ON THE COMPLEXITY OF PRECEDENCE CONSTRAINED SCHEDULING

TR-86-11

by

KRISHNA SSV. PALEM, B.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May, 1986

PREFACE

Scheduling theory which concerns the optimal allocation of resources to contending activities or computations is an area which has seen tremendous growth over the past decades. Much of the emphasis in this area centers around efficient algorithms for finding such optimum schedules. In situations where the nature of the computations and the resources are known apriori, the resulting scheduling problems display a decidedly combinatorial nature, which attracted several researchers from the active field of combinatorial optimization. However, scheduling theory also abounds in problems, which like a number of problems from combinatorial optimization, have defied repeated attacks aimed at synthesizing good algorithms. In such situations, computational complexity theory and more specifically, the theory of NP-completeness, has come to our rescue by giving us formal techniques through which we can establish that in all probability, we cannot hope to ever find efficient algorithms for many of these scheduling problems.

Within combinatorial optimization, scheduling theory stands out with respect to the number of individual problems whose complexity has been resolved, through an application of algorithmic techniques together with the theory of NP-completeness. In a sense, there is a combinatorial explosion, not only in the time and space requirements of certain scheduling problems, but also in the number of different problems from this area which have been classified into being polynomially solvable or NP-hard. Yet, despite this wealth of knowledge, this area is lacking in unifying principles or theories which capture the similarity in complexity characteristics of several of the seemingly different subproblems of the scheduling problem. In fact, the large volume of results in this area seem to reinforce the view that many of the well solvable subproblems from scheduling have associated polynomial time algorithms for quite different reasons, while their generalizations are known to be NP-hard. This is in contrast with other areas from combinatorial optimization, whose complexity, as well as the behavior at optimality of the solutions of these problems, have deep unifying characterizations.

This was the starting point of my investigation into the scheduling area, and I was really intrigued by the lack of such unifying characterizations despite the large number of individual complexity results. In particular, I was interested in scheduling problems which allow task systems with non-trivial precedence relationships. These precedence relationships account for interrelationships between the (computational) tasks, induced by data and control dependencies. Moreover, motivated by the current developments in the parallel computation field which have gained impetus from recent technological advances including VLSI, I was especially considering scenarios in which these precedence-constrained tasks are to be executed in parallel on multipipeline and multiprocessor systems, which might even have many different types of specialized processors in them.

In the context of such scheduling problems, I have been able to show in this dissertation that many of the apparently unrelated results in this area are really variants of a single class of problems, with common properties which determine their complexity. In particular, I have been able to derive a single *intrinsically ordered convergence* property under which scheduling problems are polynomial, and which is at the heart of the results of several investigators in this field.

An extremely interesting aspect of intrinsically ordered convergence is that it draws upon our ability to characterize conditions under which schedules remain optimum. This approach towards studying and characterizing the solution domain of an optimization problem is characteristic of the philosophy of combinatorial optimization. These results pertaining to the behavior of schedules at optimality are presented in Chapters 3 and 4, and are developed within the context of a hypergraph theoretic formulation of the scheduling problem; in this formulation, a schedule is an appropriately constrained *sequence* of edges of the hypergraph defined by the task set and therefore, each edge in such a sequence is a snapshot of all the tasks with identical start and finish times in the schedule. The complexity related issues are brought into play in Chapter 5, and combined with the results from Chapter 4 which gives us the central IO convergence property. The above mentioned unifying framework is completed in Chapter 6, and a detailed implementation of an algorithm which produces optimum schedules from IO convergent in-

stances is described in Chapter 7. Since this work draws heavily, both in philosophical and concrete terms, from the fields of combinatorial optimization and complexity theory, I have included a fairly detailed introduction to many of the results from these two fields in Chapter 1, especially as they relate to our problem on hand. Also, since the precedence constrained problem with its many versions and parameters can be quite specialized at times, I have decided to include a detailed description of the basic problem and its hypergraph theoretic version, respectively in Chapters 1 and 2.

Many individuals have contributed, both professionally and personally, towards the successful completion of my research effort. I am most grateful to Donald Fussell, for his advice, constant encouragement and support. He was an invaluable source of inspiration and guidance as my dissertation supervisor, which helped in making my pursuit of a Ph.D. an extremely enjoyable experience indeed. I am equally thankful to Ashley Welch for his help, for guiding me into the problem area, and for jointly supervising my dissertation. James Bitner spent an extraordinary amount of time with me during my years as a doctoral student while I grappled with a variety of topics ranging from combinatorial optimization and combinatorics to complexity, in addition to evincing an active interest in my own research; for this I am extremely thankful. I am also indebted to James Browne, who constantly helped me in evaluating my work and in placing it in the proper perspective. In addition to the above mentioned individuals, Bruce Buckman and Miroslaw Malek also served on my dissertation supervisory committee, and I wish to thank them for their time. It has also been a rare pleasure to study under Norman Martin, whose keen insight into logic, recursive function theory, and metatheory, and his philosophical orientation, have helped me in many ways. Zvi Kedem and Rao Kosaraju also deserve my gratitude for their many helpful remarks and criticisms.

Besides these individuals who contributed directly towards my dissertation, I am also thankful to several workers and authors whose work has helped and influenced me in several ways. In this respect, my initiation into this dissertation area started with Michael Gary and David Johnsons' lucid treatment of the basic theory of NP-completeness in their book: 'Computers and Intractability - A guide to the Theory of NP-completeness'. Also, their

ground breaking research into scheduling problems has formed the basis of my own research to a significant extent. Christos Papadimitriou, Kenneth Steiglitz, and Eugene Lawler, whose books and results blend ideas from combinatorial optimization, and algorithms and complexity, in many beautiful ways, also stand out in this respect. B. J. Lageweg, Jan Karel Lenstra and Alexander Rinnooy Kan, who through their work in combinatorial optimization - especially in scheduling theory - have also been a source of inspiration to me. My task of keeping abreast of the ocean of results in scheduling was greatly simplified, thanks to their novel and integrated approach towards compiling them. Finally, there are several other individuals whose work from the scheduling and closely related areas has influenced me in several ways, and without attempting to be complete, I would like to mention some of them: John Bruno, Edward Coffman, Ronald Graham, T.C.Hu, Clyde Monma, Sartaj Sahni, Ravi Sethi, and Jeffrey Ullman.

I would also like to thank my family and especially my parents for the many sacrifices that they have made, and for their untiring emphasis on my educational advancement, without which this work would not have been possible. Finally, I am thankful to my many friends, who through their unfailing support, understanding, and candid criticism, have helped me in striving towards a deeper understanding of a wide range of issues.

A significant part of this research was supported by the Office of Naval Research under contract N0014-83-K-0730.

TABLE OF CONTENTS

Preface iv

Table of Contents viii

Chapter 1. Introduction 1

1.1. Foundations 1
1.2. The Precedence Constrained Scheduling Problem 9
1.2.1. Schedules with Minimum Tardiness 16
1.2.2. Scheduling Pipelined Multiprocessors 17
1.3. Structure in Combinatorial Optimization 24

Chapter 2. On Hypergraphs and Scheduling 32

2.1. A Choice of Characterizations 33
2.1.1. Matroids 33
2.1.2. Polytopes 36
2.2. Formulating Schedules through Hypergraphs 39
2.3. On Reducing Makespan Minimization to Tardiness Minimization 45
2.4. Factors Affecting the Feasibility of Sequences with Zero Cost 49
2.4.1. The Role of the Bound Vector 49
2.4.2. Predecessor-Successor Interactions and Cost 51
2.5. On Characterizing the Feasible Set of Sequences and Some Properties 55

Chapter 3. On Projections and Affined Sequences 59

3.1. Selecting Elements Through Projections 60
3.2. Affined Sequences 63
3.3. Affined Sequences and their Cost 66

Chapter 4. On Relating Affined and Optimum Sequences 70

4.1. A Technique for Proving the Optimality of Affined Sequences 71
4.1.1. The Proof Technique 73
4.2. Perturbing Sequences Through Shifting 79
4.3. Properties Preserved Under Shifting 85
4.4. On Affined and Optimum Sequences 94

Chapter 5. Obstructions and Permutation Lists 97

5.1. A Constructive Method for Generating Sequences 98
5.1.1. The Generalized Schedule Construction Method 99
5.2. A Sufficient Condition for the Optimization Problem to be Tractable 103
5.3. Some Properties of Generated Sequences 111

Chapter 6. Unifying the Polynomially Solvable Subdomains 117

6.1. Saturated and Unsaturated Subsequences 118
6.2. Unification Results 124
6.3. More Unification Results 132

Chapter 7. The Algorithm and its Complexity 142

7.1. Algorithmic Notation 142
7.1.1. Data Structures 143
7.1.2. Notation 145
7.2. Details of the Algorithm 146
7.2.1. The Data Structures 146

7.2.2. The Algorithm 151
7.2.3. Complexity of Procedure Modify 156
7.3. Enumerating Sequences from Lists 158

Chapter 8. Concluding Remarks 162

Bibliography 168

LIST OF TABLES

Table 1-1: Polynomially solvable subdomains of the makespan minimization version of the scheduling problem. 14

Table 1-2: Polynomially solvable subdomains of the tardiness minimization version of the scheduling problem. 17

Table 1-3: Polynomially solvable subdomains with pipelined processors from [4]. 25

LIST OF FIGURES

Figure 1-1:	The precedence graph and class function of an example instance.	11
Figure 1-2:	A schedule for the instance of Figure 1-1 where each vertical line indicates the start or the finish time of a task.	12
Figure 1-3:	The process of slicing : The instance in (i) is replaced by its pipelined equivalent in (ii) by slicing the tasks and resources into four.	18
Figure 1-4:	A schedule for the instance of Figure 1-1 with two stage pipelined processors.	24
Figure 2-1:	Four of the 511 edges from the hypergraph H associated with the instance of Figure 1-1.	40
Figure 2-2:	The schedule of Figure 1-2 represented as a sequence of edges of the corresponding hypergraph.	41
Figure 2-3:	Representing the schedule of Figure 1-4 in terms of a sequence of hypergraph edges.	41
Figure 2-4:	Reducing the makespan minimization problem to the tardiness minimization problem.	46
Figure 2-5:	(i). Transforming the instance of Figure 1-1 by adding the function f_u to it and (ii.) the tardiness of these elements in the schedule of Figure 1-2.	47
Figure 2-6:	The precedence graph and the cost-determining function of an instance which shows the effect of the saturation indices on cost.	50
Figure 2-7:	The precedence graph and the cost-determining function of an instance.	51

