relationships. They are minimal in that they do not contain other incompatible subsets. Now, with all the incompatible subsets of an instance identified, the problem can be seen as that of adding precedence relationships among the activities in such a way that no incompatible subsets ensue and in such a way that the longest path of the resulting graph has the minimum possible length. Four heuristics by Alvarez-Valdés and Tamarit (1989) are proposed along those lines.

## 1.5 Complexity Measures and Test Data Sets

It has long been recognized that, irrespective of the algorithms used, some instances of RCPSP are computationally easy while others are computationally hard[21]. To comprehend the difference, researchers have focused on finding measures that potentially correlate with the computational difficulty of encountered instances. To this end, the trend in the literature, since the mid-sixties, has been to isolate relevant characteristics of RCPSP and possibly combine them in some way to come up with the measures sought. Besides merely trying to predict the difficulty of RCPSP instances, such measures, if successful, could be used in selecting algorithms and heuristics that may be more efficient than others to deal with general instances or instances having particular characteristics.

Developed measures have been used, especially in the sixties and seventies, to compare the performance of existing exact methods and heuristics on general problem instances without a prior demonstrated correlation with computational difficulty. Their purpose in many cases was not to assess computational difficulty but to show that the test data used in the study they appeared in is appropriate in that it covered a wide range of problems. Since the 'eighties, though, questions about the computational significance of the proposed measures started to emerge. The paper by Elmaghraby and Herroelen (1980) was a critique of those earlier efforts. Thereafter, several studies have appeared to refute or confirm the alleged correlation with computational difficulty of many accepted measures. Concerning the second potential use of the measures, and as far as we know, there has been only one published study that attempted to match exact methods and instance classes. This is the 1984 paper of Patterson in which he used

[21] Computational difficulty could be due to lack of computer memory space and/or extensive execution time. Generally, the computational difficulty that we discuss here is due to the latter.

110 test problems that became famous as the de facto test set for some 20 years thereafter. But only one measure was used in the Patterson study, however, omitting the many that by then were common knowledge in the field. We are also aware of two studies that aimed at matching heuristics and classes of instances: Thomas and Salhi (1997) and Schirmer (2000). The two studies used different heuristics, different measures and different test data sets of instances to come up with their recommendations, though.

The question that might be raised here is: are the characteristics that correlate with difficult instances or classes of instances inherent to the problem itself or are they rather characteristics of the algorithm, or algorithms, used to solve them? For sure, a characteristic that is intrinsic to the problem will manifest itself in whatever algorithm is used to solve the problem. The opposite question is harder, though. That is, given a characteristic that correlates with computational difficulty when using all the available algorithms to solve the problem, can we decide a priori whether this characteristic is intrinsic to the problem itself or not? In other words, will this characteristic correlate with the difficulty of instances possessing it when solved using any algorithm that could possibly be conceived? If we were interested in worst case performance only, for example, and if $P \neq NP$, then the obvious characteristic of difficulty that is intrinsic to the problem itself would be the encoding length. When considering the average case performance though, which is the more appropriate consideration here, the answer seems elusive.

We submit that settling this question is not a precondition to proposing measures of instance complexity. One can simply discuss measures of complexity relative to the known algorithms or even a particular one. If one measure correlates with the instance difficulty when using one algorithm but not another then that may only get us closer to the goal of matching algorithms

and classes of instances. Furthermore, if with the emergence of more efficient algorithms, some others become obsolete then it is natural that the complexity measures that are solely associated with those algorithms become obsolete too.

One obvious characteristic of an RCPSP instance that likely impacts the computational effort to solve it, whatever the algorithm, is the **number of activities**. In the worst case, the computational effort required to solve an instance is expected to be exponential in terms of the number of activities. But that is only a worst case scenario and does not explain why some instances with the same number of activities are easy while others are hard. This characteristic by itself, then, is not sufficient for the purpose of predicting the computational difficulty. Most other characteristics that were investigated in the literature relate to two defining elements of RCPSP: the precedence relationships and the resources. As far as we know, besides the afore mentioned measure, only one other activity related measure was proposed and no objective function related measure, in this case makespan related measure, was ever suggested. It is important here to mention one point stressed by Elmaghraby and Herroelen (1980): " one and (the) same network can be complex on one scale and easy on another". That is, computational difficulty depends on the objective function too, and not just the other defining elements of the problem.

### 1.5.1 Activity Related Measure

This is one of two measures introduced by Cooper (1976) in connection with test problems he used to compare the Parallel and the Sampling methods when paired with about two dozens priority rules. He refers to it as the project's **density** and defines it as $D = \sum_i d_i / \sum_i (d_i + ff_i)$, where $d_i$ is the duration of activity $i$ and $ff_i$ is its 'free float'. The free float of an activity is the

maximum amount of time by which the activity can be delayed while still allowing its successors to start at their earliest starting times. Clearly, $0 < D \leq 1$. If $D = 1$ then every activity of the project is critical. Cooper's primary aim in selecting this measure is to pick different types of projects on which to test the heuristics. His focus was the performance of the heuristics, not the reliability of the measures. But in a similar study of heuristics, Alvarez-Valdés and Tamarit (1989) used the same measures and claimed that projects with a higher density are "more difficult" to schedule than ones with a lower density. Again, we emphasize that the focus of the study was the heuristics and not the measures.

### 1.5.2 Precedence Related Measures

Precedence relationships and their structure are the most tempting characteristic of an RCPSP instance to look at in order to make the instance easier or harder to solve. It is no wonder, then, that this characteristic has received the most attention form researchers trying to identify the difficulties of the problem. As we shall see, several researchers have argued that RCPSP instances with many precedence relationships are easier to solve, on the average, than instances with fewer precedences. The reason, they claim, is that increasing the precedence relationships of an instance reduces the number of its feasible sequences and thus makes the search for the optimal solution simpler. Computational evidence certainly exists to support this view but we, nevertheless, would like to present the opposite perspective.

Consider two special cases of RCPSP: $P2\,||\,C_{\max}$ and $P2\,|\,chains|\,C_{\max}$. The first is the problem of scheduling jobs on two identical machines in parallel under the minimum makespan objective. The second is the same problem but with precedences among the jobs that take the simplest possible form: chains. That is, any job except the first and the last, which may be

dummy jobs, has a single predecessor and a single successor. It is well known that $P2 \,||\, C_{\max}$ is NP-complete in the ordinary sense while $P2 \,|\, chains | \, C_{\max}$ is NP-complete in the strong sense (Pinedo 1995, p.349)[22]. It follows that $P2 \,|\, prec | \, C_{\max}$ is strongly NP-complete too since chains are a special form of precedences. Hence, $P2 \,|\, prec | \, C_{\max}$ is harder than $P2 \,||\, C_{\max}$. The analysis is extendable to the general case of $m$ machines. That is $Pm \,|\, prec | \, C_{\max}$ is harder than $Pm \,||\, C_{\max}$. This suggests that $PS \,|\, prec | \, C_{\max}$ is harder than $PS \,||\, C_{\max}$ and so precedences complicate the problem rather than facilitate it.

There are three ways in which this apparent contradiction between pure reason and the empirical evidence may be resolved. First, it may be argued, the contention that $PS \,|\, prec | \, C_{\max}$ is harder than $PS \,||\, C_{\max}$ is based on a worst case analysis as is NP-Completeness Theory in general. What RCPSP measures of complexity are concerned with, by contrast, is the average case performance. A result concerning computational difficulty in one context may not necessarily hold in the other and so the contradiction has not been established to begin with. Second, the empirical evidence on the blessings of precedence relationships has been gathered by researchers working with only one particular algorithm of the researcher's choice. Algorithms designed to take advantage of a precedence structure may well be inefficient when little or no such structure exists. The apparent contradiction could be resolved if the algorithms used in the studies incorporate a component algorithm well adapted for little precedence structures. In fact, this is also an argument for establishing reliable complexity measures so as to allow targeting particular algorithms for particular classes of instances. Third, pure reasoning and the empirical evidence need not be necessarily at odds. It could well be that the average computational

[22] Pinedo actually states that $P2 \,||\, C_{\max}$ is NP-hard in the ordinary sense. But, again, Pinedo does not differentiate between NP-complete and NP-hard. We do and that is why we quote him in this way.

difficulty initially increases as a function of some measure of precedence structure but that it briskly peeks and then declines as the precedence measure increases. The increase could be so steep that it was not detectable in the studies; in fact, it may even be discontinuous.

Any and all these three possibilities could explain the apparent contradiction away and each merits an investigation. We leave their merits as open questions, though, and move on to describe the precedence related complexity measures from the literature.

The first, and most debated, precedence related measure in the literature that purports to correlate with the difficulty of RCPSP instances is the **coefficient of network complexity** (Pascoe 1966), **CNC** for short. It is defined as the ratio of the number of arcs to the number of nodes in an AoA representation of the project. Davies (1973) and Kaimann (1974) define it differently but their definitions use the same two parameters. The Pascoe definition was subsequently adapted to AoN projects by Davis (1975). Elmaghraby and Herroelen (1980) questioned its validity on the grounds that there are projects with the same numbers of arcs and nodes yet varying computational difficulty. Kolisch *et al* (1995) define it as the ratio of the number of non-redundant arcs to the number of nodes in the AoN representation. They report, using the first DH, marginal negative correlation with the average solution time and explain it by noticing that the higher the number of precedences is, the smaller the feasible set of solutions is and so the smaller the enumeration tree becomes. In a more recent paper, De Reyck and Herroelen (1996) report that the negative correlation observed by Kolisch *et al* is strongly confounded by the strong correlation this measure has with another measure called the "complexity index". They conclude that it is therefore "ambiguous to attach all explanatory power" of problem complexity to this measure.

A second measure of complexity, the **total activity density**, was proposed by Johnson in

1967 (Elmaghraby and Herroelen 1980). It is defined as

$$\sum_i \max\{0, \text{number of predecessor activities } - \text{ number of successor acttivities}\}.$$

Patterson in 1976 used the **average activity density** (Elmaghraby and Herroelen 1980). This is the total activity density averaged by the number of activities of the project.

Besides the project's density, mentioned previously, Cooper (1976) proposed a measure of complexity called **order strength, OS** for short. This has originally been proposed by Mastor for the line balancing problem (Elmaghraby and Herroelen 1980). It is defined as the number of precedences in the project, including the transitive ones, divided by the total number possible, which is $n(n-1)/2$. It turned out that this measure is closely related to a measure of complexity for the assembly line balancing problem called the **flexibility ratio, FR**. This latter one was defined by Dar El (1973) as the number of zeros in the transitive incidence matrix divided by $n(n-1)$. It is easy to see that $OS = 1 - FR$. Herroelen and De Reyck (1999) report on an experiment indicating that OS is a good measure of network complexity and that the logarithm of the average CPU-time as a function of OS exhibits a linear hard-easy behavior. Again, the more precedence relationships, the easier the instances seem to get, on the average.

Another measure of complexity was proposed by Thesen in 1977 and is called the **restrictiveness index**, **RI (**Thesen 1977**)**. It is based on the number of precedence-feasible sequences of activities of the project, $s$, irrespective of resource limitations, and the total number possible, that is $n!$: $RI = 1 - \frac{\log s}{\log(n!)}$. Note that $0 \leq RI \leq 1$; $RI = 0$ corresponds to the no-precedences case and $RI = 1$ corresponds to $s = 1$, *i.e.,* the case of a predetermined sequence and so maximum restrictiveness. Due to the difficulty of computing $s$, Thesen proposed several estimators

for RI the best of which turned out to be $RT = \frac{2\sum_{i,j} r_{i,j} - 6(n+1)}{n(n-1)}$, where $r_{ij} = 1$ if $i$ precedes $j$ and $r_{ij} = 0$, otherwise. De Reyck has reportedly shown that RT is identical to OS (Herroelen *et al* 1998 and Herroelen and De Reyck 1999).

Elmaghraby and Herroelen (1980) proposed that a complexity measure for the single resource RCPSP should be a function of an instance's number of feasible sequences, not just precedence feasible sequences as is the case of RI. They assert that, in this special case, this number is given by $perm(M^c)$, where $M^c$ is the conditional position matrix of the instance and $perm$ is the permanent function [23]. Computing $perm(M^c)$ could be exponential in terms of the number of activities, though. Nevertheless, they observe, there are special cases where it can be computed analytically or recursively. Notable among these is when the network structure is that of chains or unrelated parallel networks. Unfortunately, as they remark, the measure is not extendable to the general RCPSP.

In 1992, Bein, Kamburowski and Stallmann introduced a way of transforming a non-series-parallel directed acyclic graph[24], via node reductions, into a series-parallel graph using a minimum number of node reductions. A **node reduction** consists of eliminating a node with in-degree (out-degree) one and replacing the incoming (outgoing) arc and the outgoing (incoming) arcs by single arcs from the preceding node (nodes) to the succeeding nodes (node). The resulting graph may be further reduced to a single 'equivalent' arc via a sequence of standard series and parallel arc reductions. The minimum number of possible node reductions turns out to be a characteristic of the graph. In that respect, it can be thought of as a measure of how close to being series-parallel a graph is. This prompted De Reyck and Herroelen (1995)

[23] For details on the definition of those, the reader is referred to the original papers of the authors.
[24] The authors assume also that the nodes are topologically ordered, *i.e.* if an arc $(i, j)$ exists then $i < j$.

to suspect that it might play a role in determining the difficulty of RCPSP instances. They called it the **complexity index**, **CI** for short, and ran experiments, using the DH procedure, to gauge its significance in that regard. Their results show a negative, albeit small in magnitude, correlation between the CI and the computational time to solve instances. More importantly the results show a strong correlation between the CI and the CNC which explains the negative correlation that was observed by researchers between the CNC and the computational difficulty. Herroelen and De Reyck (1999) report that in subsequent studies the CI was found to have a strong impact on the processing time. In other studies, the CI turned out to be significant in predicting the computational difficulty when the resources are non-renewable.

### 1.5.3 Resource Related Measures

Resource availability, or tightness, is another factor besides precedence structure that could have a significant effect on the computational difficulty of an RCPSP instance. Recall that when the resources are plentiful the problem of minimizing the project duration reduces to the CPM model which is easy. When the resources are limited to the bare minimum then at least one activity, and possibly more, cannot be run in parallel with any other activity. Additionally, there would be little possibility of scheduling many activities concurrently. This potentially makes the problem easier to solve. In fact, Elmaghraby and Herroelen (1980) conjectured that the graph of the computational difficulty as a function of the resource availability is a bell shaped curve. Several measures were proposed to quantify the resource availability.

As early as 1966, Pascoe proposed a measure of resource availability which he called the **resource factor, RF** for short. This was later used by Cooper (1976) and Alvarez-Valdés and Tamarit (1989) in their studies of heuristics. It is defined as $RF = \frac{1}{nK}\sum_{i,k} r_{ik}$, where

$n$ is the number of activities, $K$ is the number of resources and $r_{ik} = 1$ if activity $i$ requires resource $k$ and $r_{ik} = 0$, otherwise. As such, RF is the average proportion of resources used by the activities. Note that $0 \leq RF \leq 1$. If $RF = 1$ then every resource is required by every activity. In their study, Alvarez-Valdés and Tamarit generated projects with RF of 0.5 and 1. They concluded that it is more difficult to schedule projects with an RF factor of 0.5 rather than projects with a factor of 1. Again, we have to caution, their focus was the heuristics not the complexity measures and "more difficult" most likely means that the solution quality of the heuristics used was bad. Using DH, Kolisch *et al* (1995) contradicted their finding. They observed that increasing the resource factor results in an increase of the computational time required.

Cooper (1976) introduced another resource related measure: the **resource strength RS**. This is the amount of a resource available per period in relation to the average activity requirement of that same resource: $RS_k = nR_k / \sum_i c_{ik}$, where $R_k$ is the per period availability of resource $k$ and $c_{ik}$ is the requirement of activity $i$ of resource $k$. This, too, was subsequently used by Alvarez-Valdés and Tamarit (1989). They report that problems with intermediate values are "more difficult" to schedule than problems with low or high values. Kolisch *et al* (1995) modified the definition to the following: $RS_k = \frac{R_k - R_k^{\min}}{R_k^{\max} - R_k^{\min}}$, where $R_k^{\min} = \min_i c_{ik}$ and $R_k^{\max}$ is the peak demand in the earliest start schedule of the corresponding CPM model. They report, again using the DH procedure, a monotone decrease in the average solution time as the resource strength increases. De Reyck and Herroelen (1996) use the same measure and contradict the findings of Kolisch *et al.* Their results show a bell shaped curve relating the solution time to the resource strength thereby confirming the conjecture of Elmaghraby and Herroelen. They attribute the monotone decrease found by Kolisch *et al* to the fact that the CI in those trials

was not held constant. Note that unlike RF and Cooper's RS, the modified RS incorporates precedence related information as $R^{\max}$ depends on the earliest start schedule which, in turn, depends on the precedence structure.

A third resource related measure was introduced by Patterson (1976). He calls it the **resource constrainedness** of resource $k$, $\mathrm{RC}_k$, and defines it as $RC_k = d_k/R_k$, where $R_k$ is defined as before, and $d_k$ is the average activity consumption of resource $k$, *i.e.*, $d_k = \sum_i c_{ik}/\sum_i r_{ik}$. In contrast with the modified RS, RC is a purely resource related measure. Another argument in favor of using it, reported by De Reyck and Herroelen (1996), is that it can distinguish between easy and hard instances when RS no longer can. Once more, confirming the Elmaghraby and Herroelen conjecture, the two researchers report a bell shaped curve of the average solution times of instances in terms of RC.

Two more resource measures were introduced by Kurtulus and Davis (1982) in a study of heuristics for multi-project scheduling. They are the **utilization factor**, **UF**, and the **average resource load factor, ARLF**. For a resource $k$, the utilization factor for a single RCPSP is the ratio of the per period requirement of resource $k$, assuming the corresponding CPM model duration, to the per period availability of the resource. In mathematical terms: $UF_k = \sum_i c_{ik}/CP \cdot R_k$, where $CP$ is the duration of the CPM model. The average resource load factor for a single RCPSP, on the other hand, is designed to identify whether the peak in the utilization of a resource occurs in the first or the second half of the CPM model duration. We refer the reader to the original paper of the authors for its precise mathematical definition and note that these two measures were conceived in an attempt to generate test data to assess the performance of some heuristics rather than the computational difficulty of RCPSP instances. Thomas and Salhi (1997) have used them to study the performance of heuristics for the single

RCPSP.

### 1.5.4 Test Data Sets

In the early days of RCPSP, there was no commonly acknowledged set of problem instances that researchers could use to test their exact methods and/or heuristics on. Researchers used to generate their own test sets which could include cases from real practice and the literature, possibly in a modified form. To widen the range of problems included in such test sets, characteristics of projects, more recently referred to as complexity measures, were defined which had to span some predetermined ranges. Often, at one end of the range, problem instances turned out to be computationally easy while at the opposite end they were harder. Comparisons of computational performance claimed in different studies could not be easily related to one another as the test instances and, frequently, even the measures were different. Examples of data sets used in those studies are the ones of Cooper (1976) and Alvarez-Valdés and Tamarit (1989), which are based on common measures, and Kurtulus and Davis(1982) and Thomas and Salhi (1997), in which the latter adds one more measure that the former did not consider. Patterson (1984) and Boctor (1990) are further examples of studies each with their own data sets.

Amongst the data sets just mentioned, the Patterson data set, sometimes referred to as **Patt**, was subsequently used by enough researchers that it became a de facto standard data set in the field. It consists of 110 instances, for the most part previously available in the literature. The number of activities ranged from 7 to 50, while the number of resources ranged from 1 to 3. The majority of projects (103) consisted of activities requiring three resources. Klein (2000) reports that the CNC of projects ranged between 1.08 and 3.05, the RF between 0.25 and 1 and the RS, in the modified sense, between 0 and 1.42. Recall that a resource strength set at 1 and

above indicates an easy instance as resources are in this case plentiful. In fact, the whole set was later recognized as 'easy', although for some solution methods, such as the Davis-Heidorn and Talbot algorithms, it was indeed challenging (cf. Patterson 1984).

In 1993, there appeared the first systematic way of obtaining data sets. Demeulemeester *et al* (1993) came up with a programmable procedure to generate what they call weakly and strongly randomized activity networks. A weekly randomized activity network is one where the size and structure of the network is predetermined, while the rest of the parameters are randomized. A strongly randomized network is one where even the size and structure are random. The networks generated, even the weakly random ones, turned out to be too random for practical purposes, though, which prevented the widespread use of the procedure. The problem was that no control over the complexity measures of generated instances was possible and this hindered the systematic generation of what are believed to be hard instances.

In the meantime, Kolisch and his colleagues were working on a similar network generator, **ProGen**, which, to some extent, alleviated the shortcoming of the Demeulemeester *et al* procedure. Besides obvious parameters such as the number of activities, the number of resources and other less obvious ones, their instance generator provided for controlling three complexity measure: the coefficient of network complexity, the resource factor and the resource strength. It also allowed for the generation of instances that are not strictly RCPSP instances. For example it allowed the use of non-renewable resources and multi-modes for activity executions. The reader is referred to Kolisch and Sprecher (1996) for the full list of control parameters.

Using ProGen, Kolisch and his colleagues generated two widely used sets of RCPSP instances. The first, **J30,** a.k.a. **SMFF**, contains 480 instances of 30 activities each, excluding two dummy activities as start and end activities. The activities required four resources. The

CNC of instances was set at one of three values: $1.5, 1.8$ and $2.1$. The RF took one of four values: $0.25, 0.5, 0.75$ and $1$; while the RS, which was set at the same value for all resources, was $0.2, 0.5, 0.7$ or $1$. Ten instances of each of the possible 48 combinations were generated. Some researchers omit from consideration the instances with RS set at 1, as for these instances the resource limits are not constraining. This leaves a data set of 360 instances. Of these, 96 have optimal durations equal to their corresponding CPM model durations (Möhring *et al* 1999). All instances of J30 were solved to optimality using several of the procedures that we previously reviewed.

The second set generated by Kolisch and his colleagues for RCPSP is called **J60**. It was generated using the same parameter values as J30 except that its instances have 60 non-dummy activities instead of 30. Again, some researchers omit from it the 120 instances with RS set at 1. Of the remaining 360, 175 have optimal durations equal to their corresponding CPM model durations (Möhring *et al* 1999). In contrast with J30, many instances of J60 could not be solved to optimality in reasonable time. Today, these two sets are considered by many researchers to be benchmark data sets in the field.

We should mention here the web site that Kolisch and colleagues have set up for benchmark data sets[25]. It contains, among other things, ProGen, the J30 and J60 data sets, further data sets **J90** and **J120**, optimal as well as best known heuristic solutions and best known lower and upper bounds for the instances. It also contains the Patterson, Alvarez-Valdéz and Tamarit and an expanded Boctor data sets. Further details can be obtained in Kolisch *et al* (1999).

A third activity network generator we mention is that of Agrawal *et al* (1996), DAGEN. As we stated in the previous subsection, at one time the CI was thought to play an important role

[25] ftp://ftp.bwl.uni-kiel.de/pub/operations-research/psplib

in determining the difficulty of an RCPSP instance. The aim of the Agrawal *et al* generator was to generate instances with a predetermined CI. This was accomplished by generating a skeleton network for a specified CI and then filling in this skeleton with the rest of parameters that an RCPSP instance would possess. All networks with the same CI had the same skeletons while the rest of parameter values were randomized. In this sense, the networks generated were not totally random.

Finally, we mention the data sets used by Thomas and Salhi (1997) in studying the performance of a heuristic they proposed. They have generated projects with 45 up to 666 activities with three predetermined complexity measures, CNC, ARLF and AUF. Three values were used for the first and third measures: low, mid and high; and four levels for the second measure: low, mid, high and two-peak. Four environments, in terms of activity count, were considered. Fifty projects for each combination of measures in each of the first three environments were generated and twenty for each combination in the fourth, for a grand total 6120 projects. The authors used the data set to select the best heuristics for the 144 problem types they considered from amongst a dozen or so that included theirs.

# Chapter 2

# Integer Programming Models

We have remarked in our discussion of exact methods that Integer Programming approaches were among the first to be tried to solve RCPSP. Over the past forty years or so, several researchers have attempted either to solve their integer programming formulation of the problem or use such a formulation to derive a lower bound. We have already mentioned, in that respect, the Pritsker *et al* (1969) formulation and its modification adopted by Möhring *et al* (1999) to obtain a lower bound and Sankaran *et al* (1999) to derive a cutting plane approach. We have also mentioned the disjunctive integer programming formulation of Alvarez-Valdés and Tamarit (1993) used to study the facets of the corresponding LP feasible region and derive valid inequalities for its IP counterpart. Further, we have reviewed the Mingozzi *et al* formulation (1998) and the important bound it generated. In addition to that, Klein (2000) lists three more integer programs that have been proposed, two of them due to him, and the literature surely contains few more.

This chapter is mainly concerned with presenting closely related integer programming models for RCPSP and its Preemptive-Resume variant, **PRCPSP**. We make no attempt at sug-

gesting exact algorithms based on these models; though, we propose a heuristic approach that can be adapted to any mathematical programming formulations for those problems.

## 2.1 An RCPSP Integer Program

We start by setting our notation. Let

$T$ : be an upper bound on the project duration,

$d_i$ : the duration of activity $i$,

$s_i$ : the starting time of activity $i$,

$x_{i,t}$ : the activity level of $i$ in period $t$, *i.e.*, 0 if $i$ is inactive and 1 if it is fully active,

$b_{i,k}$ : the per period requirement of activity $i$ from resource $k$ when $i$ has activity level 1,

$R_k$ : the per period capacity of resource $k$,

$I$ : the set of activities $1, \cdots, n$.

$H$ : the set of precedence relationships among the activities in $I$,

$K$ : the set of resources.

Then the problem is:

$$
\begin{aligned}
& Min \quad s_n \\
& s.t. \\
\sum_{i \in I} x_{i,t}.b_{i,k} \leq & \; R_k, \forall\, k \in K, \forall\, t = 1, \cdots, T, \\
s_i - s_j \leq & \; -d_i, \forall\, (i,j) \in H,
\end{aligned}
$$

For all $i \in I$ :

$$
\begin{aligned}
x_{i,1} &= y_{i,1}, \\
x_{i,t} - x_{i,t-1} &= y_{i,t} - z_{i,t},\ t = 2, \cdots, T, \\
\sum_{t=1}^{T} y_{i,t} &\leq 1, \\
\sum_{t=2}^{T} z_{i,t} &\leq 1, \\
\sum_{t=1}^{T} x_{i,t} &= d_i, \\
My_{i,t} - s_i &\leq M - t,\ t = 1, \cdots, T, \\
My_{i,t} + s_i &\leq M + t,\ t = 1, \cdots, T, \\
s_i, x_{i,t} &\geq 0 \text{ and } y_{i,t}, z_{i,t} \in \{0, 1\}, \forall i, t.
\end{aligned}
$$

The interpretation of these constraint sets is as follows. The first set insures that the caps on resource availabilities are observed. The second set enforces the precedence relationships. The block constraint for each activity $i$ (1) tracks changes in the activity level of $i$ (2) limits these changes to one start and one finish (3) makes sure $i$ is processed for $d_i$ periods and (4) sets the starting time for $i$. In these blocks, $M$ is a large number. The following observations can be made:

1. $s_i$ and $x_{i,t}$ are continuous. Also $x_{i,t}$ is implicitly limited to $[0, 1]$. Therefore, the first two sets of constraints which can be seen as coupling constraints are linear continuous while the block constraints are mixed integer linear.

2. The number of variables and constraints can be greatly reduced by using lower and upper bounds on the start time of the activities. The simplest lower bounds on the start times of the activities are the earliest start times of the corresponding CPM model. The simplest

start times upper bounds are the latest start times of the CPM model derived using a feasible project duration.

3. Instead of $\sum_{t=1}^{T} y_{i,t} \leq 1$ we can have $\sum_{t=1}^{T} y_{i,t} = 1$. That sum is 1 anyway. Also, instead of $\sum_{t=2}^{T} z_{i,t} \leq 1$ we can get $\sum_{t=1}^{T} z_{i,t} = 1$ by simply adding the slack variable $z_{i,1.}$ In any case, these constraints are the simplest of the model. They are common across integer programming formulations of many problems and are usually referred to as generalized upper bounding constraints. Frequently, they are treated in a special way apart from the rest of the constraints of a model. Note, in our model, that by guaranteeing at most one jump up in an activity level and at most one jump down, these constraints insure that no preemption is allowed.

4. The model would still be valid if the 0/1 requirement is shifted to the $x$ variables instead of the $y$ and $z$ variables. We would have half as many integer variables but the coupling constraints would become mixed integer instead of continuous.

5. The constraints

$$
\begin{aligned}
x_{i,1} &= y_{i,1}, \\
x_{i,t} - x_{i,t-1} &= y_{i,t} - z_{i,t}, \; t = 2, \cdots, T, \\
\sum_{t=1}^{T} y_{i,t} &\leq 1, \\
\sum_{t=2}^{T} z_{i,t} &\leq 1
\end{aligned}
$$

could be replaced with

$$\begin{aligned}
x_{i,1} &\leq y_{i,1}, \\
x_{i,t} - x_{i,t-1} &\leq y_{i,t},\ t = 2, \cdots, T, \\
x_{i,t} - x_{i,t-1} &\geq -z_{i,t},\ t = 2, \cdots, T, \\
\sum_{t=1}^{T} y_{i,t} &\leq 1, \\
\sum_{t=2}^{T} z_{i,t} &\leq 1.
\end{aligned}$$

That, though, increases the number of constraints.

6. If the activity levels are allowed to vary between 0 and 1 and if the precedence constraints are interpreted as start to start relationships, *i.e.*, $(i, j) \in H$ means activity $j$ can only start $d_i$ periods after the start of activity $i$, then we can eliminate the need for the $z_{i,t}$ variables by using the following block constraints:

$$\begin{aligned}
x_{i,1} &\leq y_{i,1}, \\
x_{i,t} - x_{i,t-1} &\leq y_{i,t},\ t = 2, \cdots, T, \\
My_{i,t} - s_i &\leq M - t,\ t = 1, \cdots, T, \\
My_{i,t} + s_i &\leq M + t,\ t = 1, \cdots, T, \\
\sum_{t=1}^{T} x_{i,t} &= d_i, \\
\sum_{t=1}^{T} y_{i,t} &\leq 1, \\
s_i, x_{i,t} &\geq 0\ and\ y_{i,t} \in \{0, 1\}, \forall i, t.
\end{aligned}$$

The resulting problem is no longer RCPSP but this demonstrates the flexibility potential of the model specially if a lower bound is to be derived.

7. $s_i$ could be replaced with $\sum_{t=1}^{T} t y_{i,t}$ in the coupling constraints and eliminated in the block constraints, thus eliminating $|I|$ variables and $2|I|T$ constraints. But that would put the $y_{i,t}$ variables in the coupling constraints.

8. $s_i$ can also be eliminated and the precedence constraints replaced with

$$\sum_{r=t}^{T} y_{i,r} + \sum_{r=1}^{t+d_i-1} y_{j,r} \leq 1, \ \forall \ t = 1, \cdots, T \ , \ \forall \ (i,j) \in H$$

or

$$\sum_{r=t}^{T} x_{i,r} + \sum_{r=1}^{t+d_i-1} x_{j,r} \leq d_i, \ \forall \ t = 1, \cdots, T \ , \ \forall \ (i,j) \in H.$$

The rationale for these constraint sets is that if activity $i$ is started on or after time $t$ then activity $j$ cannot be started before time $t + d_i$. The advantage of the second set over the first is that it keeps one type of variables in the coupling constraints, namely the continuous $x_{i,t}$.

Of course, the above model, in whichever form is adopted, is bound to have many more variables and constraints than the concise Pritsker *et al* formulation. The advantage of this model, though, is its structure. To our knowledge, this is the first RCPSP model in the literature that possesses a block diagonal structure. Structure is of paramount importance when it comes to solving large scale programs of any type. The block diagonal structure that this model exhibits has proved to be very beneficial in a Linear Programming setting. The most obvious way of taking advantage of it here is when solving LP or Lagrange relaxations of the model

in as part of a branch and bound solution approach. There, the Dantzig-Wolfe Decomposition is suitably designed for the linear program with block diagonal structure. A more satisfactory approach to our model, at least on a mathematical level, is the one adopted by Rana (1992) in which the structure dealt with happens to be exactly ours, *i.e.,* linear continuous coupling constraints and mixed integer linear blocks. It relies on the Sweeney and Murphy (1979) decomposition of the coupled block diagonal structure with all integer variables. The literature contains few more papers on the decomposition of such structures. Unfortunately, the ones we encountered tended to be either too theoretical for immediate purposes or else of an outright heuristic nature.

## 2.2 A PRCPSP Integer Program

Recall that in this preemptive-resume variant of RCPSP the processing of the activities can be interrupted when convenient and resumed at negligible or no cost. This can be thought of as a relaxation of one of the tenants of RCPSP and as such the optimal value of the relaxed problem is a lower bound on the optimal value of the original. Following the same notation as for our RCPSP model, here is our integer programming formulation of the problem:

$$Min \quad \sum_{t=1}^{T} t x_{n,t}$$

$$s.t.$$

$$\begin{aligned}
\sum_{i \in I} x_{i,t}.b_{i,k} &\leq R_k, \ \forall\ k \in K, \ \forall\ t = 1, \cdots, T, \\
\sum_{r=1}^{t} x_{j,r} &\leq d_j.(1 - x_{i,t}), \ \forall\ t = 1, \cdots, T, \ \forall\ (i,j) \in H,
\end{aligned}$$

$$\begin{aligned}\sum_{t=1}^{T} x_{i,t} &= d_i,\ \forall\, i \in I,\\ x_{i,t} &\in \{0,1\},\ \forall\, i \in I,\ \forall\, t = 1, \cdots, T.\end{aligned}$$

As before, the first set of constraints insures that the resource limits are not exceeded. The second makes sure that if $(i,j) \in H$ and if $t$ is the time at which activity $i$ finishes then activity $j$ cannot start before $t$ as $x_{i,t} = 1$ and $\sum_{r=1}^{t} x_{j,r} = 0$ in that case. The third set makes sure that the activities are fully processed. Note that the last activity $n$ is, as discussed in the previous chapter, by convention a dummy activity of zero duration. Of course, such an activity cannot be preempted and $x_{n,t} = 1$ for exactly one $t$. Whence its start and finish times can be represented by $\sum_{t=1}^{T} t x_{n,t}$ which can also represent the finish time of the project. The following couple of observations can be made:

1. As for RCPSP, the number of variables can be greatly reduced by employing earliest and latest start times of the corresponding CPM model or even better lower and upper bounds.

2. Also as for RCPSP, the precedence constraints can be replaced with

$$\sum_{r=t}^{T} x_{i,r} + \sum_{r=1}^{t+d_i-1} x_{j,r} \leq d_i,\ \forall\, t = 1, \cdots, T\ ,\ \forall\, (i,j) \in H.$$

   The rationale for this constraint is the same as before. Its advantage over what we have in the model is that its coefficients are all 1.

3. If we wish to allow the activity levels to be real instead of integers then the precedence

constraints can be replaced with

$$\begin{aligned}
\sum_{r=1}^{t} x_{j,r} &\leq d_j.(1-u_{i,t}),\ \forall\, t=1,\cdots,T,\ \forall\,(i,j)\in H,\\
x_{i,t} &\leq u_{i,t},\ \forall\, t=1,\cdots,T,\ \forall\, i\in I,\\
\sum_{t=1}^{T} u_{n,t} &= 1.
\end{aligned}$$

In that case, the variable constraints would be

$$x_{i,t}\geq 0,\ u_{i,t}\in\{0,1\},\ \forall\, i\in I,\ \forall\, t=1,\cdots,T.$$

## 2.3 A Look-Ahead Heuristic

Suppose we are given an RCPSP or a PRCPSP instance, $\mathcal{P}$, to solve. Then the cumulative resource requirements of the project, $R_c$, can be computed quite easily. So suppose that instead of having the means to obtain an optimal schedule for $\mathcal{P}$ in a direct way, we can solve the problem of maximizing the cumulative resource consumption on fixed intervals $[0,t]$; which would require two algorithms, one for each type of problems. Then, in order to find the optimal schedule for $\mathcal{P}$ we could apply the following algorithm presented in broad outline:

Step 1 : Obtain a lower bound for the optimal project duration, $LB$.

Step 2 : Solve the resource maximization problem over $[0,LB]$.

Step 3 : Compare the resulting cumulative resource consumption to $R_c$. If the two are equal then we are done. The schedule obtained in step 2 is an optimal schedule for $\mathcal{P}$. Else, increase $LB$ by 1 and go back to step 2.

Unfortunately, solving the resource maximization problem is not much easier than minimizing the makespan of the project. But what this illustrates is that there is a close connection between the two. In fact it is not hard to see that for the optimal project duration $T$ of $\mathcal{P}$, the optimal solution spaces of the two problems are one and the same. This, naturally, motivates the search for a heuristic algorithm for the resource maximization problem.