Figure 2-8:	Two equivalent sequences for the instance from Figure 2-7 with a cost of : (i) one and (ii) Zero.	52
Figure 3-1:	The elements selected by the projection operation.	62
Figure 3-2:	The critical distance of element a_j with respect to d' and x.	64
Figure 3-3:	The distance between elements a_i and a_j in sequence S.	65
Figure 3-4:	The relationship between subsets $S_\emptyset$ and S_α of a feasible set S whenever $S_\emptyset$ is non-empty.	67
Figure 4-1:	Property (**) satisfied by the mapping '$\rightarrow$'.	74
Figure 4-2:	Property (***) satisfied by the mapping '$\rightarrow$'.	75
Figure 4-3:	Sequence $\overline{S}$ is isomorphic within shifting by X to sequence S.	81
Figure 4-4:	The relationship 'isomorphic within shifting by X to' captured by the mapping F.	83
Figure 4-5:	The effect of perturbation on cost.	83
Figure 4-6:	The relationship between n(i,d ',x) and $\overline{n}$(i ',d '',y) stated in Lemma 4.1.	88
Figure 4-7:	The property of sequences asserted in the Shifting Lemma.	88
Figure 4-8:	The relationship between S_α and S_{min} in an arbitrary feasible set S.	95
Figure 5-1:	Identifying subdomains of an optimization problem through an algorithm A.	98
Figure 5-2:	Alternate lists for the instance of Figure 2-1.	100
Figure 5-3:	The two steps of the generalized sequence construction method.	102
Figure 5-4:	Refining the GSC method to ensure that the resulting method always solves the optimization problem.	105
Figure 5-5:	Refining the AGSC method to run in polynomial time.	110

Figure 6-1:	The edges of a (i) saturated and (ii) unsaturated subsequence of a projected sequence.	119
Figure 6-2:	The relevant components of an instance including a classification of the elements into various 'types'.	125
Figure 6-3:	The type graph of the instance from Figure 6-2.	126
Figure 6-4:	The relationship between the elements at the various levels of a cyclic forest.	131
Figure 7-1:	The list (of lists) memory.	150

Chapter 1

Introduction

1.1 Foundations

The field of combinatorial optimization has grown tremendously, especially in the last twenty years. Generally speaking, combinatorial optimization refers to problems which can be formulated as *linear programs* or *integer linear* programs. In the latter case, we are quite often interested in specialized problems which have 'additional structure', over and above that captured by a general integer linear programming formulation. These specially structured problems include such well known optimization problems as finding *shortest paths, minimum spanning trees, maximum flows, maximum matchings,* and *shortest tours* (the *travelling salesman problem*) in directed and undirected graphs. These graphs are often referred to as networks and as a consequence, the above mentioned optimization problems are quite often collectively referred to as *network optimization* problems, due to the obvious way in which they are formulated in terms of networks.

Many of the results in combinatorial optimization concern efficient algorithms for solving these problems. These algorithms are efficient in the sense that they run in time proportional to a polynomial function of the problem size (more precisely the input length); this widely understood and accepted characterization of efficiency is attributed to be originally due to Cobham [7] and Edmonds [19]. There are several good books which treat al-

gorithms for network optimization and related problems through a graph theoretic approach [31], [10], [22], [67], [88]. Also, Lawler [61] and Papadimitriou and Steiglitz [74] elucidate algorithms for solving many of the network optimization problems in the fascinating setting of matroids and polytopes. Klee's comprehensive survey [54] is another source of references to many of the original papers on network optimization. In Tarjan's extremely readable treatment [89], the best known algorithms for solving the shortest path, the minimum spanning tree, the maximum flow and the maximum matching problem are presented and analyzed. Tarjan [89] pays close attention to data structures and their role in the performance of the various algorithms. In addition, he introduces a new and promising way of analyzing algorithms based on the principle of *amortization*, and applies this idea repeatedly in several different contexts.

In addition to network optimization, combinatorial optimization also includes two other important areas namely *scheduling* and *discrete location*; areas which are also significant from a practical standpoint. The first of these two areas namely scheduling has attracted a lot of attention from researchers motivated by practical as well as theoretical interests. In fact, this field has been the focus of a lot of activity over the past several years as reflected in Johnson's comments from [49], which we now quote:

> Scheduling theory has turned out to be one of the most fertile territories for exploring the boundary between the NP-hard and the polynomial time solvable and the outpouring of results has continued unabated... .

Starting with Conway, Maxwell and Miller's book on the subject of scheduling [13], there have been a number of excellent books and papers which describe and survey algorithms and related complexity results for a variety of scheduling problems [5], [14], [24], [38], [49], [65]. Coming to the discrete loca-

tion problem, Krarup and Pruzan [55] provide a good introduction to many of the results in this area.

However, efficient algorithms are not always known for combinatorial optimization problems. In fact several important problems such as the travelling salesman problem and the general scheduling problem have resisted repeated attacks by researchers, in the sense that we do not know of any polynomial time algorithms for solving these problems. An obvious question that arises in this regard asks as to whether this failure reflects an intrinsic difficulty in these problems which precludes the possibility of their being polynomially solvable or alternately, whether it merely reflects our inability in finding efficient algorithms for solving these problems.

Questions such as these have been raised in the past in more general contexts and have also been successfully resolved. For instance, following Turing's pioneering work [91] it is now well known that there are a number of problems for which it can be shown conclusively that there is no algorithm or effective procedure - polynomial time or otherwise - for resolving them; these constitute the well known class of *undecidable* problems [3], [15], [16], [79]. Clearly, if a problem is shown to be undecidable, we can conclude that it is not possible to solve it computationally or equivalently, that it is computationally intractable.

At a more reasonable level, even problems which can be resolved computationally - which constitute the class of *decidable* problems - can prove to be computationally intractable. For example, if the running time (measured in terms of the number of steps) of an algorithm grows as an exponential function of the size of the input (say 3^n steps for an input of n bits), then even for

reasonably small inputs (let us suppose that n takes on a value of 53), the algorithm will run for extraordinarily large amounts of time (for 2×10^5 centuries even on a computer with an instruction execution time of 1 nanosecond). In this case, the problem with which we are faced is that if the number of steps needed to resolve a given problem grows rapidly enough, then even for reasonably small input sizes, this number (of steps) can be so large that even though it is finite, we cannot ever hope to run the corresponding algorithm to completion.

In contrast to these extremely computationally intensive algorithms, those algorithms whose running time and space requirements are bound above by a polynomial function of the input length are extremely desirable since these functions grow much more slowly and as such, we can hope to use the corresponding algorithms even for reasonably large sized inputs. This is in fact the motivation behind equating the polynomial time solvability of a problem with its *tractability*; a convention which we will also adopt in the sequel. Early examples of decidable but provably *intractable* problems (in the sense of having at least exponential lower bounds on their time and space complexity) can be found in [25], [64], and [69].

However, the types of problems with which we are faced, in combinatorial optimization and other areas as well, seem to be in a sense 'easier' than these decidable but intractable problems. As evidence, we cite the fact that the 'decidable but provably intractable' problems which we discussed in the previous paragraph remain intractable even if we attempt to solve them on a *nondeterministic turing machine* (or computer); the converse is true of a number of problems from combinatorial optimization or at least of their *decision* versions in that they (these decision problems) can be resolved non-

deterministically in polynomial time. Therefore, all the problems which have been shown in the past to be computationally intractable are either decidable and nondeterministically intractable, or are undecidable problems. In contrast, several apparently intractable problems from combinatorial optimization as well as from other areas, are both decidable and can be resolved in polynomial time on a nondeterministic turing machine. Thus, the techniques that have been developed thus far for establishing the intractability of problems such as those in [25] for example, are not powerful enough to establish strong lower bounds on the worst case time complexity of these problems which can be solved in polynomial time nondeterministically, but which nevertheless seem to be deterministically intractable; further evidence towards the inadequacy of known proof techniques in establishing intractability results can be found in [12], [45], [46], and [83].

While efforts to develop stronger techniques for proving intractability results are ongoing, researchers have been trying in parallel to study interrelationships between the intrinsic difficulty of various combinatorial optimization, as well as other problems. These interrelationships not only help us in understanding the way in which various problems are related in their difficulty but can also provide the algorithm designer with valuable information which can be used towards the more positive goal of solving these problems as well. A standard technique used to study such interrelationships between any two problems has involved *reducing* one of the problems to another through a constructively specified transformation which maps instances of one problem into corresponding instances of the other. The basic principle behind this idea of reducing one problem to another, which has its roots in recursion theory, has been recognized and applied in the past in the context of combinatorial optimization problems. For example, Dantzig [16] showed that a number of

combinatorial optimization problems can be cast as integer linear programming problems with zero-one solutions. These reductions were however undertaken with a more positive goal in mind since they prove to be useful especially when the structure of the second problem (the one in the range of the reduction) is better understood, as a consequence of which good (but not necessarily polynomial time) algorithms are known for solving it.

Let us digress briefly from our discussion of computational intractability and consider an example situation where these reductions have played such a positive role by helping in understanding and solving an optimization problem. In their work, Grotschel and Padberg [41, 42] and others have studied the polytopes associated with certain linear programming formulations of the travelling salesman problem. Since linear and integer linear programs have been studied extensively, many interesting and powerful techniques are known for solving them. As a consequence, there is reason to hope (which in fact proved to be the case) that these linear programming versions of the travelling salesman problem will prove to be amenable to efficient solution methods. In addition, such reductions yield interesting characterizations of the travelling salesman problem through polytopes, which sheds considerable light on its intrinsic structure [6, 41, 42].

A problem with the above mentioned approach to solving the travelling salesman problem is that solving its linear programming version does not always yield a solution to the underlying travelling salesman problem. This is because a solution to these linear programs - in fact, a 'basic feasible solution' or equivalently, a 'vertex' of the corresponding polytope - does not always correspond to a 'valid tour' in its equivalent travelling salesman problem. However, this kind of a problem is not uncommon and in fact, a variant is en-

countered in the context of solving integer linear programs by first reducing them to equivalent linear programs through 'relaxation', and then by solving the resulting linear programming problems; the *relaxation* of an integer linear program is the corresponding linear program derived by dropping the integer constraints on the solutions. Gomory [40] solved this problem through the well known *cutting plane* method which involves computing additional inequalities or constraints which represent the cutting planes. These additional constraints eliminate non-integer solutions of the (relaxed) linear programs while preserving the feasible integer solutions [74].

This basic principle has been used to devise techniques for computing such *cuts* or *valid inequalities* in several situations for the travelling salesman problem [6, 71]. Then, solving the linear programming formulations of the travelling salesman problem in conjunction with these cuts, is guaranteed to yield a correct and optimum solution to the latter problem. This shows us that reductions have been used effectively towards the positive goal of solving an optimization problem (the travelling salesman problem in our case) by reducing it to another problem (linear programming), especially when good techniques are know for solving the latter problem.