Now, suppose we look at a special instance of RCPSP or PRCPSP where all the activity durations are one[1]. Then it is intuitively appealing in this case to solve the problem by maximizing, in each period, the cumulative resource consumptions and collecting the solutions of each into one global solution for the whole problem. We will see in the next chapter that while this idea does not lead to an optimal solution to our problem, this line of thinking does. In fact, we will see that the optimal solution can be constructed using not optimal solutions of each period but using "maximal" solutions of those individual periods. But for the time being, we can do better than collecting optimal solutions of periods by collecting optimal solutions of subintervals of the planning horizon. That is, we can partition the interval over which we want to maximize the cumulative resource consumptions into disjoint subintervals, solve the maximization problem over each and collect the individual solutions into a global one.

This same idea, *i.e.,* collecting optimal solutions of each period into a global solution, can be carried to the general case of arbitrary durations but with one necessary modification in the non-preemptive case. It is quite possible, actually probable, that some activities might be started but not finished by the end of a subinterval. In that case the subproblem to solve in the consecutive subinterval will have to take this fact into consideration. In other words, the continuations of these activities will have to be "initial conditions", to borrow a term from

[1] Note that in such a case the distinction between the two problems disappears.

differential equations, for the succeeding subproblem. Such initial conditions are very easy to accommodate if the subproblems are modeled as integer programs; just set the corresponding variables to fixed values, simplify or eliminate the affected constraints and eliminate these same variables. We should add here that these integer programs may have the same variables and constraints as the integer programs we proposed for the makespan minimization problems. The only difference with those would be the objective functions which should take the form $Max \sum_{i,t}(x_{i,t} \sum_k b_{i,k})$ .

In view of the previous discussion, we propose the following outline of an algorithm to solve RCPSP or PRCPSP:

Step 1 : Obtain a reasonably good upper bound on the optimal project durations and partition the interval into, say, $m$ subintervals of equal durations $k$.

Step 2 : For $i = 1$ to $m - 1$, set up and solve the problem of maximizing the cumulative resource consumptions over subinterval $i$ taking into account the initial conditions in the preemptive case, if there is any.

Step 3 : Setup and solve the problem of minimizing the makespan over subinterval $m$, again taking into account the initial conditions if any exists. Say the makespan of this subproblem is $T'$.

Step 4 : Collect the subproblem solutions into one global solution. Its makespan should be $k(m - 1) + T'$.

Several observations can be made about this general outline:

1. It is the parameter $k$ that determines the look-ahead capability of the solution. At each

time $k.i$, $i = 0, 1, \cdots, m-1$, the solution reached is guaranteed to be the best for the foreseeable $k$ periods. If $k = 1$ then the algorithm reduces to the greedy heuristic while if $k$ is the minimum makespan possible for the project then the schedule reached is actually a makespan optimal one.

2. The parameter $k$ should be chosen small enough so that the solution of each subproblem is not expensive computationally. At the same time $k$ cannot be too small that the quality of the solution becomes unacceptable. There is a balancing act that needs to be performed here.

3. The subproblems could greatly benefit from preprocessing. Not every activity needs to be included in every subproblem. Lower and upper bounds on the start time of the activities can rule out eliminating many of them in some subproblems. Additionally, activities that are scheduled in subproblems should not reappear in subsequent ones. This preprocessing is specially important if the subproblems are modeled as integer programs.

4. From a practical point of view, changes to the structure of a project that occur while the project is under way need only be considered in the subintervals they belong to. Unless such changes occur in the immediate subinterval, there is no need to re-solve the remainder portion of the project, every time changes occur, in order to schedule the activities of the subinterval. In this sense, the procedure focuses on the immediate and relatively short horizon in anticipation of structural changes to the project in the more distant future.

# Chapter 3

# A Shortest Path Paradigm

This chapter explores a new approach to the Resource Constrained Project Scheduling Problem. It starts with establishing the rationale for a new way of looking at the problem and lays its foundations in sections 2 and 3 as a pair of networks with specially constructed nodes. Section 4 refines this foundation. Section 5 investigates a potential implication of some results in section 4 concerning the difficulty of the problem in terms of the distribution of its activity durations. Section 6 examines how to generate the networks' nodes. Section 7 develops further concepts to make the approach workable and outlines a bidirectional algorithm to solve the problem. Section 8 is concerned with further reductions in the networks' sizes as well as another aspect of the computational effort. Section 9 discusses possible implementations of the algorithm while section 10 details the experience gained from one such implementation.

## 3.1 Critique of Integer Programming and Branch and Bound Approaches

We mentioned in our first chapter that the Branch and Bound approach for solving RCPSP proved to be the more efficient approach to solve the problem; especially when compared to integer programming approaches. The main reason for this relative success, it is commonly believed, is that BaB approaches are able to take advantage of the special structure of RCPSP in ways that integer programming approaches are not. After all, the most successful approaches to solve integer programming problems were and still are themselves BaB based. It is quite natural in this case, then, for a BaB method that is specifically tailored to RCPSP to outperform another BaB approach whose purpose and design are geared towards solving general problems. In fact, we contend, as long as the BaB based algorithms are the more successful algorithms for the Integer Programming problem, integer programming procedures should not be expected to outperform special purpose BaB based methods; whether in so far as RCPSP is concerned or as far as any such difficult NP-complete problem one might face.

That is not to say that BaB methods take full or even near full advantage of the structure of RCPSP. They simply take better advantage of it than integer programming and other previously proposed methods. Indeed, we believe that they don't take nearly enough advantage of that structure in at least two aspects.

First, RCPSP has both a start and an end activity, or can be made so. While it is quite intuitive to conceive that the root node of a BaB procedure should be the start activity, it is no less conceivable that it be the end activity. Actually, one might venture to think that a procedure could start from both ends and meet somewhere in the middle. Indeed, a heuristic

along those lines was mentioned in our review of literature. But to the best of our knowledge no one has ever attempted such an exact procedure, and for good reason. Unfortunately, neither Integer Programming nor Branch and Bound are amenable to deriving a termination criterion for such a procedure that guarantees optimality. This applies to **Breadth First** BaB strategies but more especially to their **Depth First** counterparts which, conventional wisdom says, are the more efficient computationally. Moreover, it is quite conceivable that, due to an instance's structure, a one directional algorithm solving the instance is very much time consuming in one direction but quite efficient in its opposite. Not knowing which direction to choose might be disadvantageous at best. Having a bidirectional algorithm could very well mitigate the disadvantage.

Second, and perhaps more fundamentally, in all BaB methods we have reviewed except one, it is relatively easy to compute the additional amount of time incurred when moving from one node to any of its descendents before obtaining the descendent itself. To clarify our assertion, consider the prototype example of a BaB algorithm; that for the Integer Programming Problem. A node in that algorithm represents the LP-relaxed version of the original optimization problem with a set of simple inequalities that have been added to it during the branching process. Branching from a node consists of choosing a variable $x_i$ that takes a non-integer value in an LP optimal solution of the problem at the node, $a_i$, and considering at the descendent node the same optimization problem solved at the parent node but with one of two simple inequalities added to it, $x_i \leq [a_i]$ or $x_i \geq [a_i] + 1$[1]. Of course, associated with each node is also the value of the LP solution at that new problem. Notice that the only way to obtain this value at a descendent node is to set up the optimization problem and derive the solution again. This

[1] As is commonly the case, $[a_i]$ denotes the greatest integer in $a_i$.

"cost" incurred while moving from one node to its descendent can only be obtained, in a rather involved way, as the difference in the values of the two nodes. In contrast with this, the cost incurred in moving from a node to its descendent in BBLB3, GSA, Progress/Scatter and the DH procedures can be obtained as the least remaining time of any of the active activities, of the partial schedule built up to the parent node, at which enough resources are released to process the added activities in the descendent node[2]. Now, it is our contention that models where the cost of moving from a node to its descendent can be computed in a relatively simple way are better conceived as Dynamic Programming or more simply Shortest Path problems. The main reason for this, we maintain, is that in such cases the simple computation of the cost makes a labeling approach to the model, as in many network labeling algorithms, computationally feasible. At any rate, none of the BaB procedures we reviewed takes advantage of the ease with which the cost or time incurred in moving from one node to its descendent can be computed. Incidentally, we might add here, the remaining procedure where this property is not so evident, namely BKST, is also the least efficient procedure we reviewed.

With these two shortcomings of Branch and Bound as it applies to RCPSP in mind, we move to presenting our own approach to the problem.

## 3.2 Problem Setup

Consider an RCPSP instance and an arbitrary feasible schedule to it. At any time period during its execution, the state of this project can be characterized by (1) the activities that are completed, (2) those that are currently active with their remaining times and (3) those activities

[2]In these procedures also, the incurred cost up to a node can be seen as the earliest possible starting time of the activities added to the partial schedule at the said node.

that have yet to start. Using the precedence relationships of the project, this characterization of the state can be rendered more concise by eliminating from it the activities that precede the active ones and those that succeed them; as the disposition of those activities is inferable from the active ones. What remains, besides the *active* activities with their residual times, are the activities that are neither predecessors nor successors of the *active* ones. The characterization of a project's state can now be rendered even more concise by eliminating from those latter activities the ones that possess uncompleted predecessors; as, again, their disposition could be inferred from those same uncompleted predecessors. Thus, the state of the project at any time period $t$ can be characterized by the activities that are active at time $t$, with their residual times, and the precedence feasible but not yet started activities as of time $t$.

Notice that in the next time period $t + 1$, any activity that is already active at time $t$ will either be completed or else will have its residual time decrease by one. Additionally, a new activity could be started depending on the resource requirements of active activities, the resource requirements of the precedence feasible activities and the resource limits. But whether activities are completed or precedence feasible ones are started or both, the state of the project at time $t+1$ will be different from its state at time $t$. Actually, if a state of the project at time $t+1$ is reached in this way then we say that the project's state at time $t$ immediately precedes its state at time $t + 1$. Of course, there could be many different possible states at time $t + 1$ reached in this fashion in which case the state of the project at time $t$ immediately precedes all those states at time $t+1$. Naturally, a state $S_i$ precedes another $S_j$ if there is a chain of states that starts with $S_i$ and ends with $S_j$.

Now, let every possible state of the project be represented by a node and let every immediate precedence relationship between two states be represented by an arrow pointing from the

preceding state to the succeeding one. Then by adding a source, or start, node with arrows to the nodes with no predecessors and by further adding a terminal node with arrows from the nodes with no successors, we obtain a directed graph of the possible states of the project each path of which between the start and end nodes represents a feasible way to execute the project. We refer to this graph as the **state graph** of the project. Clearly, the objective of the problem can now be seen as that of finding a path between the source and terminal nodes of the state graph having the least number of intermediate nodes.

## 3.3 The State Network

Note that in the state graph representing an instance, any state has at least one immediate successor that includes no additional activity. Some have exactly one such as when only active activities of the immediate predecessor state are processed in the immediate successor state. Actually, we may obtain, within the state graph, chains of nodes which, mainly due to the non preemption requirement, have only one immediate successor. They represent situations where the set of active activities of a state is the same for several time periods; first by reason of non preemption and second for either lack of sufficient resources or lack of precedence feasible activities or both. Since the activities in the corresponding nodes are the same with the only difference between the nodes due to the decrease in the residual times of the active activities, it is feasible and even desirable to coalesce such chains into single nodes. In this way each coalesced node corresponds to the start or completion of at least one activity. Note that the number of nodes in the chains can be marked next to each outgoing arrow from their representative nodes. Moreover, this number of nodes can also be interpreted as the number of time periods the

corresponding active activities are processed concurrently. In fact, extending this interpretation and notation to chains of single nodes, the state graph of an RCPSP instance can readily be seen as transformed from a graph into a network. Thus, our problem now is to find the shortest path between the source and terminal nodes of the induced network.

The number of nodes in the induced network can be further reduced by noting that if along a path from the source node to the terminal node, a node other than the start node corresponds to only the start of some activities, *i.e.,* without any activity completion to conjunctly prompt it, then that node can be coalesced with the node immediately preceding it along the same path. In other words, those activities that engendered the node could be started at the same time as the immediately preceding node along the same path, with no other changes in the starting times of any other activities, as evidently they need not be delayed due to lack of sufficient resources. Thus, other than the start node, the network of an instance need only include nodes induced solely by the completion of activities. We call this further simplified network the **state network** of an instance and, henceforth, whenever we speak of a state we shall always mean a state of that network.

**Example 5** *To illustrate these concepts, consider the project of five activities of Figure 3-1 in its AoA mode*[3]*. Its state network of* 22 *nodes is given in Figure 3-2. The shortest path of this network, marked in bold type, has length* 11*. This, coincidentally, is also the length of the critical path. Note that the* 'start' *state in this network is actually the state representing all precedence feasible activities at time* 0*, i.e., state* $\{a_1, a_2, a_3\}$*. In fact, in any state network, the* start *state represents the set of all activities precedence feasible at time* 0*.*

[3] *Note that in this representation we have dealt away with the convention that two or more activities shall not share the same end nodes; for we see no advantage of following it here.*

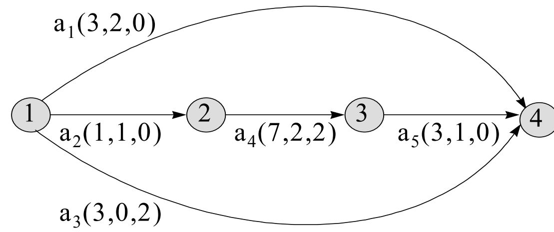


Figure 3-1: A project to illustrate the state network concept - AoA representation

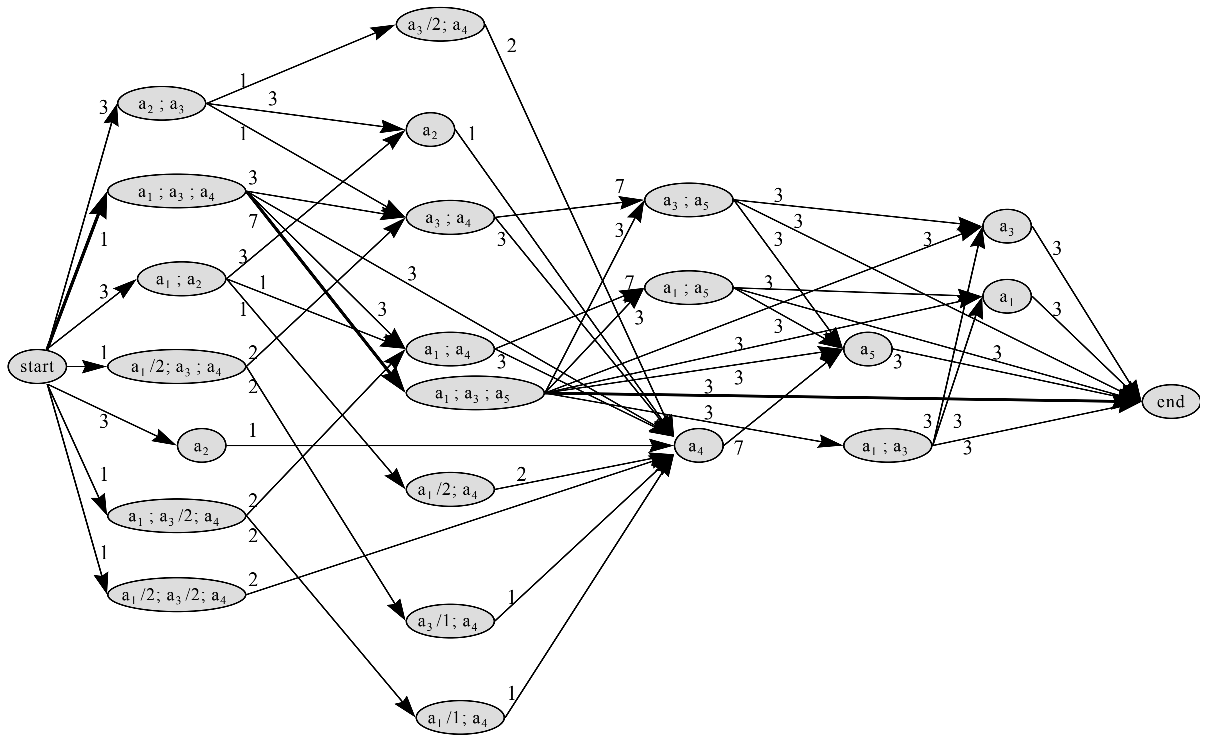


Figure 3-2: The state network of the project in Figure 3-1.

## 3.4 Node Reductions

Since the Shortest Path problem is in the class of P problems, this transformation of the strongly NP-Complete RCPSP into Shortest Path is attractive if the number of nodes of the state network can be kept under control. We should not expect this number to be polynomial in the size of an instance as that would imply that RCPSP is polynomial. Nevertheless, the smaller this number is the larger the size of instances we can realistically expect to solve. In this regard, it is very important to realize that we need not generate the complete state network of an instance before we set to find its shortest path. For example, if we are using Dijkstra's algorithm for Shortest Path then, at each iteration of the algorithm, we need only generate the immediate successor nodes of the node most recently labeled. By the time the end node is labeled, which terminates the algorithm, we are likely to have generated only part of the state network.

Note that an upper bound on the number of immediate successors for a given node is $2^m - 1$, where $m$ is the number of precedence feasible activities at the node. This, even, is an attainable upper bound as the start node, among several others, in Figure 3-2 indicates. Obviously, this exponential number of immediate successors could result in a significant total number of nodes in the state network as compared to the number of activities of the corresponding project[4]. A way out of this difficulty, it may seem, is to set up one or more mathematical programs to come up with the most promising immediate successor node or nodes to include in a reduced state network. Indeed, such an idea could be used as part of a heuristic procedure but, we think, is unlikely to be sufficient for an exact method. For to be easily solvable, the mathematical

[4] The reader may note here that an estimate of the total number of nodes in the state network is complicated by the possibility of different nodes sharing the same immediate successors.

program would have to rely on the activities of the node only or at most few of their descendents. But as the projects in Theorem 4 of Chapter 1 show, those "few" activities and their descendents could well be all the projects's unscheduled activities. In fact, the following example shows that even for far simpler projects with only one resource and unit activity durations, such mathematical programs are unlikely to yield the nodes that need to be selected.

**Example 6** *Consider the project activities in Figure 3-3. With only four activities, a single resource and unit activity durations, this is one of the simplest projects there can be. The precedence feasible activities at time zero are* $\{a_1, a_2, a_3\}$*. A reasonable mathematical program to characterize the best decision to be made at this time is to maximize the resource consumption. This is easily set up as a Knapsack problem that yields* $\{a_1, a_3\}$ *to start, for a project duration of three time units. But, clearly, the optimal duration is two. Hence maximizing the resource consumption is not necessarily a good idea here. Notice that had we maximized the number of activities started, subject to the resource availability, we would have obtained* $\{a_1, a_2\}$ *or* $\{a_1, a_3\}$ *or* $\{a_2, a_3\}$ *to start; which would not have ruled out the optimal solution as before.*
*Now, consider the same project but with different resource availability and requirements as depicted in Figure 3-4. The precedence feasible activities at time zero are again* $\{a_1, a_2, a_3\}$*. Maximizing the number of activities started yields* $\{a_1, a_3\}$ *to start, for a project duration of three time units. This, however, is suboptimal as maximizing the resource consumption happens to yield the optimal duration of two.*

The lesson we draw form this example is that even for one of the simplest classes of projects, those with a single resource and unit durations, it is very hard, if not impossible, to characterize the best decision to take at a given state of the project solely in terms of the activities of the

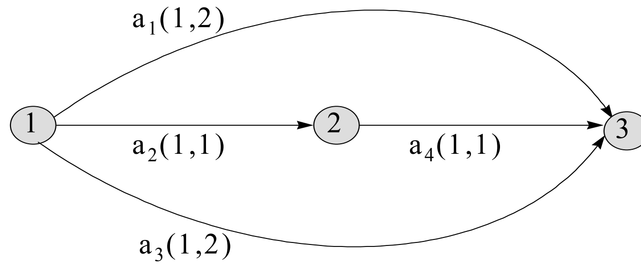


Figure 3-3: Counterexample to maximizing resource consumption - single resource, unit duration case.

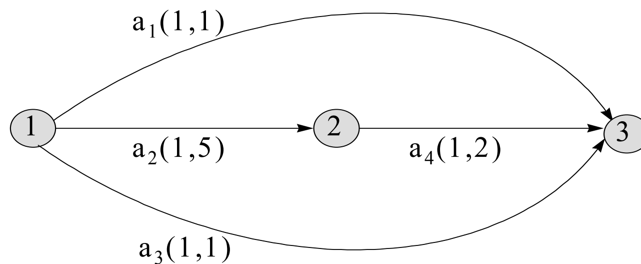


Figure 3-4: Counterexample to maximizing the number of activities started - single resource, unit duration case.

given state. We are forced, then, to consider in all cases, simple and general, all the subsets of resource compatible activities at the given state for inclusion as nodes in the state network. For this reason, we think the emphasis of research on an exact procedure should be placed not on the few subsets that might be promising to include but rather on the subsets that need not be considered. It is in this spirit that we present Theorem 8 whose proof requires the following intuitive and well known lemma.

**Lemma 7** *Consider an RCPSP instance with a minimum completion time of $d_1$ and let $a$ be one of its activities. Modify the project by reducing the duration of $a$. Let $d_2$ be the minimum duration of the new project. Then $d_2 \leq d_1$.*

**Proof.** To see this, consider the graphic representation of the original instance, whether AoN or AoA. Identify the paths between the start and end activities of lengths $d_1$. Then either activity $a$ belongs to all of those paths or it belongs to some of them or it belongs to none of them. In the first case, since those paths determine the minimum completion time of the instance, reducing the duration of $a$ reduces the completion time[5]. In contrast, in the second and third cases reducing the duration of $a$ has no effect on the makespan but certainly cannot increase it. Thus in all cases we get $d_2 \leq d_1$. ■

**Theorem 8** *Let $S$ be a state of the state network of an RCPSP instance and $A$ be its subset of ongoing activities. Let $V \subset S$, $V \cap A = \emptyset$, be another subset of resource compatible activities which processing, along with $A$, induces a successor state, $S_1$, to $S$. Let $l$ be the shortest start to end path of the state network passing through $S$ and $S_1$ and assume there exists an activity*

[5]Note, in this case, that the reduction in the duration of $a$ reduces the minimum project duration up and until some limit for $a$'s duration beyond which other paths of the instance become dominant; in which case they now determine the makespan. This limit can be the duration of $a$ itself but that possibility can easily be lumped with the second case.

*$a \in S - (A \cup V)$ which is resource compatible with the activities of $A \cup V$ and not longer in duration than* (1) *the activities in $V$ and* (2) *the residual durations of the activities in $A$. Let $S_2$ be the successor state to $S$ induced the processing of the activities in $A \cup V \cup \{a\}$.Then the shortest start to end path through $S$ and $S_2$ has length at most $l$.*

**Proof.** Note first that if $d_a$ denotes the duration of $a$ and $d$ denotes the time to reach state $S_1$ through state $S$ then, by assumption, $d \geq d_a$.

Subdivide the shortest start to end path through $S$ and $S_1$ into three parts as follows. The first part starts at the start node, $s$, and ends with $S$. Say its length, or duration, is $l_0$. The second part consists of $S$, $S_1$ and the arc between them. Its length is $d$. The third and last part is the rest of the shortest path through $S$ and $S_1$. It starts at $S_1$, ends at the end node and has length, say, $l_1$. Note that $l_0 + d + l_1 = l$.

In the same way, subdivide the path through $S$ and $S_2$. Its first part is the same as the first part of the shortest path through $S$ and $S_1$. It starts at $s$ and ends at $S$ and is of length $l_0$. The second part consists of $S$, $S_2$ and the arc between them. Its length is $d_a$. The final part consists of the rest of the shortest path through $S$, $S_2$. Say its length is $l_2$. In this way, the length of the path through $S_2$ is $l_0 + d_a + l_2$.

Note now that the third part of the path through $S_1$ represents almost the same subproject as the one represented by the third part of the path through $S_2$ with the only difference owing to the presence of activity $a$ in the first subproject. This absence of $a$ in the second subproject can be interpreted as reducing $a$'s duration to zero in that subproject. Hence, by the previous lemma, we may conclude that the minimum duration of the second subproject is at most the minimum duration of the first. That is $l_2 \leq l_1$. This says that $l_0 + d_a + l_2 \leq l$, *i.e.*, the shortest

path through $S$ and $S_2$ has length at most $l$. ■

An immediate consequence of this theorem is the following result of considerable significance for a special type of projects. We precede it by formalizing a basic concept it uses.

**Definition 9 (Maximal Subset)** *A subset of the set of precedence feasible activities to start at a given time during the execution of a project instance is said to be* ***maximal*** *if it cannot be executed concurrently with any other precedence feasible activity without violating a resource limit.*

Observe here that if sufficient resources are available to process all precedence feasible activities concurrently then the only maximal subset would be the set of precedences feasible activities itself.

**Corollary 10** *If all activity durations of an RCPSP instance are the same then, in a shortest path approach to the problem, the states of the corresponding state network need only include those induced by the processing of maximal subsets of activities.*

**Remark 3** *Although in practice RCPSP instances do not usually have the same duration for their activities, preemptive RCPSP instances do. In fact, one way to solve the preemptive case is to replace each activity with a chain of subactivities of unit durations. This is how Demeulemeester and Herroelen (1996), for example, approached the problem and used their DH procedure on it.*

Unfortunately, as might be construed from the previous remark and as we shall see in the following example, this corollary is hard to generalize. In fact, this example is especially instructive for showing what the theorem does *not* imply not only what it does.

**Example 11** *It may be tempting, in light of Theorem 8, to generalize Corollary 10 in the following way. If, according to the theorem, at any state we can pack the subset of activities to process with as many activities as the resource limits allow, subject to the condition on durations, then we should be able to pick up a duration, pack up for processing subsets of activities according to the condition, find all such induced states and claim that this is all we need to look for. More formally, to construct the immediate successor states of a given, list all durations of precedence feasible activities. For each such duration, find all subsets of resource compatible activities such that a subset* (1) *contains at least one activity of the chosen duration,* (2) *has all its activities of durations at most the chosen one, and* (3) *is maximal with respect to the subset of all activities having at most the chosen duration and also with respect to the residual resources due to the already active activities. Corresponding to each such subset, find the state induced by processing it and include it in the state network. This is not the same as including the states generated by processing all maximal subsets, as in the DH procedure, because we allow here the inclusion of states induced by processing certain subsets that, due to the restriction on durations, may not be maximal. This attempt at generalizing the corollary may seem reasonable and in line with the theorem and the corollary but actually goes further than the theorem allows. To see this, consider the RCPSP instance in Figure 3-5. Notice that there are only two durations for the activities:* $2$ *and* $1$*. To construct the states to succeed the start node of the state network, the only maximal subset with respect to duration* $2$ *is* $\{a_1,\ a_2,\ a_3\}$ *which happens to be maximal also. Similarly, the only maximal subset with respect to duration* $1$ *is* $\{a_2\}$*. The best schedule initiated with state* $\{a_1, a_2, a_3\}$ *is:* $a_1$, $a_2$ *and* $a_3$ *to start at time* $0$, $a_4$ *to start at time* $2$ *and* $a_5$ *at time* $3$ *for a makespan of* $5$*. One of the best schedules that start with the second state is:* $a_2$ *to start at time* $0$, $a_1$ *and* $a_4$ *to start at time* $1$, $a_3$ *at time* $2$ *and* $a_5$ *at time* $3$*; also for a*

$a_i(d_i, r_i)$
$r_i$ : activity $a_i$ resource requirement
$d_i$ : duration of activity $a_i$
R : per period availability of the resource

R = 4

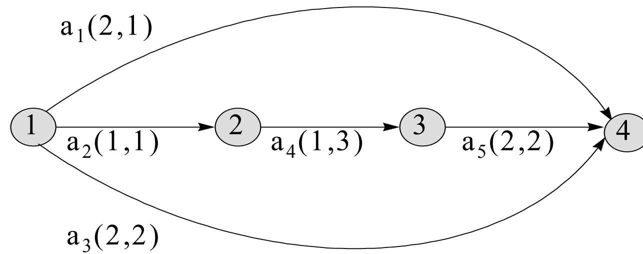


*Figure 3-5: Counterexample to generalizing Corollary 10.*

*makespan of* 5. *Now it is easy to see that the optimal schedule here is:* $a_1$ *and* $a_2$ *to start at time* 0, $a_4$ *at time* 1, *and* $a_3$ *and* $a_5$ *at time* 2 *for an optimal makespan of* 4. *The set* $\{a_1, a_2\}$ *is of course not maximal in any sense. In fact, following Theorem 8, it can simply be constructed by choosing the subset* $\{a_1\}$ *and adding a compatible activity to it,* $a_2$, *of duration at most the duration of* $a_1$. *Note that before adding* $a_2$ *it is possible to add* $a_3$, *since it has the same duration as* $a_1$. *This would have led to the first subset we considered and which produced the suboptimal solution.*

The conclusion we draw from this example is that seemingly arbitrary states which Theorem 8 helps identify can be on an optimal path and that it is hard to further reduce the number of states that have to be considered, at least as far as states immediately succeeding the start node are concerned. Actually, those states which the theorem identifies are induced by processing subsets that are structurally so different from other types of subsets that we distinguish them with a special definition.

**Definition 12 (Submaximal Subset)** *Let $S$ be a state of the state network of an RCPSP instance. Let $A$ be the subset of already active activities of $S$ and $d$ be the minimum residual duration among the activities of $A$. Let $P$ be the set of precedence feasible activities of $S$ and $d_i$ be the duration of an activity of $P$ such that $d_i \leq d$. Partition $P$ into subsets $P_1$ and $P_2$, where $P_1$ consists of the activities of $P$ of durations at most $d_i$ and $P_2$ consists of the activities of duration strictly larger than $d_i$ (i.e., $P_2 = P - P_1$). A subset $N \subset P$ is said to be $d_i$-**submaximal** if it can be written as $N = N_1 \cup N_2$, where $N_1 \subset P_1$ and $N_2 \subset P_2$ with $N_1$ being maximal with respect to $P_1$ and the residual resources of $A$ and $N_2$.*

To emphasize, determining whether or not, for some $d_i$, a given subset of $N \subset P$ is $d_i$-submaximal entails first obtaining $d$ in the definition and verifying that $d_i \leq d$. Second, it necessitates determining the subsets $P_1$, $P_2$, $N_1 = N \cap P_1$ and $N_2 = N \cap P_2$. Third, it requires confirming, when the resource requirements of all activities in $A$, $N_1$ and $N_2$ are considered, that no activity of $P_1$ can be added to $N_1$ without violating a resource limit. The following diagram and example should help further clarify the definition.

**Example 13** *The subsets considered in the previous example, that is $\{a_1, a_3, a_2\}$, $\{a_1, a_2\}$ and $\{a_2\}$ as well as $\{a_3, a_2\}$ are all $1$-submaximals. To see this, note that for $d = 1$, $P_1 = \{a_2\}$ and $P_2 = \{a_1, a_3\}$. Therefore, $N_2$ can only be one of the four possibilities: $\emptyset, \{a_1\}, \{a_3\}, \{a_1, a_3\}$. But for each of those possibilities $N_1$ can only be $\{a_2\}$. Whence, the four $1$-submaximals are the only ones possible. Additionally, the first of those, that is $\{a_1, a_3, a_2\}$, is also $2$-submaximal. This follows from the fact that for $d = 2$, $P_1 = \{a_1, a_2, a_3\}$ and $P_2 = \emptyset$ and from the fact that the three activities are resource compatible. Since these four subsets are all the 'submaximals' that can be generated at time $0$, we conclude, following Theorem 8, that processing them induces*

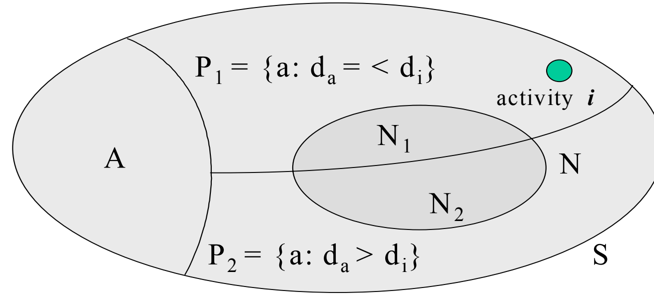


Figure 3-6: Graphic depiction of a $d_i$-submaximal subset.

*all the states that need to be included in the state network of the instance of that example and which are immediate successors to the start node.*

**Remark 4** *Note that if $A$ in the definition is not empty and $d \geq \max\{d_i : i \in P\}$ then a d-submaximal subset is in fact maximal. In case $A = \emptyset$, we propose as a convention that $d$ be considered infinite to allow the definition in this instance to be substantive. Following this convention, if $A = \emptyset$ and $d_j = \max\{d_i : i \in P\}$ then a $d_j$-submaximal subset is also maximal. On the other hand, if $P = \emptyset$ then the only $d_i$-submaximal of $P$, for any $d_i \leq d$, is the empty set itself.*

Note that this definition of submaximals prompts a reformulation of our previous theorem:

**Restatement of Theorem 8** In a shortest path approach to RCPSP, let $S$ be a state in the state network. Let $A$ be the set of *active* activities of $S$ and $d$ be the minimum residual duration of the activities of $A$. Let $P$ be the set of precedence feasible activities of $S$ and

let $d_i$, $i = 1, ..., p$, be the durations of activities of $P$ not exceeding $d$. Then the states that immediately succeed $S$ need only be the ones induced by processing the $d_i$-submaximals of $P$, $i = 1, ..., p$.

Further, by relying on the following lemma, this reformulation can be simplified to produce a corollary to Theorem 8 that represents a general case counterpart to Corollary 10.

**Lemma 14** *Let $S$, $A$ and $P$ be as in Definition 12. Let $d_1$ and $d_2$ be two durations of activities of $P$ such that $d_1 \leq d_2$ and $N$ be a $d_2$-submaximal subset of $P$. Then $N$ is $d_1$-submaximal as well.*

**Proof.** Write $P = P_1^2 \cup P_2^2$, where $P_1^2$ contains all the activities of $P$ of duration at most $d_2$ and $P_2^2 = P - P_1^2$. Then $N = N_1^2 \cup N_2^2$, where $N_1^2 \subset P_1^2$ and $N_2^2 \subset P_2^2$. Since $N_1^2$ is maximal with respect to $P_1^2$ and the residual resources of $A$ and $N_2^2$, the subset $N_1^1 = \{a \in N_1^2 :$ duration of $a$ is at most $d_1\}$ must be maximal with respect to $P_1^1 = \{a \in P_1^2 :$ duration of $a$ is at most $d_1\}$ and the residual resources of $A$ and $N_2^2 \cup (N_1^2 - N_1^1)$. Since $P_1^1 = \{a \in P :$ duration of $a$ is at most $d_1\}$ , $P = P_1^1 \cup (P - P_1^1)$, $N = N_1^1 \cup (N - N_1^1)$ and $N_1^1 \subset P_1^1$, we conclude that $N$ is $d_1$-submaximal. ■

**Remark 5** *Clearly, every maximal subset of $P$ is also $d_i$-submaximal for any $d_i$. Hence, if $d_1 < d_2 < ... < d_p$ are the durations of the activities in $P$, then we could say that the set of maximals is contained within the set of $d_p$-submaximals which itself is contained within the set of $d_{p-1}$-submaximals ... which is contained within the set of $d_1$-submaximals. Now, it is possible to state conditions which collapse this nested sequence or even only parts of it. We do not find this of practical value at this point, though, and so we choose not to pursue it. Along these*

*lines, we are satisfied to only point out that it may seem that when the activities of duration at most $d_i$ are not resource compatible then a $d_i$-submaximal is also maximal. This actually may not be the case as there could be activities with duration greater than $d_i$ that could very well be resource compatible with a $d_i$-submaximal.*

**Corollary 15** *In a shortest path approach to the arbitrary durations RCPSP, let $S$ be a state of the state network. Let $A$ be the set of active activities of $S$ and $d'$ be its minimum residual duration. Let $P$ be the set of precedence feasible activities in $S$ and, similarly, let $d''$ be the minimum duration of its activities. Then the states of the state network that immediately succeed $S$ need only be the ones induced by processing the $d$-submaximals of $P$, where $d = \min(d', d'')$.*

**Remark 6** *Note first that the $d$ in the previous corollary is different from the $d$ in Definition 12. Second, in case $P = \emptyset$, the only $d$-submaximal of $P$ is the empty set and so the only state that immediately succeeds such a state is the one expressing the continuation of the already started activities, if there is any. If, furthermore, no continuing activity exists then naturally the only immediate successor state in this case is the end state itself.*

Obviously, this general case corollary would allow a lot more states in the state network than would Corollary 10 in the special case. But in view of Example 11 and the discussion therein, there isn't much room left for further reduction in the number of such states in a way that is in line with Theorem 8. As a matter of fact, this corollary is a significant improvement over the reformulation of that theorem as it determines that only one $d_i$ needs to be used as far as obtaining the $d_i$-submaximals. To get an idea of the practical significance of this corollary, we present in Figure 3-7 the state network of the project instance in Figure 3-1 as reduced by it. Note that its number of states is 13 whereas the original state network in Figure 3-2 has

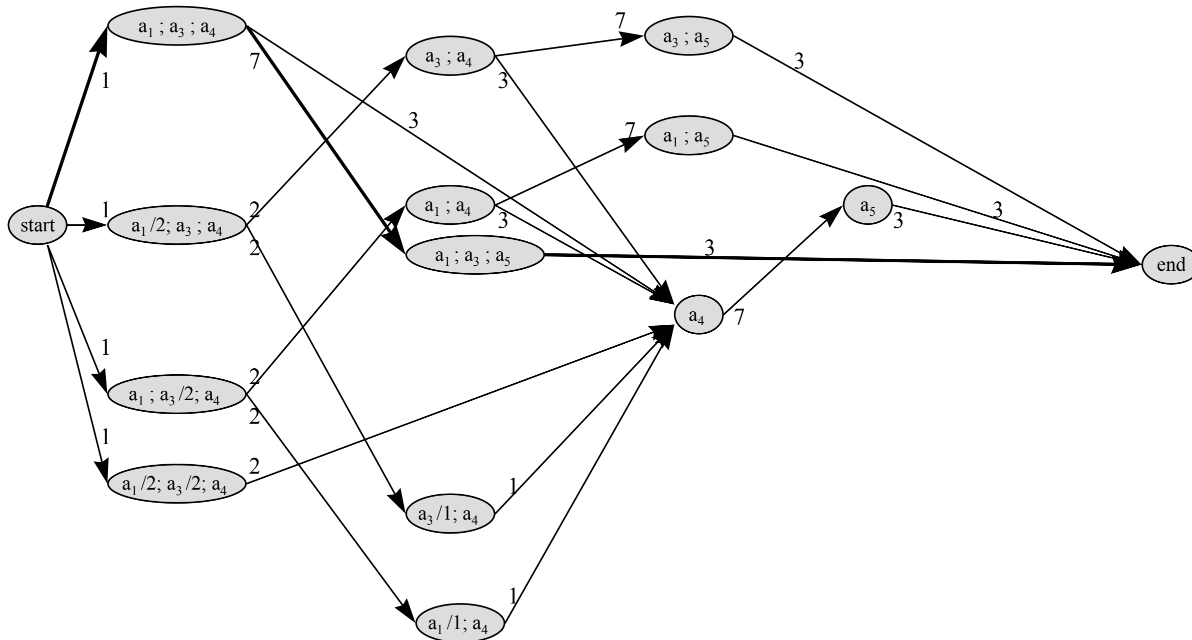


Figure 3-7: The state network of the project instance in Figure 3-1 as reduced by Corollary 15.