Let us now return to the issue of studying the interrelationship between the intrinsic difficulty or intractability of various problems through reductions. It was Cook who laid the foundation towards understanding these relationships through the theory of *NP-completeness* in his classic paper [11]. In this paper, he delineated the class NP of decision problems which can be resolved in polynomial time on a nondeterministic turing machine. But more importantly, through the notion of *polynomial time reducibilities*, he showed that the *satisfiability* problem is in a sense the *hardest* problem in NP. He

also suggested that this property of being the hardest problem in NP is shared by other problems (in NP) which have since been collectively referred to as the class of NP-complete problems. As a consequence of his results, we now know that if any NP-complete problem is intractable, then they all are; in the same vein, if a single NP-complete problem can be solved in polynomial time, then so can any problem from NP. Cook's original ideas have proved to be extremely powerful since they allow us to unify several individual questions regarding the complexity of a number of problems into a single question concerning the intractability of the class of NP-complete problems.

Subsequently, Karp showed in his influential paper [50] that the decision problem versions of many of the combinatorial optimization problems including the travelling salesman problem, are all NP-complete. Since then, the number of NP-completeness results has grown rapidly, especially since the publication of Garey and Johnson's remarkable exposition of this theory [35]. Even though the question as to whether the NP-completeness of a problem implies its inherent computational intractability remains one of the deepest and most important open questions, the NP-completeness or the NP-hardness of a problem (a problem is NP-hard if an NP-complete problem can be reduced or transformed to it in polynomial time; it need not necessarily be in NP), and its being computationally intractable, are widely accepted as being synonymous and as such, we will also refer to NP-complete or NP-hard problems as being computationally intractable in what follows.

1.2 The Precedence Constrained Scheduling Problem

So far, we have been looking at various types of well known combinatorial optimization problems. We have also looked at the developments in complexity theory and its implications to the question of whether or not we can solve these optimization problems efficiently. Part of this discussion involved formalizing the intuitive notion of the efficiency of an algorithm by equating it with polynomial time bounds on its running time. We will now outline several known complexity results from the fertile area of precedence constrained scheduling. To do this, we have to first introduce this scheduling problem formally. Having outlined these complexity results from scheduling theory, we will then be able to use them in the next section (Section 1.3) to establish the main motivating them of this dissertation and highlight our approach towards tackling the many problems which arise out of this theme.

An instance of the precedence constrained scheduling problem has a *precedence graph* which is a directed acyclic graph $P = \langle A, \alpha \rangle$. Each *element* a_i in the vertex set A for $1 \leq i \leq n$ is interpreted as a (computational) *task*. The edge set α represents interdependence of tasks; if edge $\langle a_i, a_j \rangle$ is in α, then task a_i has to be completed before task a_j can be started. In this manner, a precedence graph P represents an entire task system with several interrelated tasks.

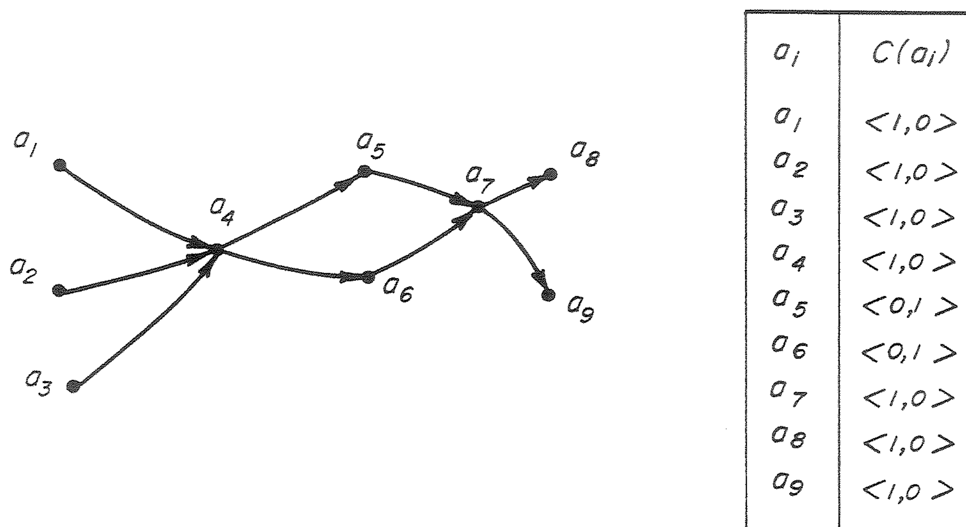
This task system is to be executed on a multiprocessor system with many processors and resources. In particular, there can be many distinguishable types of specialized processors with different characteristics. For purposes of our present discussion, let us consider processors as just being another (special) type of resource. Then, a multiprocessor system has a number of different *types* of *resources*, say m types. Some of these types can represent dif-

ferent types of processors and the remaining types can be non-processor type resources such as memory, and so on. We denote the number of units of the vth type resource in the system by $B_v > 0$ for $1 \leq v \leq m$. Therefore, the bound on the total resource availability in the system is specified through the *bound vector* $\mathbf{B} = \{B_1, ..., B_m\}$.

We also have a way of specifying the resource requirements of each element or task through the *class function* $\mathbf{C}$, which maps the task set A into the m-tuples of non-negative reals. So, for any task a_i, $\mathbf{C}(a_i) = \langle C_{i1}, ..., C_{im} \rangle$ represents its resource requirements where C_{iv} is the amount of the vth resource that is needed by task a_i during its execution. We also have a task *length* $\mathbf{l}_i > 0$ associated with each task a_i. This length $\mathbf{l}_i$ represents the duration for which task a_i will use the resources needed by it, from its start to its completion.

The precedence graph and class function of an example instance are both indicated in Figure 1.1. There are nine elements (or tasks) in this example task system, and the edge set represents the precedence relationships between the elements. There are two types of resources in the multiprocessor system on which this task system is to be scheduled equivalently, m is two. There is exactly one unit of each of these (two types) of resources and therefore, the bound vector $\mathbf{B} = \langle 1 , 1 \rangle$ for this instance; also, all the task lengths are unity.

Given such an instance, we are interested in *schedules*, where a schedule is a specification of a *start time* s_i and a *finish time* f_i for each task or element a_i where $f_i > s_i \geq 0$; the task a_i is to be started at time s_i and is to be completed by time f_i. We are only interested in non-preemptive



a_i	$C(a_i)$
a_1	<1,0>
a_2	<1,0>
a_3	<1,0>
a_4	<1,0>
a_5	<0,1>
a_6	<0,1>
a_7	<1,0>
a_8	<1,0>
a_9	<1,0>

Figure 1-1: The precedence graph and class function of an example instance.

schedules in which tasks are processed *continuously* from start to finish, that is they are not interrupted for any non-zero interval of time once they are started. In such schedules, any element or task a_i is *active* in the (half-open) interval of time $[s_i, f_i)$ and also, $\mathbf{l}_i = (f_i - s_i)$.

We also want to make sure that our schedules respect the *precedence* and *resource constraints*. The precedence constraints represented by the edge set α of the precedence graph P, require that in any schedule, if $\langle a_i, a_j \rangle \in \alpha$ for any two tasks a_i and a_j in A, then element or task a_i must finish before element a_j can start or equivalently, $s_j \geq f_i$. The resource constraints require that the total number of resources of the various types which are needed by the tasks that are being processed at any given instant of time in the schedule, do not exceed the total (available) amount of resources as specified in the bound vector $\mathbf{B}$. To formulate this constraint more precisely, let $\mathbf{t}$ represent

the set of all elements which are *active* (being processed) at some instant of time t in a given schedule. Then the resource constraints require that for all $t > 0$ and each v where $1 \leq v \leq m$,

$$\sum_{a_i \in \mathbf{t}} C_{iv} \leq B_v.$$

Schedules which respect the precedence and resource constraints will be referred to as *valid* schedules and in what follows, when we refer to a schedule, we actually mean a valid schedule.

In particular, we are interested in optimum schedules which are schedules with minimum *makespan*, where the makespan of a schedule is the maximum over all the finishing times, of all the tasks or elements in it. Intuitively speaking, optimum schedules are those in which all the tasks are completed as quickly as possible. A schedule for the example instance of Figure 1-1 is shown in Figure 1-2. In this schedule, element a_1 is active during the interval of time $[0,1)$, element a_9 during the interval $[8,9)$ and so on. Also, its makespan is nine and it is easy to see that it is optimum for the given instance.

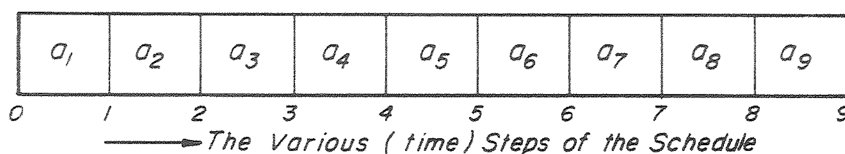


Figure 1-2: A schedule for the instance of Figure 1-1 where each vertical line indicates the *start* or the *finish* time of a task.

A special class of problem instances which have attracted considerable interest in the past in the context of finding optimum schedules is

one in which the multiprocessor system being modelled has a *single* processor-type resource. Thus, in our terminology, this class would correspond to the restricted case where m is always unity, and for any instance from this class, we have B_1 units of these processors in the system. Also, since these resources represent identical processors, it is assumed in each of these cases that any element or task a_i needs exactly one of these B_1 processors for $\mathbf{l}_i$ units of time from start to finish.

It was shown by Ullman in [92] that even for this special class of instances, the problem of finding schedules with minimum makespan is NP-hard if we were to allow instances with:

1. arbitrary precedence graphs, two processors (B_1=2), and the task lengths $\mathbf{l}_i$ are either 1 or 2 or,

2. arbitrary precedence graphs, variable number of processors (B_1 is an integer greater than zero) and all the task lengths equal unity.

Motivated by result (1), further restrictions were placed on the problem instances being considered, with the aim of finding polynomially solvable classes of instances or subdomains. In this connection, a commonly placed restriction requires that all the task lengths be equal; it is easy to see that there is no loss of generality in extending this restriction to require that all the task lengths equal unity. Problem instances which satisfy this constraint are often referred to as *equal execution time* systems. In what follows, we will be considering only such task systems and will use $\mathbf{l}$ to denote this (common) task length attribute. Even for this rather specialized class of equal execution time systems, we know from result (2) above that finding optimum schedules remains an NP-hard problem if multiprocessors with an arbitrary number of processors are considered or equivalently, if instances in which B_1 is allowed to assume arbitrary positive integer values are admitted.

However, with additional restrictions on the allowed instances, several polynomially solvable subproblems of the scheduling problem have been discovered starting with Hu's classical result [48], as shown in Table 1-1. Note that all the results in Table 1-1 deal with subdomains in which the instances have a single (identical) processor-type resource (m=1), and each task requires exactly one unit of this resource ($\mathbf{C}(a_i = \langle 1 \rangle$ for any element a_i in an instance from this subdomain). The label β_i in this (and other) table(s) will be used subsequently to refer back to the result listed in the corresponding row. Also, in each case, 'complexity' refers to the bound on the running time of the algorithm including the best known techniques for the various preprocessing steps, as reported in the original paper cited in the column entitled 'appeared in'. The entries in the remaining columns reflect the constraints on the instances in the corresponding subdomains.

Domain	Appeared in	Number of Processors	Precedence Graphs	Complexity (Time)
β_1	[48]	≥ 1	in-trees (or out-trees)	$O(n)$
β_2	[23]	two	arbitrary	$O(min(\lvert\alpha\rvert \cdot n, n^{2.61}) + n^{2.5})$
	[9]			$O(min(\lvert\alpha\rvert \cdot n, n^{2.61}) + \lvert\alpha\rvert + n \cdot alpha(n))$
	[29]			$O(\lvert\alpha\rvert + n.alpha(n))$
β_3	[75]	≥ 1	interval orders	$O(n + \lvert\alpha\rvert)$

Table 1-1: Polynomially solvable subdomains of the *makespan* minimization version of the scheduling problem.

Subdomain β_2 in this table requires special mention since it has been the focus of attention of three different research efforts, starting with that of Fujii et al. [23]. Since then, Coffman and Graham [9] and more recently Gabow [29] have designed algorithms with improved efficiency as indicated in Table 1-1. The Coffman-Graham algorithm is based on a lexicographic numbering scheme which works on a transitively closed or reduced graph. Sethi [82] has shown that this lexicographic numbering can be done in time $O(|\alpha| + n \cdot alpha(n))$ where *alpha* is the very slow-growing inverse of the Ackerman's function [90]. Finally, the $alpha(n)$ factor in the running time of Gabow's algorithm has since been shown to be unnecessary, following a result from [32].

Subsequently, Goyal [39] considered the closely related problem of multiprocessor systems with many different types of specialized (processor-type) resources or in other words, m is allowed to be greater than one for these instances. However, multiprocessors with only *one unit* of each of the m types of resources were considered, and since these resources were processors, it is once again assumed that each task requires exactly *one* unit of *one* of these resource types (or for each a_i, there exists a unique v' such that $C_{iv'}$ equals unity and C_{iv} equals zero whenever $v \neq v'$). For instances which satisfy these restrictions, Goyal was able to provide a linear time algorithm for finding schedules with minimum makespan, if in addition, the precedence graphs of these instances are *cyclic forests*; let β_4 denote this subdomain of instances.

1.2.1 Schedules with Minimum Tardiness

Another variant of the scheduling problem has also been studied which involves determining schedules for precedence constrained tasks, which are once again specified as precedence graphs. However, in addition to the class function and length attributes which were associated with the tasks in the previous case, each task a_i also has a *deadline* d_i associated with it. Once again, the resource availability or the resource constraints are specified through the bound vector **B**. However, in this case, we would like to finish each task not only before the tasks which it precedes, but also as close as possible to the explicitly specified deadline which is associated with it.

To make this notion formal, let us define the *tardiness* of a task a_i with finishing time f_i in a given schedule, to be the minimum of $(f_i - d_i)$ and zero. Then, the *tardiness* of the schedule is the maximum tardiness of all the tasks in it, and we are seeking schedules with minimum tardiness. Intuitively, the tardiness of a task in some schedule is a quantitative measure of the extent to which it has been delayed beyond its deadline before it is completed. These deadlines can be used to capture the typical constraints on the tasks from a *real-time* environment, such as those discussed by Leinbaugh in [62] for example.

Even in this case, the problem of finding schedules with minimum tardiness is NP-hard. This follows from the results of Brucker, Garey and Johnson [2], Ullman [92] and others [35, 49, 65]. Once again, polynomially solvable subproblems have been discovered for unit execution time task systems as shown in Table 2. As in the case of the makespan minimization version, multiprocessor systems with a single 'processor-type' resource are considered in each of these cases. Also, since all these processors are considered to

be identical, any task can be executed on any one of the processors. It is therefore the case that any task a_i requires exactly one unit of this 'processor-type' resource or in other words, $\mathbf{C}(a_i) = \langle 1 \rangle$ for any element a_i.

Domain	Appeared in	Number of Processors	Allowed Precedence Graphs	Complexity (Time)
β_5	[2]	≥ 1	in-trees	$O(n \log n)$
	[68]			$O(n)$
β_6	[33]	two	arbitrary	$O(n^2 \log n)$

Table 1-2: Polynomially solvable subdomains of the *tardiness* minimization version of the scheduling problem.

1.2.2 Scheduling Pipelined Multiprocessors

With the increasing popularity and commercial viability of *pipelined computers* [53, 77], various researchers have studied the associated scheduling problem in *scalar* [4, 78] and *vector* [81, 84] environments of computing. To understand this version of the scheduling problem, let us consider the pipelining concept informally in the light of the scheduling model which we have been considering thus far. In Figure 1-3(i), we have a collection of n tasks each of which needs the (single) processor indicated alongside for 1 units of time for it to be completed. Each of the computation tasks a_1 to a_n can be viewed as a specific representation of some computable function.

Now, consider replacing each of the computational tasks in Figure 1-3(i) by an equivalent 'composition' of four (in general k) tasks as shown in Figure 1-3(ii). In this manner, task a_1 gets *sliced* into the *chain* of four tasks $a_{1,1}$, $a_{1,2}$, $a_{1,3}$, and $a_{1,4}$. Also, suppose that this slicing is such that the composition (in the usual sense of a functional composition) of the four computable functions represented by the tasks $a_{1,1}$ to $a_{1,4}$ is equivalent - once

again in the usual sense of functional equivalence - to the original computable function represented by task a_1. In the same manner, we also replace the processor by a four *stage* (in general k-stage) *pipeline*. The replaced or sliced version in Figure 1-3(ii) is referred to as the *pipelined equivalent* of the original problem, which includes the task system and the processor shown in Figure 1-3(i).

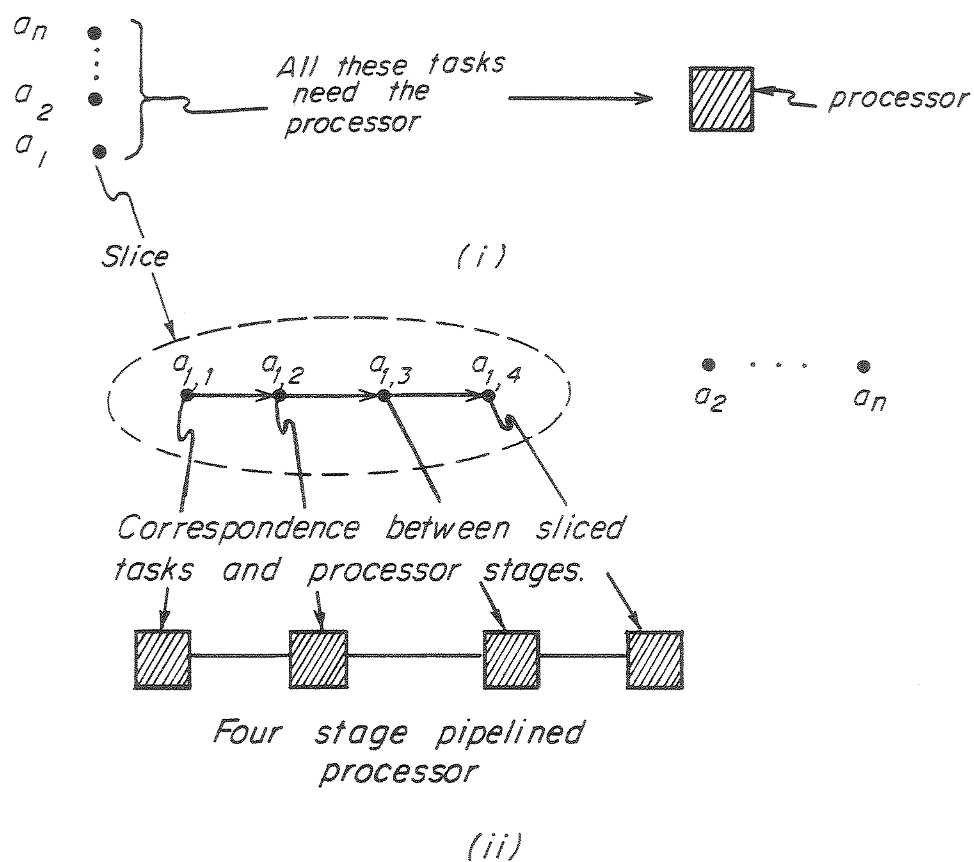


Figure 1-3: The process of *slicing* : The instance in (i) is replaced by its *pipelined* equivalent in (ii) by slicing the tasks and resources into four.

We will now describe the way in which each of the chains of tasks in the pipelined equivalent is to be executed on the corresponding four stage processor. We start off by executing some eligible task, say $a_{1,1}$ at the first stage of the pipelined processor. As soon as it is completed, we are ready to start task $a_{1,2}$, this time at the second stage of the four stage processor. We continue in this fashion where in general, we execute the γth task from the chain at the γth stage of the pipelined processor at a point in time after the $(\gamma - 1)$th task in the chain has been completed at the $(\gamma - 1)$th stage of the processor.

What is most interesting about the pipelined equivalent (shown in Figure 1-3(ii)) of the original problem is that once task $a_{1,1}$ is completed at the first stage, this stage of the processor becomes free and therefore, a different task, say $a_{2,1}$ can now be started concurrently with task $a_{1,2}$. In this manner, we can have up to four (in general k) different tasks being processed simultaneously in the four stage pipelined processor. This idea can be generalized in an obvious way when we have a system with many different types of k-stage pipelined processor and non-processor type resources in the system. Then, a task essentially starts at the first stage of all the resources it needs, including processors. As it completes getting processed at a certain stage, it moves on to the next stage of all these resources simultaneously and so on.

The motivation behind slicing up the tasks and resources in the above described manner follows from the following reasons. Firstly, it is quite often the case that each individual task in the pipelined equivalent - say task $a_{i,j}$ in the chain of tasks corresponding to the original task a_i - is much simpler than the original task a_i. As a consequence, the length of task $a_{i,j}$ is much smaller than that of the original task. Actually, if we assume that over-

heads are negligible, the length of the original task a_i will be very close in value to the sum of the lengths of all the tasks in the corresponding chain in its pipelined equivalent. It turns our that it is reasonable to make the above mentioned assumption in a number of realistic and practical environments. Under these circumstances, the length of the original task a_1 in our example would (be almost) equal (to) the sum of the lengths of tasks $a_{1,1}$ to $a_{1,4}$.