22. As a consequence of this reduction in the number of nodes, the number of arcs has also decreased; which in itself is important too.

Now, before we proceed any further with the arbitrary durations case, we need to revisit the single duration case for some motivational insight. Recall the concepts of semi-active and active schedules defined in Chapter 1. Their importance is derived from that they constitute classes of schedules for the problem that are significantly smaller than The class of all schedules, yet contain at least one optimal solution. Whence a search of those classes is presumably more likely to yield an optimal solution faster than a search of the larger class. It is worth reiterating here that the class of semi-active schedules is the larger of the two. It is also worth recalling

that every procedure we have reviewed in Chapter 1 purports to limit the search to one of those classes.

**Proposition 16** *In the single duration case, any path between the start and end nodes of the state network as reduced by use of Corollary 10, i.e., with nodes induced by processing only maximal subsets, represents an active schedule.*

**Proof.** Suppose not. That is suppose there exists a path between the end nodes that does not represent an active schedule. Then an activity of the schedule represented by this path can be started at an earlier time than in the schedule while maintaining the starting times of the rest of the activities. In other words, the activity can be started at a node of the state network that precedes the one the activity actually starts at. But the schedule was derived from a path in the state network as reduced by Corollary 10. That is, at the node the activity can supposedly start at, only maximal subsets are processed. But, by definition now, activities cannot be added to maximal subsets. This, clearly, is a contradiction. Hence, every path between the end nodes must represent an active schedule. ■

**Remark 7** *Recall that the exact procedures we previously reviewed implemented a Local and/or Global Left Shift at additional computing costs. Besides allowing for a reduction in the size of the state network for the single duration case, Corollary 10 via this proposition eliminates all non active schedules from consideration at no extra costs. It may be argued here that the corollary may charge the cost indirectly as an increase in the cost of generating the successor nodes. We will discuss this issue in a later section; but for the moment, we maintain that this is not the case.*

Back to the general case of arbitrary activity durations. One may wonder whether the same

claim as in the proposition can be made? In other words, do all paths of the state network as reduced by Corollary 15 represent active schedules? It turns out that the answer to this question is negative. To see this, consider the paths of the reduced state network in Figure 3-7. There, the subpath: $start - a_1/2, a_3, a_4 - a_3, a_4$ , no matter how completed, would represent a non semi-active schedule not only a non-active one. The reason, of course, is that $a_3$ could have been started at the first state and completed by the third. This motivates the extensions of Theorem 8 and Corollary 15 in parts $(i)$ and $(ii)$ of the following theorem. It also motivates the third part of the theorem which we present as a general case counterpart to the proposition.

**Theorem 17** *Let $\{S_i\}_{i=1}^m$ be a sequence of states in the state network of an arbitrary durations RCPSP instance where each state $S_{i+1}$ is induced from its predecessor $S_i$, $1 < i < m$, by processing a $d_i$-submaximal subset as in Corollary 15. Then in a shortest path approach to the problem in which the states of the state network are generated on an as needed basis, the following results hold:*

$(i)$ *The sequence $\{S_i\}_{i=1}^m$ can be fathomed (in the sense that its continuations need not be further explored) if state $S_m$ includes any activity which* (1) *is precedence feasible to start at states $S_i$, $i = 1, ..., m-1$, but isn't started* (2) *has its resource requirements at those states unused, and* (3) *has duration $d$ satisfying: $d \leq d_1 + \cdots + d_m$.*

$(ii)$ *The sequence $\{S_i\}_{i=1}^m$ can be fathomed, following the same sense adopted in part* $(i)$, *if state $S_m$ includes any* ***active*** *activity which* (1) *is precedence feasible at state $S_1$ but isn't started, and* (2) *has its resource requirements in the execution of the sequence unused prior to becoming active.*

$(iii)$ *Any start to end path of the state network as reduced in parts* $(i)$ *and* $(ii)$ *with respect to*

*all possible sequences represents an active schedule.*

**Proof.**

($i$) The proof of part ($i$) is based on the same idea as the proof of Theorem 8. Suppose $S_m$ contains an activity, $a$, satisfying the three conditions and let $l$ be the length of the shortest path between the end nodes of the network passing through $\{S_i\}_{i=1}^m$. To show that $\{S_i\}_{i=1}^m$ can be fathomed, we devise a path between the end nodes of the network of length at most $l$ and which bypasses $S_m$. This renders exploring successor states to $S_m$ via $\{S_i\}_{i=1}^m$ useless for our purpose; effectively fathoming the sequence.

To this end, subdivide the shortest path through $\{S_i\}_{i=1}^m$ into four parts. The first part starts at the start node and leads into the state immediately preceding $S_1$on the path, including the arc into $S_1$. Let $l_1$ denote its length and assume without loss of generality that $d > d_1$. Otherwise, if $d \leq d_1$ then Theorem 8 applies and we need continue no further. With this assumption, let the second part of the path start at $S_1$ and include $S_2, ..., S_{m'-1}$, where $m'$ is such that $d_1 + \cdots + d_{m'-1} < d \leq d_1 + \cdots + d_{m'}$. Clearly, $m' > 1$ due to the assumption. This second part of the path, of length $d$, ends with an arrow leading into a state $\bar{S}_{m'}$ which may not be part of the original states of the network but which we may need to introduce at time $l_1 + d$ for a reason that will soon be apparent. The third part consists of state $\bar{S}_{m'}$ with an arrow leading into state $S_{m'}$ along the path. It is of length $d_1 + \cdots + d_{m'} - d$. The fourth and last part is the rest of the path of length, say, $l_2$. Clearly,

$$l = l_1 + d + (d_1 + \cdots + d_{m'} - d) + l_2 = l_1 + d_1 + \cdots + d_{m'} + l_2.$$

Now, suppose activity $a$ is started following state $S_1$ and let $S_2'$ be the state similar to $S_2$ except for the addition of activity $a$ among its active activities. This is possible because by assumption $a$ is precedence feasible to start at $S_1$ and its resource requirements are available. In the same way, construct states $S_i'$ from states $S_i$, $i = 3, ..., m' - 1$.

Devise a path between the start and end nodes of the state network consisting of three parts as follows. Let the first part of the path be the same as for the first part of the path through $\{S_i\}_{i=1}^{m}$ which leads into $S_1$ and has length $l_1$. Obviously, $S_m$ does not belong to this part for otherwise it would precede $S_1$. Let the second part of the devised path consist of the states $S_1, S_2', ..., S_{m'-1}'$ and an arrow extending out of $S_{m'-1}'$ and into $\bar{S}_{m'}$. Its length is $d$. Note that $S_m$ is not in this second part of the path either because the only state in that part that has $a$ as precedence feasible is $S_1$. But that state contains at least one activity completed in the processing of the $d_1$-submaximal inducing $S_2$ which, hence, cannot belong to $S_m$. Finally, let the third part of our path be a shortest path continuation between state $\bar{S}_{m'}$ and the end node. Since this third part is the minimum makespan of the project consisting of the activities in the third and fourth part of the path through $\{S_i\}_{i=1}^{m}$ except for $a$, we conclude by Lemma 7 that its length is at most $(d_1 + \cdots + d_{m'} - d) + l_2$. Note that $S_m$ does not belong to this part because none of those states contains $a$. Thus our devised path bypasses $S_m$. Moreover, its total length is at most $l_1 + d + (d_1 + \cdots + d_{m'} - d) + l_2 = l$. It follows that $S_m$ may be discarded as far as finding the shortest start to end path in the state network through the sequence $\{S_i\}_{i=1}^{m-1}$ is concerned. In other words, $\{S_i\}_{i=1}^{m}$ can be fathomed.

*(ii)* The proof that $\{S_i\}_{i=1}^{m}$ can be fathomed in this case too uses the same idea as the proof in

part $(i)$. Essentially, it compares the lengths of two start to end paths. The first of those paths is the shortest one through the sequence $\{S_i\}_{i=1}^m$ while the second is the shortest through a modified sequence $\{S_i'\}_{i=1}^{m'}$ in which $a$ is started at $S_1'$. Since the details are similar to those in the proof of the previous part, we shall omit them for the sake of avoiding repetition.

$(iii)$ To show part $(iii)$ we note that, while building a schedule, the possibility to locally or globally left shift an activity, and thus fathoming the corresponding partial schedule, may be detected before the activity starts, while it is in process, or after its completion. The first of these three possibilities is totally covered in part $(i)$. That is, without sacrificing optimality, when considering the continuation of a sequence we may not left shift an activity before it starts unless the three conditions in that part of the theorem are satisfied. In fact, if either of the first two conditions is not satisfied then left shifting is not even feasible. Similarly, the second possibility is totally accounted for in part $(ii)$. If in a given sequence, the two conditions are not satisfied then left shifting in that sequence is infeasible. Concerning the third possibility, on the other hand, note that a left shifting possibility should be detectable as soon as an activity completes. That is, whatever the conditions that allow for a left shift to be feasible some time units after an activity completes, they should equally be applicable the moment after the activity completes. In fact, we can make and prove the following claim:

The sequence $\{S_i\}_{i=1}^m$ can be fathomed if an activity completed in the processing of the $d_{m-1}$-submaximal subset inducing $S_m$ satisfies the two conditions in part $(ii)$.

But note that this claim follows from part $(i)$, using $\{S_i\}_{i=1}^{m-1}$ instead of $\{S_i\}_{i=1}^m$, when

the duration of the activity is $d_{m-1}$. It also follows from part $(ii)$, also using $\{S_i\}_{i=1}^{m-1}$ instead of $\{S_i\}_{i=1}^{m}$, when the activity duration is strictly larger than $d_{m-1}$. Hence this third possibility for detecting possible left shifts is implicitly considered in parts $(i)$ and $(ii)$. It follows that implementing parts $(i)$ and $(ii)$ prohibits left shift possibilities from occurring. Hence a state network as reduced by the application of those two propositions can not contain a path that yields a non-active schedule. That is the schedules yielded by such paths are necessarily active. ■

**Remark 8** *The number of sequences of states of an RCPSP instance may be quite large. In this case, the question arises as to whether every sequence needs to be tested to determine the presence of some activity in one of its states which may lead to the fathoming of the sequence. If the answer to this concern is in the positive then we submit that Theorem 17 may practically be useless, even for small $m$. Fortunately, the situation is not nearly so hopeless. In applying a shortest path algorithm, such as Dijkstra's Algorithm for example, the node we need to find successors for is one that has already been labeled. Hence, we know exactly a shortest path from the start node that leads into it. This is the sort of sequences that should concern us as far as the theorem goes. The rest of sequences can at best lead to other optimal solutions and so need not be investigated any further. For this reason, the theorem effectively fathoms states not just sequences; as to each sequence we care to fathom corresponds a unique state, namely its end. And vice versa: to each state corresponds the unique sequence determined by the shortest path, from the start node and to the state, produced by an algorithm such as Dijkstra's. We remark that an arbitrary start to end path of the state network in which the theorem may fathoms nodes, instead of all the possible sequences, can actually yield a non-active schedule. That, however,*

*will not be the case for the optimal start to end path, again obtained in an algorithm such as Dijkstra's, which after all is what we are after. For this reason, and from now on, when we specify the state network as reduced by Theorem 17, we really mean the state network as reduced by the partial application of Theorem 17 to 'fathom states' of that network; as that partial implementation is sufficient for our purposes.*

Note that in Example 11 we had to explore all four 1-submaximals to construct the successors of the start node and reach an optimal schedule to the project instance. Theorem 17 provides a means for fathoming some such states in constructing a reduced state network. To get an idea of what this means in practical terms, consider in Figure 3-8 the state network of the project instance in Figure 3-1 as reduced by Corollary 15 and further reduced by the theorem. As can be observed, the number of nodes has been reduced from 13 to 7; two of which have all their descendents fathomed effectively reducing the number of nodes to only 5. Needless to say, the number of arcs has significantly decreased also.

In view of the potential savings in the number of nodes of the state network, most of the rest of this chapter is devoted to investigating solving RCPSP by exploring shortest path solutions of the state network as reduced by Corollaries 10, 15 and Theorem 17. Henceforth, the terminology we use to refer such a state network is simply the **reduced state network**.

To conclude this section, we point out that it may be argued that Corollaries 10, 15 and parts (*i*) and (*ii*) of Theorem 17 are all an implicit way of implementing Local and Global Left Shift Rules to come up with active schedules[6]. Although that was not our intention in developing these results, the argument may certainly be made. We point out in that case,

[6] As a matter of fact, it is easy to see that any active schedule corrsponds to a start to end path of the reduced state network.

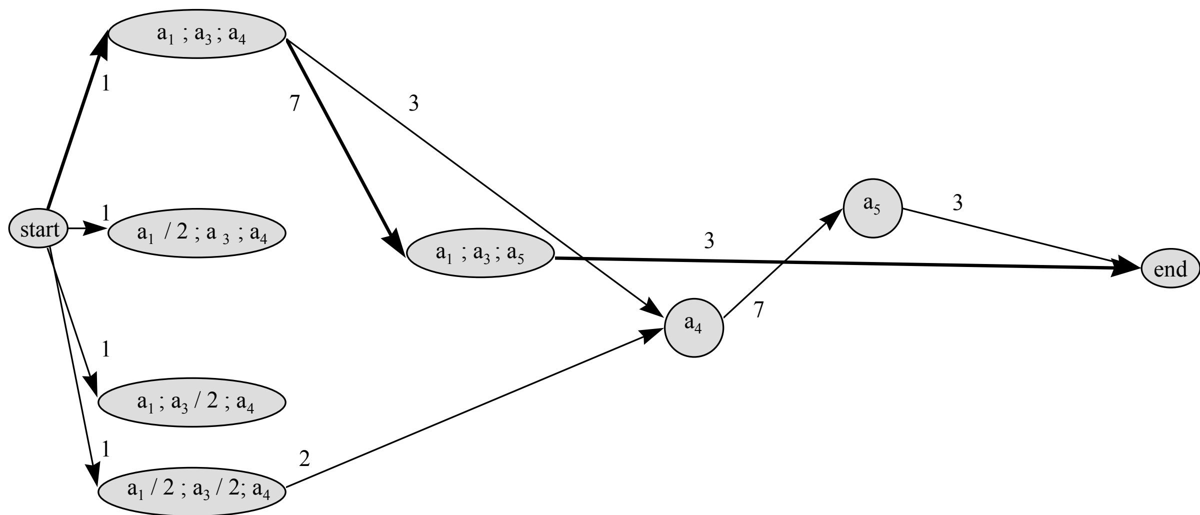


Figure 3-8: The state network of the project instance in Figure 3-1 as reduced by Corollary 15 and Theorem 17.

though, that this implementation of the rules is a relatively efficient one, computationally, as Corollaries 10, 15 have the potential to eliminate from consideration a significant proportion of states without even requiring that they be generated first. We will elaborate on this issue in following sections.

## 3.5 Computational Complexity Implications

The computational efficiency of a shortest path approach to solving RCPSP depends to a large extent on the size of the state network that needs to be considered. It is natural then, now that a reduced state network is the focus of our attention, to reflect on the size of this network. For sure, a precise description of the size of a reduced state network should factor in, in addition to considerations of precedence relationships and resource consumptions, the precise effects of using the reduction techniques in Corollary 10, for the single duration case, or Corollary 15 coupled with Theorem 17 in the general case. We frankly find such a precise description of the network size beyond our reach. But, this does not prevent us from qualitatively exploring this network size by focusing on only one of its determining factors and, even with that, through simplifying assumptions. Thereby, drawing possible inferences about the case where all factors are considered and the assumptions no longer hold true. Needless to say, the general validity of such inferences would ultimately have to be statistically tested in computational experiments.

To this end, we propose studying the number of possible $d$-submaximals that a state could have. Granted, this number heavily depends on the configuration of resource consumptions of the different activities of the state. So we propose to deduce this number subject to the simplifying assumption that any fixed number of the activities of the state can be started

simultaneously. Moreover, constructing the $d$-submaximals of the instances in Figures 3-1 and 3-5, we suspect that this number of $d$-submaximals is influenced by the number of activities of the state having the minimum original or residual duration. So we intend that this be factored in our study too.

With these notions in the background, consider a state of a project instance with $n$ activities, $a_1,\ a_2,\ ...,\ a_n$, of respective durations, or residual durations, $d_1,\ d_2,\ ...,\ d_n$. Assume without loss of generality that $d_1 \leq d_2 \leq ... \leq d_n$. Then, for the purpose of computing the number of possible $d_1$-submaximals of the state, subject to our simplifying assumption, we distinguish the following cases and subcases.

**Case 1**: All $n$ activities can be processed simultaneously.

**Subcase 11**: $d_1 < d_2 \leq ... \leq d_n$.

In this subcase, every $d_1$-submaximal has to contain activity $a_1$. To obtain such $d_1$-submaximals, we form subsets of $\{a_2,\ ...,\ a_n\}$ and add $a_1$to them. Hence the number of $d_1$-submaximals in this subcase is $2^{n-1}$.

**Subcase 12**: $d_1 = d_2 < d_3 \leq ... \leq d_n$.

Now, every $d_1$-submaximal has to contain activities $a_1$ and $a_2$. To obtain those $d_1$-submaximals, we form subsets of $\{a_3,\ ...,\ a_n\}$ and add $a_1$ and $a_2$ to them. It follows that the number of $d_1$-submaximals in this subcase is $2^{n-2}$.

$\vdots$

**Subcase 1i**: $d_1 = d_2 = \cdots = d_i < d_{i+1} \leq ... \leq d_n$.

Following the same logic as before, we obtain $2^{n-i}$ $d_1$-submaximals.

$\vdots$

**Subcase 1n**: $d_1 = d_2 = \cdots = d_n$.

Following the same logic as before, we obtain $2^{n-n} = 1$ $d_1$-submaximal; which in this subcase is actually a maximal subset. This, of course, is the set $\{a_1, ..., a_n\}$ itself.

**Case 2**: Only $n-1$ activities can be processed simultaneously..

**Subcase 21**: $d_1 < d_2 \leq ... \leq d_n$.

In this subcase, we have:

$\binom{n}{n-1} = n$ $d_1$-submaximals with $n-1$ activities.

Thereafter, $a_1$ belongs to every single remaining $d_1$-submaximal. Therefore, we have:

$\binom{n-1}{n-3}$ $d_1$-submaximals with $n-2$ activities,

$\binom{n-1}{n-4}$ $d_1$-submaximals with $n-3$ activities,

$\vdots$

$\binom{n-1}{1}$ $d_1$-submaximals with 2 activities,

1 $d_1$-submaximal with 1 activity, which is $\{a_1\}$ itself,

for a total of $n + \left[\binom{n-1}{n-3} + \binom{n-1}{n-4} + \cdots + \binom{n-1}{1} + 1\right]$ $d_1$-submaximals.

**Subcase 22**: $d_1 = d_2 < d_3 \leq ... \leq d_n$.

In this subcase, we have:

$\binom{n}{n-1} = n$ $d_1$-submaximals with $n-1$ activities.

Thereafter, $a_1$ and $a_2$ belong to every single remaining $d_1$-submaximal. Therefore, we have:

$\binom{n-2}{n-4}$ $d_1$-submaximals with $n-2$ activities,

$\binom{n-2}{n-5}$ $d_1$-submaximals with $n-3$ activities,

$\vdots$

$\binom{n-2}{1}$ $d_1$-submaximals with 3 activities,

1 $d_1$-submaximals with 2 activities,

for a total of $n + \left[\binom{n-2}{n-4} + \binom{n-2}{n-5} + \cdots + \binom{n-2}{1} + 1\right]$ $d_1$-submaximals.

$\vdots$

**Subcase 2i**: $d_1 = d_2 = \cdots = d_i < d_{i+1} \leq ... \leq d_n$.

In this subcase, we have:

$\binom{n}{n-1} = n$ $d_1$-submaximals with $n-1$ activities.

Thereafter, $a_1, a_2, ..., a_i$ belong to every single remaining $d_1$-submaximal. Therefore, we have:

$\binom{n-i}{n-i-2}$ $d_1$-submaximals with $n-2$ activities,

$\binom{n-i}{n-i-3}$ $d_1$-submaximals with $n-3$ activities,

$\vdots$

$\binom{n-i}{1}$ $d_1$-submaximals with $i+1$ activities,

1 $d_1$-submaximals with $i$ activities,

for a total of $n + \left[\binom{n-i}{n-i-2} + \binom{n-i}{n-i-3} + \cdots + \binom{n-i}{1} + 1\right]$ $d_1$-submaximals.

$\vdots$

**Subcase 2(n − 1)**: $d_1 = d_2 = \cdots = d_{n-1} < d_n$.

In this subcase, we have:

$\binom{n}{n-1} = n$ $d_1$-submaximals with $n-1$ activities and 0 $d_1$-submaximal with $n-2$ activities, $n-3$ activities ...

for a total of $n$ $d_1$-submaximals.

**Subcase 2n**: $d_1 = d_2 = \cdots = d_n$.

In this subcase, we have:

$\binom{n}{n-1} = n$ $d_1$-submaximals with $n-1$ activities and 0 $d_1$-submaximal with $n-1$ activities, $n-2$ activities ...

for a total of $n$ $d_1$-submaximals.

$\vdots$

**Case i**: Only $n-i+1$ activities can be processed simultaneously..

**Subcase i1**: $d_1 < d_2 \leq ... \leq d_n$.

In this subcase, we have:

$\binom{n}{n-i+1}$ $d_1$-submaximals with $n-i+1$ activities.

Thereafter, $a_1$ belongs to every single remaining $d_1$-submaximal. Therefore, we have:

$\binom{n-1}{n-i-1}$ $d_1$-submaximals with $n-i$ activities,

$\binom{n-1}{n-i-2}$ $d_1$-submaximals with $n-i-1$ activities,

$\vdots$

$\binom{n-1}{1}$ $d_1$-submaximals with 2 activities,

1 $d_1$-submaximals with 1 activity,

for a total of $\binom{n}{n-i+1} + \left[\binom{n-1}{n-i-1} + \binom{n-1}{n-i-2} + \cdots + \binom{n-1}{1} + 1\right]$ $d_1$-submaximals.

**Subcase i2**: $d_1 = d_2 < d_3 \leq ... \leq d_n$.

In this subcase, we have:

$\binom{n}{n-i+1}$ $d_1$-submaximals with $n-i+1$ activities.

Thereafter, $a_1$ and $a_2$ belong to every single remaining $d_1$-submaximal. Therefore, we have:

$\binom{n-2}{n-i-2}$ $d_1$-submaximals with $n-i$ activities,

$\binom{n-2}{n-i-3}$ $d_1$-submaximals with $n-i-1$ activities,

$\vdots$

$\binom{n-2}{1}$ $d_1$-submaximals with 3 activities,

1 $d_1$-submaximal with 2 activities,

for a total of $\binom{n}{n-i+1} + \left[\binom{n-2}{n-i-2} + \binom{n-2}{n-i-3} + \cdots + \binom{n-2}{1} + 1\right]$ $d_1$-submaximals.

$\vdots$

**Subcase ik**: $d_1 = d_2 = \cdots = d_k < d_{k+1} \leq ... \leq d_{n-i+1} \leq ... \leq d_n$.

In this subcase, we have:

$\binom{n}{n-i+1}$ $d_1$-submaximals with $n-i+1$ activities.

Thereafter, $a_1$, $a_2, ...,$ $a_k$ belong to every single remaining $d_1$-submaximal. Therefore, we have:

$\binom{n-k}{n-i-k}$ $d_1$-submaximals with $n-i$ activities,

$\binom{n-k}{n-i-k-1}$ $d_1$-submaximals with $n-i-1$ activities,

$\vdots$

$\binom{n-k}{1}$ $d_1$-submaximals with $k+1$ activities,

1 $d_1$-submaximal with $k$ activities,

0 $d_1$-submaximal with less than $k$ activities,

for a total of $\binom{n}{n-i+1}+\left[\binom{n-k}{n-i-k}+\binom{n-k}{n-i-k-1}+\cdots+\binom{n-k}{1}+1\right]$ $d_1$-submaximals.

⋮

**Subcase ik′**: $d_1 = d_2 = \cdots = d_{n-i+1} = \cdots = d_{k'} < d_{k'+1} \leq ... \leq d_n$.

In this subcase, we have:

$\binom{n}{n-i+1}$ $d_1$-submaximals with $n-i+1$ activities and 0 $d_1$-submaximal with $n-i$ activities, $n-i-1$ activities, ...

for a total of $\binom{n}{n-i+1}$ $d_1$-submaximals.

⋮

**Subcase in**: $d_1 = d_2 = \cdots = d_n$.

In this subcase, we have:

$\binom{n}{n-i+1}$ $d_1$-submaximals with $n-i+1$ activities and 0 $d_1$-submaximal with $n-i$ activities, $n-i-1$ activities, ...

for a total of $\binom{n}{n-i+1}$ $d_1$-submaximals also.

⋮

**Case n**: Only 1 activity can be processed at a time.

Here, there is no need to consider subcases as there are $n$ $d_1$-submaximals each consisting of a single activity irrespective of the durations $d_1,\ d_2, ...,\ d_n$.

Note that for each case, the number of $d_1$-submaximals is monotonically decreasing in the number of activities having the minimum duration and/or residual duration. As a matter of

fact, this number of $d_1$-submaximals strictly decreases, up to a lower limit, as the number of activities with $d_1$ duration, or residual duration, increases. The decrease stops when the number of activities with minimum duration, or residual duration, equals the number of activities that can be simultaneously processed. Obviously, the number of $d_1$-submaximals generated in a shortest path approach to the problem must have an effect on the difficulty of solving the problem; eventhough, many of the corresponding states may be fathomable in some way. Combining those two arguments and extrapolating to the case where our simplifying assumption no longer holds, we suspect that the larger the proportion of activities of an instance that have the minimum duration is, the easier it is to solve the instance computationally; at least via a shortest path approach. We also suspect that this relationship between the proportion of activities with minimum duration and computational difficulty is affected by the tightness of the resource limits; as comparing the number of submaximals with fixed number of activities across the different cases suggests. The following conjecture comprises our first suspicion in a more formal way.

**Conjecture 18** *Consider an arbitrary RCPSP instance of $n$ activities. Let the durations of the activities be drawn from a discrete uniform distribution and let $n_1$ be the number of those activities having the minimum duration. Then the expected computational time required to solve the instance decreases as the ratio $n_1/n$ increases.*

Note that $0 < n_1/n \leq 1$. When $n_1/n = 1$, we have the single duration case. Here, states are induced by maximals rather than submaximals and so are significantly less in number. This should lead to a far more compact state network than otherwise; which, additionally, is in no small part due to the absence of residual durations. Consequently, the computational time to

solve this case should be smaller than the computational time required to solve any other case. The conjecture, in a sense, puts this simple observation in a larger framework.

The conjecture doesn't explicitly reflect the fact that the number of $d$-submaximals decreases only up to a limit as a function of the ratio because that conclusion only holds for the number of $d$-submaximals that could be generated from a single state. We are not sure whether possibly different such limits for the different states of the network combine to produce a noticeable effect on the computational time. In any case, those limits are more the result of the resource consumptions and precedence relationships which, again, we find hard to combine with our ratio.

**Remark 9** *By requiring the durations to be drawn from a uniform distribution, these durations are the most arbitrary they can be, subject only to the lower and upper bounds of the distribution. This requirement should not be interpreted as a constraint on the durations but rather the means to enforce the absence of constraints.*

The conclusion of this conjecture is framed in terms of an expectation precisely because of the fact that a relationship that determines the computational time in terms of the ratio should also contain other complicating factors, such as precedences and resource consumptions, which we cannot cope simultaneously with in a precise way. Actually, there could well be instances that may be easier to solve computationally when the ratio is lower rather than higher. We don't know how to predict all those instances but nevertheless we suspect those cases are either relatively so uncommon that the conjecture still holds in expectation or else that those instances can be decomposed in some way where each part dealt with separately so that overall the conjecture remains valid. Those two eventualities, *i.e.,* the relatively uncommon cases and

the possibility of some sort of decomposition, merit some further elaboration.

First, suppose we consider an instance of RCPSP where all but one of the $n$ activities have the same duration and the remaining one has a duration that happens to be smaller. Clearly, if the conjecture were not framed in terms of expectation then we would have had to conclude that this configuration of durations results in one of the most difficult cases computationally. But intuitively, that case should be a little more difficult computationally than the single duration case. In fact, if the conjecture were not framed in terms of an expectation we would have had a counter example. Obviously now, we don't. We believe that such cases are relatively uncommon enough that the conjecture still holds.

Second, regarding the possibility of some sort of decomposition, consider an RCPSP instance with an AoN diagram that, with the help of cuts, can be clearly subdivided into relatively few regions each of which having a single activity duration. Again, intuitively, this case should be a lot easier computationally than the case where the durations are the same but positioned randomly among the activities of the diagram. Clearly, this case calls for a specially tailored solution technique but may not be relatively as uncommon as the first. Again, do such cases constitute a sizable set of counter examples to the conjecture. We don't believe so; as long as specially tailored techniques can readily deal with them. We will encounter one such technique later in this chapter.

We end this section by pointing out that the simplest form that the function expressing the expected computational time in terms of the ratio could take is a linear form. That should be the first form to be tested in an effort to statistically corroborate the conjecture. Another form that this function might take, and that we think is worth testing, is the linear form in the logarithm of the ratio. But whatever the form of this function, it probably can and

should be considered as another complexity measure for the problem. We note here that the only reported measure in the literature we consulted that claims to capture the effects of the activity durations on the complexity of the problem is Cooper's density measure (Cooper 1976) mentioned in Chapter 1. We believe that our would be measure can justifiably claim the same and, for the time being, relegate this issue to future research.

## 3.6 Node Generation

Recall that nodes, or states, in the reduced state network are induced by the processing of maximal and $d$-submaximal subsets of activities. In the previous section, we were able to compute their number using simple combinatorial arguments. With some additional effort we could have obtained the subsets themselves if required. But recall also that this was possible under a simplifying assumption (*cf.* page 127). In the general case where the assumption does not hold, obtaining the number of maximals or $d$-submaximals may not be so simple and in fact one may even have to list them and count them to obtain their number. But the question then is: how do we generate them? The straight forward way would generate all possible subsets of the set of precedence feasible activities and check each and every subset for being maximal or $d$-submaximal according to the definitions. Computationally, this is an undertaking of exponential growth in terms of the number of precedence feasible activities. In fact, in cases where few or even one maximal or $d$-submaximal subset exists, this method obviously amounts to a considerable waste of computing resources that could well be a major drag on any approach to RCPSP that attempts to take advantage of the reduced state network. Clearly then, alternatives to this direct approach to the issue are in order. This section discusses two

such alternatives. One is based on an integer programming model while the other is a branch and bound approach. Chronically, the integer programming approach was the first we devised. But as we will see in the following discussion, it has a shortcoming which, if possible, should best be avoided. This prompted research into the second approach; the one we adopt as the basis of our node generation scheme.

### 3.6.1 An Integer Programming Based Scheme

This approach is based on mathematically modelling the characteristics that a subset should satisfy in order be considered maximal or $d$-submaximal. Consider a set of $n$ precedence feasible activities which require $r$ types of resources for processing. Let $c_{ik}$ be the resource consumption of activity $i$ from resource $k$ and let $R_k$ be the availability of that resource. Let $x_i$ be a 0/1 variable equal to 1 if activity $i$ is part of a maximal subset we want to determine and equal to 0 otherwise. A maximal subset of the set of $n$ activities should satisfy the following constraints:

$$\sum_{i=1}^{n} c_{ik}.x_i \leq R_k,\ k = 1, \ldots, r \tag{3.1}$$

and for all $j = 1, \ldots, n$:

$$x_j + t_{j1} + \cdots + t_{jr} \geq 1, \tag{3.2}$$

$$\sum_{i=1}^{n} c_{ik}.x_i + c_{jk}.(1 - x_j) + M.(1 - t_{jk}) \geq R_k + 1,\ k = 1, \ldots, r, \tag{3.3}$$

where

$$x_i, t_{ik} \in \{0, 1\}\ \forall i, k.$$

These constraints are to be interpreted as follows. The first constraint set (3.1) ensures that the $r$ resource limits are respected. The second constraint set (3.2) and (3.3) insures that if activity $j$ is not selected to be part of a maximal subset we are seeking then adding it to the subset violates at least one resource limit. In these constraints $M$ is a large number and $t_{jk}$, $k = 1, ..., r$, are simply 0/1 variables one of which, at least, is forced to be 1 if activity $j$ is not part of the subset. In that case, $\sum_{i=1}^{n} c_{ik}.x_i + c_{jk} \geq R_k + 1$ for at least one $k$; which is exactly what the definition of maximality of a subset calls for. Notice also that in case the available resources are sufficient to process all the activities concurrently, the only solution to the system is the set of $n$ activities itself; which, again, is in agreement with the definition.

Clearly now, a feasible solution $(x, t)$ of the system leads to the maximal subset determined by $x$. Hence, to find all the maximal subsets it is sufficient to find all 0/1 solutions to the system (3.1-3.3).

We note here that the solutions to the above system are all extreme points of the polytope determined by the LP relaxation of this same system. This LP relaxation, of course, consists of (3.1-3.3) and $0 \leq x_i, t_{ik} \leq 1$, $\forall i, k$. It follows that to find all maximal subsets, we need to enumerate those extreme points. We should be mindful though that some of the extreme points of the polytope might not be 0/1points[7] and also that there may be more than one feasible solution corresponding to the same maximal subset as there may be several values of $t$ that would satisfy the system for the same value of $x$. Still, if $p$ is the total number of extreme points of the polytope then to be truly efficient the algorithm enumerating the extreme points has to be an $O(p)$ algorithm rather than an $O(2^n)$ as the direct approach is. Undoubtedly, $p$ itself may be $O(2^n)$ but what is more likely is that it is much smaller. Note that an algorithm

[7] A point $x = (x_1, ..., x_n)$ is said to be 0/1 if $x_i \in \{0, 1\}$ $\forall$ $i = 1, ..., n$.

to enumerate the 0/1 points directly without passing over the other extreme points is the best that could be hoped for. But, unfortunately, we don't know of any such algorithm and in fact we doubt that one exists with a linear complexity in terms of the number of 0/1 points.

Now, let $d$ be the minimum duration, or residual duration, of the activities considered. Let $P_1$ be the set of activities of duration $d$ and $P_2$ the set of activities with duration strictly larger than $d$. Then to generate the $d$-submaximal subsets we simply modify our previous system for generating maximal subsets by requiring that constraints (3.2) and (3.3) be satisfied for only $j \in P_1$. That is, $d$-submaximals should satisfy:

$$\sum_{i=1}^{n} c_{ik}.x_i \leq R_k,\ k = 1, \ldots, r,$$

and for all $j \in P_1$:

$$\begin{aligned} x_j + t_{j1} + \cdots + t_{jr} &\geq 1, \\ \sum_{i=1}^{n} c_{ik}.x_i + c_{jk}.(1 - x_j) + M.(1 - t_{jk}) &\geq R_k + 1,\ k = 1, \ldots, r, \end{aligned}$$

where

$$x_i, t_{ik} \in \{0, 1\}\ \forall i, k.$$

We note that not every solution generated by this system represents a maximal subset as we now require that only adding an activity of $P_1$ need violate a resource constraint. In the same way as before, this system can generate as a solution all the activities of $P_1$, if resources permit. It can also generate as a solution all activities of $P_1$ added to any subset of activities of $P_2$, again as resources permit. Needless to say, any solution to the system represents a $d$-submaximal subset.

Hence to find the $d$-submaximal subsets all we need is to enumerate the $0/1$ extreme points of the LP relaxation of this second system. Naturally, its number of solutions is expected to be larger than the number of solutions to the previous one as this system is constructed from the first by deleting constraints. Otherwise, the same remarks as before concerning the complexity an algorithm to generate those extreme points apply.

### 3.6.2 A Branch and Bound Scheme

The basic idea behind this scheme for uncovering all maximal subsets, of a given set of activities, is that of constructing those subsets one activity at a time. At each step of the process, the question is whether or not to include an activity or not and what could the consequences of such a decision be. This is best visualized as the decision tree in Figure 3-9. Each level of the tree, except the first, corresponds to an activity. If the given set of activities has cardinality $n$ then the tree has $n+1$ levels. Branching is on the activity, whether it is part of some subset or not. Each path from the Begin node to one of the leaves represents a subset of activities. In fact, there is a one to one and onto correspondence between the set of such paths and the set of all subsets of activities. Clearly then, the tree contains $2^n$ such paths. Clearly also, not all paths correspond to maximal subsets. Therefore, our task should be to come up with bounding rules so as to fathom nodes of the tree that could not lead to maximal subsets and even ones at which a maximal subset is within reach. In that respect, the following notation is needed. Let:

$w_i$ denote the $i^{th}$ activity of the given set, $i = 1, ..., n$.

$R_k$, as before, denote the resource availability of resource $k$, $k = 1, ..., r$.

$c_{ik}$, also as before, denote the resource consumption of activity $i$ from resource $k$, $i = 1, ..., n$

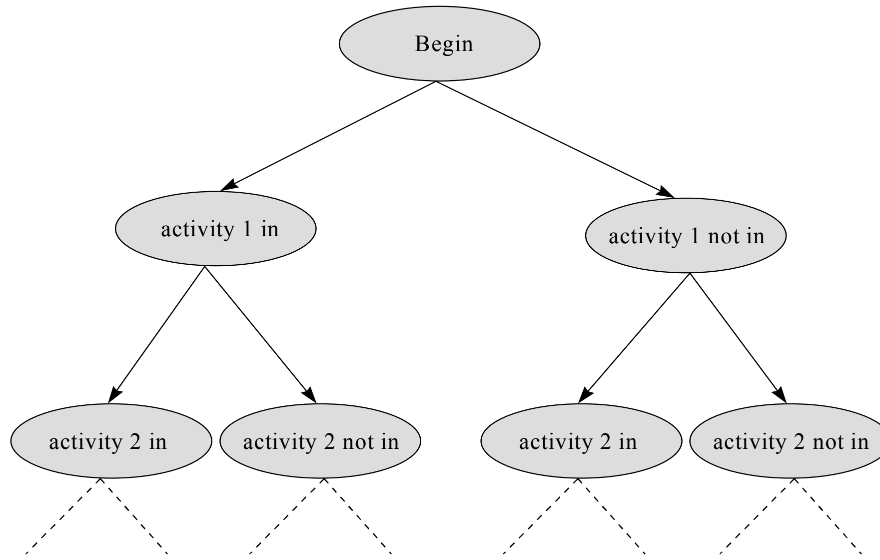


Figure 3-9: A decision tree to generate all maximal subsets.

and $k = 1, ..., r$.

$u_{ik} = \min\{c_{jk} : j > i\}$, $i = 1, ..., n-1$ and $k = 1, ..., r$. This is the smallest usage from resource $k$ among the activities $w_{i+1}, ..., w_n$. It is best computed backwards from $i = n-1$ to 1.

$r_{ik} = \sum_{j=i+1}^{n} c_{jk}$, $i = 1, ..., n-1$ and $k = 1, ..., r$. For each resource $k$, this is the total requirement of the activities $w_{i+1}, ..., w_n$. It, too, is best computed backwards.

$a_k$ denote the residual availability of resource $k$ after making a decision at level $i$, $k = 1, ..., r$. This variable is updated at every level which alleviates the need for the explicit dependence on $i$.

$v_k = \min\{c_{jk} : j \leq i$ and $w_j$ was not included in the path being explored$\}$. This variable depends of the level $i$ as well as the subpath already explored. It is updated for every subpath, though, which alleviates the explicit dependence on the index $i$ and the subpath.

The following bounding rules could be implemented:

**Rule 1**: If $\sum_{i=1}^{n} c_{ik} \leq R_k$, then the set of $n$ activities itself is the unique maximal subset.

**Rule 2**: The height of the tree, *i.e.,* its number of levels, is $n+1$.

**Rule 3**: Suppose that at a node at level $i$, and after making the decision to include $w_i$ or not, the condition:

$$(iii)\ c_{jk} + r_{ik} \leq a_k, \forall\ k = 1, ..., r$$

holds for some $j \leq i$ for which $w_j$ was not already included in the subset leading up to the current node. Then the current node can be fathomed as it cannot lead to a maximal subset. Since this condition is computationally expensive to check at every node, we precede it by checking the condition:

$$(ii)\ v_k + r_{ik} \leq a_k, \forall\ k = 1, ..., r$$

which has to hold before $(iii)$ holds. On the other hand, if condition:

$$(i)\ r_{ik} \leq a_k, \forall\ k = 1, ..., r$$

holds with either

$$(ii)'\ v_k + r_{ik} > a_k,\ \text{for some}\ k = 1, ..., r$$

or both $(ii)$ and

$$(iii)'\ c_{jk} + r_{ik} > a_k,\ \text{for some}\ k = 1, ..., r$$

as $(iii)'$ applies to all $j \leq i$ for which $w_j$ was not already included in the subset leading up to the current node, then the current node can be fathomed as the already accumulated activities together with $w_{i+1}, ..., w_n$ form a maximal subset.

**Rule 4**: Suppose that at a node at level $i$, and after making the decision to include $w_i$ or not, the condition:

$$(v)\ u_{ik} > a_k,\ \text{for some } k = 1, ..., r$$

holds together with the condition:

$$(vii)\ c_{jk} \leq a_k, \forall\ k = 1, ..., r$$

which, in turn, holds for some $j \leq i$ for which $w_j$ was not already included in the subset leading up to the current node. Then the current node can be fathomed as it cannot lead to a maximal subset. Since condition $(vii)$ is computationally expensive to check at every node condition $(v)$ holds for, we precede it by checking the condition:

$$(vi)\ v_k \leq a_k, \forall\ k = 1, ..., r$$

which has to hold before $(viii)$ holds. On the other hand, suppose condition $(v)$ holds together with either the condition:

$$(vi)'\ v_k > a_k,\ \text{for some } k = 1, ..., r$$

or both $(vi)$ and

$$(vii)' \; c_{jk} > a_k, \text{ for some } k = 1, ..., r$$

as $(vii)'$ applies to all $j \leq i$ for which $w_j$ was not already included in the subset leading up to the current node. Then the current node can be fathomed as the subset of activities already accumulated is maximal.

**Remark 10** *The implementation of Rules* 3 *and* 4 *can be significantly simplified if* $r = 1$, *i.e., if only one resource is significant. In that case, the activities can be ordered in non-increasing or non-decreasing resource consumptions which allows a faster implementation of the rules than otherwise. Also in that case, the vector* $u$ *is easily obtainable and so is not needed anymore.*

**Remark 11** *Bounding Rules* 3 *and* 4 *are consistent with one another. That is, a node that is fathomed by one rule because it cannot yield a maximal subset cannot be fathomed by the other because it does. This is easily seen when considering the flowcharts of the rules in Figure 3-10. There,* $test3$ *is* 0 *if, according to Rule* 3, *the node can be fathomed without yielding a maximal subset. It is* 1 *if the node can be fathomed yielding a maximal subset and* $test3$ *is* 2, *otherwise.* $test4$ *has the same interpretation in the context of Rule* 4. *Note that the only possible values for* $(test3, test4)$ *are* $(0, 2)$, $(1, 2)$, $(2, 2)$, $(2, 1)$ *and* $(2, 0)$. *Needless to say, among these pairs the only case that merits further branching is the* $(2, 2)$ *case.*

The following recursive algorithm implements our branching idea together with bounding Rules $2 - 4$. The search strategy is Depth First organized as a laser search, *i.e* DFSL. Rule 1 could be checked before the algorithm is called, and so is omitted, while Rule 2 is implicitly implemented.

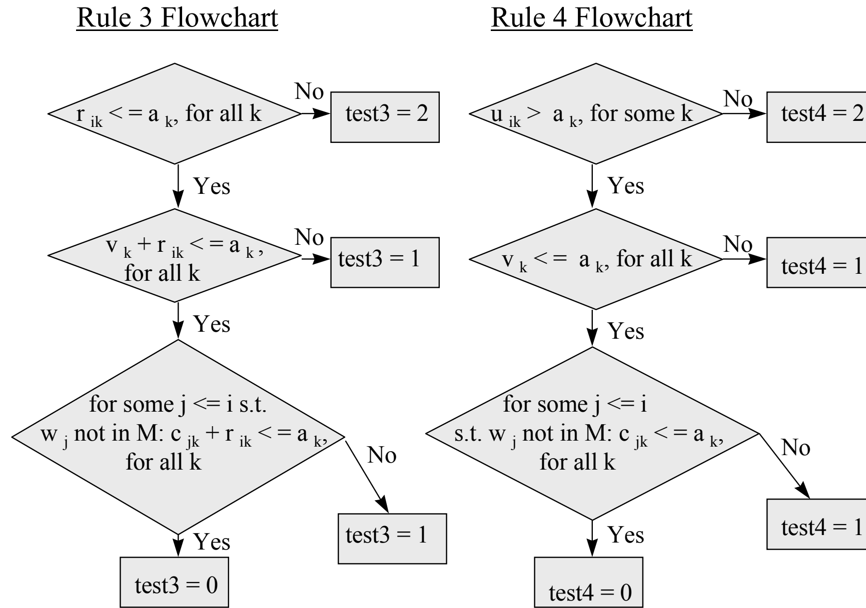


Figure 3-10: The flowcharts for implementing Rules 3 and 4 in Algorithm 1.

**Algorithm 1**

**begin** enum_maxmls $(i, M, a, v)$

/* $i$ denotes a tree level, $M$ is a maximal subset under construction, $a$ and $v$ are as previously defined. Initial call with $i = 0$, $M = \emptyset$, $a(k) = R_k$ and $v(k) = 0$, $\forall\ k = 1, ..., r$ */

$\mathbf{t}est3 \leftarrow rule3(i, M, a, v)$ /* test3 is 0 if node $i$ cannot yield a maximal subset, 1 if it does and 2 if Rule 3 is inconclusive */

**i**f $test3 = 1$

**o**utput $M \cup \{w_{i+1}, ..., w_n\}$

**e**lseif $test3 = 2$

$\mathbf{t}est4 \leftarrow rule4(i, M, a, v)$ /* test4 has the same interpretation as test3 but w.r.t. Rule 4 */

```
    if test4 = 1
      output M
    elseif test4 = 2
      enum_maxmls (i + 1, M ∪ {w_{i+1}}, a − c_{i+1,.}, v)
      enum_maxmls ( i+1, M, a, min{v, c_{i+1,.}})   /* min here as applied to vectors
is meant to yield a vector of component minima. */
    endif
  endif
end
```

---

```
function rule3 (i, M, a, v)
begin
  if i = 0
    test3 ← 2, return
  endif
  k ← 1
  while k < r + 1 and r_{ik} ≤ a_k
    k + +
  endwhile
  if k < r + 1
    test3 = 2, return
  else
    k1 ← 1
```

**w**hile $k1 < r + 1$ and $r_{i,k1} + v_{k1} \leq a_{k1}$

**k**$1 + +$

**e**ndwile

**i**f $k1 = r + 1$

**j** $= 1$

**w**hile $j \leq i$

**i**f $w_j \notin M$

**k**$2 \leftarrow 1$

**w**hile $k2 < r + 1$ and $r_{i,k2} + c_{j,k2} \leq a_{k2}$

**k**$2 + +$

**e**ndwhile

**i**f $k2 = r + 1$

**t**$est3 \leftarrow 0$, return

**e**ndif

**e**ndif /* if $w_j \notin M$ */

**j** $+ +$

**e**ndwhile /* while $j \leq i$ */

**t**$est3 \leftarrow 1$, return /* this line is reached when all $w_j \notin M$ s.t. $j \leq i$ could not fit in $M$ */

**e**lse

**t**$est3 \leftarrow 1$, return

**e**ndif /* if $k1 = r + 1$ */

**e**ndif /* if $k < r + 1$ */

**end**

---

**function** rule4$(i, M, a, v)$

**begin**

**i**f $i = 0$

**t**$est4 \leftarrow 2$, **r**eturn

**e**ndif

**k** $\leftarrow 1$

**w**hile $k < r + 1$ and $u_{ik} \leq a_k$

**k** $++$

**e**ndwhile

**i**f $k = r + 1$

**t**$est4 = 2$, return

**e**lse

**k**$1 \leftarrow 1$

**w**hile $k1 < r + 1$ and $v_{k1} \leq a_{k1}$

**k**$1 ++$

**e**ndwile

**i**f $k1 = r + 1$

**j** $= 1$

**w**hile $j \leq i$

**i**f $w_j \notin M$

**k**$2 \leftarrow 1$

**w**hile $k2 < r + 1$ and $r_{i,k2} + c_{j,k2} \leq a_{k2}$

**k**$2 + +$

**e**ndwhile

**i**f $k2 = r + 1$

**t**$est4 \leftarrow 0$, return

**e**ndif

**e**ndif /* if $w_j \notin M$ */

**j**$+ +$

**e**ndwhile /* while $j \leq i$ */

**t**$est4 \leftarrow 1$, return /* this line is reached when all $w_j \notin M$ s.t. $j \leq i$ could not fit in $M$ */

**e**lse

**t**$est4 \leftarrow 1$, return

**e**ndif /* if $k1 = r + 1$ */

**e**ndif /* if $k < r + 1$ */

**end**

**Remark 12** *Algorithm* 1 *does not implement bounding Rule* 2 *explicitly for a good reason. This rule is implicitly implemented by the combined effect of implementing Rules* 3 *and* 4*. To see this, note that enum_maxmls is called in* 1 *increments in terms of $i$. That is, enum_maxmls cannot be called with $i = j$, for any $j > 0$, unless the call originated in a call with $i = j - 1$. Suppose now that enum_maxmls is called with $i = n - 1$. Rule* 3 *is, then, immediately tested. If $r_{n-1,k} \leq a_k$ for all $k = 1, ..., r$, the node will either yield a maximal subset or else we know*

*it cannot lead into one. But in either case it will be fathomed and so a follow up call to enum_maxmls is not placed. That is, in that case, no call with $i = n$ is originated. On the other hand, if $r_{n-1,k} > a_k$ for some $k = 1, ..., r$, the algorithm calls for testing Rule 4. But it so happens that $r_{n-1,k} = u_{n-1,k}$ for all $k = 1, ..., r$. Hence Rule 4 cannot be inconclusive now and has to either yield a maximal subset or else decide the node cannot lead into one. Again, that node is fathomed and a call to enum_maxmls with $i = n$ is not originated. Thus, in both cases, Rule 2 is implicitly implemented. Observe here that this argument cannot be modified to show that a call to enum_maxmls with $i = n - 1$ cannot be not originated; as both Rules 3 and 4 can be inconclusive when $i = n - 2$.*

**Theorem 19** *Algorithm 1 generates exclusively all maximal subsets of a set of $n$ activities, requiring $r$ limited resources, in $O(rn^2m)$ time, where $m$ is the number of maximal subsets.*

**Proof.** We note first that without Rules 3 and 4 and by conditioning $i$ to be at most $n$, at which value $M$ is outputted whatever it contains, Algorithm 1 generates all subsets of the set of $n$ activities[8]. Recall also that the only nodes of the search tree that bounding Rules 3 and 4 fathom are the ones that cannot possibly lead to maximal subsets or the ones at which such subsets are readily available and so outputted. It necessarily follows then, from those two statements, that Algorithm 1 outputs all maximal subsets. Moreover, the algorithm outputs no subset that is not maximal since subsets are outputted only when either $test3 = 1$ or else $test4 = 1$. Thus, the algorithm produces exclusively all maximal subsets.

To estimate the running time of the algorithm, observe that a maximal subset is outputted only once. To see this recall that a maximal subset is generated as a branch of the search tree.

[8]Eliminating Rules 3 and 4 can essentially be accomplished by discarding the calls to functions *rule*3 and *rule*4 and setting *test*3 and *test*4 to 2, instead.

Two different branches may share the same nodes at the first $j+1$ levels of the tree, for some $j$ s.t. $0 \leq j < n$, but ultimately have to split to be different. If the two branches split at level $j+2$ then one of the subsets they represent must contain activity $w_{j+2}$ while the other must not. This implies that if the two branches are ultimately fathomed yielding maximal subsets, they must represent different maximal subsets.

Additionally, due to the fact that a node of the search tree has only one immediate predecessor, a reexamination of a previously examined node cannot take place unless the node is directly accessed a second time or else its immediate predecessor is. Now, a direct access to the node for the second time is not possible as the calls to enum_maxmls are never with both the same level $i$ and subset $M$. For this same reason, a second examination of the immediate predecessor node is not possible also. Hence, we conclude, a node of the tree can be explored during the search at most once. This implies that a branch of the tree that leads to a maximal subset can be examined at most once. Since we already have established that different branches that lead to maximal subsets lead to different ones and since we have also already established that every maximal subset is outputted, we conclude that a maximal subset is outputted by Algorithm 1 exactly once.

Now, it remains to estimate the maximum cost of obtaining a maximal subset and the maximum cost involved in fathoming nodes because they cannot lead to maximal subsets. To this end, consider the complete search tree and two of its branches that each lead to a maximal subset. Say the two branches are adjacent to one another in the sense that no third branch in between the two leads to a third maximal subset. Say also that the two branches split at a level

$j$, $0 \leq j < n$[9]. It is easy to see that each node on levels $j+2, ..., n$, that is in between the two branches, is not part of any branch leading to a maximal subset precisely because it is either an immediate descendent of a node on one of the branches and must have been fathomed or else one of its predecessor nodes was such. Hence, the cost of eliminating those nodes from the search is actually the cost of eliminating from consideration the nodes bordering the two branches. Note that given a branch of the tree representing a maximal subset, one can fathom at most $n-1$ nodes that border this branch regardless of which side they might be on. This doesn't include the Begin and the leaf nodes as those cannot be fathomed. Each of the fathoming tests yielding no maximal subsets costs the most when only Rule 3 is used and not both rules. This is due to the additions and comparisons instead of just comparisons involved in the last step of that rule. Hence fathoming the nodes bordering a branch representing a maximal subset adds up to at most $\sum_{i=1}^{n-1}[r + r + 2r(i-1)] = rn(n-1)$ operations. Additionally, this branch costs at most $n-2$ inconclusive tests for fathoming and one last conclusive test in order to decide on the maximality of the subset it represents. Those $n-2$ inconclusive tests cost at most $r$ comparisons each, for every node and rule, for a total of $2r(n-2)$ operations for the two rules. The last conclusive test costs at most $2r(n-1)$, for a total of $2r(2n-3)$ operations. Thus, a branch representing a maximal subset costs at most $O(rn^2)$ operations to construct and to fathom nodes bordering it. Suppose now we have $m$ possible maximal subsets. Clearly, any two branches representing two of those subsets share the same first several nodes. But regardless, the total cost of those branches, and thus subsets, is at most $O(rn^2m)$ operations. This says Algorithm 1 is an $O(rn^2m)$ algorithm. ■

[9] The two branches cannot split at level $n$ and still lead to maximal subsets because the only difference between the two would be due to activity $w_n$ which contradicts the maximality of the subset that does not contain that activity.

**Remark 13** *In the previous subsection, the idea to generate the maximal subsets of a set was to enumerate the* 0/1*extreme points of a polytope. Several such extreme points, however, may correspond to the same maximal subset. The best algorithm we could hope for following that idea was one that is linear in the total number of extreme points itself, not even the number of* 0/1 *points. Well, we have now, in Algorithm* 1*, a means to generate the maximal subsets that is linear in the number such subsets. This must be the best we could hope for as far as the number of those subsets is concerned. This, clearly, is an important theoretical advantage for the latter algorithm over a treatment that enumerates all the extreme points of a polytope.*

Now, recall that what we really want to generate are the $d$-submaximal subsets not just the maximal subsets. In that respect, let $P_1$ and $P_2$ be as in the previous subsection, *i.e.,* the sets of activities with durations $d$ and with durations strictly larger than $d$, respectively. To generate all $d$-submaximals of $P = P_1 \cup P_2$, under limited $r$ resources, we can simply form all subsets of compatible activities of $P_2$ and complement them as much as resources permit with activities from $P_1$. Note that Algorithm 1 can be slightly modified to accomplish the second task but that we need a way of accomplishing the first. To do this, we adopt the same basic BaB idea to generate all maximals. That is we want a binary tree with some bounding rules whose search will yield all subsets of compatible activities of $P_2$. To this end, we use the second of the previous four rules and a simplified version of the fourth:

**Rule 5** : In a BaB effort to produce all resource compatible subsets of a set of activities, suppose that at a node at level $i$, and after making the decision to include $w_i$ or not, the condition:

$$(v)\ u_{ik} > a_k,\ \text{for some } k = 1, ..., r$$

holds. Then the subset of activities already accumulated can be outputted as no subsequent activity can fit.

The following algorithm implements those ideas to generate all $d$-submaximals of $P$. It uses enum_maxmls which needs to be modified in the two output statements of its main function so that it returns a given subset $M'$, namely a resource feasible subset of $P_2$, in addition to the subsequently maximal subsets of $P_1$. The call to enum_maxmls would then be: enum_maxmls$(i, M, a, v, M')$. In this algorithm we use $w_i$ to denote the $i^{th}$ activity of $P_1$ and $w'_i$ to denote the $i^{th}$ activity of $P_2$. Additionally, $c_{i,\cdot}$ and $c'_{i,\cdot}$ denote the resource consumption vectors corresponding to those activities, respectively, and $|S|$ denotes the cardinality of a set $S$.

**Algorithm 2**

**begin** enum_ $d$_submaxmls$(i, M', a)$

/* $M'$ is a subset of compatible activities of $P_2$ which is under construction, $a$ and $v$ are as previously defined. Initial call with $i = 0$, $M' = \emptyset$, $a(k) = R_k \ \forall \ k = 1, ..., r$ */

**if** $i = 0$

**if** $\sum_{i \in P_1} c_{i\cdot} \leq a$

**o**utput $P_1$

**e**lse

**e**num_maxmls$(0, \emptyset, a, 0, \emptyset)$ /* output all maximals of $P_1$ */

**e**ndif

**e**lseif $i < |P_2|$ /* cardinality of $P_2$ */

**t**$est5 \leftarrow rule5(i, M', a)$ /* test5 is 1 if node $i$ can be fathomed yielding a resource

feasible subset that cannot fit further activities down the list It is 2 if the rule is inconclusive

*/

**i**f $test5 = 2$

**i**f $c'_{i+1,.} \leq a$

**i**f $\sum_{i \in P_1} c_{i.} \leq a - c'_{i+1,.}$

**o**utput $M' \cup \{w'_{i+1}\} \cup P_1$

**e**lse

**e**num_maxmls$(0, \emptyset, a - c'_{i+1,.}, 0, M' \cup \{w'_{i+1}\})$

**e**ndif /* if $\sum_{i \in P_1} c_{i.} \leq a - c'_{i+1,.}$ */

**e**num_$d$_submaxmls$(i+1, M' \cup \{w'_{i+1}\}, a - c'_{i+1,.})$

**e**num_$d$_submaxmls$(i+1, M', a)$

**e**lse

**e**num_$d$_submaxmls$(i+1, M', a)$

**e**ndif /* if $c'_{i+1,.} \leq a$ */

**e**ndif /* if $test5 = 2$ */

**e**ndif /*if $i = 0$*/

**e**nd

---

**function** rule5$(i, M', a)$

**begin**

**k** $\leftarrow 1$

**w**hile $k < r + 1$ and $u_{ik} \leq a_k$

**k** $++$

**e**ndwhile

**i**f $k = r + 1$

**t**$est5 = 2$, return

**e**lse

**t**$est5 = 1$, return

**e**ndif

**e**nd

**Remark 14** *If rule*$5$ *returns* $1$ *at a node at level* $i$ *of the search tree then the two subtrees descending from this node can be eliminated (cf. Figure.3-9). In contrast, if* $c'_{i+1,.} > a$ *then only the subtree corresponding to "activity* $(i+1)$ *in" can be eliminated. The other subtree is still open to further exploration.*

**Theorem 20** *Consider a set of activities written as* $P_1 \cup P_2$ *and requiring* $r$ *limited resources, where* $d$ *is the minimum duration of the activities and* $P_1$ *and* $P_2$ *are as defined previously. Let* $m'$ *be the number of* $d$*-submaximal subsets of* $P_1 \cup P_2$*. Then Algorithm* $2$ *generates exclusively all such* $m'$ *subsets in* $O(r\{|P_1|^2 + |P_2|\}m')$ *time.*

**Proof.** To see that Algorithm 2 generates all $d$-submaximals, set $test5$ to 2 regardless of the outcome from function rule5; which effectively cancels that function. Furthermore, disregard the direct output statements and the calls to enum_maximals.. It is clear, in this case, that the algorithm acts only on activities of $P_2$ and that the only branches of the binary search tree that Algorithm 2 fathoms are the ones corresponding to adding an activity, to a subsets under construction, that does not fit with the already accumulated ones due to resource limitations. Except for those fathomed branches, all the branches of the search tree are traced up to the

leaf nodes. Since each branch traced, whether fathomed or not, represents a unique subset of compatible activities, we conclude that Algorithm 2 without rule5 and the direct output statements and enum_maxmls calls, potentially produces all subsets of compatible activities of $P_2$. Now, as rule5 additionally fathoms only branches that could not yield more activities added to the subsets already accumulated, we conclude that Algorithm 2, without the output and enum_maxmls calls, traces all branches of the search tree corresponding to resource compatible subsets of activities. Additionally, no branch corresponding to a subset of activities with resource requirements larger than one of the limits could be traced because precisely those fathoming rules are tested prior to branching. Therefore, the algorithm without the output statements and enum_maxmls calls traces exclusively all the branches of the search tree representing resource compatible activities of $P_2$. Now, the only effect of the output statements and the enum_maxmls calls is to add subsets of activities of $P_1$ to each resource feasible subset of $P_2$ in such a way that the $P_1$ subsets are maximal w.r.t. the residual resources. Since enum_maxmls produces exclusively all possible maximal subsets of a set of activities, we conclude that all subsets outputted by Algorithm 2 are $d$-submaximals. For this same reason, we also conclude that Algorithm 2 does not output a subset that is not $d$-submaximal.

Note that two different branches representing subsets of compatible activities of $P_2$ represent different such subsets. Since, similarly to enum_maxmls, calls to enum_$d$_submaximals do not revisit the same node, and since Algorithm 2 outputs exclusively all $d$-submaximals, we conclude that it exclusively outputs them exactly once.

Now, the cost of outputting a $d$-submaximal can be seen as that of producing the $P_2$ subset of resource compatible activities plus that of producing the $P_1$ subset to complement it. In the worst case, the $P_2$ subset is produced after $|P_2| - 1$ recursive calls. Each of those costs $O(r)$

operations for calling rule5 and an additional $O(r)$ for miscellaneous operations. Hence, the cost of producing a $P_2$ subset is at most $O(r|P_2|)$. Following the proof of Theorem 19, the cost of complementing it with a $P_1$ subset is $O(r|P_1|^2)$. Hence, the cost of a of a $d$-submaximal is at most $O(r\{|P_1|^2 + |P_2|\})$. We note that the fathoming cost of a branch is $O(r)$. This can be absorbed by the cost of the last resource feasible subset produced by the branch and as such, this cost is negligible. It follows, then, that if the whole set of activities under consideration, i.e. $P_1 \cup P_2$, has $m'$ $d$-submaximals then the cost of Algorithm 2 is at most $O(r\{|P_1|^2 + |P_2|\}m')$ operations. ■

**Remark 15** *As in the case of maximal subsets, we now have in Algorithm* 2 *a means to generate all d-submaximals of a set of activities that is linear in the number of such subsets. Theoretically, again, this algorithm has a considerable advantage over one based on enumerating the extreme points of a polytope as not all such points may be* 0/1 *and as several of them may lead to the same d-submaximal.*

## 3.7 A Bidirectional Approach

Recall that the idea behind the reduced state network of an instance, or even the plain state network and state graph, is to apply a shortest path algorithm whose output yields a minimum makespan schedule of the instance. In the examples we have seen, the state networks were always constructed forward, *i.e.,* in the same direction as the precedence relationships. But, just as well, we could have constructed the state networks in the reverse direction, *i.e.*, starting from the terminal activity and going opposite the precedence relationships. But for that to be possible, the definition of a state has to be altered from being the set of all activities that

are precedence feasible to start or that are already active, with their residual processing time, to alternatively become the set of all activities that are precedence feasible to finish or that already are active, with their initial processing time. To differentiate between these two types of states, we call a state satisfying the first notion a **forward state** and one satisfying the second a **reverse state**. Now the network of reverse states can be referred to as the **reverse state network** while our usual state network is simply the **forward state network**. Of course, results to reduce the reverse state network can be obtained in a way similar to the results that enable reducing the forward state network. This naturally extends the forward/reverse terminology to the reduced state network concept.

Note that with different definitions of a state at their core, the reduced forward state network and its counterpart the reduced reverse state network should not be expected to be the same. This dissimilarity is further reinforced by, in effect, right shifting the activities in a reduced reverse network instead of left shifting them as in the reduced forward network. This could definitely cause the two reduced networks to be markedly different from one another not only in terms of state designations but also in the number of actual states and network configuration. As a matter of fact, since according to Theorem 17 the schedules yielded by the forward reduced state network are active in the sense that no activity can be left-shifted without altering the start time of other activities, we could say that the schedules yielded by the reverse reduced network are active in the sense that no activity can be right-shifted without altering the finish time of other activities. That statement would simply follow from a reverse state network counterpart to the cited theorem which may analogously be stated and proven, and this even includes the single duration case. But the point we wish to make here is that the two reduced networks should not in general be expected to be the same as they yield different types of

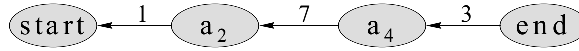


Figure 3-11: The reverse reduced state network of the project instance in Figure 3-1.

"active" schedules. To differentiate between those two types, we call the former **left-active** and the latter **right-active**.

To illustrate the difference between the two networks, we display in Figure 3-11 the reverse reduced state network of the project instance in Figure 3-1. We remark that its forward reduced state network, in Figure 3-8, has seven nodes instead of only two.Note also that the only reason the arrows in that reduced state network point from the end node towards the start one is to indicate that the network is constructed backwards. Reading backwards, the schedule is to start $a_1$, $a_2$, $a_3$ simultaneously, then $a_4$ and then $a_2$. Interestingly enough**,** reversing that order, we obtain the optimal schedule for the instance. Notice, in this case, that finding the shortest start-to-end path happens to be a lot simpler for the reverse network. The same applies when solving the instances in Theorem 4 of Chapter 1. There too, the reverse reduced state network is much simpler a network to work with and so yields the optimal solution faster.

It is worthwhile pointing out here that the ease with which the optimal solution to these instances was obtained helps foil what might have been a possible complexity measure for RCPSP. Recall in Theorem 4 of Chapter 1, the ratio of optimal non-delay to delay schedule makespans is unbounded. This might suggest that a complexity measure for the problem could

be based on this ratio. Indeed, the more possibilities for delaying activities the more one has to explore them and so the more computationally expensive the instance gets relative to the case of no delays. In fact, in the forward direction, the reduced state networks for the instances in the theorem are relatively involved due to this same possibility. In the reverse direction, on the other hand, it can be verified that the reduced state networks are not nearly as large and hence their shortest paths can be obtained faster. This says that the many possibilities for delay need not be examined; which implies that they need not affect the computational burden. Whence, the ratio of delay to non-delay schedule makespans would not be a good measure of complexity as one instance could have two possibly far apart such measures. In fact, not even the average of the two such measures for an instance is likely to be a good complexity measure as this same average is likely to mask whatever computational difficulty is encountered in the forward direction. One might wonder if the same is true of a potential complexity measure based on the ratio of number of activities with the minimum duration to the total number of activities. We, therefore, remark that the proposed ratio in such a would be measure is not tied to any notion that is computationally dependent on the direction of its computing, as the optimal delay makespan is. That is, the argument against the ratio of delay to non-delay makespans simply does not apply to the conjectured measure.

One probably suspects by now that the reverse direction does not always yield the simpler network. To see this, consider an instance whose reverse reduced state network is simpler than the forward one. Construct a new instance by maintaining the same activities but reversing all the precedence relationships of the first. Clearly now, the forward network is the simpler of the two. But, unfortunately, given an arbitrary project instance, we don't know, a priori, how to predict which of the two directions is the easier to go with. For this reason, one way

to take advantage of the easier direction, whichever it may be, is to combine the searches for a shortest start-to-end path from each direction. This can be done by alternating the search from the start node of the reduced forward network and the end node of the reverse reduced network, despite the networks' difference, but requires (1) a means to verify that two subpaths, one from each direction, lead to a "complete start-end path" and (2) a means to verify that such a path is optimal so that an algorithm based on this idea eventually concludes before a terminal node is reached. Additionally, if Dijkstra's Shortest path algorithm is to be the basis for obtaining the shortest path, as we intend it to be, then a by-product of this bidirectional approach, one hopes, would be a relatively reduced number of nodes to generate as compared to the unidirectional approach. Intuitively, the reason for this would be that the higher the eventual label of a node, the more nodes would need to be explored in order to label that node. Whence, if we can avoid labeling nodes that would necessarily have a relatively high label, such as the end node, we are likely to explore, and thus generate, a relatively smaller number of nodes.

### 3.7.1 Path Completion

To reiterate, we aim that our bidirectional approach to finding the shortest path would alternate iteratively between the two reduced networks. For each network, the goal is to adapt Dijkstra's labeling algorithm to find the shortest path between the terminal nodes of that network. In this process, path completion and optimality conditions are checked whenever a node is generated so that we may possibly terminate the two searches before the other terminal node of one of the networks is reached. Of course, this requires expanding the commonly agreed upon notion of a path as we are contemplating here a path between two nodes in two different networks. Indeed,

by a **mixed path** between the start node of the reduced forward state network and the end node of the reduced reverse network of a project instance we mean a pair of subpaths[10], one from each node and each totally within their respective networks, in which every activity of the instance is jointly sufficiently processed. That is to say, for each activity, we want either that one of the subpaths, at the least, yields its starting and completion time or else that the total processing times allocated to the activity in the two subpaths be, at the least, the duration of the activity. One might wonder here what the need to over-process or even double-process an activity might be. The answer is that the potential of over-processing is a consequence of combining two partial schedules one of which cannot be but left-active while the other cannot be but right-active. There simply may not be a mixed path that does not over-process any of the activities. This is even true of the single duration case as the requirement that each state be maximal could well force the same activity to be in both partial schedules. Of course then, the process of deriving a schedule from a mixed path should eliminate the over-processing in a way that maintains feasibility.