For purposes of our present example, let us also suppose that the lengths of all the tasks in the chains of the pipelined equivalent are all equal or equivalently, the processing delays are the same at all stages of the pipelined resources. Then, based on our explanation from the previous paragraph, the length of each task in the chains of the pipelined equivalent in Figure 1-3(ii) equals $\mathbf{l}/4$, where $\mathbf{l}$ denotes the length of a task from the original instance from Figure 1-3(i). Also, note that the makespan of any valid schedule for the n tasks in Figure 1-3(i) with single stage processors and resources is at least $n \cdot \mathbf{l}$. In contrast, it is easy to see that by pipelining, we can get schedules with lower makespans; for our example of Figure 1-3(ii), the makespan can be as low as $\{ ((n-1) \cdot \mathbf{l}/4) + \mathbf{l} \}$ (or more generally, $\{ ((n-1) \cdot \mathbf{l}/k) + \mathbf{l} \}$ when k-stage pipelines are used). It is also easy to show similar benefits, even when the task lengths in the chains of the pipelined equivalent are not all equal and in addition, when we have task deadlines and our goal is to find schedules with minimum tardiness. It is this performance benefit which pipelined systems have to offer with relatively negligible accompanying control overhead which makes them practical, and which is responsible for their widespread use.

Let us now extend our scheduling model to account for pipelined multiprocessor systems as well. To do this, we simply need to add an integer

parameter $k \geq 1$ which we will refer to as the *degree of slicing*. Clearly, k represents the number of stages into which the tasks and resources in the original problem are sliced. Viewed in the light of this generalization, it is easy to see that the results listed in Tables 1-1 and 1-2 correspond to the special case of the scheduling problem in which the degree of slicing k equals unity. The various other components of an instance which were identified and defined earlier on in the context of non-pipelined systems including the precedence graph, the class function, the task length and the bound vector attributes, are all defined in exactly the same manner in the case of pipelined systems as well. In effect, to specify instances of the scheduling problem with pipelined multiprocessors and resources, we simply take the way in which we specified these instances in the case of non-pipelined systems in the earlier subsections and add an extra parameter k to this specification to indicate the number of stages into which the tasks and the resources are sliced.

Also, we continue to be interested only in equal execution time systems, and in determining non-preemptive schedules. As a consequence of the latter fact, schedules are still completely determined by specifying the start and finish times $[s_i, f_i)$ for each task a_i in the system. However, in this interval, a task will be 'switching' from one stage of the resources it is executing on, to the next. If this process of switching is to be explained in terms of slicing, we will have to say that for any task a_i, the various tasks $a_{i,j}$ for $1 \leq j \leq k$ in the corresponding chain in the pipelined equivalent are being successively processed at various points in the interval of time $[s_i, f_i)$.

We now need to ensure that the schedules for these pipelined systems also obey the precedence and resource constraints. The precedence constraints are still enforced by requiring that $f_i \leq s_j$ whenever $\langle a_i, a_j \rangle \in \alpha$. However,

since tasks contend for resources only when they are active at the same stage of the pipelined resources, we have to modify our earlier formulation of resource constraints to account for pipelining. In other words, we only need to make sure that the resource requirements of all the tasks in the schedule which are being processed at the same stage and at a given instant of time, do not exceed the total resource availability. If this has to be done, we need to first formalize the notion of a task switching from one stage to the next, while it is being processed on the pipelined resources.

To do this, let us for the moment concern ourselves with instances in which the processing delay is the same at all stages of the pipelined resources. This fact in conjunction with our present focus on equal time systems implies that the lengths of all the tasks in the pipelined equivalent of any instance that is of interest to us, must all be equal (to $1/k$); let l denote this length of a task from the pipelined equivalent. In the rest of this dissertation, we are going to concern ourselves with instances which satisfy the above constraint and will refer to them as *uniformly k-sliced* systems. Then, a given task a_i - actually task $a_{i,1}$ in its pipelined equivalent - is being processed at the first stage in the interval of time $[s_i, s_i + l)$. During the second time interval $[s_i + l, s_i + 2 \cdot l)$ when task a_i (or task $a_{i,2}$ in the pipelined equivalent) is being processed at the second stage of its resources, the corresponding first stages of all these resources are free to be assigned to a different task. Thus, for a given task a_i, we can associate an active period at each stage as it is being processed - formally, we say that task a_i is *active* in the given schedule at the γth *stage* for positive integer $\gamma \leq k$, in the interval of time $[s_i + (\gamma - 1) \cdot l, s_i + \gamma \cdot l)$; in 'slicing' terminology, task $a_{i,\gamma}$ from the corresponding chain of tasks in the pipelined equivalent is being processed during this time interval. Another way of stating this idea is to say that task a_i is

active in a schedule at the γth *stage* at time t, whenever $t \in [s_i + (\gamma - 1) \cdot l, s_i + \gamma \cdot l)$.

We are now in a position to state the resource constraints precisely. To this end, let $\mathbf{t}_\gamma$ be the set of all tasks which are active in the given schedule at the γth stage, at time instant t. Then the resource constraints are such that for each γ and v where $1 \leq \gamma \leq k$ and $1 \leq v \leq m$,

$$\sum_{i\,:\,a_i \in \mathbf{t}_\gamma} C_{iv} \leq B_v.$$

Once again, we say that a schedule is valid if it satisfies the precedence and resource constraints and we continue to be interested in schedules with minimum makespan and, when local deadlines are associated with the tasks, in schedules with minimum tardiness; both makespan and tardiness are defined in exactly the same way as before. Let us now consider the example schedule in Figure 1-4 for the example instance whose precedence graph and class function were shown in Figure 1-1. In this instance, we have a single two stage ($k = 2$) processor (instead of two single-stage processors in the original example). Also, since we are concerned only with uniformly k-sliced systems, the processing delay is assumed to be equal at both stages of this pipelined processor and since all the task lengths equal unity, we have $l = 1/2$. Note that through pipelining, we have a schedule with improved in makespan in Figure 1-4, when compared to its counterpart in Figure 1-2 which corresponds to an instance with single stage processors.

Bruno, Jones and So [4] have studied the problem of scheduling parallel tasks systems on pipelined processors. As is the case with the earlier results which we outlined in Tables 1-1 and 1-2, they consider multiprocessor

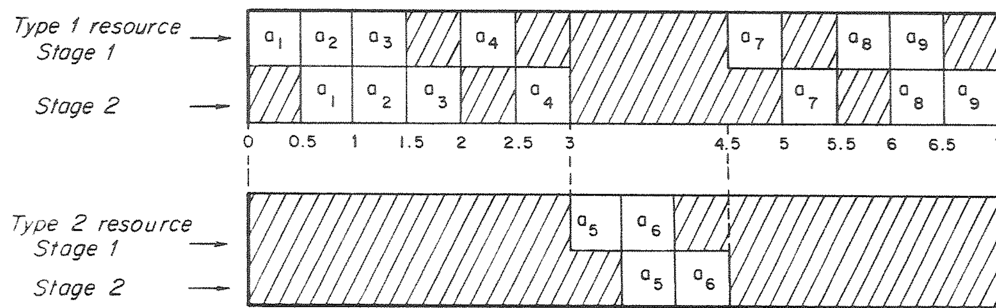


Figure 1-4: A schedule for the instance of Figure 1-1 with two stage pipelined processors.

systems with a *single* type of identical processor-type resources ($m = 1$ in our formulation). Once again, since these resources are identical processors, Bruno et al. expect the tasks to require exactly one of these processors ($\mathbf{C}(a_i) = 1$ for all tasks a_i). They also restricted their attention to equal execution time systems and in particular, those in which the task length $\mathbf{l}$ equals the degree of slicing k, as a consequence of which l equals unity for any instance which was considered in [4]. These results are listed in Table 1-3 and as we have indicated in the second column of this table, in some cases, they constitute a proper generalization of the corresponding results for multiprocessors with non-pipelined or single-stage processors (those with $k = 1$) which we listed earlier in Tables 1-1 and 1-2.

1.3 Structure in Combinatorial Optimization

As we have seen Section 1.1, combinatorial optimization problems can vary in their complexity from being polynomially solvable to being NP-hard. Nevertheless, one thing that is common to a number of these problems, be they tractable or intractable, is that many of them are well understood in the sense that a lot of insight has been gained into their inherent structure. The problem of finding matchings in graphs is a notable example of such a

Domain	Generalization of	Number of Processors	Precedence Graphs	K	Objective Function
β_7	β_1	≥ 1	in-trees (or out-trees)	≥ 1	makespan
β_8	*	one	arbitrary	2	makespan
β_9	β_5	≥ 1	in-trees	≥ 1	tardiness
β_{10}	*	one	arbitrary	2	tardiness

Table 1-3: Polynomially solvable subdomains with *pipelined* processors from [4].

problem [21]. A lot is understood about this problem and the incisive algorithmic results due to Micali and Vazirani [70] and more recently, due to Karp, Upfal and Wigderson [59] bear further testimony to this fact. Papadimitriou and Steiglitz discuss this problem and its weighted version in detail and the reader is referred to their book [74] if a comprehensive treatment is desired. Efficient algorithms such as the Micali and Vazirani [70] or the Karp, Upfal and Wigderson [59] results give us significant insight into the structure of this problem, albeit indirectly; by studying these algorithms, we can learn a lot about the nature of the information that is really essential towards solving this problem.

There are other well understood optimization problems as well. For example, the capabilities of the greedy algorithm and in particular, its ability to solve problems such as the minimum spanning tree (or the equivalent maximum weighted forest) problem are captured remarkable well through the matroidal formulations of Edmonds [20]. Another instance of a well understood issue in combinatorial optimization is embodied in the classical duality theorem for linear programming [30, 37, 94]. This theorem and other duality

results for combinatorial extremum problems [17, 28, 80] help elucidate the behavior of the solutions of these optimization problems - in particular, at optimality. Therefore, insight in this case takes the shape of good characterizations of the behavior of optimum solutions, of the optimization problem in question, through such duality results.

Even NP-hard optimization problems have good characterizations at times. In this context, we have already seen some of the results for the well known travelling salesman problem through polytopes and linear programming formulations; Burkard [6] surveys these results. We will return in Section 2.1 to discuss some of the above mentioned results related to matroids and polytopes in greater detail.