Now, how do we know that we have secured a mixed path in the first place? Are we to simply verify that each activity of an instance is sufficiently processed in two subpaths which we suspect determine a mixed path? Note that the number of possibilities to combine two subpaths might be quite large. In fact, if we were to follow this direct approach then every state created would have to carry with it information specifying exactly what activities have already been completed. This forces the inclusion of too many activities in each state, in a way that cumulatively adds up to a relatively considerable usage of memory space, and renders

[10]Note that the only reason we use this terminology is to emphasize that a subpath yields a subschedule, *i.e.* an incomplete schedule, as opposed to a schedule that can only be induced by a start-end or an end-start path.

our concise description of a state irrelevant. Surely, this can be done for smaller problem sizes. But given limited amounts of time and memory space available, a less demanding alternative is indeed desirable. To this end, we explore three ways of verifying whether or not a mixed path is at hand. All are based on indirectly verifying that every activity of an instance has been completed in one of two constituent subpaths of a possible mixed path or sufficiently jointly processed. All presuppose that the following qualification condition holds:

**Mixed Path Qualification Condition.** Let $\mathcal{P}^f$ and $\mathcal{P}^r$ be two subpaths of the reduced forward state network and the reduced reverse state network of a project instance, respectively. Let $I^c$ be the index set of activities that are active at both the last nodes of $\mathcal{P}^f$ and $\mathcal{P}^r$. Then a necessary condition for the two subpaths to determine a mixed path is for the total remaining processing times of any activity of $I^c$ not to exceed its duration.

**Remark 16** *This condition is necessary only and by no means sufficient; as no activities beyond $I^c$ are tested for completion. Nevertheless, the condition can be a useful tool to screen out subpath combinations that do not constitute mixed paths. We point out here that had we required, in the definition of mixed path, that every activity be completed in one of the subpaths not only 'jointly sufficiently processed', then there would be no need for the qualification condition. In fact, the results in the rest of this section can easily be written for the simpler definition. This of course brings up the question of why the need for a more demanding definition of mixed path when a simpler one can do. The answer is that the more elaborate definition allows, potentially, for detecting that all instance activities have been sufficiently processed sooner than what otherwise might be the case. To illustrate this point in its extreme manifestation, consider an example in which a single activity runs throughout an optimal project duration. Clearly, if*

*we require that each activity be completed in one of two subpaths then a mixed path will not be detected before a shortest path in one of the two reduced networks is identified first. Hence in this case, the whole concept of mixed path is rendered useless. If on the other hand, we allow that each activity be jointly sufficiently processed then chances are a mixed path is detected a lot sooner.*

In the rest of this section, and unless otherwise stated, we will assume that the Mixed Path Qualification Condition holds.

**The Powers of 3 Method**

Let $\mathcal{P}^f$, $\mathcal{P}^r$ and $I^c$ be as in the statement of the Mixed Path Qualification Condition. Let $I = \{1, ..., n\}$ be the index set of all activities of the project instance. Let $I^f$ be the index set of activities completed in subpath $\mathcal{P}^f$ and let $I^r$ the index set of activities completed in subpath $\mathcal{P}^r$. Clearly, $I^f$, $I^r$, $I^c \subset I$ and $I^c \cap (I^f \cup I^r) = \emptyset$. To verify whether $I^f \cup I^r \cup I^c = I$, *i.e.,* whether $\mathcal{P}^f$ and $\mathcal{P}^r$ constitute a mixed path, without carrying explicit information about the activities, this method aims to verify that the difference of two numbers could be written in a particular way. It is based on an extension to the following lemma.

**Lemma 21** *Let $x \geq 2$. Then, $I^f$ , $I^r$ and $I^c$ form a partition of $I$* iff $\sum_{i \in I^f} x^i + \sum_{i \in I^r} x^i + \sum_{i \in I^c} x^i = \sum_{i \in I} x^i$.

**Proof.** Clearly, the forward implication is immediate. To prove the converse, suppose the equality holds. If $I^f \cap I^r = \emptyset$ then $I^f \cup I^r = I - I^c$. For otherwise, $\sum_{i \in I^f} x^i + \sum_{i \in Ir} x^i < \sum_{i \in I} x^i - \sum_{i \in I^c} x^i$; a contradiction. Hence in this case the converse is true. So suppose $I^0 = I^f \cap I^r \neq \emptyset$. The equality implies that $\sum_{i \in I^0} x^i + \sum_{i \in I^f - I^0} x^i + \sum_{i \in I^r} x^i + \sum_{i \in I^c} x^i = \sum_{i \in I} x^i$.

That is $\sum_{i\in I^0} x^i = \sum_{i\in I-(I^f-I^0)-I^r-I^c} x^i$, as $(I^f - I^0)$, $I^r$, $I^c \subset I$ and all three sets are pairwise disjoint. Now, note that $I^0 \cap [I - (I^f - I^0) - I^r - I^c] = \emptyset$ and let $i^* = \max\{i : i \in I^0 \cup \{I - (I^f - I^0) - I^r - I^c]\}$. Suppose without loss of generality (WLOG) that $i^* \in I^0$. Then $\sum_{i\in I^0} x^i > x^{i^*} > x^{i^*} - 1 \geq \frac{x^{i^*}-1}{x-1} > \frac{x^{i^*}-1}{x-1} - 1 = \sum_{i=1}^{i^*-1} x^i \geq \sum_{i\in I-(I^f-I^0)-I^r-I^c} x^i$; a contradiction. Hence, $I^0 = I^f \cap I^r \neq \emptyset$ cannot hold. This says that $I^0 = \emptyset$ and so $I^f$, $I^r$ and $I^c$ partition $I$. ■

**Remark 17** *Given arbitrary numbers $y_i$, $i = 1, ..., n$, associated with the activities instead of the $x^i$'s, for some $x \geq 2$, the lemma does not hold. That is we could have $\sum_{i\in I^f} y_i + \sum_{i\in I^r} y_i + \sum_{i\in I^C} y_i = \sum_{i\in I} y_i$ yet $I^f \cup I^r \cup I^c \subsetneq I$. For example, if $y_i = i$, for $i \in I = \{1, 2, 3, 4\}$, $I^f = \{1, 2\}$, $I^r = \{1, 2, 4\}$ and $I^c = \emptyset$ then $\sum_{i\in I^f} y_i + \sum_{i\in I^r} y_i + \sum_{i\in I^c} y_i = \sum_{i\in I} y_i$. Yet $I^f \cup I^r \cup I^c \neq I$.*

Note that this lemma would be ideal to use if it weren't for the double-processing issue. Each node of the reduced networks would carry with it the sum of $x^i$ 's, either $\sum_{i\in I^f} x^i$ or $\sum_{i\in I^r} x^i$, where each $i$ would be the index of an activity completed up to that node. When a pair of subpaths one from each network are up for mixed path verification, the set of activities active at both end nodes, $I^c$, is determined. If the total remaining times of any activity in that set exceeds its duration then no mixed path is available. Otherwise, $\sum_{i\in I^c} x^i$ is computed. If the equality in the lemma is satisfied then we have a mixed path at hand. The problem with this scheme, however, is that it detects only partitions of the set of activities, *i.e.,* mixed paths with no double-processing. Unfortunately, though, double-processing is unavoidable. This motivates the following extension of the lemma formulated specifically to deal with this issue.

**Theorem 22** *Let $x \geq 3$ and $I^0 \subset I$. Then, $I = I^f \cup I^r \cup I^c$ and $I^0 = I^f \cap I^r$ iff $\sum_{i \in I^f} x^i + \sum_{i \in I^r} x^i + \sum_{i \in I^c} x^i = \sum_{i \in I} x^i + \sum_{i \in I^0} x^i$ .*

**Proof.** Again, the forward implication is immediate. To prove the converse, suppose that

$$\sum_{i \in I^f} x^i + \sum_{i \in I^r} x^i + \sum_{i \in I^c} x^i = \sum_{i \in I} x^i + \sum_{i \in I^0} x^i. \tag{3.4}$$

We will consider two cases. The first is when $I^f \cup I^r \cup I^c = I$. In that case we have

$$\sum_{i \in I^f} x^i + \sum_{i \in I^r - (I^f \cap I^r)} x^i + \sum_{i \in I^f \cap I^r} x^i + \sum_{i \in I^c} x^i = \sum_{i \in I} x^i + \sum_{i \in I^0} x^i$$

which implies that

$$\sum_{i \in I^f \cap I^r} x^i = \sum_{i \in I^0} x^i, \tag{3.5}$$

since $I^c \cap \big(I^f \cup I^r\big) = \emptyset$. Let $j = \max\{i : i \in I^f \cap I^r \text{ or } i \in I^0\}$. Suppose WLOG that $j \in I^f \cap I^r$. If $j \notin I^0$ then $\sum_{i \in I^f \cap I^r} x^i > x^j > x^j - 1 > \frac{x^j - 1}{x - 1} > \frac{x^j - 1}{x-1} - 1 = \sum_{i=1}^{j-1} x^i \geq \sum_{i \in I^0} x^i$ which is a contradiction. Hence $j \in I^0$. Now simplifying $x^j$ from both sides of (3.5) and repeating the same argument enough times, we conclude that $I^f \cap I^r = I^0$. To sum up, if (3.4) holds and $I^f \cup I^r \cup I^c = I$ then $I^f \cap I^r = I^0$.

In the second case, now, assume that $I^f \cup I^r \cup I^c \subsetneqq I$. By (3.4),

$$\sum_{i \in I^f} x^i + \sum_{i \in I^r - (I^f \cap I^r)} x^i + \sum_{i \in I^f \cap I^r} x^i + \sum_{i \in I^c} x^i = \sum_{i \in [I - (I^f \cup I^r \cup I^c)] \cup I^f \cup [I^r - (I^f \cap I^r)] \cup I^c} x^i + \sum_{i \in I^0} x^i$$

which implies that

$$\sum_{i \in I^f \cap I^r} x^i = \sum_{i \in I - (I^f \cup I^r \cup I^c)} x^i + \sum_{i \in I^0} x^i. \tag{3.6}$$

Note that $(I^f \cap I^r) \cap [I - (I^f \cup I^r \cup I^c)] = \varnothing$ so rewrite (3.6) to get

$$\sum_{i \in (I^f \cap I^r \cap I^0) \cup [(I^f \cap I^r) - (I^f \cap I^r \cap I^0)]} x^i = \sum_{i \in I - (I^f \cup I^r \cup I^c)} x^i + \sum_{i \in (I^0 \cap I^f \cap I^r) \cup [I^0 - (I^0 \cap I^f \cap I^r)]} x^i$$

and simplify to obtain

$$\sum_{i \in (I^f \cap I^r) - (I^f \cap I^r \cap I^0)} x^i = \sum_{i \in I - (I^f \cup I^r \cup I^c)} x^i + \sum_{i \in I^0 - (I^0 \cap I^f \cap I^r)} x^i. \tag{3.7}$$

Note that $[(I^f \cap I^r) - (I^f \cap I^r \cap I^0)] \cap [I - (I^f \cup I^r \cup I^c)] = \emptyset$ and $[(I^f \cap I^r) - (I^f \cap I^r \cap I^0)] \cap [I^0 - (I^0 \cap I^f \cap I^r)] = \emptyset$. Let $i^* = \max\{i : i \in [(I^f \cap I^r) - (I^f \cap I^r \cap I^0)]$ or $i \in [I - (I^f \cup I^r \cup I^c)]$ or $i \in [I^0 - (I^0 \cap I^f \cap I^r)]\}$. If $i^* \in [I - (I^f \cup I^r \cup I^c)]$ then using an argument similar to one used before we conclude that $\sum_{i \in I - (I^f \cup I^r \cup I^c)} x^i > \sum_{i \in (I^f \cap I^r) - (I^f \cap I^r \cap I^0)} x^i$; which violates (3.7). Hence $i^* \notin [I - (I^f \cup I^r \cup I^c)]$. Similarly, $i^* \notin [I^0 - (I^0 \cap I^f \cap I^r)]$.

Now, if $i^* \in [(I^f \cap I^r) - (I^f \cap I^r \cap I^0)]$ then $i \leq i^* - 1$ for all $i \in [I - (I^f \cup I^r \cup I^c)] \cup [I^0 - (I^0 \cap I^f \cap I^r)]$. Hence

$$\sum_{i \in I - (I^f \cup I^r \cup I^c)} x^i + \sum_{i \in I^0 - (I^0 \cap I^f \cap I^r)} x^i \leq 2[\frac{x^{i^*} - 1}{x - 1} - 1].$$

But $2[\frac{x^{i^*} - 1}{x - 1} - 1] < x^{i*}$ since $2[x^{i^*} - 1 - x + 1] < x^{i^*}(x - 1)$, where this latter inequality holds because $3x^{i^*} - 2x < x^{i^* + 1}$ when $x \geq 3$. Hence

$$\sum_{i \in I - (I^f \cup I^r \cup I^c)} x^i + \sum_{i \in I^0 - (I^0 \cap I^f \cap I^r)} x^i < x^{i^*}$$

which says (3.7) cannot hold; a contradiction to the assumption that $i^* \in [(I^f \cap I^r) - (I^f \cap I^r \cap I^0)]$. Thus $i^* \notin [(I^f \cap I^r) - (I^f \cap I^r \cap I^0)]$; which now contradicts the definition of $i^*$. It follows that (3.7) cannot hold so that (3.6) cannot hold so that (3.4) does not either. To sum up the argument, $I^f \cup I^r \cup I^c \subsetneqq I$ implies that (3.4) does not hold. This says that the second case cannot be and so the only case left is the $I^f \cup I^r \cup I^c = I$ case. But we had already seen there that $I^f \cap I^r = I^0$. Hence the converse is proven. ■

Assuming that the Mixed Path Qualification Condition holds, this theorem can be used to detect mixed paths as follows. Say, again, that each node of the reduced networks carries with it $\sum_{i \in I'} x^i$, where $I'$ is the index set of completed activities by that node. To check whether or not two subpaths of end nodes carrying $\sum_{i \in I^f} x^i$ and $\sum_{i \in I^r} x^i$, respectively, constitute a mixed path we determine $\sum_{i \in I^c} x^i$ and $Y = (\sum_{i \in I^f} x^i + \sum_{i \in I^r} x^i + \sum_{i \in I^c} x^i) - \sum_{i \in I} x^i$. By the theorem, a mixed path is at hand *iff* $Y = \sum_{i \in I^0} x^i$ for some $I^0$. Note that we don't even need to determine that $I^0 = I^f \cap I^r$; for if $Y$ is of the required form then the theorem asserts that $I^0 = I^f \cap I^r$and $I = I^f \cup I^r \cup I^c$. Now, to verify that $Y$ may be written in that form, we will assume that $x$ is integer. This makes computing $x^i$ easier and makes the verification task itself simpler. With this assumption, one may apply the following algorithm:

**Step 1 :** Find the largest $r$ such that $Y = x^r Y_1$ for some integer $Y_1$. If $r = 0$ then $Y$ cannot be written in the required form.

**Step 2 :** If $Y_1 \neq 1 \bmod(x)$ then $Y$ cannot be written in the required form. Else if $Y \geq x$, replace $Y$ with $Y_1$ and go to step 1. If $Y < x$ then only if $Y = 1$ does $\sum_{i \in I^f} x^i + \sum_{i \in I^r} x^i - \sum_{i \in I} x^i$ possess the required form.

This algorithm, of course, relies on the fact that $Y = \sum_{i \in I^0} x^i$ *iff* $Y = x^{r_1}(1 + x^{r_2}(1 + x^{r_3}(...)))$, for some positive integers $r_1, r_2, ...$

An alternative to this algorithm is the following:

Consider the sequence 1, $x$, $x^2$, ..., $x^n$.

**Step 1 :** Find the largest $r$ s.t. $x^r \leq Y$. If $x^r = Y$, then stop for $Y$ has the required form.

**Step 2 :** Having determined that $x^r \neq Y$, if $x^r \leq Y - x^r$ then $Y$ cannot have the required form. Otherwise, if $Y \geq x$ then replace $Y$ with $Y - x^r$ and go to step 1. If $Y < x$ then only if $Y = 0$ does $\sum_{i \in I^f} x^i + \sum_{i \in I^r} x^i - \sum_{i \in I} x^i$ possess the required form.

Note that in order to avoid generating large numbers, as much as possible, it is desirable to keep $x$ as small as possible. In other words, the smallest possible value of $x$, 3 that is, should be the one used. Whence the name of the method. We also note that in order to use at most 16 significant digits to represent the powers of 3, which is the number of digits in a double precision representation of real numbers, the largest number of activities in a project instance should be no more than 33 [11]. For larger size instances, the way around this limitation, we believe, is to subdivide all the activities of the project into sets all of which have 33 activities except the last one which could have less than that number. Each of these sets is then treated as before with the information stored at each generated node of the reduced networks taking the form of an array instead of a single number.

**Remark 18** *If $x$ were chosen to be less than 3, namely 2, if $x$ must be integer, then a necessary and sufficient condition that $Y$ be in the required form is that it be divisible by 2. This, of course,*

[11] This is derived by noticing that the largest integer $n$ for which $\sum_{i=1}^{n} 3^i = \frac{3^{n+1}-1}{3-1}$ uses at most 16 digits is $n = 33$.

*eliminates the need for the verification algorithms. But unfortunately, the theorem is not valid for $x = 2$. As a counter example let $I^f = I^r = \{2, ..., n\}$ and $I^c = \emptyset$. Clearly, two subpaths with end nodes corresponding to these three subsets do not constitute a mixed path. But*

$$Y = (2^2 + 2^3 + ... + 2^n) + (2^2 + 2^3 + ... + 2^n) - \sum_{i=1}^{n} 2^i = -2 + 2^2 + 2^3 + ... + 2^n = 2 + 2^3 + ... + 2^n.$$

*That is $Y$ has the required form eventhough no mixed path is possible.*

Observe that besides the additions of powers of 3 at the nodes, which can be computed ahead of time and stored in an array, this method requires, using the second algorithm, $O(\log_3 Y)$ operations to determine whether $Y$ has the required form or not.

**A Prime Numbers Based Method**

In this method the first $n$ prime numbers are given and instead of assigning $x^i$ to activity $i$, for some $x \geq 3$, we assign to it $p_i$, the $i^{th}$ prime number. Then in the same notation and assumptions as the previous method we have the following theorem,

**Theorem 23** *$I = I^f \cup I^r \cup I^c$ and $I^0 = I^f \cap I^r$* iff $\prod_{i \in I^f} p_i . \prod_{i \in I^r} p_i . \prod_{i \in I^c} p_i = \prod_{i \in I} p_i . \prod_{i \in I^0} p_i$.

**Proof.** Again, the forward implication is easy. To prove the converse, assume that

$$\prod_{i \in I^f} p_i . \prod_{i \in I^r} p_i . \prod_{i \in I^c} p_i = \prod_{i \in I} p_i . \prod_{i \in I^0} p_i. \tag{3.8}$$

Since $p_j | \prod_{i \in I} p_i . \prod_{i \in I^0} p_i$ for all $j \in I$, $p_j | \prod_{i \in I^f} p_i . \prod_{i \in I^r} p_i . \prod_{i \in I^c} p_i$. Since $\prod_{i \in I^f} p_i . \prod_{i \in I^r} p_i . \prod_{i \in I^c} p_i$ is a product of primes, $p_j$ must be one of those primes for every $j$. That is $j \in I^f$ or $j \in I^r$or $j \in I^c$. This says that $j \in I^f \cup I^r \cup I^c$. Since $I^f$, $I^r$ and $I^c \subset I$,

$I = I^f \cup I^r \cup I^c$. Now note that $I = I^f \cup I^r \cup I^c = I^f \cup [I^r - (I^f \cap I^r)] \cup I^c$. Then (3.8) implies that

$$\prod_{i \in I^f} p_i . \prod_{i \in I^r \cap I^r} p_i . \prod_{i \in I^r - (I^r \cap I^r)} p_i . \prod_{i \in I^c} p_i = \prod_{i \in I^f} p_i . \prod_{i \in I^r - (I^r \cap I^r)} p_i . \prod_{i \in I^c} p_i . \prod_{i \in I^0} p_i .$$

That is

$$\prod_{i \in I^r \cap I^r} p_i = \prod_{i \in I^0} p_i .$$

Since $p_i$ is prime for all $i \in I^0$ and all $i \in I^r \cap I^r$, we conclude that $I^0 = I^f \cap I^r$. ■

This theorem could be used in much a similar way as Theorem 22. That is, the product of primes corresponding to the completed activities are stored at nodes of the reduced networks. To verify whether a mixed path is at hand, we first verify that every common activity to the end nodes of the subpaths is sufficiently processed, *i.e.,* we verify the Mixed Path Qualification Condition. If the condition is verified, we perform a product and division: $\frac{\prod_{i \in I^f} p_i . \prod_{i \in I^r} p_i . \prod_{i \in I^c} p_i}{\prod_{i \in I} p_i}$. The result is an integer *iff* we have a mixed path. The trouble with this approach, though, is that the product of primes can grow very fast. In fact the product of the first $n$ prime numbers is much larger than $n!$. If a way to mitigate this difficulty is possible, however, then this method should be reasonable. Note that it may seem that the difficulty can be eased by using the logarithm of the prime numbers instead of the prime numbers themselves. This idea is formalized in the following corollary to the theorem which proof is simple enough for us to omit.

**Corollary 24** *In the same notation as the theorem, let* $\alpha = \sum_{i \in I^f} \ln p_i + \sum_{i \in I^r} \ln p_i - \sum_{i \in I} \ln p_i$. *Then* $I = I^f \cup I^r \cup I^c$ *and* $I^0 = I^f \cap I^r$ iff $e^\alpha$ *is integer. Moreover,* $I^f$, $I^r$ *and* $I^c$ *partition* $I$ iff $\alpha = 0$.

The logarithms of the $p_i$'s, of course, should be computed beforehand and not at every node, just like $x^i$ in the previous method. Therefore, this is not the difficult issue in applying the corollary. The issue is the precision with which the logarithmic function needs to be computed. The following analysis sheds the light on what we mean.

Say $\varepsilon$ is the error in computing $\ln x$, for any $x > 0$, and $\eta$ is the error in computing $e^x$. Clearly, the error in computing the $\alpha$ in the corollary is at most $3n\varepsilon$. Using the notation $\bar{x}$ for the computer representation of $x$, we have

$$\bar{\alpha} - 3n\varepsilon < \alpha < \bar{\alpha} + 3n\varepsilon.$$

Hence

$$e^{\bar{\alpha}-3n\varepsilon} < e^{\alpha} < e^{\bar{\alpha}+3n\varepsilon}.$$

But

$$\overline{e^{\bar{\alpha}}} - \eta < e^{\bar{\alpha}} < \overline{e^{\bar{\alpha}}} + \eta.$$

Therefore

$$(\overline{e^{\bar{\alpha}}} - \eta).e^{-3n\varepsilon} < e^{\alpha} < (\overline{e^{\bar{\alpha}}} + \eta).e^{3n\varepsilon}.$$

It follows that if we want $\overline{e^{\bar{\alpha}}}$ to be in an interval around $e^{\alpha}$ of length at most $\theta$, to decide whether $e^{\alpha}$ is integer or not, then we want

$$(\overline{e^{\bar{\alpha}}} + \eta).e^{3n\varepsilon} - (\overline{e^{\bar{\alpha}}} - \eta).e^{-3n\varepsilon} < \theta.$$

That is we want

$$
\begin{aligned}
e^{\bar{\bar{\alpha}}} &< \frac{\theta - \eta(e^{3n\varepsilon} + e^{-3n\varepsilon})}{e^{3n\varepsilon} - e^{-3n\varepsilon}} \\
&< \frac{\theta - 2\eta}{6n\varepsilon} \\
&< \frac{\theta}{6n\varepsilon}
\end{aligned}
$$

as can be verified by expanding the exponentials and as $\eta$ is supposed to be small. Now, as can be seen, the smaller the error in computing the logarithmic function, the larger $e^{\bar{\bar{\alpha}}}$ can be. But the trouble is that the fraction doesn't decrease fast enough in terms of $\varepsilon$. For example, if $\theta = 10^{-2}$, $n = 100$ and $\varepsilon = 10^{-6}$ then $\bar{\alpha}$ can be at most about 2.81. If $\varepsilon = 10^{-8}$ then $\bar{\alpha}$ can be at most about 7.41. While if $\varepsilon = 10^{-12}$ then $\bar{\alpha}$ can approach 16.63 at the most ... Obviously, such a range for $\alpha$ would be too small for even small size project instances. Whence, the logarithmic approach to solving the large products issue is not appealing at all.

**A Semi-direct Method**

Again assuming that the Mixed Path Qualification Condition holds, this method is based on the following lemma.

**Lemma 25** *Let $U^f = \{u_1, ..., u_m\}$ and $V^r = \{v_1, ..., v_p\}$ be the sets of activities, whether active or precedence feasible, of the last states of two subpaths, $\mathcal{P}^f$ and $\mathcal{P}^r$, of the reduced forward and reverse state networks, respectively. Then $\mathcal{P}^f$ and $\mathcal{P}^r$ constitute a mixed path* iff *$u_i$ does not precede $v_j$ for any $i = 1, ..., m$ and any $j = 1, ..., p$, or equivalently,* iff *$v_j$ does not succeed $u_i$ for any $i = 1, ..., m$ and any $j = 1, ..., p$.*

**Proof.** We will prove only the first necessary and sufficient condition as the second is an obvious consequence of the first and vice versa.

Suppose that the two subpaths constitute a mixed path. If $u_i$ precedes $v_j$ for some $i$ and $j$ then $u_i$ has not been completed in the subschedule yielded by $\mathcal{P}^f$, as it appears in the state $U^f$, and furthermore $u_i$ has not been completed in the reverse subschedule yielded by $V^r$, as $u_i$ precedes a yet unstarted activity $v_j$ in that subschedule. In other words, $u_i$ has not been completed by either subpaths. This says that $u_i$ is not part of the mixed path; a contradiction.

Conversely, suppose that $u_i$ does not precede any $v_j$ for any $i = 1, ..., m$ and any $j = 1, ..., p$. Observe that the set of completed activities by the two subpaths is the union of the complements of two sets. The first is the set activities in $U^f$ and their successors and the second is the set of activities in $V^r$ and their predecessors. Let $w$ be an activity of the project instance. If $w$ is not completed in $\mathcal{P}^f$ then $w \in U^f$ or else $w$ is a successor to an activity in $U^f$. In both cases $w$ does not precede any activity of $V^r$. Hence $w$ belongs to the complement of the set formed by the activities of $V^f$ and their predecessors. Hence $w$ is completed. Since $w$ is arbitrary, we conclude that all activities of the instance are completed. Hence the two subpaths constitute a mixed path. ■

Note that implementing the lemma to check whether $\mathcal{P}^f$ and $\mathcal{P}^r$ form a mixed path requires $O(mp + rs)$ operations, where $r$ and $s$ are the cardinalities of the set of active activities at the end nodes of the two subpaths, respectively. It also does not require storing the activity status information for each activity and at each node as the direct method requires. Additionally, the implementation of the lemma could be preceded by simple to implement rules such as:

If the total consumption of a resource in the two subpaths is less than the total

that the project instance requires, then the two subpaths do not constitute a mixed path.

A similar condition could be formulated with respect to the time requirement of the project instance whereby, in the above rule, time is considered as any other resource, albeit an unlimited one. Note that implementing such rules requires relatively little additional memory space and little computational time but helps rule out obvious cases where a mixed path does not exist.

We note also that eventhough this method is mathematically less involved than the previous two, it suffers not from their limitations. Namely, this method is not nearly as sensitive to an instance size, in terms of number of activities, as the previous two. *For this reason we choose it as the means to identify mixed paths rather than any of the other two.*

### 3.7.2 Optimality Condition

Again, given that it is a mixed path that which will ultimately yield an optimal schedule, and given that we now have established how to recognize a mixed path, we need to address the issue of recognizing an "optimal" mixed path. Namely, what is the criterion, or perhaps criteria, that we should look for in a mixed path so that it qualifies as optimal. The length of a mixed path defined as the duration of the schedule induced by the mixed path might be the obvious criterion. This, however, is complicated by the possibility, indeed likelihood, of over- and double-processing and the way to resolve them[12]. For, as the following example illustrates, several schedules with different durations could be derived from the same mixed path.

[12] As the name may suggest, double processing is the special case of over processing where an activity is started and completed in each subpath of the mixed path.

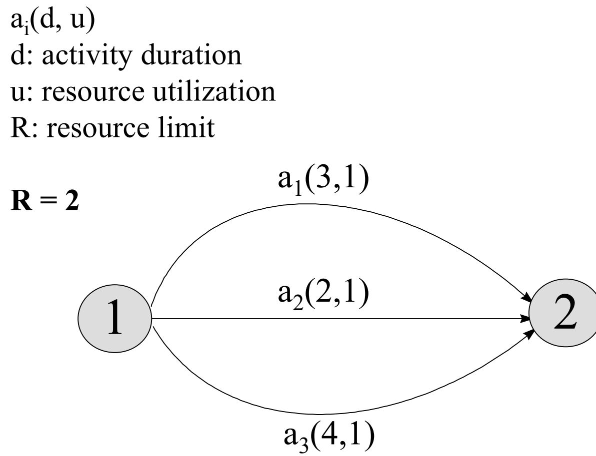


Figure 3-12: The project instance of Example 26 - AoA representation

**Example 26** *Consider the project instance in Figure 3-12 and its forward and reverse subpaths* $\{start, S^f\}$ *and* $\{end, S_1^r, S_2^r\}$*, where* $S^f = \{a_1, a_3\}$*,* $S_1^r = \{a_2, a_3/1\}$ *and* $S_2^r = \{a_2/1\}$*. Note that the length of the forward subpath is* 2*, corresponding to the duration of* $a_2$*, while the length of the reverse subpath is* $3 + 1 = 4$*, corresponding to the duration of* $a_1$ *and the remaining duration of* $a_3$ *in* $S_2^r$*. Furthermore, the two subpaths determine a mixed path. Clearly, activity* $a_2$ *is over-processed. To deduce a schedule from the mixed path, one can either adopt the start time of* $a_2$ *in the forward subpath as its scheduled start time or else adopt the finish time of* $a_2$ *in the reverse subpath as its scheduled finish time. But the first alternative induces a schedule of makespan* 6 *while the second leads to a schedule makespan of* 5 *!*

In light of this example now, are we to generate all schedules induced by a mixed path so that the mixed path length could be defined as the minimum duration of these schedules?[13] Or are we to somehow deduce that shortest duration indirectly? Fortunately, it turns out that

[13] If $p$ activities of a mixed path are over- or double- processed, the number of such schedules could be $2^p$.

these complications can be dealt with by defining the length of a mixed path in the following way.

Given a mixed path with end states $S^f$ and $S^r$ in the forward and reverse reduced state networks, respectively, with distances $d_f$ and $d_r$ from their immediate predecessor states, let the **forward over-processing** of the mixed path be the minimum of $d_f$ and the over-processings of the activities of $S^f$. Similarly, let the **reverse over-processing** be the minimum of $d_r$ and the over-processings of the activities of $S^r$. Now define the **mixed path length** as the sum of the lengths of its constituent subpaths minus the maximum of the forward and reverse over-processings.

It is important to note here that our choice of a state network as the medium in which the mixed paths are to be found compels this somewhat elaborate definition of a mixed path length. For had we instead chosen the state graphs of an instance as that medium then a mixed path's length simply defined as the sum of lengths of the constituent subpaths would have been amply sufficient. The reason for deducting the maximum of the forward and reverse over-processings when considering reduced state networks, or even plain state networks, is that in such networks states are only generated at the completion of activities. This naturally causes activity over-processing for some mixed paths. In fact, for some instances, there may not exist a mixed path of minimum possible length, according to our definition, which does not include some forward or reverse over-processing. Were it not for deducting the over-processing in these instances, chances are the optimal makespan would never be found. At any rate, the next lemma establishes the adequacy of this definition of a mixed path's length for our purpose. It makes use the following observation.

Consider a feasible schedule to a project instance of makespan $\Delta$ and let $t \in [0, \ \Delta]$ . It is quite possible to locally and globally left-shift all activities of the schedule that are completed by time $t$ and locally and globally right-shift all activities started after time $t$. It is also possible to arbitrarily either left-shift or right-shift, locally and globally, all remaining activities just as well. In that sense, the resulting schedule could be seen as left-active up to time $t$ and right-active thereafter. We will call such schedules **$t$-active**. Notice that a derived $t$-active schedule cannot have makespan larger than that of the original schedule. Notice also that $t$-active schedules originate in mixed paths identified in a process that explores shortest paths from the start node of the reduced forward state network and the end node of its counterpart the reduced reverse state network in an alternating fashion. They are the result of eliminating the over- and double-processing of activities in such mixed paths in a feasible way.

We are now ready to state and prove the lemma.

**Lemma 27** *All feasible schedules derivable from a minimum length mixed path of the two reduced state networks of a project instance are of minimum makespan which also is the mixed path length.*

**Proof.** Note first that an upper bound on the makespan of any feasible schedule derived from a mixed path is the length of the mixed path itself. In particular, this is true of minimum length mixed paths. Suppose now that a feasible schedule of makespan smaller than the minimum length mixed path could be derived, whether via a minimum length mixed path or any other means. Consider a time $t$ before the completion this schedule and transform the schedule into a $t$-active one as described in the previous discussion. The new $t$-active schedule can now be seen to originate in a mixed path of length equal to the smaller makespan. But this contradicts

the minimality of the original mixed path. Hence a lower bound on the makespans of feasible schedules is the minimum of mixed path lengths. It follows that a schedule derived from a minimum length mixed path has for makespan the mixed path length and is of minimum makespan. This implies that all schedules derived from a minimum length mixed path have the same makespan. ■

In other words, this lemma asserts that over- and double-processing of activities in a minimum length mixed path, as we defined it, can be eliminated without the need to devise a special way to deduce an optimal schedule. The over- and double-processing can be resolved almost arbitrarily as the only restriction that needs to be considered is the feasibility of the resulting schedule. Moreover, this feasibility concerns the precedence relationships only; as eliminating an activity, or part of it, in a mixed path can not result in a resource limit violation. In this regard, note that the over-processing of an activity that is active at both the last nodes of the two subpaths of a mixed path can be eliminated by adopting either the start time of the activity in the forward subpath or its completion time in the reverse subpath as the reference to its processing time slot in the resulting schedule.

Now, having identified the minimum length mixed path as the means to deducing an optimal schedule, it remains to specify how to obtain one. But first we need some more observations.

We have mentioned that $t$-active schedules originate in mixed paths generated in a process that explores shortest paths from the start node of the reduced forward state network and the end node of its counterpart the reduced reverse state network in an alternating fashion. A natural question that arises in this process is the following. Once a mixed path is identified, should we investigate the length of the mixed path derived by extending one of the two subpaths while shortening the other in the hope that the new subpath would be of smaller length? In

other words, could eliminating the activities completed last in one of the subschedules and adding them to the other result in a smaller schedule makespan? In the terminology we have been using, this question is about transforming a $t$-active schedule into a $(t \pm i)$-active one, for some appropriate $i$, and the effect the transformation can have on the makespan of the resulting schedule.

It is not hard to see that this question cannot be settled in general terms. The transformation may or may not result in makespan reduction depending on the mixed path and the parameters of the project instance. But regardless, our focus is only on minimum length mixed paths which, by virtue of their minimality, a schedule they yield can be transformed from being $t$-active into becoming $t'$-active, for any feasible $t'$, while maintaining the minimality of the resulting schedule. To see this, recall that transforming any schedule, including a $t$-active one, into a $t'$-active schedule can not result in an extension of the schedule makespan. In particular then, a minimum makespan schedule is not extended by modifying its "$t$-characteristic"; which simply says that such a modified schedule maintains its optimality.

This observation about minimum length mixed paths can be exploited as follows. Suppose that while alternating between the two reduced networks, we identify a mixed path of suboptimal length and that we are contemplating extending one of its subpaths at the expense of the other in the hope of reaching a mixed path of minimum length. The observation in that case simply points out that if the transformed mixed path were optimal then the original one must have been so too. That is, since the original mixed path is suboptimal, its transformations cannot be optimal.

We note that this observation that does not render needless the checking for mixed path status of one end of an established mixed path against the immediate predecessor of the other

end. Such checking does not fall under the category of "$t$-characteristic" modification and in fact is necessary when iteratively and alternately developing the reduced networks; for example, in the case where mixed path completion hinges upon the completion of one activity in one of the subpaths and the opposite subpath is extended by states solely containing activities already completed in the first subpath.

We now are ready to establish how to obtain a minimum length mixed path. Our intention is: (1) to label the nodes of the reduced state networks as in Dijktsra's labeling algorithm, (2) to develop the successors of a state as soon as the state is labeled and (3) to check, directly or indirectly, for mixed path the newly generated states, as soon as they are generated, against all developed stated of the opposite network so as to continuously update the minimum length mixed path at hand.

**Theorem 28** *In a process that attempts to determine a minimum makespan schedule to an RCPSP instance by iteratively and alternately exploring shortest paths of the two reduced networks using Dijkstra's labeling algorithm as described above, the mixed path with minimum length at hand when the first mixed path with both ends labeled is identified qualifies as a minimum length mixed path for the instance.*

**Proof.** We start by setting the notation $(W^f, W^r)$ for the mixed path with end state $W^f$ in the forward reduced network and end state $W^r$ in the counterpart reverse reduced network. We reserve the notations $l^f(W^f)$ and $l^r(W^r)$ for the labels of $W^f$ and $Wr$ in their respective networks..

Let $(S^f,S^r)$ be the first mixed path obtained where both states $S^f$ and $S^r$ are labeled in their respective networks. Suppose that none of the mixed paths uncovered by the time $S^f$ and $S^r$ are

labeled, including $(S^f,S^r)$, is a minimum length mixed path for the instance. Let $U_0, U_1, \ldots, U_p$ and $V_0, V_1, \ldots, V_q$ be two subpaths in the forward and reverse reduced networks, respectively, that actually do constitute a minimum length mixed path[14]. For simplicity, assume that both the forward and reverse over-processings of both mixed paths are zero. If this does not hold for one then back off from the state corresponding to the largest of the two over-processings a distance equal to the over-processing and include in the appropriate network the corresponding new state[15]. The lengths of $(S^f,S^r)$ and $(U_p, V_q)$ are then assumed to be $l^f(S^f)+l^r(S^r)$ and $l^f(U_p)+l^r(V_q)$, respectively.

Since $(U_p, V_q)$ is a minimum length mixed path, we can transform any schedule it yields into a $t$-active one for any $t$ not exceeding the mixed path length. In particular, we can choose $t = l^r(V_j)$, $j = 0, 1, \ldots, q$. With a fully developed reduced forward network, each $l^r(V_j)$-active schedule corresponds to a mixed path $(U_{p+j}, V_{q-j})$, where $l^f(U_{p+j}) = l^f(U_p)+l^r(V_q)-l^r(V_{q-j})$. We note that $U_{p+j}$ may not be generated from $U_{p+j-1}$ due to the possibility that $U_{p+j-1}$ contains an activity with a remaining time strictly less than $l^f(U_{p+j}) - l^f(U_{p+j-1})$. In a similar way, we can choose to transform a schedule yielded by $(U_p, V_q)$ into an $l^r(U_i)$-active for any $i = 0, 1, \ldots, p$. Again, with a fully developed reduced reverse network, each such schedule corresponds to a mixed path $(U_{p-i}, V_{q+i})$, where $V_{q+i}$ may not be generated from $V_{q+i-1}$ and $l^r(V_{q+i}) = l^f(U_p)+l^r(V_q)-l^f(U_{p-i})$.

Note that $S^f$ cannot be the source node of the forward network. Otherwise, $S^r$ would be the terminal node of the reverse network and so $(S^f,S^r)$ would be a minimum length mixed path.

[14] $U_0$ and $V_0$ are assumed here to be the source nodes of their respective networks.

[15] This new state would be derived form the corresponding subpath end state by simply adding the over processing to the remaining times of the active activities. The result would indeed be a new state as otherwise, due to our definition of over processings, it would have to be the immediate predecessor of said end state. This cannot be as $(S^f,S^r)$ would not have been the first mixed path with both end states labeled as we assumed it is.

Similarly, $S^f \neq U_{p+q}$, the terminal node of the forward network, and $S^r \neq V_0,\ V_{p+q}$. It follows that in the fully developed reduced networks there exist $m$ and $m'$, $0 < m, m' < p+q$, such that

$$l^f(U_m) \leq l^f(S^f) < l^f(U_{m+1}) \tag{3.9}$$

$$l^r(V_{m'}) \leq l^r(S^r) < l^r(V_{m'+1}). \tag{3.10}$$

Three cases for $l^f(S^f)$ and $l^r(S^r)$ can now be distinguished.

**case 1:** $l^f(U_m) < l^f(S^f) < l^f(U_{m+1})$ and $l^r(V_{m'}) < l^r(S^r) < l^r(V_{m'+1})$.

We know that

$$l^f(S^f) + l^r(S^r) > l^f(U_i) + l^r(V_{p+q-i}),\ \text{for all } i = 0, 1, ..., p+q. \tag{3.11}$$

Since *($S^f$,$S^r$)* is the first mixed path detected with both ends labeled, $l^f(S^f) > l^f(U_i)$ and $l^r(S^r) > l^r(V_{p+q-i})$ cannot hold simultaneously. This says that either $l^f(S^f) > l^f(U_i)$ and $l^r(S^r) < l^r(V_{p+q-i})$, or else $l^f(S^f) < l^f(U_i)$ and $l^r(S^r) > l^r(V_{p+q-i})$. In the first subcase, $l^f(S^f) > l^f(U_i)$ implies that $i \leq m$ whereas $l^r(S^r) < l^r(V_{p+q-i})$ implies that $p+q-i \geq m'+1$. Hence for that subcase, $i \leq \min(m, p+q-m'-1)$. Similarly for the second subcase, $i \geq \max(m+1, p+q-m')$.

Note that $\min(m, p+q-m'-1) = p+q-m'+1$ *iff* $\max(m+1, p+q-m') = m+1$. If $m+1 \geq p+q-m'$ we deduce that either $i \leq p+q-m'-1$ or else $i \geq m+1$. On the other hand, if $m+1 \leq p+q-m'$ then either $i \leq m$ or else $i \geq p+q-m'$. In both cases, unless $m+1 = p+q-m'$ there exists at least one $i$, $p+q-m'-1 < i < m+1$

or $m < i < p+q-m'$, for which (3.11) does not hold. Since $i$ spans $0, 1, ..., p+q$ , we conclude that $m+1 = p+q-m'$. Therefore,

$$\begin{aligned} l^f(U_m) &< l^f(S^f) < l^f(U_{m+1}) \\ l^r(V_{p+q-m-1}) &< l^r(S^r) < l^r(V_{p+q-m}). \end{aligned}$$

Note now that if $m < p$ then $U_{m+1}$ is generated from $U_m$ as soon as the latter is labeled. It follows that by the time $S^f$ and $S^r$ are labeled $U_{m+1}$ is generated and the mixed path $(U_{m+1}, V_{p+q-m-1})$ is detected. If $m \geq p$ then $V_{p+q-m}$ in turn is generated from $V_{p+q-m-1}$ as soon as $V_{p+q-m}$ is labeled. So there too, by the time $S^f$ and $S^r$ are labeled $V_{p+q-m-1}$ is generated and the mixed path $(U_m, V_{p+q-m})$ is detected. But both of these mixed paths have length strictly less than that of $(S^f, S^r)$; a contradiction to the assumption that a minimum length mixed path is not at hand by the time $(S^f, S^r)$ with both ends labeled is detected. Hence this case is vacuous.

**case 2:** $l^f(U_m) = l^f(S^f)$ and $l^r(V_{m'}) \leq l^r(S^r) < l^r(V_{m'+1})$.

In this case, $l^f(S^f) + l^r(S^r) > l^f(U_m) + l^r(V_{p+q-m})$ implies that $l^r(S^r) > l^r(V_{p+q-m})$. Further, $l^f(U_m) = l^f(S^f)$ implies that $U_m$ was generated before $S^f$ was labeled as distances in the reduced forward network are positive. Therefore, the mixed path $(U_m, V_{p+q-m})$ must have been detected before $(S^f, S^r)$ was; a contradiction to the assumption that a minimum length mixed path is not at hand by the time both ends of $(S^f, S^r)$ are labeled. Hence this case too is impossible.

**case 3:** $l^f(U_m) \leq l^f(S^f) < l^f(U_{m+1})$ and $l^r(S^r) = l^r(V_{m'})$.

Obviously, this case is similar to the previous one and so is itself unrealizable too.

Since all three cases cannot occur, (3.9) and (3.10) do not hold and ultimately our assumption that a minimum length mixed path is not at hand by the time both ends of $(S^f, S^r)$ are labeled does not hold. ■

Clearly, this theorem can be used as a stopping rule in a bidirectional approach to solve RCPSP. We will do so but in a slightly indirect way. Instead of directly verifying whether or not two ends of a mixed path are labeled, we propose to store and update as necessary two variables, one for each network, representing the smallest source to node distance found for which the node and its predecessors determine a mixed path with a labeled node of the opposite network and its predecessors. Now when a node of one network is labeled with the same such smallest source to node distance of its network, the stopping rule is satisfied.

**Remark 19** *This approach to RCPSP might suggest a parallel one to solve the Shortest Path Problem, a much simpler type of problems. In fact, a bidirectional approach to Shortest Path based on Dijkstra's algorithm has been mentioned in Minieka and Evans (1989)* [16]*. The reader familiar with that approach will notice that there is no need to consider two networks, develop the concept of mixed path and come up with a means to verify its existence. Nevertheless, the basic idea behind that approach can be seen as a much simplified version of our approach here.*

We end this subsection with recalling that it is perfectly reasonable to adopt a definition of a mixed path whereby every activity must be completed in one of the subpaths not just 'jointly sufficiently processed'. Of course, then, a mixed path in the second sense is one in the first sense as well. We also note that all the machinery we have subsequently developed around this

[16] *The theorem specifying the stopping rule contains quite a misleading typo, though, and no proof is provided.*

concept would still be adaptable with minimal effort to the new concept. In fact this machinery is simpler with the new concept. The only reason we choose to adopt the first sense, however, is that a mixed path in this second sense can not be detected before a mixed path in the first sense; which makes an algorithm using it run through more iterations than it absolutely has to.

### 3.7.3 Outline of an Algorithm

We close this section by outlining our bidirectional approach to solve RCPSP, which can very simply be adapted to solve Preemptive-Resume RCPSP also. We reserve the notation $N^f$ for nodes of the reduced forward state network, the notation $S^f$ and $\bar{S}^f$ for nodes of that same network, the notation $d^f(N^f)$ for the smallest known length of a path from the start node of the forward reduced network to $N^f$ and the notation $l^f(S^f)$ for the label of $S^f$ in the forward network, *i.e.,* the length of the absolute shortest path from the start node of the forward reduced network to $S^f$. We also extend this notation to cover the same entities in the context of the reduced reverse network by replacing the superscript " $f$ " with the superscript " $r$ ". Lastly, let $t^f$ be the smallest source to node distance in the forward network for which a mixed path is detected with a labeled or smallest distance node (*i.e.,* about to be labeled) of the reverse network. Let $t^r$ be the similar variable corresponding to the reverse network. Given an RCPSP instance, our algorithm's outline to solve it is then as follows:

**Step 0**: Label the start node of the reduced forward network and the start node of the reduced reverse network, which corresponds to the end node of the instance, with 0. Set $t^f = t^r = M$, a large number.

**Step 1**: Let $S^f$ be the last labeled node of the forward network. Compute $d$ as the minimum

of the residual durations of the continuing activities of $S^f$ and the durations of precedence feasible ones at the node. Generate all $d$-submaximals of $S^f$. For each $d$-submaximal, generate the corresponding successor state and test it for fathoming. If $N^f$ is an unfathomed successor node to $S^f$ induced by the processing of a $d$-submaximal and if $N^f$ does not already exist then $d^f(N^f) = l^f(S^f) + p$, where $p$ is the smallest of the processing times of the activities of the $d$-submaximal and the residual times of its continuing activities. Else, if $N^f$ already exists then $d^f(N^f) = \min\{l^f(S^f) + p, d^f(N^f)\}$.

**Step 2**: Directly or indirectly, test all newly generated states of the reduced forward network against all labeled states of the reduced reverse network for mixed path. Update $t^f$ and the minimum length mixed path if necessary. Let $\bar{S}^f$ be an unlabeled node of the reduced forward network of minimum $d^f$, ties broken arbitrarily. Label $\bar{S}^f$ by letting $l^f(\bar{S}^f) = d^f(\bar{S}^f)$. If $l^f(\bar{S}^f) = t^f$ then a mixed path is at hand so go to step 5.

**Step 3**: Let $S^r$ be the last labeled node of the reverse network. Compute $d$ as the minimum of the residual durations of the continuing activities of $S^r$ and the durations of reverse precedence feasible ones at the node. Generate all $d$-submaximals of $S^r$. For each $d$-submaximal, generate the corresponding reverse successor state, *i.e.* predecessor, and test it for fathoming. If $N^r$ is an unfathomed reverse successor node to $S^r$ induced by the processing of a $d$-submaximal and if $N^r$ is newly generated then $d^r(N^r) = l^r(S^r) + p$, where $p$ is the smallest processing time of any of the activities of the $d$-submaximal and the residual times of its continuing activities. Else, if $N^r$ already exists then $d^r(N^r) = \min\{l^r(S^r) + p, d^r(N^r)\}$.

**Step 4**: Directly or indirectly, test all newly generated states of the reduced reverse network

against all labeled states of the reduced forward network for mixed path. Update $t^r$ and the minimum length mixed path if necessary. Let $\bar{S}^r$ be an unlabeled node of the reduced reverse network of minimum $d^r$, ties broken arbitrarily. Label $\bar{S}^r$ by letting $l^r(\bar{S}^r) = d^r(\bar{S}^r)$. If $l^r(\bar{S}^r) = t^r$ then a mixed path is at hand so go to step 5. Go to Step 1.

**Step 5**: Eliminate the over and double-processing of activities in the optimal mixed path in any way that satisfies the precedence relationships. The result is a feasible minimum makespan schedule.

The action of this algorithm on our project instance in Figure 3-1 is depicted in Figure 3-13. The stopping condition is satisfied when node $\{a_2\}$ in the reverse network is labeled and the mixed path is detected. The optimal makespan is indeed 11, as before. But notice that the total number of nodes generated is 8, instead of 7 for the reduced forward state network and 2 for the reduced reverse state network. We don't think this will be the case for large examples, though. We believe there is much to save with this method when larger size examples are solved.

## 3.8 Further Reductions

Recall that the aim in section 3.4 was to significantly reduce the size of the state networks we need to consider. This was accomplished by use of Corollary 15 and Theorem 17. Our rationale was that the smaller the number of nodes generated the more efficient a search algorithm would be. But looking at the outline of the bidirectional algorithm, one may point out to the number of mixed path verifications as an additional potential source of computational difficulty. In this section, we discuss further means to reduce the number of nodes generated as well as some

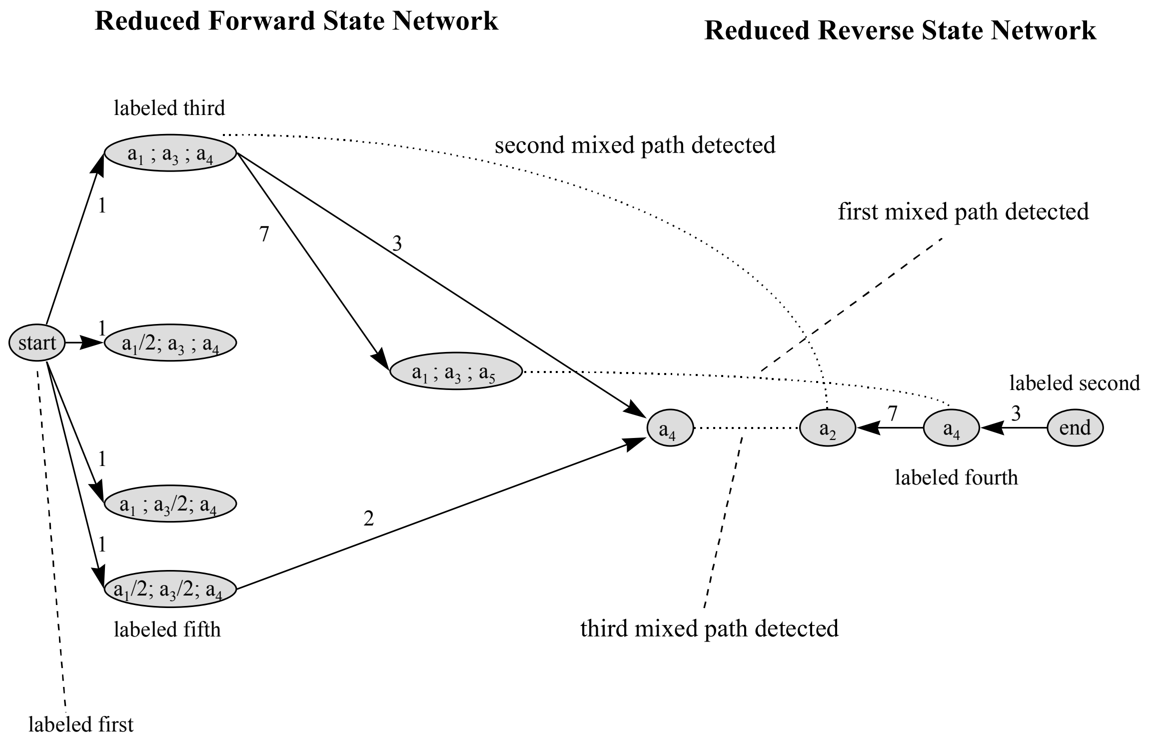


Figure 3-13: The Bidirectional Approach solving the project instance in Figure 3-1.

simple rules to decrease the number of mixed path verifications. Most definitely, these are not the only rules that could be considered. In fact, we have already mentioned several others in our literature review.

**Lower Bound/Upper Bound Rule**

This is the basic rule to fathom nodes in the Branch and Bound methodology. Given an upper bound on the optimal makespan of an RCPSP instance and a state of the forward or reverse reduced networks, a lower bound on the makespans derived from the mixed paths passing through the state is computed. If the lower bound is not strictly smaller than the upper bound at hand, the state is fathomed. Naturally, the upper bound on the makespan can be repeatedly updated as smaller length mixed paths are detected by the algorithm. This causes the rule to be more efficient as the algorithm progresses until the optimal makespan is reached at which time the efficiency levels off till optimality is proven.

A variation on this rule, for the reduced forward network say, is to compute the latest start of the activities given the upper bound on the makespan. Now, if a state contains an activity with a latest start strictly smaller than the source to state distance then said state is fathomed. Of course, a similar rule can be devised for the reduced reverse network.

**Cutset Dominance Rule**

This is the dominance rule used by Demeulemeester and Herroelen (1992) in their DH and DH1 procedures. In our context, it goes as follows. Given a state $S$, the cutset determined by $S$ and denoted by $\mathcal{C}(S)$ is defined as the set of all unscheduled activities of the instance which either belong to $S$ or, if not, have their predecessors either completed or active in $S$. Now, if

two states $S_1$ and $S_2$, with respective distances $t_1$ and $t_2$ from the source node of their network, are such that (1) $\mathcal{C}(S_1) = \mathcal{C}(S_2)$, (2) $t_1 \geq t_2$, and (3) $t_2 +\ r(S_2, j) \leq t_1 + r(S_{1,}j)$, where $r(S, j)$ is the remaining time for activity $j$ in state $S$ (considered to be 0 if $j$ has been completed by time $t_1$ ) then $S_1$ can be fathomed.

**Mixed Path Verification Reduction Rules**

One simple rule to reduce the number of mixed path verifications is to only test end nodes with respective source to node distances that add up to the makespan lower bound, at the least.

A similar and previously mentioned rule is to only test subpaths with accumulated resource consumptions adding up to the cumulative resource requirements of the instance. This, of course, requires storing with each state the different resource consumptions accrued by the time the state is reached.

## 3.9 Implementation Issues and Decomposition

We have seen in previous sections that RCPSP can and probably should be conceived as a shortest path problem in which the objective is to find a shortest mixed path between the start and end nodes of two 'reduced state networks'. Our motivation and intuition, recall, was that such a shortest path, or mixed path, can ultimately be found based on an extension, however involved, of Dijkstra's labeling algorithm. In this section we start by presenting an overview of single source shortest path algorithms and their implementations. Next, we present our implementations of the bidirectional algorithm for the single and multiple durations cases together with possible data structures supporting them. We end with a discussion of decomposition approaches to the problem and the potential for multiple or parallel processing.

### 3.9.1 Brief Survey of Relevant Shortest Path Algorithms for the Arbitrary Arc Lengths Case

We begin our survey by recalling that Dijkstra's algorithm for Shortest Path (Dijkstra 1959) is an algorithm for graphs with undirected edges having non-negative weights, or lengths. Therefore, in view of the reduced state networks being special cases of such graphs, namely being directed and acyclic, the question justifiably arises as to whether Dijkstra's algorithm is the most efficient algorithm to adapt in the first place. After all, an algorithm for Shortest Path in directed acyclic graphs that is linear in the number of nodes and arcs is well known[17]; so why not adopt it instead as our basis? The answer to this concern is that the special case linear algorithm, and most likely any other algorithm for that particular case, relies on either the nodes of the network being topologically sorted or, if not, on topologically sorting them[18]. But unfortunately, while topological sorting is linear in the number of nodes and arcs, one must have determined the whole network beforehand in order to sort it. Indeed it is quite feasible, while generating nodes, to have one node point to another which was generated and sorted much earlier. Not knowing what nodes are to be ultimately generated and how they might relate to previous nodes, it is generally impossible to execute a topological sort.

Essentially, what a shortest path algorithm suited for a network under construction must possess is a mechanism whereby once a shortest path to a node in the partially generated network has been determined, this 'so far' shortest path retains its minimality even when the network is complete. In the case of Dijkstra's algorithm, this mechanism is realized by labeling, or scanning as sometimes it is referred to, the nodes of the graph. The nodes are labeled in non

[17] See Cormen *et al.* (1990), Chapter 25, for example.

[18] Topological sorting of a network refers to ranking the nodes of the network in a way that any node has for sucessors higher ranking nodes only.

decreasing order of the shortest known distances to all nodes. Once a node is labeled then any other subsequently labeled node cannot have a lower label. Thus, since arc lengths are assumed non-negative, a node that thereafter points to the first cannot determine a shorter distance to it than what has already been established. This is one major reason why Dijkstra's algorithm is suitable for our purposes.

The second major reason is that Dijkstra's algorithm has proven to be, in theory and practice and until the very recent past, the most efficient algorithm for Shortest Path. Indeed, while other algorithms for Shortest Path with non-negative edge weights are known, the research into the problem for the last fifteen years or so has mostly focused on efficient implementations of Dijkstra's algorithm. It is worthwhile noting here that Dijkstra's original implementation is of complexity $O(n^2)$, where $n$ is the number of nodes of the graph under consideration. At that time though, 1959, 'efficient implementation' and 'computational complexity' were not part of the lexicon. Subsequently of course, those concepts became critical from a practical point and further research into the algorithm resulted in major improvements.

To comprehend how implementation issues of Dijkstra' s algorithm could inspire all the research it has, it is essential that we review how the algorithm would actually execute. This algorithm, recall, partitions the set of nodes in a graph into two sets: one for nodes with known shortest paths from the source node, termed 'labeled nodes', and the other for nodes with a tentative shortest path if at all, the 'unlabeled nodes'. The idea is for the former set to start with the source node, of zero shortest distance to itself, and iteratively grow to encompass all the nodes of the graph. This is accomplished as follows:

(1) Select the shortest of the known tentative distances to unlabeled nodes. Declare this

distance to be the shortest to the corresponding node. Label that node and removing it from the set of unlabeled nodes.

(2) Set tentative shortest paths to the unlabeled nodes of the graph which are directly linked to the newly labeled one but have so far no tentative shortest paths.

(3) Update the tentative distances to the unlabeled nodes which are directly linked to the newly labeled one if the path through the latter node proves shorter.

Each of these tasks is implemented as a function or procedure and are commonly referred to as *delete_ min(h)*, *insert(h,x)* and *decrease(h,x,value)*, respectively, where $h$ is the data structure used in the implementation of the algorithm, $x$ is a node of the graph linked to the newly labeled node and *value* is the new tentative shortest distance replacing the old. Since, for the most part, besides input and output operations those functions are all what the algorithm requires, the key to an efficient implementation of Dijkstra's algorithm is a data structure that allows efficient implementation of those functions. Note that if $m$ is the number of arcs we are dealing with and $n$ is as before the number of nodes then $n$ *delete_ min* calls, $n$ *insert* calls and at most $m-(n-1)$ *decrease* calls are required. Usually, *insert* and *decrease* calls are the less costly functions to implement while *delete_ min* is considerably more expensive and thus controls the complexity of the whole algorithm.

Dijkstra's original implementation of his algorithm uses an array for the data structure. This leads to an $O(1)$ incurred cost per *insert* or *decrease* and an $O(n)$ cost per *delete_ min* for the aforementioned worst case total time of $O(n^2)$. Using a binary tree for the data structure, however, the worst case running time can be reduced to $O(m \lg n)$ (Cormen *et al.* 1990, for instance). Fredman and Tarjan (1987) have reduced this requirement even further when

considering amortized time instead of worst case time[19]. With their Fibonacci heap data structure, an extension of the Binomial heaps developed specifically for Shortest Path, the authors achieve an $O(m + n \lg n)$ amortized time. They note that this is the best possible when the arc lengths, or edge weights, are real numbers and only binary comparisons are used in the heap implementation. These assumptions, however, are by no means inevitable. Research had previously been carried, and was subsequently continued, into efficient implementations that result from tightening the first and/or relaxing the second by considering variations of the RAM computing model[20]. But regardless, however significant the Fibonacci heap implementation was in theoretical terms, it proved unattractive in practice.

Since the late sixties, several authors have noticed that by restricting the arc lengths to be integers instead of non-negative real numbers they could achieve better complexity results. Bucket data structures, which require the RAM model for their implementation, were proposed by several authors, for example Dial *et al.* (1969). In such implementations, each bucket is a list of nodes with the same tentative distance, or range of tentative distances, from the source. A node is simply moved from one bucket to another in the implementation of *decrease* and is merely chosen from the top in the *delete_min* implementation. Thus, using this data structure, all functions are implemented in $O(1)$ time. Additionally, the number of buckets that needs to be maintained is only $C + 1$, where $C$ is the largest arc length. We note here that to obtain the complexity of the resulting algorithm, one also has to take into account the number of times the buckets are accessed. Indeed, we could have a situation where most buckets are empty most of the time and where the non-empty ones are last to be accessed most

[19] Amortized time refers to the total running times possible divided by the number of those running times.

[20] RAM stands for 'random access machine': a computing model much similar to today's computer capable of supporting the C programming language, for example.

of the time[21]. This necessitates accessing the buckets an $O(nC)$ times yielding a worst case $O(m+nC)$ algorithm. Clearly, if $C$ is large then maintaining a large number of empty buckets wastes space and running time. This prompted Denardo and Fox (1979) to develop the $k$-level bucket data structure with a resulting worst case implementation of $O(m+n(k+C^{1/k}))$ time and $\Theta(kC^{1/k})$ buckets[22][23]. In an experimental evaluation, Cherkassky *et al.* (1996) showed that a 2-level bucket implementation is generally the best or nearly the best among an extensive array of codes they tested. Goldberg and Silverstein (1997) experimented with $1, 2, 3$ and 4-level implementations. They concluded that 3 and 4-level implementations are even more robust choices than the 2-level implementation. In both papers, the 1-level implementation was not advised unless $C$ is small.

Along different lines of thought but still for the integer arc lengths, van Emde Boas *et al.* (1977) devised a data structure that yields a worst case running time of $O(m \log\log C)$. In 1990, Ahuja *et al* came up with three implementations the best of which achieved an amortized bound of $O(m+n\sqrt{\log C})$ using a 1-level radix heap, a bucket based structure, combined with a Fibonacci heap on the second level. Fredman and Willard (1994) developed the atomic priority queue concept and used it to achieve an $O(m+n\log n/\log\log n)$ time. They made no pretense about the practicality of their algorithm, though, as $n$ is required be larger than $2^{12^{20}}$.

Still for integer arc lengths and variations of the RAM model, a flurry of complexity bounds were derived by Cherkassky and his colleagues, by R. Raman and by M. Thorup starting in

[21] An extreme manifestation of this situation is when the network is a path each arc of which has duration $C$.

[22] Given two functions $f(n)$ and $g(n)$, $f(n)$ is said to be $\Theta(g(n))$ *iff* there exist positive constants $c_1$, $c_2$ and $N$ such that $0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$ , for all $n \geq N$. That is $f(n)$ and $g(n)$ are roughly of the same order of magnitude. In contrast, $f(n)$ is $O(g(n))$ *iff* $0 \leq f(n) \leq c_2.g(n)$, for all $n \geq N$. That is, for large enough $n$, $g(n)$ is an upper bound of $f(n)$ to within a constant factor $c_2$.

[23] It would seem that a way to deal with the empty buckets issue is to maintain an ordered doubly linked list of the non-empty buckets. Each time a new bucket is needed it is inserted and each time one is emptied it is removed. The problem with this approach, though, is that *insert* will not be $O(1)$ anymore.

1996. Following are the best bounds from each. In 1996, Cherkassky *et al* (1999) proposed their so called 'heap-on-top (hot)' data structure which combines the multi-level bucket structure with a monotone s-heap. They obtained an expected running time for their algorithm of $O(m + n(\log C)^{\frac{1}{3}+\varepsilon})$ for any $\varepsilon > 0$. Later that year, Raman (1997) improved on this result by showing that Dijkstra's algorithm can be implemented in $O(m + n(\log C)^{\frac{1}{4}+\varepsilon})$ expected time, for any $\varepsilon > 0$. He also showed that it can be run in $O(m+n(\log C \log\log C)^{\frac{1}{3}})$ worst case time. Both results, however, rely on Fredman and Willard 's atomic priority queues and are therefore unsuitable for practice.

Finally, last but not least, while all research efforts into the problem were aimed at efficiently implementing Dijkstra's algorithm, Thorup (1999) ended the quest for the theoretical best Shortest Path bound by proposing a non-Dijkstra algorithm for the integer case that runs in $O(m)$ time, at the worst, that also uses atomic heaps. The same paper contains an $O(m)$ algorithm for the non-integer case but under the additional assumption of presorting a given sequence of integers defined by the arc lengths. In a subsequent paper, the author dealt away with the assumption while still maintaining the $O(m)$ bound (Thorup 2000). Evidently, no lower bound can exist for Shortest Path as merely inputting an instance requires $O(m)$ time. Nevertheless, recognizing that the use of atomic heaps renders his algorithm impractical, Thorup proposed a variant he claims should work well in practice. Theoretically at least, this variant is almost linear as it runs in $O(\alpha(m, n)\ m)$ time, where $\alpha(m, n)$ is the inverse Ackerman function known, for all practical purposes, to have an upper bound of 4 (Cormen *et al.* 1990).

### 3.9.2 Shortest Path in the Identical Arc Lengths Case

As RCPSP instances with unit activity durations are a distinct class of instances in so far as their reduced state networks are concerned, one might wonder whether a special implementation of the bidirectional algorithm might exist to take advantage of their structure. Recall that for this class of state networks no remaining activity durations are possible and all states are induced by maximal subsets. These, of course, are important features of this class but what is also significant about it is that the arcs of the corresponding state networks all have the same length. This opens up the possibility of devising a shortest path algorithm to take advantage of this property.

To see how such an algorithm could be obtained, note that in Dijkstra' s algorithm when all arc lengths are 1 the *decrease* operation need never be called. That is the first estimate of a shortest distance from the source node to any node $v$ of the graph is actually optimal. This is due to the fact that an estimate of a shortest distance to $v$ is made through a labeled node $u_1$. That is when a node $u_1$, which happens to be directly linked to $v$, is labeled then the estimate for the shortest distance to $v$ is $d(u_1)+1$, where $d(u_1)$ is the shortest distance to $u_1$. If the first estimate for the distance to $v$ is through $u_1$ but the shortest distance possible to $v$ is through some other node $u_2$, of label $d(u_2)$, that also is directly linked to $v$ then $d(u_2)+1 < d(u_1)+1$ so that $d(u_2) < d(u_1)$. This says that $u_2$ must have been labeled before $u_1$. But then, the first estimate for the distance to $v$ should have been through $u_2$ not $u_1$; a contradiction to our assumption about $u_1$. We conclude that the shortest distance to $v$ could not be through $u_2$ and so the first estimate to the distance to $v$ is actually optimal.

It follows that to implement Dijkstra's algorithm when all arc lengths are 1, it is sufficient to

implement the *insert* and *delete_min* operations only. This can be done using a simple queue for the data structure. As a node has its shortest distance from the source node determined, it enters the queue. It leaves the queue labeled and on a 'First In First Out' basis. The complexity of this special case implementation of Dijkstra's algorithm can easily be seen to be $O(m)$, where $m$ is the number of arcs.

### 3.9.3 Shortest Path Implementations for the Reduced State Networks

Having surveyed a significant portion of the relevant repertoire of algorithms and implementations for Shortest Path, it is time to grapple with the issue of choosing a suitable algorithm and an appropriate data structure to support the efficient implementation of our approach to RCPSP. We reiterate that this choice should be compatible with developing networks and reasonably efficient in terms of computing time and space. We stipulate as well that it be easily codable, for obvious reasons, and easily executable in the sense of not requiring extensive experimental fine tuning.

In this regard, and for the arbitrary durations case, the first candidate we look for to fill our requirement is the almost linear variant of Thorup's shortest path algorithm for integer arc lengths. As it turns out, however, this algorithm requires presorting the network's arc lengths; an impossible task, in our case, as we are dealing with networks under construction. Hence, this algorithm is unsuitable for our purpose. Skipping Raman's algorithms which rely on atomic heaps, the second algorithm we investigate is that of Cherkassky *et al* (1999) based on the authors' 'hot' data structure. According to its authors, the efficient implementation of this algorithm requires carefully chosen parameter values; a requirement inconsistent with the 'easily executable' condition we imposed. Additionally, the authors own computational experiments

indicate that when $C$ is small, *i.e.* about a thousand or less, hot queues behave similarly to multi-level bucket data structures. Since the largest value of $C$ we expect to encounter is almost surely less than a thousand, we do not anticipate an implementation of this algorithm to be much better, for our purpose, than that of a multi-level bucket algorithm[24].

Now, in view of the conclusion in Cherkassky *et al* (1996) that this same multi-level bucket data structure implementation of Dijkstra's algorithm is preferable to the multitude of implementations and algorithms they tested, including the Fibonacci heap implementation, the 2-level radix/Fibonacci heap implementation and a $d$-ary[25] heap implementation, we can justifiably eliminate from further consideration these aforementioned three. What remains for us to decide between, then, are the single level and the multi-level bucket implementations. But again, the experimental results of Cherkassky and his colleagues and, further, Goldberg and Silverstein (1997) indicate that generally unless $C$ is larger than a thousand or so, the single level bucket implementation is the faster of the two. In addition, and expectably, the single level bucket implementation is the simpler of the two to code and execute. Therefore, we conclude, among all the shortest path algorithms and implementations we reviewed, the single level bucket implementation is, for our purposes, the preferable one.

Having decided that the single level bucket implementation of Dijkstra's algorithm is the most appropriate for us, at least for arbitrary durations instances, we are now in a position to propose a data structures to support the implementation of the bidirectional algorithm

[24] To see why, for our purposes, $C$ would not exceed a thousand, note that the largest arc length of the reduced networks is at most the largest activity duration of the corresponding instance. Since, in practice, we do not expect the largest activity duration to approach let alone exceed a thousand, we are not concerned with solving shortest path instances with such large $C$'s.

[25] The $d$-ary heap implementation of Dijkstra's algorithm is an extension of the correponding binary heap implementation. For $d = 2$, a $d$-ary heap is in fact a binary heap. Cherkassky and his colleagues had tested the $d$-ary heap for $d = 3$.

for RCPSP. Recall that such data structures have to efficiently support the operations *insert*, *delete_min* and also *decrease*, in the arbitrary durations case. In the reduced networks, though, the *insert* and *decrease* operations have to be somewhat altered. More specifically, when a submaximal is generated, which in turn generates a state, the *insert* operation has to be first preceded by a check of whether or not the state has been previously generated. If not then the state can be inserted. If the state exists and the current path leading into it has a shorter distance than the established path then *decrease* is performed. Otherwise, no action needs to be taken and the state is discarded. Hence, not only does a data structure we may use need to efficiently support *delete_min*, *insert* and *decrease*, it also has to efficiently support a 'verification' step. Next, we discuss two ways of accomplishing these objectives, each leading to a different data structure.

### An Index Function Based Data Structure

Note that each node of a developed network has an index associated with it. This says that the verification step need only involve checking the current source to node distance associated with a node under consideration to verify whether the node has been previously inserted or not. Such distances may simply be stored in an array and accessed through the indices corresponding to the network's nodes. Of course, we may try to emulate this scheme for networks under construction by assigning indices to the states as they are generated. We need to make sure, though, that a state is assigned only one index. That is, if and when a state is generated a second time, the index assigned to it is the same as the one it was previously assigned. Needless to say, we could not ensure that the same index is assigned to a state by executing a linear search that detects whether the state was previously generated or not. Linear search would be

too time consuming.