However, what is extremely surprising is that despite the large number of complexity results that are known for the scheduling problem and its obvious importance, it is not well understood in any of the above senses. In fact, so many individual complexity results are known for the scheduling problem in terms of its subdomains being classified into polynomially solvable or NP-hard categories, that a computer is used to maintain these results and their interrelationships [66]. Despite this wealth of information, it is not really clear as to what factors of the scheduling problem are really contributing to its complexity and this is especially true if one considers the scheduling problems with non-trivial precedence constraints.

For example, Brucker, Garey and Johnson [2] have shown that the tardiness minimization problem is polynomially solvable for in-trees (row one of Table 1-2). What is intriguing is that this problem becomes NP-hard even if everything about this subdomain β_5 is kept the same except that the

precedence graphs are now allowed to be out-trees [2]. To cite another instance, it has been shown in [36] that the precedence constrained scheduling problem is NP-hard if the polynomially solvable subdomain discovered by Hu [48] (row one of Table 1-1) is generalized to include precedence constraints which are *opposing forests*; opposing forests are the disjoint unions of in- and out-forests. This does not however help us understand as to why the minimization problem (both the makespan and tardiness versions) remains polynomially solvable even if we have arbitrary precedence graphs with two processors (respectively subdomain β_2 in row 2 of Table 1-1 and subdomain β_6 in row 2 of Table 1-2); in general, we do not have an explicit and fundamental understanding of the structure and properties of subdomains such as these, which are ensuring their polynomial solvability, and which are being lost, thereby rendering the tardiness minimization problem for out-trees [2] or the makespan minimization problem for opposing forests [36] NP-hard. This situation deteriorates further if in addition, we start considering the extra dimension spanned by pipelining.

Thus, the problem on hand is two fold in that firstly, we do not seem to have good unifying characterizations or theories through which we can explain the polynomially solvable, nature of the many apparently unrelated subdomains listed in the various tables earlier on. Secondly, as we described in the previous paragraph, we do not always have an explicit understanding of the structure which is being lost in going from a polynomially solvable to an NP-hard subdomain of the scheduling problem. Therefore, our interest in unifying complexity characterizations and theories stems from the fact that such characterizations will provide us with a unified basis or point of reference for elucidating the intrinsic nature of these polynomially solvable subdomains. Another equally important benefit of having such unifying complexity charac-

terizations is that our understanding of the boundary between easy and hard problems will also be greatly improved as a consequence; this follows since any collection of conditions which are shared by, and determine the complexity of the various polynomially solvable subdomains of a problem must be such that there will be instances from the corresponding NP-hard subdomains (in fact, there can be no finite upper bound on the size of such instances) which explicitly violate (some of) these conditions, unless P = NP of course.

In this dissertation, we remedy this situation by working towards synthesizing such unifying conditions or characterizations of the polynomially solvable subdomains of the precedence constrained scheduling problem. In Chapter 2, we abstract out the essential aspects of a schedule in terms of sequences of edges of a hypergraph. The elements or 'vertices' of the hypergraph are the tasks. Given that we are going to be dealing with uniformly k-sliced equal time systems, each edge in the sequence - say the qth - represents the collection of all the tasks with identical start and finish times (respectively $l \cdot (q-1)$ and $l \cdot q$). Then, using this hypergraph theoretic formulation of schedules, we characterize the 'output or solution domain' of a given instance of the scheduling problem in terms of sets of such sequences.

This formulation of schedules in terms of hypergraph edges greatly simplifies our job of studying their behavior. In doing this (studying the behavior of sets of sequences that is), we are motivated by the following line of reasoning. To start with, the properties which are used to delineate tractable subdomains of an optimization problem are typically drawn from the input or problem domain. In-trees, cyclic forests and other such constraints all testify to this fact in the case of the scheduling problem. However, if we reconsider these constraints or properties from Tables 1-1, 1-2 and 1-3, they all seem to

lend support to the thesis that these subdomains are perhaps tractable for intrinsically different reasons. As a consequence, it seems extremely unlikely that we will have much success in synthesizing unifying complexity characterizations by perusing these properties and attributes from the input domain. This motivated us to take a different approach towards seeking unifying conditions: one which involves studying the output or solution domain of the scheduling problem instead.

As a first step in this direction, we try to understand the basic factors which distinguish optimum sequences from sub-optimum ones. This knowledge will prove to be of great value subsequently as we will see in the next couple of paragraphs. In Chapter 2, we also integrate both the makespan and tardiness minimization versions of the precedence constrained scheduling problem into a single optimization problem associated with hypergraphs; in subsequent chapters, we concern ourselves only with this single 'integrated' problem.

In Chapters 3 and 4, we develop a very strong characterization of minimum cost sequences through the rich class of affined sequences. Sequences from this class have the extremely interesting property that they are always optimum. We do this by taking a deep look at the behavior of sequences, and especially at the effect of the precedence relationships on their cost, which could be either the makespan or the tardiness, depending on the particular special case of the integrated problem which we might be considering. Also, we use a simple and yet quite powerful proof technique to establish this important property of affined sequences, and the fundamental principles on which this technique rests are elucidated in Chapter 4. In this chapter, we also establish some very interesting and counterintuitive properties of affined sequences

which play a central role in applying the above mentioned proof technique to show that affined sequences are always optimum.

In Chapter 5, we develop our unifying complexity characterization by building on the results of Chapters 3 and 4. This is done by formulating the *intrinsically ordered convergence* property or IO convergence, which is essentially a sufficient condition for polynomial solvability of our scheduling problem, in that if an instance of the problem satisfies it, then a corresponding minimum cost sequence (or schedule) can be enumerated in polynomial time. This property rests heavily on our earlier results including the important class of affined sequences. Another interesting and useful aspect of this unifying IO convergence property is that it has a constructive or algorithmic component, which automatically gives us an algorithm for solving this scheduling problem, hand in hand with the unifying framework.

Subsequently, we are in a position to show in Chapter 6 that surprisingly enough, instances drawn from any of the subdomains of the scheduling problem which were described earlier on, are all IO convergent, and as such the polynomially solvable nature of these subdomains essentially follows as a corollary to this single unifying fact. In Chapter 6, we will also see evidence that IO convergence has also guided us in discovering new and useful polynomially solvable subdomains of the scheduling problem.

In Chapter 7, we describe our algorithm for solving this optimization problem in greater detail. This algorithm basically runs in time $O(n^2 \cdot logn)$ and constructs optimum sequences or schedules for any IO convergent instance. In specifying this algorithm, we adopt the algorithmic notation and constructs used by Tarjan in [89] to a significant extent. Also, as in Tarjan's

development [89], we pay close and explicit attention to issues related to the data structure aspects of our algorithm. Our goal is to present the algorithm at a sufficiently general level to where it can be easily understood, and at the same time, we include enough detail such that specific implementations can be easily realized.

Chapter 2

On Hypergraphs and Scheduling

In this chapter, we formulate the notion of a schedule in terms of sequences of edges of the hypergraph H defined by the set of tasks or elements A. In Section 2.1, we argue for our formulation over alternate choices in terms of polytopes and matroids, and in Section 2.2, we will introduce and describe our hypergraph theoretic formulation of schedules. In Section 2.3, we show that the problem of finding a schedule with minimum makespan can be reduced to that of finding one with minimum tardiness. As a consequence, we can formulate both the makespan and tardiness minimization problems in terms of a single optimization problem associated with hypergraphs, with an appropriately defined cost function. In Section 2.4, we identify the significant factors influencing the cost of a sequence or schedule. This step involves developing the important notion of the modified cost of a sequence of edges of H. In Section 2.5, we characterize the space of solutions associated with a problem instance in terms of the feasible set $\mathbf{S}$ of sequences and establish a relationship between certain of its key subsets. This relationship leads us (in Chapter 3) to a characterization of subsets of $\mathbf{S}$, all of whose member sequences have zero cost. In Chapter 4, we will further extend this result to a characterization of minimum cost sequences.

2.1 A Choice of Characterizations

2.1.1 Matroids

Matroids have played a significant role in characterizing the behavior of combinatorial algorithms in the past and as such, we wish to consider them as a possible basis for formulating and studying the complexity of scheduling problems. Whitney [95] introduced matroids, although his interest was in characterizing the abstract properties of independence from linear algebra. It was Edmonds [20] who recognized the connection between these mathematical structures and the behavior of the *greedy algorithm*. A matroid essentially consists of a finite set E of *elements* and a collection $\mathcal{I}$ of its subsets. Subsets of E which are in $\mathcal{I}$ are called *independent subsets* of E. Then, the pair $\langle E, \mathcal{I} \rangle$ is a *matroid* if the family of subsets is such that:

1. every subset of an independent subset is independent and

2. all the maximal independent subsets in $\mathcal{I}$ are of the same size.

The maximal independent subsets are referred to as the *bases* of the matroid.

Given that numerical weights are associated with the elements of E, the optimization problem is to find an independent subset of maximum *total* weight, where the total weight of a subset of E is the sum of the weights of its constituent elements. Edmonds [20] showed that the greedy algorithm in which we start off with the empty subset of E and on each step, choose an element with maximum weight such that independence is maintained, correctly solves this optimization problem. Given that the collection $\mathcal{I}$ is closed under (subset) inclusion (follows from condition 1 above), the correctness of the greedy algorithm is essentially equivalent to condition 2. This matroidal formulation throws considerable light on the *minimum spanning tree* problem or

its closely related *maximum weighted forest* problem because, in applying the greedy algorithm to solve these problems, we are actually solving the optimization problem associated with the corresponding *graphic* matroid. The set E of the graphic matroid corresponding to an instance of the minimum spanning tree problem with graph $G = \langle V, E \rangle$ is its edge set, and the family I is the collection of all *forests* of G.

Since this early result, matroid optimization theory has become an important field of research in its own right [61, 74]. Recent results in this area include efficient algorithms for a generalization of the well known *two matroid intersection* problem where the elements of these matroids, in addition to having a weight, also have one of two colors associated with them. In this case, we are interested not only in finding a base of the matroid with maximum total weight, but also in one which has the specified number of elements of a given color; the latter constraint is specified through the second matroid and the desired solution is a base which is common to both matroids. Fredrickson and Srinivas [26, 27] solve the *on-line update* version of this problem efficiently through the clever use of data structures.

The two matroid intersection problem is of interest to people working in scheduling theory since it embodies a certain simple scheduling problem as a special case. In this scheduling problem, we have a collection of n tasks and each task has an associated *deadline* and *release time* (a time before which the task cannot be started). The idea is to try and schedule such a task system on a *single* processor and all the tasks have unit length. A subset of these tasks is said to be *feasible* if they can all be sequenced on the processor within their release time and deadline constraints. Then, the desired solution to this scheduling problem involves finding the maximum cardinality feasible subsets

of tasks, which are actually the bases of the *simple scheduling matroid* [43]. Variations of these matroids and the corresponding two matroidal intersection problems can be found in [26], [27], [43] and [61].