An alternative to linear search is for the index of the array to be determined by a one-to-one function of the state. The range of such a function, however, should be of the same order as the number of states we expect to examine. Otherwise, we may end up with an array of such a large size that it may be impossible to store. For example, for the single duration case where states $S$ do not contain continuing activities, the index function $I(S) = \sum_{i \in S} 2^i$ can be shown to be one-to-one. But its range of $[1, 2^{n+1} - 2]$, where $n$ is the number of activities of an instance, is likely to be far larger than the number of states that need to be generated. It is even larger than the number of subsets of the set of $n$ activities. In any case, even for rather small $n$, this range is impractical.

For lack of an explicit one-to-one index function with a manageable range, we propose using the well known hashing function technique (Cormen *et al* (1990), for example). The idea is to drop the one-to-one requirement and use a function, or procedure, to map a set of $n$ keys, or objects, into an array of size $m$, where $m$ could be smaller than $n$. In this way, when an array index is already 'occupied', the next key mapped to the index is linked to the previous key to form a chain of keys associated with that index. Thus, in a hashing function technique, searching for a key can involve a linear search but that search is performed on a much smaller list of keys than the original one. We adapt this hashing function idea to our problem as follows.

Let $\{a_1, ..., a_p\}$ be the set of activity indices of a state $S$, where $a_1 < ... < a_p$. By the key of $S$, or its numerical representation, we mean the number $k$ defined by the concatenation of indices $a_1 ... a_p$. Clearly, two states with the same key are identical up to the remaining processing times of their activities. As such keys are potentially very large numbers, they are unsuitable for use as array indices to reference their corresponding states. To remedy the situation, we

define the index of state $S$ as $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$, where $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod (m-2))$ and where $m$ is a prime number upper bound to the number of states that may be generated, up to different remaining durations, and $i$ is the number of the failed trials to map key $k$, soon to be explained. In this way, an entry of the array is a pointer to a linked list the head of which represents the set of activities of a state while each of the other elements represents a set of remaining durations for the possibly continuing activities of that state. In the process of mapping a state with key $k$ for the $i^{th}$ time, if the pointer referenced by $h(k,i)$ is not nil, in other words the $h(k,i)$ entry references some state, a check is made as to whether the activities of the state we seek to map are the same as the activities referenced by $h(k,i)$. If there is no match, we increase $i$ by 1 and try again for the $(i+1)^{th}$ time. We note that this $h(k,i)$ function, known as a double hashing function, is designed so that any of the $m$ indices of the array is approximately as likely to be an image for state $S$ as any other; thus alleviating the clustering problems associated with other hashing techniques (Cormen *et al* 1990).

We point out that besides storing the activities' remaining durations, the elements of the linked lists indirectly pointed to by the array entries consist of nodes that store various other information such as the shortest distance to the corresponding state from the source node and pointers to other nodes in the same bucket. Additionally, and instead of storing information specifying whether a node was generated in the forward or reverse state networks, the head node of that list could point to two lists instead of one, each corresponding to one state network. Figure 3-14 partially depicts this data structure. There, the top row represents part of the array of pointers and two nodes: one storing the head node information, and the other storing the working information for a state are expanded. Note that with this data structure, the

verification step is not an $O(1)$ operation as more than one trial to map a state may be required. In fact, the worst case complexity of each unsuccessful trial depends on the number of state activities whilst the worst case complexity of the successful one depends also on the length of the list of states with the same activities. Nevertheless, the number of trials depends on how full the array is at the time of the trials and has no useful and simply derivable upper bound. But if, at the time of mapping a state, $\theta$ is the ratio of occupied entries in the array to the total number of entries, *i.e.,* $m$, then the expected number of trials is approximately $\sum_{j=1}^{(1-\theta)m}(1-\theta)\theta^{j-1}j \simeq \sum_{j=1}^{\infty}(1-\theta)\theta^{j-1}j = \frac{1}{1-\theta}$; yielding an expected $O((\frac{1}{1-\theta}+\beta)\alpha)$ time for the verification step, where $\alpha$ is the average number of activities in a state and $\beta$ is the average number of states with the same activities.

Two important issues related to this data structure still need to be addressed. First, how do we secure an upper bound on the number of states that the bidirectional approach may require, up to differences in activity durations. Second, how do we transform this most likely composite upper bound into a prime one.

Regarding the first issue, we note that the number of states that the bidirectional algorithm may require, up to differences in durations, depends not only on the number of activities, precedence relationships and resource limits of an instance but also on the stopping criterion and the rules we use to control the number of states. But, even though we can prove that the rules and stopping criterion help reduce the number of states, it is very difficult, if not impossible, to quantify the cumulative effect of all of the factors involved. Actually, quantifying the combined effect of any two of those factors, not including the number of activities, is out of our reach. We, therefore, choose to limit our search for an upper bound to only the quantification of the combined effect of the number of activities and the precedence relationships.

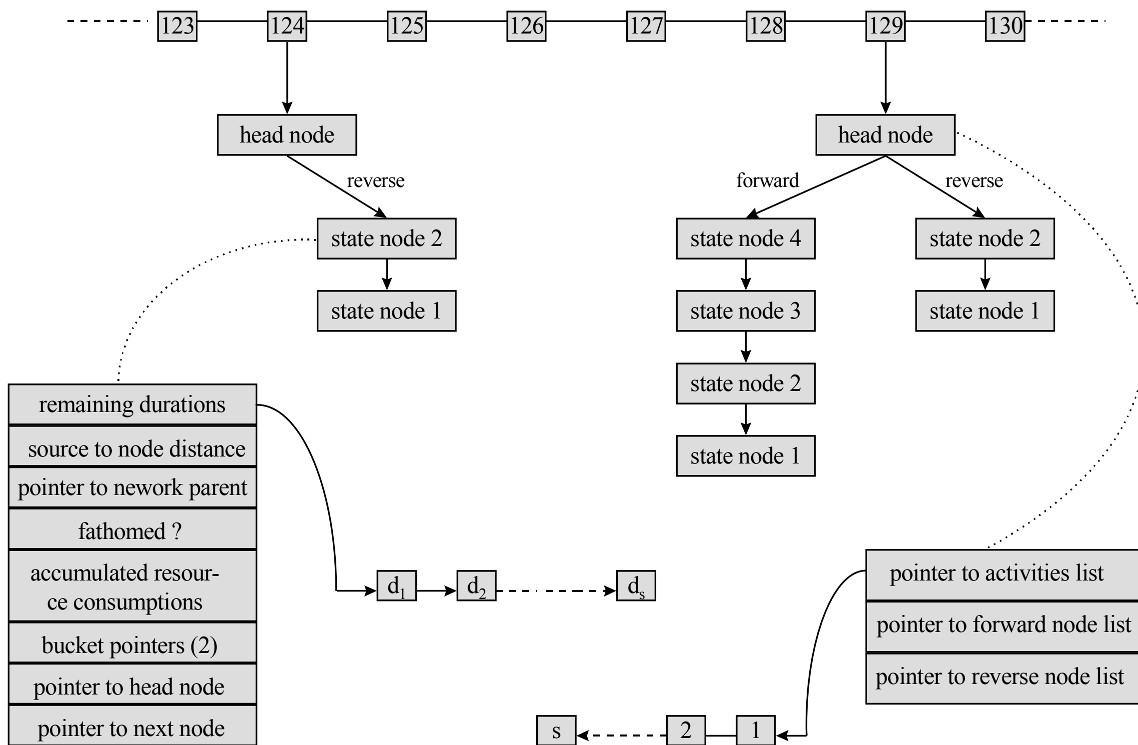


Figure 3-14: A partial depiction of the data structure supporting the index function approach.

For any activity $i$ of an instance, the precedence relationships of that instance induce a partition of the whole set of activities into two subsets. One is for the activities that are precedence feasible to be processed with $i$, $P(i)$, and the second is for the activities that precede or succeed $i$. The states containing $i$ that the bidirectional algorithm may generate, then, can only be subsets of $P(i)$. Determining $P(i)$ for every activity, we obtain $\sum_{i=1}^{n} 2^{|P(i)|}$ as an upper bound to the number of possible states, up to differences in durations.

To improve on that bound, note that not all activities within one of those subsets can be concurrently processed. Depicting the subset $P(i) - \{i\}$ as a graph with vertices representing activities and edges representing precedence relationships between the corresponding adjacent vertices, we observe that a state that the bidirectional algorithm may generate cannot contain two activities that are reachable from one another. Let $P_1(i), ..., P_{r_i}(i)$ be the connected components of $P(i) - \{i\}$. Then the number of subsets of $P(i) - \{i\}$ containing only precedence compatible activities is $\prod_{j=1}^{r_i}(P_j(i) + 1)$. Thus an upper bound on the number of states, up to differences in duration, that the bidirectional algorithm may generate is $UB = \sum_{i=1}^{n}(\prod_{j=1}^{r_i}(P_j(i) + 1))$. We note that this count surely includes non-submaximal subsets, and even duplicates of the same subsets. However, further refinement of this upper bound to eliminate counting occurrences such as these is rather cumbersome, to say the least. We also note that the validity of this procedure hinges mainly on the transitivity property of the 'precedence relationship'; which, essentially, is what allows the decomposition of $P(i) - \{i\}$ into distinct subsets[26]. Such a property does not hold for the 'resource compatible' relationship, for example, which explains why the procedure is not extendable to quantify the effects of the

[26] To be more precise, if $i$ and $j$ are two activities and $R$ denotes the precedence relationship then by '$R$ relates $i$ and $j$ ' we mean that either $i$ precedes $j$ or $j$ precedes $i$. In this way, $R$ is reflexive, symetric and transitive. This, of course, partions the set of activities into the equivalent classes we used.

resource limits.

The second issue concerning this data structure we still need to address is that of finding a prime upper bound to a possibly composite $UB$. The obvious way to accomplish this is through sequentially testing for primality the odd integers larger than $UB$ until the test returns positive. We claim that this method does not require too many iterations, on the average. To see this, recall the Prime Numbers Theorem which asserts that $\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1$, where $\pi(n)$ denotes the number of primes less than or equal to $n$. Another way to formulate the theorem is to state that $\lim_{n\to\infty} \frac{n/\pi(n)}{\ln n} = 1$. But $n/\pi(n)$ can be interpreted as the average gap between the consecutive primes up to $n$. This says that the number of iterations this prime finding method is expected to run is $\Theta(\ln UB)$, with $1/4$ as the coefficient of $\ln UB$, which obviously is not too many iterations[27]. The maximum number of iterations that may be required, on the other hand, is a lot more challenging question. Reisel (1994) reports that the maximum gap between two consecutive primes, $p_n$ and $p_{n+1}$, is the subject of two conjectures. One is by H.Cramér who, in1936, proved that $\limsup_{n\to\infty} \frac{p_{n+1}-p_n}{(\ln p_n)^2} = 1$ with probability 1 and conjectured that the same identity holds without qualifications and the other is by D.Shanks who, in 1964 and based on experimental results, made the even stronger conjecture that $\lim_{n\to\infty} \frac{\max_{k\leq n} \sqrt{p_{k+1}-p_k}}{\ln n} = 1$. Both conjectures suggest that the maximum number of iterations required is $O((\ln n)^2)$; with the second implying yet $\Theta((\ln n)^2)$.

Clearly now, the complexity of this procedure depends to a large extent on the complexity of the underlying primality test. This, in turn, depends on the size of $UB$ as some tests are preferable to others depending on the input size. For our purposes, the historically first and

[27] If $n - [\ln n]$ and $n$ are consecutive primes and $n - [\ln n] < UB \leq n$, then on the average the number of iterations required to find prime $n$ is $\frac{1}{2}[\ln n]$. Additionally, we can skip testing about half the integers in that range as they must be even. Whence, the average number of iterations required is $\frac{1}{4}[\ln n]$.

simplest primality test, consisting of trial divisions by prime numbers at most the square root of the integer tested, should work well. This test does require storing these primes but for a small trade-off in execution speed, we can generate and use the integers not divisible by the first few primes, say 2 and 3, instead. Note that the limit for which this method ceases to be an efficient primality test is reportedly around $25 \times 10^6$ (Bressoud 1989)[28]. More sophisticated primality tests do exist but are not likely to be significantly more efficient, on the average, for the range of upper bounds we are interested in. Note also that the number of primes less than or equal to 5000 is 669; not too large a list to generate and store for repeated use.

Besides its efficiency at testing the primality of integers up to, approximately, $25 \times 10^6$ this trial division method can be implemented 'concurrently' not just sequentially. In other words, the result of testing the divisibility of $UB$ by a prime $p$ can also be used to deduce whether $UB+1$, $UB+2, ...$ are or are not divisible by $p$. In mathematical terms, if $UB \equiv c \bmod p$ then $UB + i \equiv c + i \bmod p$, for any $i$. Hence, if $c + i \equiv 0 \bmod p$ then $UB + i$ is not prime. But $c + i \equiv 0 \bmod p$ if and only if $i = kp - c$, for some integer $k$. Thus, while testing the divisibility of $UB$ we may be able to decide the compositeness of some other integers in the sequence. This, of course, makes the trial division method even more computationally attractive for the upper bounds we expect. The following is the prime finding procedure in pseudo-code.

**Algorithm 3**

**begin** find_prime ($UB$)

/* This algorithm finds the first prime in the sequence $UB, UB+1, ..., UB+2\lfloor \ln UB \rfloor^2$.

If the sequence contains no prime then the next similar sequence starting with $UB +$

[28] For an extensive study of primality testing and other 'prime' issues, the interested reader is refered to the thorough treetment of the subject by Riesel (1994). Also, besides Bressoud (1989), another good reference is by Ribenboim (1995).

$2\lfloor \ln UB \rfloor^2 + 1$ is examined. Array $u$ stores the sequence and array $p$ stores the 669 primes up to 5000, in increasing order. $UB+2\lfloor \ln UB \rfloor^2$ should around $25\times 10^6$ at the most. Some of the comments contain the total number of operations that the corresponding statement entails, up to a constant such as 1 or $-1$. There, $UB + d$ denotes the prime to be found and $p(r-1)$ is the last used prime of the sequence. */

**i**_ max $\leftarrow 2\lfloor \ln UB \rfloor^2$

**f**or $i = 0$ to $i_\max -1$

**u**$(i) \leftarrow UB + i$

**e**ndfor /* $2i_\max$ additions, $i_\max$ assignments */

$i \leftarrow 0$, $j \leftarrow 0$

**w**hile $i < i_\max$ and $j < 669$ and $p(j)^2 \leq u(i)$ /* $d$ inequalities, $2r$ inequalities, $r$ products */

**c** $\leftarrow u(i) \bmod p(j)$ /* $r$ modulus, $r$ assignments */

**k** $\leftarrow p(j) - c$ /* $r$ differences, $r$ assignments */

**w**hile $k < i_\max$

**u**$(k) \leftarrow 0$ /* $u(k)$ is set to 0 to indicate that $UB + k$ is composite */

**k** $\leftarrow k + p(j)$

**e**ndwhile /* $i_\max(\sum_{i=2,\ i \text{ prime}}^{p_r} \frac{1}{i})$ inequalities, $2i_\max(\sum_{i=2,\ i \text{ prime}}^{p_r} \frac{1}{i})$ assignments, $i_\max(\sum_{i=2,\ i \text{ prime}}^{p_r} \frac{1}{i})$ additions */

**i**f $c = 0$

**u**$(i) \leftarrow 0$

**w**hile $u(i) = 0$ and $i < i_max$

**i** $\leftarrow i + 1$

endwhile

endif /* $d$ equalities, $d$ inequalities, $d$ additions, $2d$ assignments */

**j** $\leftarrow j+1$ /* $r$ additions, $r$ assignments */

endwhile

**if** $u(i) > 0$

**output** $u(i)$ /* this is the smallest prime of the sequence. */

**else**

**find**_prime($UB + 2\lfloor \ln UB \rfloor^2 + 1$) /* recursive call seldom to be executed, if at all. */

**endif**

**end**

**Remark 20** *It is possible to fine tune Algorithm* 3 *by choosing a smaller* $i_\max$ *and storing, for each prime* $p(j)$ *used, the largest index,* $index(j)$*, s.t.* $u(index(j)) \equiv 0 \bmod (p(j))$*. In this way, if all integers of the sequence under consideration are composite then before searching the next sequence for a prime we can eliminate some of its integers using the results of the previous modulo operations. The purpose, of course, is to minimize some basic operations while at the same time economize modulo operations. We choose to forgo the fine tuning , however, as we believe the algorithm is fast enough as it is. Note that this algorithm has a complexity of* $O(qr) = O(q\sqrt{UB}/\ln UB)$*, where* $r$ *is the number of primes used and* $q$ *is the number of recursive calls needed*[29].

[29] *The* $\sum_{i=2,\ i\ prime}^{p_r} \frac{1}{i}$ *term in the number of operations count is* $O(\ln p_r)$ *at the most and may even be of lower complexity. In any case, it is less than* 5 *for* $p_r < 5000$ *and so, as far as our range for* $UB$ *is concerned, this term can be treated as a constant.*

*Moreover,* $q$ *is* 1 *in view of Shanks and Cramér's conjectures and some experimental findings in Reisel (1994, p. 80). But those are only conjectures and limited evidence not proofs of upper bounds.*

Finally, we note that the issue of finding an appropriate prime upper bound can be dealt with ahead of runtime by figuring out the largest prime number that the size of the pointers' array can have. For example, if we estimate to have 64 MB of RAM to store the nodes of our data structure and if each node requires about 64 bytes, then we know we may have at most $2^{20} = 1048576$ nodes. So we look for the largest prime number less than $2^{20}$and use it as the size of the array, whatever the RCPSP instance we wish to solve is[30]. In this way, the prime number we use is machine dependent not instance dependent. Of course, the significant drawback of this approach is that the ensuing algorithm may not be portable.

**An Alphabetic Data Structure**

Unlike the previous approach, the second data structure we discuss to implement the bidirectional algorithm does not index the nodes of the state network in order to gain access to them. Rather, much as a word dictionary depends on the ordering of the alphabet letters, it uses the ordering of the activities of a state to store the corresponding node. This, of course, assumes that the activities of a state are ordered to begin with so we choose the increasing order in the activities indices. The data structure consists of a collection of trees each with a root corresponding to an activity index. The nodes of the trees are of three types: directional nodes, head nodes and state nodes.

As the designations may indicate, the state nodes represent the nodes we seek to store in the first place, *i.e.,* the states of the reduced forward and reverse networks, and the directional nodes are the means we use to access the state nodes. In fact, the state and head nodes are the

[30]Note that the prime, in this instance, is most definitely not $2^{20} - 1$, however tempting it might be to test. Numbers of the form $2^n - 1$ are known as Mersenne numbers and it is not difficult to prove that when $n$ is composite then so is the corresponding Mersenne number. Actually, those numbers have their own primality test known as Lucas's test for the primality of Mersenne numbers (Riesel 1994 and Bressoud 1989).

same as in the previous data structure while the directional nodes consist of a key and three pointers each. The key in a directional node represents the index of some activity that is in the same position in all state nodes succeeding the directional node. One of the three pointers of a directional node is to a 'child' of the node, another is to its 'sibling', both of which are also directional nodes, and the third is to a head node. The head node, in turn, points to a list of activities and two lists of state nodes for the forward and reverse state networks.

A schematic depiction of this data structure is given in Figure 3-15. The siblings of directional nodes are drawn on the same level as the nodes themselves whilst their children are one level lower. A node of each of the three types is expanded.

When accessing a state node of $p$ activities $\{a_1, a_2, ..., a_p\}$, or verifying that it already exists, at least $p$ directional nodes are passed. With the assumption that $a_1 < a_2 < ... < a_p$, the search starts at the root of the tree corresponding to activity $a_1$. It then moves to the child of the root node and its iterative siblings in a linear search to locate the directional node with key $a_2$ among those siblings. If no directional node with key $a_2$ exists then one is created and placed consistently with an increasing order for the keys of the iterative siblings. If, on the other hand, that directional node is found then the search moves through its child and the iterative siblings of that child to locate the directional node with key $a_3$. The same process is repeated until the proper directional node with key $a_p$ is located or created. At this point the search moves to the head node pointed to by the last directional node and then to one of the lists of state nodes that may attached to that node, depending on whether the state to be located or stored is one of the forward or reverse networks. From there, the search proceeds linearly through the appropriate list of state nodes to determine a match or store the state node. If no proper directional node with key $a_p$ exists then one is created together with the corresponding

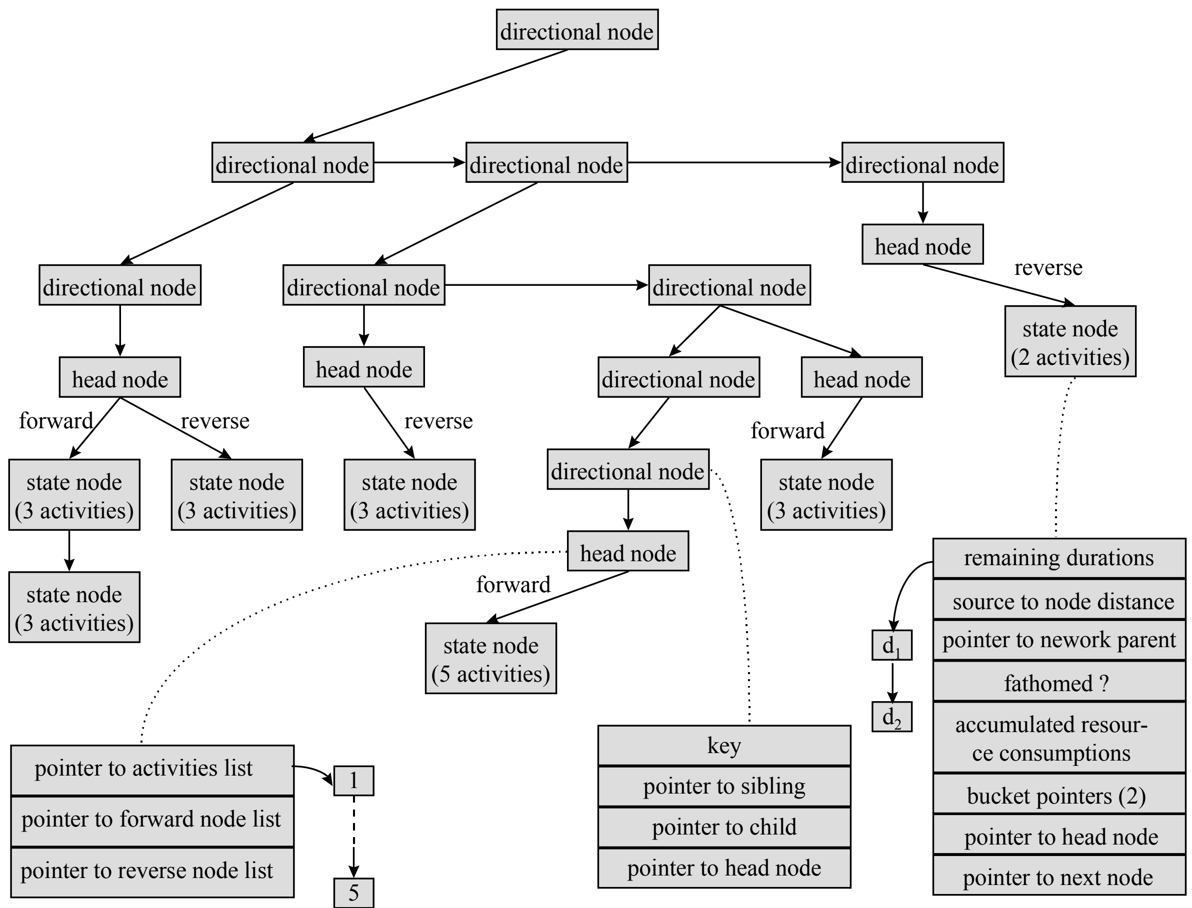


Figure 3-15: A partial depiction of the data structure supporting the alphabetic approach.

head node, state activities' list and attached state node list. To facilitate the search, state node lists, which actually consist of lists of remaining activity durations, are kept lexicographically ordered.

Note that with this data structure, locating or storing a state is not subject to trial and error as with the previous approach. The process does not depend on how full or empty the main component of the data structure is. As a matter of fact, locating or storing a state node of activities $\{a_1, ..., a_p\}$, increasingly ordered, requires passing through at most $(a_p - a_1 + 1)$ directional nodes. That is, the number of directional nodes passed in a verification step is $O(1)$. This says that if, as before, $\alpha$ is the average number of activities in a state and $\beta$ is the average number of states with the same activities then the average case complexity of a verification step is $O(\alpha\beta)$. Since we don't expect $\alpha$ to be large, a verification step is expected to cost practically $O(1)$ time. Similar reasoning and a similar conclusion hold for the worst case complexity too. The drawback of this data structure, however, is the additional memory space required to store the directional nodes.

### 3.9.4 Decomposition Approaches

As the name suggests, decomposition of a project instance refers to splitting the instance into two or more parts, obtaining a solution to each part and then somehow combining the individual solutions into a single whole. The purpose is to reduce the instance's solution time by either reducing the computational burden required or at the least, in the case of parallel or multiple processing, by distributing this burden. As might be expected, not every project instance is well suited for decomposition just as not every Linear Program is amenable for a fast solution via the LP Decomposition Principle. The obvious and ideal instance for decomposition is one where one

of the activities precedes some others and succeeds the rest. There, combining the individual solutions into one for the whole instance is straightforward. Our aim in this subsection is to emulate this ideal situation by considering two cases of instances that strictly are not 'ideal' yet sufficiently resemble the ideal so as to still permit an advantageous decomposition.

**An Almost Ideal Situation**

One extension of this 'ideal' situation is the case where an instance contains an activity $i$, of duration $d_i$, that has the property that all activities that are precedence feasible to be processed with it determine a subproject, of a makespan not exceeding $d_i$, which can be run in parallel with it. In this case, the project instance is decomposable into three parts the first of which consists of the activities preceding activity $i$ while the third is determined by the activities succeeding it. We note that if the subproject of activities concurrent with $i$ has minimum makespan exceeding $d_i$ then the makespan of the solution to the instance consisting of juxtaposing the optimal solutions of the three parts depends on the precedence relationships and resource consumptions of the activities on the juxtaposed ends; so optimality in that case cannot not guaranteed. We also note that, when feasible, the decomposition of an instance in this fashion does reduce the computational burden that otherwise would be required to optimally solve the instance; even on a single processor.

**A Node Cutset Approach**

Another way to realize decomposition is as follows. Assume, for a moment, that the forward reduced state network of a project instance is fully developed. If this network is large and two processors are available for its solution then to find the shortest path between the end nodes,

we can introduce a so called **node cutset** to partition the network into three parts as follows. The first part consists of the nodes with no arcs except in between nodes of this same part and those pointing to nodes of the second part. The second part, which is the **node cutset** itself, consists of nodes with arcs incoming from nodes in the first part and/or arcs outgoing into nodes of part three only; no arcs in between nodes of this part is allowed. Finally, the third part consists of the rest of the network nodes where all arcs, except the ones incoming from the second part are in between nodes of this part[31]. Now, to obtain the shortest path between the two end nodes of the network, each processor can solve the problem of finding the shortest paths between one end node and each node of the node cut. The shortest start to end path of the network is then deduced by comparing the shortest paths through the nodes of the node cut.

If $p > 1$ processors are available and the network is large enough to accommodate $p - 1$ pairwise disjoint node cuts then, assuming bucket implementations with $C + 1$ buckets are used for the subproblems, the complexity of this decomposition is

$$\{O(m_1 + (n_1 + \nu_1)C) + \sum_{i=2}^{p-1} O(\nu_{i-1}[m_i + (n_i + \nu_i)C]) + O(m_p + (n_p + \nu_{p-1})C)\}$$
$$+O(\nu_1 + \textstyle\sum_{i=2}^{p-1} \nu_{i-1}\nu_i + \nu_{p-1}),$$

where $m_i$ is the number of arcs in part $i$, $n_i$ is its the number of nodes, excluding nodes on the cuts, and $\nu_i$ is the number of nodes of the $i^{th}$ cut. In this estimate, the last term represents the cost of finding the shortest path in the topologically sorted network consisting of the terminal nodes and the node cuts. We note that this total cost is surely larger than if one were to solve

[31] Note that this partition prohobits arcs in between nodes of the first part and others in the third and that this concept is somewhat analogous to the concept of uniformly directed cutset (UDC).

the problem without resorting to node cuts. Still, the decomposition can reduce the solution time on the $p$ processors, specially when $\nu_1, ..., \nu_{p-1}$ are relatively small.

Now, to achieve this type of decomposition in solving an RCPSP instance when the fully developed reduced state networks are not available, we need to be able to generate node cutsets without generating the networks. One might think that such a cutset can be obtained by choosing an activity, with at least one predecessor, and finding all possible states containing the activity. Unfortunately, such a set of states does not necessarily constitute a node cutset due to the possibility that some of its states may precede others in the same set. As an example, consider the forward reduced state network in Figure 3-8. There, the set of states containing activity $a_4$ cannot be considered a node cutset due to state $\{a_1; a_3; a_4\}$ preceding state $\{a_4\}$.

To remedy the situation, we propose to choose an activity $i$ with both predecessors and successors, to find all possible states containing $i$ and to generate the successors of those states until $i$ has been processed, obtaining in the process a subnetwork of states. For a node cutset, then, choose any in this subnetwork. To find all possible states containing $i$, note that those states are subsets of the uniformly directed cutsets (UDC's) containing $i$ in the AoA representation of the instance. Therefore, finding those subsets involves finding the UDC's containing $i$. Generally, all subsets of those UDC's are considered as candidates for possible states, with the appropriate variations in remaining times of all activities except $i$ . This is potentially a large number of states. But potentially also, many of those states can be quickly eliminated by reduction rules. As an example of a node cutset found in this way, consider the set $\{\{a_1; a_3; a_5\}, \{a_4\}\}$ in Figure 3-8. In that instance, the only UDC containing activity $a_4$ is $\{a_1; a_3; a_4\}$.

One might notice that to reduce the size of the subnetwork containing the cut, it makes sense to choose activities that belong to as few UDC's as possible. In particular, introducing

decomposition in this way is suitable for instances that have few of their activities preceding or succeeding the rest of the activities [32].This decomposition is even more suitable in such instances when all activity durations are identical as then only maximal subsets of the UDC's need be considered; which reduces a subnetwork's size considerably.

Finally, we note, this decomposition approach can be implemented bidirectionally. That is, given a node cutset in the forward network, we can easily construct its corresponding one in the reverse network and hence move from both sides of the cut. In this way, every two processors solve the subproblem of finding the shortest paths between the nodes of opposite node cuts. This procedure would require finding shortest paths from multiple sources of a network, in parallel, instead of repeating several single source searches. But we believe this is not too hard to accomplish efficiently.

## 3.10 Computational Experience

To test the theory developed for the bidirectional approach, the algorithm was coded. The data structure used to store the states was the alphabetic data structure as it avoids the additional code required to obtain a prime upper bound on the number of possible states. The language used was Microsoft Visual C++ 6.0. A vector class and a subset class were used to implement various vector and subset operations.

All the node reduction rules discussed in this chapter were implemented in some form. Corollary 15 was implemented as the 'enum_$d$_submaxmls' procedure of subsection 3.6.2. States were tested for fathoming according to Theorem 17 as they were generated and again, if

[32] These include a type of instances studied by Thomas and Salhi (1997) which the authors refer to as 'two-peak ARLF networks'.

not fathomed, as they were developed.

The LB/UB rule was implemented in a way that takes advantage of the state definition. At the start of the algorithm, given an upper bound $UB$ on the optimal makespan, latest starts and earliest finishes of the activities can be obtained and later easily updated whenever $UB$ is. With that, a lower bound on the length of a start to end path passing through a state $S^f$ of the reduced forward network is then $l^f(S^f) + \max_{i \in S^f}\{EF(i) + rem_dur(i)\}$, where $l^f(S)$ is the label of $S^f$, $EF(i)$ is the earliest finish time of activity $i$ and $rem_dur(i)$ is the remaining duration for $i$ to be completed. A similar bound can be obtained for the reduced reverse network states. Note that in this fashion, the lower bounds are not recomputed from scratch for the uncompleted part of a project but are simply updated.

The cutset dominance rule, on the other hand, was not implemented in full. Only a weak version of it was implemented due to an additional data structure to store cutsets that the full rule requires. This version is different from the original one in that it compares states with the same sets of ongoing and precedence feasible activities rather than states with the same cutsets.

Still another means to reduce the number of nodes generated we utilized is to test newly generated nodes, directly or indirectly, against all undeveloped nodes of the opposite network not just the labeled ones. The effect of adopting such tests is to uncover mixed paths earlier than otherwise would be the case and therefore to update the upper bound on the makespan as soon as is feasible. This, of course, brings in the LB/UB rule into play.

The algorithm was tested on the 480 instances of the J30 test set developed by Kolisch *et al* (1995) and reviewed in Chapter 1. As the optimal makespans of those instances is known, we set the initial lower and upper bounds of the instances at the integers in 90% and $1 + 110\%$ of optimal makespan, respectively. These are not unusually tight bounds as they can be achieved

by existing lower and upper bound procedures.

Comparisons of the bidirectional algorithm with the corresponding forward only and reverse only algorithms were made on an IBM Personal Computer 750 with 167 Mhz Pentium processor and 64 MB of RAM. Though we acknowledge that the codes for the three algorithms can be made a lot more efficient, the comparisons are fair as the only difference between these codes is the stopping criterion which for the bidirectional case relies on mixed path versifications while for the other two relies on labeling the end node of the corresponding reduced state network.

The computational results obtained are summarized in Tables 3.1 to 3.4. Recall that the 480 instances of J30 are in fact 48 groups of 10 instances each where all group instances have the same network complexity, the same resource factors and the same resource strengths.

Table 3.1 presents the per group average computational times, in seconds, of each of the three algorithms. Column 5 shows the number of instances in each group for which the computational time of the Bidirectional algorithm is less than or equal to the corresponding computational time of the other two. Column 6 displays the number of instances for which the Bidirectional algorithm performs at least as well as the average of the other two but not better than both. This average performance of the two unidirectional algorithms is a legitimate point of comparison as it is unknown, a priori, which of the two directions is preferable for any given instance. Column 8 contains the number of instances for which the bidirectional algorithm performed worse than the average but not worse than both the other two. Column 9 is reserved for the number of instances for which the Bidirectional algorithm performed worse than both the forward and reverse counterparts. Column 7 of that table includes the number of instances in the group for which the computational time of all three algorithms did not exceed one second. We believe that the one second precision of our time measurements does not permit drawing meaningful

conclusions with these instances and so we did not include them in the counts of the other four statistics. Finally, the last five entries of the last row of this table comprise the totals in their corresponding columns. The second, third and fourth entries total the computational times of the corresponding algorithm over all 480 J30 instances.

Overall, the bidirectional algorithm performed measurably worse than the average of the forward and reverse counterparts on 27 out of the 480 instances. It performed measurably better than that average on 285 instances. On the remaining 168 instances, all three algorithms performed so fast that we could not detect a measurable difference. The computational times for the instances on which the bidirectional algorithm fails are in Table 3.2. The number of states, or nodes, generated is also included in parentheses besides the computational time.

Looking at these two tables, we notice that the range of computational times for the Bidirectional algorithm is quite large. Many instances were solved in almost zero time while one required 19607 seconds! By comparison the longest running instance of the forward only algorithm required 2686 seconds and that of the reverse algorithm necessitated 1624 seconds. Additionally, for many instances, it seems that when the Bidirectional algorithm performs worst than the other two it does so in a dramatic way. The performance, in those cases, is often worse by an order of magnitude. Now, we believe that most of the computational time of that algorithm is spent either generating states or else checking for mixed paths. We therefore kept track of the number of states generated. Unfortunately, keeping track of the number of mixed path verifications done is problematic due to the bounding rules we used. For all counts of mixed path verifications to be registered we would need to include the bounding rules within the verification routine; which has the unintended effect of very significantly slowing down the bidirectional algorithm to the point where tracking the number of verifications becomes

Table 3.1: J30 - Time Analysis per Group

| Group | Bidirectional | Forward | Reverse | ≤both | ≤average | all≤1 sec | >average | >both |
|---|---|---|---|---|---|---|---|---|
| j301_..sm | 2 | 3 | 7 | 7 | 1 | 2 | 0 | 0 |
| j302_..sm | 0 | 2 | 3 | 6 | 2 | 2 | 0 | 0 |
| j303_..sm | 0 | 1 | 1 | 3 | 0 | 7 | 0 | 0 |
| j304_..sm | 0 | 0 | 2 | 3 | 0 | 7 | 0 | 0 |
| j305_..sm | 40 | 77 | 48 | 8 | 2 | 0 | 0 | 0 |
| j306_..sm | 5 | 19 | 37 | 10 | 0 | 0 | 0 | 0 |
| j307_..sm | 1 | 5 | 7 | 10 | 0 | 0 | 0 | 0 |
| j308_..sm | 0 | 0 | 3 | 4 | 1 | 5 | 0 | 0 |
| j309_..sm | 481 | 117 | 119 | 1 | 2 | 0 | 0 | 7 |
| j3010_..sm | 14 | 84 | 164 | 10 | 0 | 0 | 0 | 0 |
| j3011_..sm | 3 | 10 | 73 | 9 | 1 | 0 | 0 | 0 |
| j3012_..sm | 0 | 1 | 94 | 5 | 2 | 3 | 0 | 0 |
| j3013_..sm | 4972 | 527 | 386 | 0 | 1 | 0 | 0 | 9 |
| j3014_..sm | 72 | 167 | 188 | 8 | 1 | 0 | 1 | 0 |
| j3015_..sm | 7 | 31 | 134 | 10 | 0 | 0 | 0 | 0 |
| j3016_..sm | 0 | 1 | 51 | 5 | 0 | 5 | 0 | 0 |
| j3017_..sm | 0 | 1 | 2 | 4 | 0 | 6 | 0 | 0 |
| j3018_..sm | 0 | 1 | 1 | 0 | 0 | 10 | 0 | 0 |
| j3019_..sm | 0 | 1 | 1 | 4 | 2 | 4 | 0 | 0 |
| j3020_..sm | 0 | 0 | 1 | 0 | 0 | 10 | 0 | 0 |
| j3021_..sm | 2 | 7 | 16 | 10 | 0 | 0 | 0 | 0 |
| j3022_..sm | 2 | 8 | 15 | 9 | 0 | 1 | 0 | 0 |
| j3023_..sm | 0 | 2 | 2 | 5 | 0 | 5 | 0 | 0 |
| j3024_..sm | 0 | 0 | 1 | 2 | 0 | 8 | 0 | 0 |
| j3025_..sm | 40 | 37 | 44 | 6 | 2 | 0 | 0 | 2 |
| j3026_..sm | 2 | 18 | 12 | 9 | 1 | 0 | 0 | 0 |
| j3027_..sm | 1 | 6 | 3 | 9 | 0 | 1 | 0 | 0 |
| j3028_..sm | 0 | 0 | 1 | 3 | 0 | 7 | 0 | 0 |
| j3029_..sm | 215 | 72 | 54 | 2 | 0 | 0 | 0 | 8 |
| j3030_..sm | 5 | 31 | 67 | 10 | 0 | 0 | 0 | 0 |
| j3031_..sm | 2 | 12 | 13 | 10 | 0 | 0 | 0 | 0 |
| j3032_..sm | 0 | 0 | 3 | 3 | 0 | 7 | 0 | 0 |
| j3033_..sm | 0 | 1 | 1 | 0 | 0 | 10 | 0 | 0 |
| j3034_..sm | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 |
| j3035_..sm | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 |
| j3036_..sm | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 |
| j3037_..sm | 1 | 2 | 3 | 9 | 0 | 1 | 0 | 0 |
| j3038_..sm | 0 | 2 | 1 | 7 | 1 | 2 | 0 | 0 |
| j3039_..sm | 0 | 1 | 1 | 2 | 0 | 8 | 0 | 0 |
| j3040_..sm | 0 | 0 | 1 | 1 | 0 | 9 | 0 | 0 |
| j3041_..sm | 2 | 5 | 5 | 10 | 0 | 0 | 0 | 0 |
| j3042_..sm | 1 | 5 | 4 | 10 | 0 | 0 | 0 | 0 |
| j3043_..sm | 1 | 2 | 4 | 8 | 1 | 1 | 0 | 0 |
| j3044_..sm | 0 | 0 | 1 | 2 | 0 | 8 | 0 | 0 |
| j3045_..sm | 2 | 5 | 4 | 10 | 0 | 0 | 0 | 0 |
| j3046_..sm | 3 | 11 | 10 | 10 | 0 | 0 | 0 | 0 |
| j3047_..sm | 1 | 4 | 5 | 9 | 0 | 1 | 0 | 0 |
| j3048_..sm | 0 | 0 | 2 | 2 | 0 | 8 | 0 | 0 |
| Total | 58836 | 12810 | 15935 | 265 | 20 | 168 | 1 | 26 |

Table 3.2: J30 - Instances for which the Bidirectional Algorithm Performs Slower than Average

| Instance | Bidirectional sec (nodes) | Forward sec (nodes) | Reverse sec (nodes) |
|---|---|---|---|
| j309_1.sm | 3165 (149186) | 284 ( 343989) | 181 ( 248505) |
| j309_2.sm | 171 ( 43316) | 49 ( 63994) | 100 ( 131961) |
| j309_3.sm | 165 ( 41891) | 113 ( 120086) | 121 ( 150754) |
| j309_4.sm | 611 (101845) | 210 ( 197532) | 297 ( 323792) |
| j309_8.sm | 107 ( 42380) | 56 ( 67221) | 100 ( 114077) |
| j309_9.sm | 174 ( 37933) | 103 ( 118448) | 56 ( 73537) |
| j309_10.sm | 235 (100919) | 110 ( 123560) | 113 ( 134159) |
| j3013_1.sm | 19607 (400443) | 2686 (1261991) | 1624 (1056598) |
| j3013_2.sm | 5178 (216791) | 338 ( 480615) | 492 ( 689089) |
| j3013_3.sm | 1305 (114019) | 233 ( 282446) | 203 ( 264734) |
| j3013_4.sm | 106 ( 33592) | 103 ( 148931) | 57 ( 91224) |
| j3013_5.sm | 12472 (312178) | 518 ( 774370) | 485 ( 768147) |
| j3013_6.sm | 9437 (262897) | 443 ( 694124) | 536 ( 886635) |
| j3013_7.sm | 714 ( 80823) | 150 ( 219333) | 143 ( 259676) |
| j3013_8.sm | 76 ( 28696) | 47 ( 59361) | 48 ( 65772) |
| j3013_9.sm | 616 ( 83542) | 231 ( 300573) | 182 ( 244882) |
| j3014_2.sm | 375 ( 71980) | 290 ( 266896) | 436 ( 437892) |
| j3025_1.sm | 183 ( 47459) | 89 ( 99895) | 118 ( 145224) |
| j3025_4.sm | 42 ( 22828) | 36 ( 45199) | 32 ( 42192) |
| j3029_1.sm | 329 ( 69590) | 127 ( 163621) | 75 ( 104918) |
| j3029_2.sm | 58 ( 27307) | 57 ( 70486) | 50 ( 58810) |
| j3029_3.sm | 150 ( 33042) | 52 ( 76872) | 43 ( 70949) |
| j3029_4.sm | 40 ( 16036) | 21 ( 30217) | 32 ( 49924) |
| j3029_6.sm | 123 ( 33109) | 49 ( 73155) | 51 ( 83266) |
| j3029_8.sm | 1304 (116465) | 299 ( 422708) | 177 ( 261211) |
| j3029_9.sm | 57 ( 24221) | 46 ( 61871) | 35 ( 46906) |
| j3029_10.sm | 38 ( 6481) | 7 ( 9080) | 11 ( 15269) |

unattractive.

Nevertheless, the number of nodes generated by all three algorithms was obtained. This is presented in Table 3.3 which we organized along the same lines as Table 3.1. Column 6 in that table includes the number of instances that required at most 1000 states for each of the three algorithms. All three algorithms required less than a second to solve these instances and so including them, we believe, would not have been meaningful. The instances for which the Bidirectional algorithm performed worse than the average of the other two in terms of states generated are examined in Table 3.4. We notice that none of these instances took more than four seconds to solve on any of the algorithms and that even on these instances, the Bidirectional algorithm was faster than the average of the other two.

We conclude that the Bidirectional algorithm, as it was implemented and coded, generated significantly less states than the other two on all instances that required more than four seconds and that its under performance in terms of computational time on the instances in Table 3.2 is due to the number of mixed path verifications that were required. We believe this number was large in those cases due to the data structure we used in implementing the algorithm. In retrospect, we think a multi-level bucket data structure that implicitly incorporates the mixed path verification bounding rules, instead of the single level bucket structure we used, would have been more suitable.

But another way to improve the performance is to introduce more state bounding rules; certainly, implement the full version of the Cutset Dominance Rule. For as the number of states decreases, the number of mixed path verifications has to surely decrease by, roughly, a quadratic order. This, unfortunately, not only points out a way to improve the algorithm but also points out its fundamental limitation. At some point, the number of states of an instance

can be so large that mixed path verifications become prohibitive before the burden of generating the states becomes so. Of course now, any algorithm to solve RCPSP is very likely to have serious limitations as the problem itself is NP-Complete. The only question is the point at which these limitations become practically insurmountable.

Still, another way to take advantage of the bidirectional approach is to develop rules, exact or heuristic, to control the speed with which each reduced network is developed. For example, instead of alternatingly developing one node from each network, as the Bidirectional algorithm currently does, it may be worthwhile at some point to develop more than one node from one network before switching to its opposite. This surely takes advantage of each network to a fuller extent than we do.

Table 3.3: J30 - Node Analysis per Group

| Group | Bidirectional | Forward | Reverse | ≤both | ≤average | all≤1000 | >average | >both |
|---|---|---|---|---|---|---|---|---|
| j301_..sm | 2601 | 2471 | 5843 | 0 | 5 | 1 | 1 | 3 |
| j302_..sm | 2980 | 6407 | 9575 | 0 | 8 | 1 | 1 | 0 |
| j303_..sm | 870 | 848 | 557 | 0 | 1 | 6 | 0 | 3 |
| j304_..sm | 303 | 170 | 1255 | 0 | 3 | 7 | 0 | 0 |
| j305_..sm | 16851 | 54783 | 45486 | 10 | 0 | 0 | 0 | 0 |
| j306_..sm | 7096 | 14532 | 30963 | 6 | 4 | 0 | 0 | 0 |
| j307_..sm | 2487 | 3701 | 5880 | 5 | 5 | 0 | 1 | 0 |
| j308_..sm | 599 | 297 | 2332 | 0 | 4 | 4 | 1 | 1 |
| j309_..sm | 59655 | 130219 | 142299 | 10 | 0 | 0 | 0 | 0 |
| j3010_..sm | 21846 | 67715 | 119836 | 9 | 1 | 0 | 0 | 0 |
| j3011_..sm | 5373 | 9077 | 52314 | 5 | 5 | 0 | 0 | 0 |
| j3012_..sm | 747 | 387 | 66892 | 0 | 6 | 2 | 1 | 1 |
| j3013_..sm | 157789 | 496559 | 444308 | 10 | 0 | 0 | 0 | 0 |
| j3014_..sm | 32267 | 154906 | 203028 | 10 | 0 | 0 | 0 | 0 |
| j3015_..sm | 13536 | 28186 | 127519 | 5 | 5 | 0 | 0 | 0 |
| j3016_..sm | 530 | 331 | 43008 | 0 | 5 | 4 | 0 | 1 |
| j3017_..sm | 1052 | 948 | 2112 | 1 | 3 | 6 | 0 | 0 |
| j3018_..sm | 483 | 519 | 458 | 0 | 0 | 10 | 0 | 0 |
| j3019_..sm | 592 | 728 | 461 | 0 | 3 | 6 | 0 | 1 |
| j3020_..sm | 298 | 188 | 377 | 0 | 0 | 10 | 0 | 0 |
| j3021_..sm | 3297 | 6311 | 9252 | 10 | 0 | 0 | 0 | 0 |
| j3022_..sm | 2980 | 6407 | 9575 | 8 | 1 | 1 | 0 | 0 |
| j3023_..sm | 861 | 1813 | 1798 | 5 | 3 | 2 | 0 | 0 |
| j3024_..sm | 342 | 190 | 732 | 0 | 2 | 8 | 0 | 0 |
| j3025_..sm | 18585 | 41107 | 52637 | 10 | 0 | 0 | 0 | 0 |
| j3026_..sm | 3380 | 15130 | 11179 | 8 | 2 | 0 | 0 | 0 |
| j3027_..sm | 1872 | 4564 | 2615 | 4 | 4 | 1 | 1 | 0 |
| j3028_..sm | 234 | 138 | 1372 | 0 | 4 | 6 | 0 | 0 |
| j3029_..sm | 36237 | 99561 | 78047 | 10 | 0 | 0 | 0 | 0 |
| j3030_..sm | 7934 | 29043 | 59756 | 10 | 0 | 0 | 0 | 0 |
| j3031_..sm | 3505 | 10257 | 12811 | 6 | 4 | 0 | 0 | 0 |
| j3032_..sm | 269 | 155 | 2677 | 0 | 3 | 7 | 0 | 0 |
| j3033_..sm | 411 | 540 | 494 | 0 | 0 | 10 | 0 | 0 |
| j3034_..sm | 283 | 259 | 361 | 0 | 0 | 10 | 0 | 0 |
| j3035_..sm | 242 | 266 | 221 | 0 | 0 | 10 | 0 | 0 |
| j3036_..sm | 201 | 148 | 196 | 0 | 0 | 10 | 0 | 0 |
| j3037_..sm | 1573 | 2384 | 2511 | 9 | 1 | 0 | 0 | 0 |
| j3038_..sm | 642 | 1304 | 1382 | 4 | 4 | 2 | 0 | 0 |
| j3039_..sm | 431 | 704 | 458 | 1 | 3 | 6 | 0 | 0 |
| j3040_..sm | 224 | 134 | 456 | 0 | 1 | 9 | 0 | 0 |
| j3041_..sm | 2991 | 5897 | 6538 | 10 | 0 | 0 | 0 | 0 |
| j3042_..sm | 1538 | 4012 | 4138 | 7 | 3 | 0 | 0 | 0 |
| j3043_..sm | 1122 | 2049 | 3456 | 4 | 5 | 1 | 0 | 0 |
| j3044_..sm | 197 | 134 | 738 | 0 | 3 | 7 | 0 | 0 |
| j3045_..sm | 2981 | 5830 | 5836 | 10 | 0 | 0 | 0 | 0 |
| j3046_..sm | 4091 | 10651 | 10890 | 10 | 0 | 0 | 0 | 0 |
| j3047_..sm | 1384 | 3270 | 5111 | 9 | 1 | 0 | 0 | 0 |
| j3048_..sm | 193 | 135 | 1341 | 0 | 2 | 8 | 0 | 0 |
| Total | 4238882 | 12205814 | 15840282 | 205 | 104 | 155 | 6 | 10 |

Table 3.4: J30 - Instances for which the Bidirectional Algorithm Requires more Nodes than Average

| Instance | Bidirectional sec (nodes) | Forward sec (nodes) | Reverse sec (nodes) |
|---|---|---|---|
| j301_1.sm | 2 (3124) | 4 (3539) | 3 (2048) |
| j301_3.sm | 1 (1392) | 2 (1185) | 1 (923) |
| j301_6.sm | 1 (1548) | 1 (1001) | 1 (1350) |
| j301_8.sm | 1 (2568) | 3 (1940) | 3 (2072) |
| j302_7.sm | 1 (1381) | 2 (1464) | 1 (842) |
| j303_3.sm | 1 (3155) | 4 (2823) | 2 (1701) |
| j303_5.sm | 0 (1348) | 1 (832) | 2 (993) |
| j303_9.sm | 0 (1133) | 1 (691) | 1 (1060) |
| j307_1.sm | 2 (2371) | 2 (1355) | 4 (3039) |
| j308_3.sm | 1 (898) | 0 (353) | 2 (1235) |
| j308_9.sm | 1 (1398) | 1 (721) | 1 (951) |
| j3012_8.sm | 2 (2759) | 3 (1497) | 2 (1417) |
| j3012_10.sm | 0 (937) | 0 (426) | 1 (1261) |
| j3016_3.sm | 1 (1028) | 2 (753) | 1 (631) |
| j3019_1.sm | 1 (1208) | 2 (981) | 1 (844) |
| j3027_8.sm | 1 (2632) | 3 (2440) | 3 (2744) |

# Chapter 4

# Future Research

Throughout this dissertation, many issues were raised which justifiably could serve as subjects for more detailed investigations. Many of these have a direct impact on the computational performance of the bidirectional approach we proposed. Others relate to its possible extensions or even to the separate approaches we discussed. This chapter is devoted to reiterating these issues and suggesting a few more.

The first of these issues is how to take advantage of the special structure of the integer programming model for RCPSP presented in Chapter 2. Recall that the constraint matrix in that model was block diagonal with coupling constraints; the same type that the LP decomposition principle was developed for. The difference here is that some of our model variables are 0/1 rather than continuous.

Moving to Chapter 3, the first issue that merits further research is the conjecture we state in section 5. It surmises, recall, that RCPSP instances tend to be easier as the proportion of their activities having minimum duration increases. Recall also that this conjecture is not likely to be established by analytical means but should rather be supported or refuted based on statistical

tests performed on data collected in solving appropriate samples of RCPSP instances.

Another point for future research in the same chapter should be the development of a lower bound for the problem that is tighter than the CPM bound we used yet is as easily, or almost as easily, implementable. Again, the desirable feature of our CPM bound is the ease with which it can be updated; thereby, eliminating the need for repeated re-computations.

A fourth issue for future research is as follows. From the implementation point of view, a full implementation of the Cutset Dominance rule that fits well with the adopted data structure is instrumental. For example, an efficient means to eliminate 'old' cutsets generated so as to make room for 'new' ones could be an important factor in the solution of relatively large size instances.

More generally, introducing further node reduction rules could significantly improve the performance of the bidirectional approach. The same goes for mixed path verification reduction rules. Ideally, all of these reduction rules should be implicitly implemented, or at the least easily implementable, in the data structure selected. This suggests designing the data structure with all of these rules in mind rather than coming up with the data structure and implementing the rules as an afterthought. Of course, this may complicate the design of the data structure more than is desirable; in which case necessary compromises would have to be made. But at the least, these compromises would have been thought out in advance.

A sixth issue for future research is the development of rules to control the expansion of the reduced networks. Currently, each algorithm is given an equal chance for expansion at every iteration. But there could be a situation where one network's size could be increasing a lot faster than the other. In that case, it may be preferable to develop the smaller network faster than its opposite in the hope that the algorithm terminates earlier than otherwise would be the

case. The mechanism to successfully accomplish this is not clear to us at this point.

A seventh issue for future research is the decomposition of an instance and the development of a parallel algorithm to solve the problem. We discussed this issue in section 9 of the preceding chapter but much details and computing experience have yet to be worked out.

An eighth issue is the development and evaluation of heuristics based on the bidirectional approach. This is an important issue as our exact approach, similar to every other exact approach for an NP-Complete problem, cannot be practical for large enough problems.

Finally, a ninth issue for future research is the implementation of our algorithm to preemptive RCPSP. We touched upon this issue in several places but especially in section 9 of Chapter 3 when we discussed Shortest Path in the single arc length case. It certainly is worth more attention.

# Bibliography


[1] Agrawal, M.K.; S.E. Elmaghraby and W.S. Herroelen (1996): DAGEN: A Generator of Testsets for Project Activity Nets. European Journal of Operational Research 90, 376-382.

[2] Ahuja, R.K.; K. Melhorn; J.B. Orlin and R.E. Tarjan (1990): Faster Algorithms for the Shortest Path Problem. Journal of the Association for Computing Machinery 37, 213-223.

[3] Alvarez-Valdés, R. and J.M. Tamarit (1989): Heuristic Algorithms for Resource Constrained Project Scheduling: A Review and Empirical Analysis. In: R. Słowiński and J. Weglarz. (Ed.): Advances in Project Scheduling. Elsevier. New York, New York.

[4] Alvarez-Valdés, R. and J.M. Tamarit (1993): Project Scheduling Polyhedron: Dimension, Facets and Lifting Theorems. European Journal of Operational Research 67, 202-220.

[5] Bandelloni, M.; M. Tucci and R. Rinaldi (1994): Optimal Resource Leveling Using Non-Serial Dynamic Programming. European Journal of Operational Research 78, 162-177.

[6] Bein, W.W.; J. Kamburowski and M.F.M. Stallmann (1992): Optimal Reduction of Two-Terminal Directed Acyclic Graphs, SIAM Journal on Computing 21, 1112-1129.

[7] Bell, C.E. and K. Park (1990): Solving Resource-Constrained Project Scheduling Problems by A* Search. Naval Research Logistics 37, 61-84.

[8] Blazewicz, J.; J.K. Lenstra and A.H.G. Rinnooy Kan (1983): Scheduling Subject to Resource Constraints: Classification and Complexity. -Discrete Applied Mathematics 5, 11-24.

[9] Boctor. F (1990): Some Efficient Multi-Heuristic Procedures for Resource-Constrained Project Scheduling. European Journal of Operational Research 49, 3-13.

[10] Böttcher, J; A. Drexl; R. Kolisch and F. Salewski (1999): Project Scheduling under Partially Renewable Constraints. Management Science 45, 543-559.

[11] Bouleimen, K. and H. Lecocq (1998): A new Efficient Simulated Annealing Algorithm for Resource-Constrained Project Scheduling Problem. Working Paper, Service de Robotique et Automatisation, Université de Liège, Belgique. To appear in the European Journal of Operational Research.

[12] Bressoud, D.M. (1989): Factorization and Primality Testing. Springer-Verlag New York, Inc. New York.

[13] Brucker, P.; S. Knust; A. Schoo and O. Thiel (1998): A Branch and Bound Algorithm for the Resource-Constrained Project Scheduling Problem. European Journal of Operational Research 107, 272-288.

[14] Brucker, P; A.Drexl; R. Möhring; K. Neumann and E. Pesch (1999): Resource Constrained Project Scheduling: Notation, Classification, Models and Methods. European Journal of Operational Research 122, 3-41.

[15] Brucker, P. and S. Knust (1999): Solving Large-Sized Resource-Constrained Project Scheduling Problems. In Weglarz, J. (Ed.): Project Scheduling: Recent Models, Algorithms and Applications. Kluwer Academic Publishers. Boston, Massachusetts.

[16] Cherkassky, B.V.; A.V. Goldberg and T. Radzik (1996): Shortest Paths Algorithms: Theory, Experiment and Evaluation. Mathematical Programming 73, 129-174.

[17] Cherkassky, B.V.; A.V. Goldberg and C. Silverstein (1999): Buckets, Heaps and Monotone Priority Queues. SIAM Journal of Computing 28, 1326-1346.

[18] Christofedes, N.; R. Alvarez-Valdés and J.M. Tamarit (1987): Project Scheduling with Resource Constraints: A Branch and Bound Approach. European Journal of Operational Research 29, 262-273.

[19] Cooper, D. F. (1976): Heuristics for Scheduling Resource-Constrained Projects: An Experimental Investigation. Management Science 22, 1186-1194.

[20] Cormen, T.H.; C.E. Leiserson and R.L. Rivest (1990): Algorithms. The MIT Press, Cambridge, Massachsetts.

[21] Dar-El, E.M. (1973): A Heuristic Technique for Balancing Single-Model Assembly Lines. AIIE Transactions 5, 343-356.

[22] Davies, E. M. (1973): An Experimental Investigation of Resource Allocation in Multiactivity Projects. Operational Research Quarterly 24, 587-591.

[23] Davis, E.W. and G.E. Heidorn (1971): Optimal Project Scheduling under Multiple Resource Constraints. Management Science 17, B803-B816.

[24] Davis, E.W. (1975): Project Network Summary Measures and Constrained Resource Scheduling. IIE Transactions 7, 132-142.

[25] Demeulemeester E.L. and W.S. Herroelen (1992): A Branch and Bound Procedure for the Multiple Resource Constrained Project Scheduling Problems. Management Science 38, 1803-1818.

[26] Demeulemeester E.L.; B. Dodin and W.S. Herroelen (1993): A Random Activity Network Generator. Operations Research 41, 972-980.

[27] Demeulemeester E.L.; W.S. Herroelen; W.P. Simpson; S. Baroum; J.H. Patterson and K.K. Yang: On a Paper by Christofedes *et al* for Solving the Multiple-Resource Constrained, Single Project Scheduling Problem. European Journal of Operational Research 76, 218-228.

[28] Demeulemeester, E.L. (1995): Minimizing Resource Availability Costs in Time-Limited Project Networks. Management Science 41, 1590-1598.

[29] Demeulemeester E.L. and W.S. Herroelen (1996): An Efficient Optimal Solution Procedure for the Preemptive Resource Constrained Project Scheduling Problem. European Journal of Operational Research 90, 334-348.

[30] Demeulemeester E.L. and W.S. Herroelen (1997 a): New Benchmark Results for the Resource Constrained Project Scheduling Problem. Management Science 43, 1485-1492.

[31] Demeulemeester E.L. and W.S. Herroelen (1997 b): A Branch and Bound Procedure for the Generalized Resource-Constrained Project Scheduling Problem. Operations Research 45, 201-212.

[32] Denardo, E.W. and B.L. Fox (1979): Shortest Routes Methods: Reaching, Pruning and Buckets. Operations Research 27, 161-186.

[33] De Reyck, B. and W.S. Herroelen (1996): On the Use of the Complexity Index as a Measure of Complexity in Activity Networks. European Journal of Operational Research 91, 347-366.

[34] De Reyck, B. and W.S. Herroelen (1998): An Optimal Procedure for the Resource Constrained Project Scheduling Problem with Discounted Cash Flows and Generalized Precedence Relations. Computers and Operations Research 25, 1-17.

[35] Dial, R.B. (1969): Algorithm 360: Shortest Path Forest with Topological Ordering. Communications of the Association for Computing Machinery 12, 632-633.

[36] Dijkstra, E.W. (1959): A Note on Two Problems in Connection with Graphs. Numerische Mathematik 1, 269-271.

[37] Drexl, A; J. Juretzka; F. Salewski and A. Schirmer (1999): New Modeling Concepts and Their Impact on Resource Constrained Project Scheduling. In: Weglarz, J. (Ed.): Project Scheduling: Recent Models, Algorithms and Applications. Kluwer Academic Publishers. Boston, Massachusetts.

[38] Elmaghraby, S. E. (1977): Activity Networks: Project Planning and Control by Network Models. John Wiley and Sons, New York.

[39] Elmaghraby, S. E. and W.S. Herroelen (1980): On the Measurement of Complexity in Activity Networks. European Journal of Operational Research 5, 223-234.

[40] Evans, J.R. and E. Minieka (1992): Optimization Algorithms for Networks and Graphs. Marcel Dekker Inc., New York, New York.

[41] Fredman, M.L. and R.E. Tarjan (1987): Fibonacci Heaps and their Uses in Improved Network Optimization Algorithms. Journal of the Association for Computing Machinery 34, 596-615.

[42] Fredman, M.L. and Willard, D.E. (1994): Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. Journal of Computing and System Science 48, 533-551.

[43] Garey, M.R.; D.S. Johnson and R. Sethi (1976): The Complexity of Flowshop and Jobshop Scheduling. Mathematics of Operations Research 1, 117-129.

[44] Goldberg, A.V. and C. Silverstein (1997): Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets. In Lecture Notes in Economics and Mathematical Systems 450, 292-327. P.M. Pardalos, D.W. Hearn and W.W. Hages (eds), Springer-Verlag, New York.

[45] Golumbic, M.C. (1980): Algorithmic Graph Theory and Perfect Graphs. Academic Press, New York.

[46] Hartmann, S (1998): A Competitive Genetic Algorithm for Resource-Constrained Project Scheduling. To appear in Naval Research Logistics.

[47] Hartmann, S and A. Drexl (1998): Project Scheduling with Multiple Modes: A Comparison of Exact Algorithms. Networks 32, 238-257.

[48] Herroelen, W; B. De Reyck and E. Demeulemeester (1998): Resource Constrained Project Scheduling: A Survey of Recent Developments. Computers and Operations Research 25, 279-302.

[49] Herroelen, W. ; E. Demeulemeester and B. De Reyck (1999): A Classification Scheme for Project Scheduling. In: Weglarz, J. (Ed.): Project Scheduling: Recent Models, Algorithms and Applications. Kluwer Academic Publishers. Boston, Massachusetts.

[50] Herroelen, W. and B. De Reyck (1999): Phase Transitions in Project Scheduling. Journal of the Operational Research Society 50, 148-156.

[51] Icmeli, O and W.L. Rom (1996): Solving the Resource Constrained Project Scheduling Problem with the Optimization Subroutine Library. Computers and Operations Research 23, 801-817.

[52] Icmeli, O and W.L. Rom (1997): Ensuring Quality in Resource Constrained Project Scheduling. European Journal of Operational Research 103, 483-496.

[53] Icmeli, O and Rom W.L. (1998): Analysis of the Characteristics of Projects in Diverse Industries. Journal of Operations Management 16, 43-61.

[54] Kaimann, R.A. (1974): Coefficient of Network Complexity. Management Science 21, 172-177.

[55] Kaimann, R.A. (1975): Coefficient of Network Complexity: Erratum. Management Science 21, 1211-1212.

[56] Klein, R. (2000): Scheduling of Resource Constrained Projects. Kluwer Academic Publishers, Boston, Massachusetts.

[57] Kohlmogren, U.; H. Schmeck and K. Hasse (1999): Experiences with Fine-Grained Parallel Algorithms. Annals of Operations Research 90, 203-219.

[58] Kolisch, R. (1995): Project Scheduling under Resource Constraints- Efficient Heuristics for Several Classes. Phisica, Heidelberg.

[59] Kolisch, R.; A. Sprecher and A.Drexl (1995): Characterization and Generation of a General Class of Resource-constrained Project Scheduling Problems. Management Science 41, 1693–1703.

[60] Kolisch, R. (1996): Serial and Parallel Resource-Constrained Project Scheduling Methods Revisited: Theory and Computation. European Journal of Operational Research 90, 320-333.

[61] Kolisch, R. and A. Drexl (1996): Adaptive Search for Solving Hard Project Scheduling Problems. Naval Research Logistics 43, 23-40.

[62] Kolisch, R.; A. Sprecher (1996): PSPLIB - A Project Scheduling Problem Library. European Journal of Operational Research 96, 205-216.

[63] Kolisch, R. and S. Hartmann (1999): Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis. In: Weglarz, J. (Ed.): Project Scheduling: Recent Models, Algorithms and Applications. Kluwer Academic Publishers. Boston, Massachusetts.

[64] Kolisch, R.; C. Schwindt and A. Sprecher (1999): Benchmark Instances for Project Scheduling Problems. In: Weglarz, J. (Ed.): Project Scheduling: Recent Models, Algorithms and Applications. Kluwer Academic Publishers. Boston, Massachusetts.

[65] Krishnamoorthy, M. and N. Deo (1979): Complexity of the Minimum Dummy Activities Problem in a PERT Network, Networks 9, 189-194.

[66] Kurtulus, I. and E.W. Davis (1982): Multi-Project Scheduling: Categorization of Heuristic Rules Performance. Management Science 28, 161-172.

[67] Michael, D.; J. Kamburowski and M. Stallmann (1993): RAIRO Rechèrches Opérationelles 27, 153-168.

[68] Mingozzi, A; V. Maniezzo; S. Ricciardelli and L. Bianco (1998): An Exact Algorithm for the Resource-Constrained Project Scheduling Problem Based on a New Mathematical Formulation. Management Science 44, 714-729.

[69] Möhring, R.H.;F.J. Radermacher and G. Weiss (1984): Stochastic Scheduling Problems I - General Strategies. Zeitschrift für Operations Research Se. A 28, 193-260.

[70] Möhring, R.H.;F.J. Radermacher and G. Weiss (1984): Stochastic Scheduling Problems II - Set Strategies. Zeitschrift für Operations Research Se. A 29, 65-104.

[71] Möhring, R.H. (1985): Algorithmic Aspects of Comparability Graphs and Interval Graphs. In: Rival, I. (Ed.): Graphs and Order: The Role of Graphs in the Theory of Ordered Sets and its Applications. Nato Advanced Science Institute, Series C, Mathematical and Physical Sciences, 41-101.

[72] Möhring, R.H. and F.J. Radermacher (1989): The Order Theoretic Approach to Scheduling - The Deterministic Case. In: Slowinski, R. and J. Weglarz (Ed.): Advances in Project Scheduling. Elsevier Science Publishing Company, New York, NY.

[73] Möhring, R.H.; A.S. Schultz; F. Stork and M. Uetz (1999): Resource-Constrained Project Scheduling: Computing Lower Bounds by Solving Minimum Cut Problems. In: Nesetril, J. (Ed.): Algorithms - ESA 1999: Proceedings of the 7th Annual European Symposium, Prague, Czech Republic, July 16-18, 1999. Lecture Notes in Computer Science 1643.

[74] Nemhauser, G.L. and L.E. Trotter (1975): Vertex Packings: Structural Properties and Algorithms. Mathematical Programming 8, 232-248.

[75] Özdamar, L. and Ulusoy, G. (1995): A Survey on the Resource Constrained Project Scheduling Problem. IIE Transactions 27, 574-586.

[76] Pascoe, T.L. (1966): Allocation of Resources - CPM. Revue Française de Rechèrches Opérationelles 38, 31-38.

[77] Patterson, J.H.; R. Slowinski; F.B. Talbot and J. Weglarz (1989): An Algorithm for a General Class of Precedence and Resource Constrained Scheduling Problems. In: Slowinski, R. and J. Weglarz (Ed.): Advances in Project Scheduling. Elsevier Science Publishing Company, New York, NY.

[78] Pinedo, M. (1995): Scheduling - Theory, Algorithms and Systems. Prentice-Hall, Englewood Cliffs, New Jersey.

[79] Pritsker, A.A.B.; L.J. Watters and P.M. Wolfe (1969): Multiproject Scheduling with Limited Resources: A Zero-One Programming Approach. Management Science 16, 93-108.

[80] Raman, R. (1997): Recent Results on the Single-Source Shortest Paths Problem. SIGACT News 28, 81-87.

[81] Rana, K. (1992): A Decomposition Technique for Mixed Integer Programming Problems. Computers and Operations Research 19, 505-519.

[82] Reisel, H. (1994): Prime Numbers and Computer Methods for Factorization, Second Edition. Bikhäuser. Boston, Massachusetts.

[83] Ribenboim, P. (1995): The New Book Of Prime Number Records. Springer-Verlag New York, Inc. New York.

[84] Sankaran, J.K.; D.L. Bricker and S.H. Jyang (1999): A Strong Fractional Cutting-Plane Algorithm for Resource-Constrained Project Scheduling. International Journal of Industrial Engineering 6, 99-111.

[85] Schäffter, M. (1997): Scheduling with respect to Forbidden Sets. Discrete Applied Mathematics 72, 155-166.

[86] Schirmer, A. (2000): Case-Based Reasoning and Improved Adaptive Search for Project Scheduling. Naval Reseach Logistics 47, 201-222.

[87] Schrage, Linus (1970): Solving Resource-Constrained Network Problems by Implicit Enumeration - Non-Preemptive Case. Operations Research 18, 263-278.

[88] Sprecher, A (1994): Resource-Constrained Project Scheduling: Exact Methods for the Multi-Mode Case. Lecture Notes in Economics and Mathematical Systems, No. 409. Springer, Berlin.

[89] Sprecher, A.; R. Kolisch and A.Drexl (1995): Semi-active, Active and Non-delay Schedules for the Resource Constrained Project Scheduling Problem. European Journal of Operational Research 80, 94-102.

[90] Sprecher, A. and A.Drexl (1995): On Semi-Active Timetabeling in Resource Constrained Project Scheduling. Management Science 45, 452-454.

[91] Sprecher, A. and A.Drexl (1998): Solving Multi-Mode Resource-Constrained Project Scheduling Problems by a Simple, General and Powerful Sequencing Algorithm. European Journal of Operational Research 107, 431-450.

[92] Sprecher, A. (2000): Scheduling Resource-Constrained Projects Competitively at Modest Memory Requirements. Management Science 46, 710-723.

[93] Slowinski, R. and J. Weglarz (1989): Advances in Project Scheduling. Elsevier Science Publishing Company, New York, New York.

[94] Stinson, J.P.; E.W. Davis and B.M. Khumawala (1978): Multiple Resource Constrained Scheduling Using Branch and Bound. AIIE Transactions 10, 252-259.

[95] Sweeney, D.J. and R.A. Murphy (1979): A Method of Decomposition of Integer Programs. Operations Research 27, 1128-1141.

[96] Talbot, F.B. and J.H. Patterson (1978): An Efficient Integer Programming Algorithm with Network Cuts for Solving Resource-Constrained Scheduling Problems. Management Science 24, 1163-1174.

[97] Thesen, A. (1977): Measures of the Restrictiveness of Projects Networks. Networks 7, 193-208.

[98] Thomas, .R. and S. Salhi (1997): An Investigation into the Relationship of Heuristic Performance with Network Resource Characteristics. Journal of the Operational Research Society 48, 34-43.

[99] Thorup, M. (1999): Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. Journal of the Association for Computing Machinery 46, 362-394.

[100] Thorup, M. (2000): Floats, Integers and Single-Source Shortest Paths. Journal of Algorithms 35, 189-201.

[101] Van Emde Boas, P.; R. Kaas and E. Ziljstra (1977): Design and Implementation of an Efficient Priority Queue. Mathematical Systems Theory 10, 99-127.

[102] Xue, J. (1994): Edge-Maximal Triangulated Subgraphs and Heuristics for the Maximum Clique Problem. Networks 24, 109-120.