An important restriction that is inherent to all of the above mentioned scheduling problems is that they do not allow precedence constraints in any form, or equivalently, α is empty in all of these cases. In addition, they also deal with systems with a single processor. If these constraints are relaxed along these two important dimensions, that is if we consider instances with non-trivial precedence constraints and many processors, it seems extremely unlikely that we will find interesting and meaningful (from the viewpoint of helping us understand the similarity in the complexity characteristics of the apparently dissimilar polynomial subdomains of the precedence constrained scheduling problem from the previous chapter) formulations of the scheduling problem. The general precedence constrained scheduling problem being NP-hard, it clearly cannot be directly formulated in terms of any of the polynomially solvable optimization problems associated with matroids which we described in the previous paragraphs, or their tractable generalizations. Even if we restrict our attention to the polynomially solvable cases of our scheduling problem and consider formulating them in terms of matroids, we encounter difficulties in that the bases of the resulting matroidal subset systems do not always correspond to valid schedules.

Under these circumstances, solving these matroidal formulations by finding specific types of minimum cost bases and so on, does not always yield a valid or feasible solution to the original scheduling problem. Based on our experience, it seems unlikely that matroids will even play an important and meaningful role as that played by matching theory in the case of algorithms

for solving the two processor scheduling problem due to Fujii, Kasami and Ninomiya [23] or Vazirani and Vazirani [93]. The problem seems to be inherently due to the highly 'structured' nature of matroidal problems which seems to be destroyed when precedence constraints - especially those characterized by arbitrary precedence graphs - are allowed. This motivates us to seek alternate settings for characterizing and explaining the complexity of our scheduling problems, and we will now consider polytopes as a possible choice to this end.

2.1.2 Polytopes

The role of *convex polytopes* in characterizing the behavior of linear programming problems in beautiful and intuitive ways is well known. The algebraic representation of a convex polytope always defines the constraint matrix of a linear program and conversely, a constraint matrix uniquely specifies a convex polytope. There are many other interesting correspondences, for example between *basic feasible solutions* of the linear program and the *vertices* of the corresponding polytopes and the reader is referred to [74] for details.

Another instance in combinatorial optimization where polytopes have played an interesting role, is in characterizing the behavior of the widely used class of *local search* algorithms [74]. In this connection, if we consider the very general class of *discrete linear subset* problems as an example, it has been shown that the solutions of this problem which have to be searched by any local search algorithm relative to some given solution if global optimality is to be guaranteed, is exactly the set of neighbors of the given solution point on the corresponding polytope [74]. Papadimitriou and Steiglitz [73] have studied the complexity of searching this set of neighbors on the polytopes defined by

the travelling salesman problem. Owing to their work, it is now well known that indeed, the travelling salesman problem is extremely hard to solve since it follows from their results in [73] that there is no local search algorithm for solving this problem which even has polynomial time complexity *per iteration* unless P=NP!

Clearly, linear programs and integer linear programs and their corresponding polytopes are powerful enough to express and capture the behavior of combinatorial optimizations problems even when they are NP-hard. For example, there are several obvious ways of formulating the general scheduling problem in terms of integer linear programs. In the same vein, any discrete linear subset problem, which might even be NP-hard (the NP-hard travelling salesman problem is a discrete linear subset problem), can be reduced to linear programming. However, as an aside, before we get too excited about reducing a potentially NP-hard discrete linear subset problem to linear programming, we have the following sobering result due to Karp and Papadimitriou [52]: if the original discrete linear subset problem is NP-hard, then there is no polynomially concise characterization of the rows of the constraint matrix A of the linear program to which it (the discrete linear subset problem) is reduced, unless NP = co-NP; an extremely unlikely possibility indeed.

Thus, although (integer) linear programs and their polytopes have a lot to offer, let us reconsider our goals for a minute. We are attempting to take an existing set of polynomially solvable subdomains of the precedence constrained problem and seek out unifying complexity determining characteristics. To do this, we do feel the need for working at a higher level of abstraction relative to the scheduling problem itself in order to isolate and focus on the really essential components and attributes of the problem on

hand. However, an important issue which we need to resolve in this connection is to determine as to how much more abstract a level it is at which we are going to be working. Clearly, integer linear programs seem to be powerful enough and therefore abstract enough to this end.

But one has to also consider the possibility that they might be too abstract from the viewpoint of our present goals. For, while we seek to explore and thereby synthesize relationships between the polynomially solvable subdomains by working at an abstract level, we also have to retain some of the invaluable intuition that has already been gained through existing results. This is especially important if we have to build inductively upon existing knowledge about the complexity of the scheduling problem. We support our arguments by quoting Hilbert and Cohn-Vossen [44] from their beautiful work on geometry:

> In mathematics, as in any scientific research, we find two tendencies present. On the one hand, the tendency towards *abstraction* seeks to crystallize the *logical* relations inherent in the maze of material that is being studied, and to correlate the material in a systematic and orderly manner. On the other hand, the tendency towards *intuitive understanding* fosters a more immediate grasp of the objects one studies; a live *rapport* with them, so to speak, which stresses the concrete meaning of these relations.
>
> As to geometry, in particular, the abstract tendency has here led to the magnificent systematic theories of Algebraic Geometry... . Notwithstanding this, it is still true today as it ever was that *intuitive* understanding plays a major role in geometry. And such concrete intuition is of great value not only to the research worker, but also for anyone who wishes to study and appreciate the results of research in geometry.

We feel that these observations are extremely apt in our present con-

text and are by no means restricted in their scope to the field of geometry. More specifically, (integer) linear programs tend to be too abstract for our present purposes in the sense that a lot of valuable semantic information from the original scheduling problem is lost in the process of formulating it in these terms. As we saw above, this semantic information is important not only from our point of view during the actual research phase, but it also makes our results much more intuitive. Therefore, we choose to formulate the scheduling problem in the setting of hypergraph theory which is less 'structured' and therefore less restrictive when compared to matroidal theories. In contrast, hypergraphs offer a less abstract and more intuitive vehicle for formulating the scheduling problem when compared to polytopes or their algebraic equivalents.

2.2 Formulating Schedules through Hypergraphs

Let us now address the issue of casting the precedence constrained scheduling problem in terms of hypergraphs. To start with, given the precedence graph $P = \langle A, \alpha \rangle$, the *hypergraph* H associated with A is a collection of all non-empty subsets of A. A non-empty subset E_p of A (in H) is referred to as the pth *edge* of H. Matroids are essentially a special class of hypergraphs or subset systems which satisfy the two axioms which we stated earlier on in section 2.1.1. For the example instance of Figure 1-1, the corresponding hypergraph H has 511 edges four of which are shown in Figure 2-1.

For a given instance, we can specify a schedule by choosing the appropriate edges from the associated hypergraph H and composing them to form a sequence. Such a sequence is shown in Figure 2-2 for the example instance of Figure 2-1. Note that each step in the original schedule illustrated in Figure 1-2 (for this instance) corresponds to an edge in the sequence of edges in Figure 2-2. In this manner, each edge essentially represents a snapshot of all

the tasks which are scheduled to to be active during the appropriate time interval; for example, the elements in the second edge in the example sequence are exactly those tasks which are active during the second time step of the schedule, and so on.

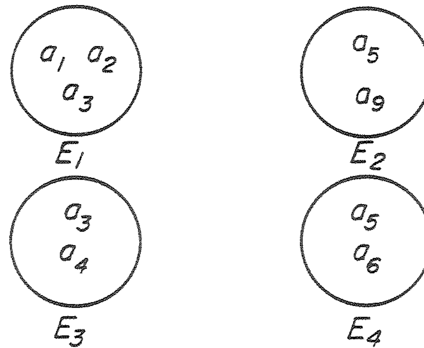


Figure 2-1: Four of the 511 edges from the hypergraph H associated with the instance of Figure 1-1.

Therefore, to construct a schedule, we consider *sequences* S of some L edges $\{E'_1, E'_2, \ldots, E'_L\}$ from hypergraph H of the corresponding instance. We say that an edge E'_q (from H) *is in sequence* S whenever it is part of the sequence. Also, an element a_i is said to be *contained in an* edge E'_q (in some sequence S) provided $a_i \in E'_q$. Given edges E'_q and $E'_{q'}$ in sequence S, we say that edge E'_q occurs *before* (*after*) after $E'_{q'}$ whenever $q < q'(q > q')$. By interpreting the edges in these sequences as being in one-to-one correspondence with the steps of a schedule, we can replace the problem of dealing with schedules with that of finding optimum sequences. The resulting problem associated with hypergraph sequences is much more convenient from the viewpoint of our further development, but before we move on to these issues, we first need to constrain sequences to account for such things as precedence constraints, resource constraints and pipelining.

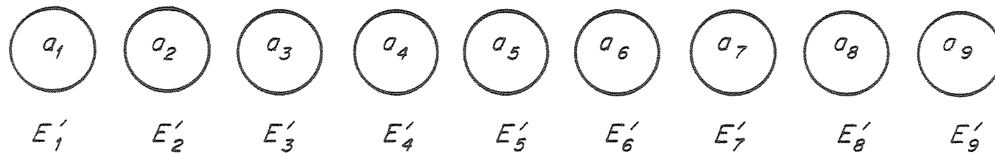


Figure 2-2: The schedule of Figure 1-2 represented as a sequence of edges of the corresponding hypergraph.

First, let us consider dealing with the issue of pipelining, that is with instances in which the degree of slicing k is allowed to be greater than unity. To do this, let us recall the example schedule from Figure 1-4 for the instance from Figure 1-1 with two stage processor-type resources ($k = 2$). To represent this example schedule in terms of hypergraph edges, we have to construct sequences in such a way that each element is contained in two different edges as shown in Figure 2-3; element a_1 is contained in the first and the second edges of this sequence to reflect the fact that it is being processed respectively at the first stage of the resources it needs during the interval of time $[0, 0.5)$ and at the second stage during the interval of time $[0.5, 1)$ in the corresponding schedule.

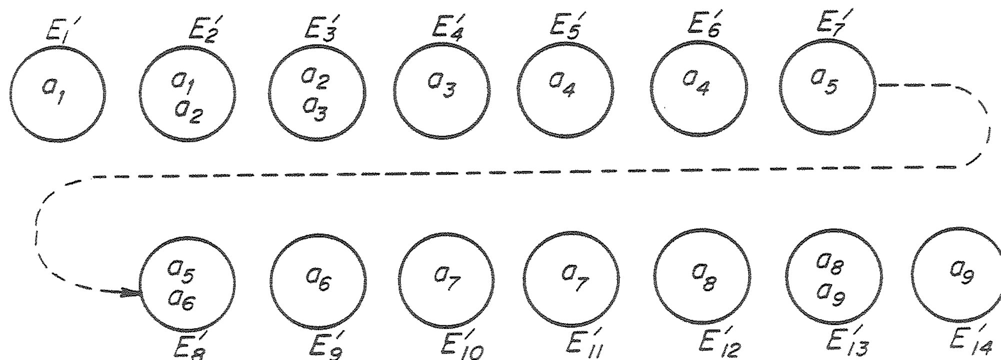


Figure 2-3: Representing the schedule of Figure 1-4 in terms of a sequence of hypergraph edges.

More generally, to formulate schedules for arbitrary k-sliced systems in terms of hypergraphs, we need to consider sequences in which each element is contained in k distinct edges. More precisely, we say that the ηth copy of an element a_i is in edge E'_q of a sequence S if and only if there exists exactly $(\eta - 1)$ edges, each of which contain element a_i, and which occur before edge E'_q in sequence S. Also, element a_i *occurs* η times in sequence S provided its ηth copy is in some edge of sequence S, whereas its $(\eta + 1)$th copy is not. Then, the types of sequences which we will use to replace schedules are *weak k-matchings*, which are defined to be sequences in which *every* element a_i occurs exactly k times for some positive integer k. For the sake of convenience, we will refer to the 1st and kth copies of an element in a sequence as its first and last copies respectively.

We can continue to view the member elements of each edge of a weak k-matching as a snapshot of all the tasks which are being processed at the various stages of the resources they need. In particular, since we are concerned only with instances which are uniformly k-sliced, it is easy to see that there is no loss in generality if we restrict our attention to schedules were all the start and finish times - even the points in time at which the tasks switch from one stage of the pipelined resources to the next - are all multiples of l (recall that $l = 1/k$). As a consequence, in all our weak k-matchings,the qth edge corresponds to the interval of time $[l \cdot (q-1)\,,\, l \cdot q)$ in the corresponding schedule. Therefore, if the ηth copy of an element a_i is in edge E'_q, then in scheduling terminology, it is to be processed at the ηth stage of the resources it needs starting at time $l \cdot (q-1)$ and ending at time $l \cdot q$.

However, weak k-matchings need not always correspond to valid schedules since they need not obey the equivalent of resource or precedence

constraints which schedules need to conform to. Recall that the precedence and resource constraints are specified through the bound vector $\mathbf{B}$, and the edge set α of the precedence graph. Let us first consider the issue of enforcing the resource constraints. To do this, we now define the *cumulative weight* of an edge E'_q in some sequence S with respect to positive integers v and η to be $\sum_{i \cdot i \in \mathbf{i}} C_{iv}$ where an $i \in \mathbf{i}$ if and only if the ηth copy of element a_i is in edge E'_q in sequence S. Stated in terms of schedules and pipelined resources, the cumulative weight of the qth edge in some sequence for a given v and η essentially gives the total number of units of the type-v resources which are needed by all the tasks that are active during the interval of time $[l \cdot (q-1), l \cdot q)$ at the ηth stage of the resources they need.

Then, in order to encode resource constraints, we simply have to restrict our attention to *bound preserving* sequences which are defined as follows. A sequence S is bound preserving with respect to a m-tuple $\mathbf{B} = \langle B_1, ..., B_m \rangle$ of positive rationals provided that for each edge E'_q in it, its (this edge's) cumulative weight with respect to each v and η is bound above by B_v where $1 \leq v \leq m$ and $\eta \leq k$. It is not difficult to see that bound preserving sequences always obey the resource constraints if the m-tuple $\mathbf{B}$ in the above definition is interpreted to be the bound vector which, is a specification of the resource availability.

Similarly, in order to ensure that sequences also respect precedence constraints which are specified in terms of the edge set α of the problem instance, we restrict our attention to *order preserving* sequences which satisfy the property that for any i where $1 \leq i \leq n$, if the first copy of element a_i is in edge E'_q, then the last copy of each element a_j in sequence S is in some

edge $E'_{q'}$ with $q' < q$ whenever $\langle a_j, a_i \rangle \in \alpha^+$ where α^+ is the transitive closure of α. Note that both the sequences of Figures 2-2 and 2-3 are order as well as bound preserving.

So far, we have identified and incorporated all the really essential constraints from the domain of schedules into the domain of sequences. Recall however from Chapter 1 that we are particularly interested in the special class of non-preemptive schedule. Therefore, in order to replace schedules with sequences, we need to identify sequences which correspond not just to valid schedules, but those which represent non-preemptive schedules as well. This can be accomplished quite easily by requiring that our sequences be *proper*; a sequence S is proper provided for all i, if the ηth copy of an element a_i is in its qth edge, then its (element a_i's) $(\eta - 1)$th copy is in its $(q-1)$th edge whenever $\eta > 1$. Intuitively, proper sequences are those in which all the k copies of a given element occur in consecutive edges, to denote that the task which corresponds to the element in question is to be processed continuously from start to finish in the corresponding schedule. Therefore, the types of sequences which correspond to valid and non-preemptive schedules are *interesting* sequences which are: *(i)* weak k-matchings, *(ii)* bound preserving, *(iii)* order preserving and, *(iv)* proper. In the sequel, we will simply refer to interesting sequences as sequences, and whenever a sequence is not interesting, we will state this explicitly.

2.3 On Reducing Makespan Minimization to Tardiness Minimization

In this section, we will integrate both the makespan as well as the tardiness minimization versions of the precedence constrained scheduling problem into a single optimization problem cast in terms of hypergraph sequences. As a first step in this direction, we will reduce the problem of finding schedules with minimum makespan to that of finding schedules with minimum tardiness. To accomplish this reduction, let us start off by recalling that an instance of the makespan minimization problem is specified in terms of: *(i)* a precedence graph P, *(ii)* a class function $\mathbf{C}$, *(iii)* a task length $\mathbf{l}$, *(iv)* a bound vector $\mathbf{B}$ and, *(v)* a degree of slicing k. The only difference between an instance of this problem and one of tardiness minimization is that in the latter, in addition to these five components, we also have a cost-determining function f which associates a deadline with each element $a_i \in A$. Then, given an instance of the scheduling problem with makespan as the objective function, consider transforming it, as shown in Figure 2-4, by adding a *uniform* cost-determining function f_u to it; the cost-determining function f_u is uniform provided $f_u(a_i) = f_u(a_j) = 0$ for all elements a_i and a_j from A. Then, the resulting transformed instance has all the six attributes of an instance of the tardiness minimization problem.

Let us now consider the problem of determining a schedule with minimum tardiness for this transformed instance (Figure 2-4). An interesting thing about the instances created by this transformation is that the tardiness of the resulting schedules always equals $l \cdot L$, where the schedule is L steps long and the function f_u assigns a value of zero to all the elements a_i. Moreover, it is easy to see that in this case, the elements in the last step of these schedules always determine their tardiness which in turn equals their makespan.

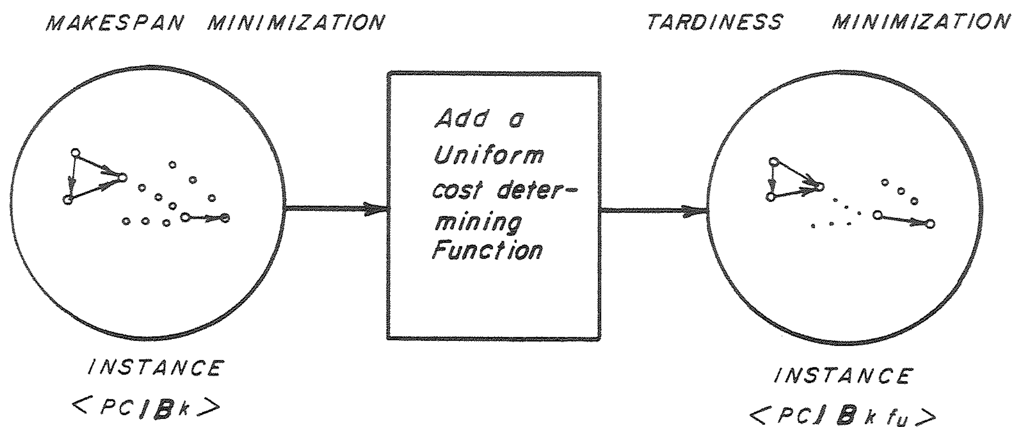


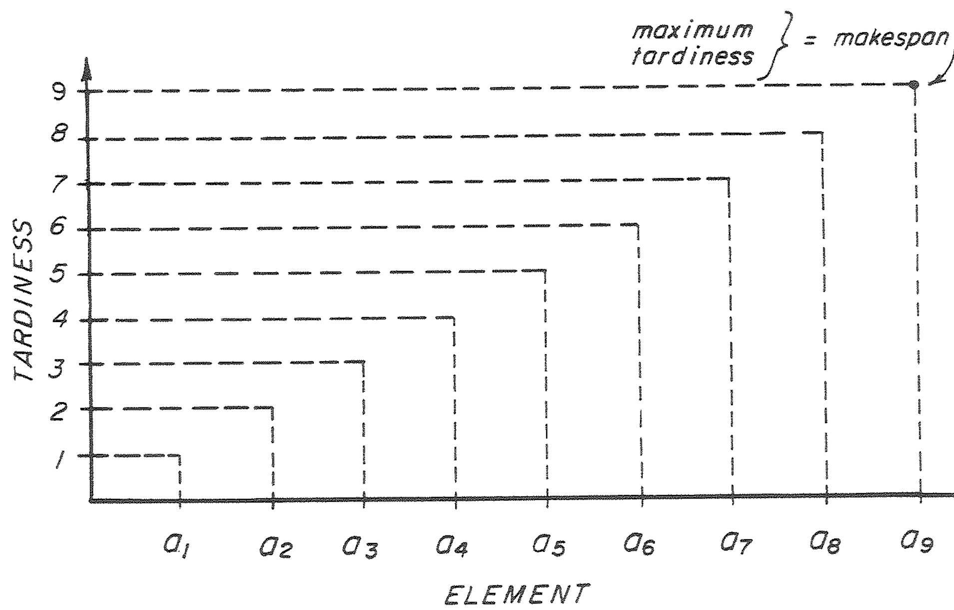
Figure 2-4: Reducing the makespan minimization problem to the tardiness minimization problem.

To illustrate these points, let us consider transforming the instance from Figure 1-1 by adding the cost-determining function shown in Figure 2-5 (i) to it. Also, let us reconsider the schedule from Figure 1-2; this schedule which was meant for the original instance from Figure 1-1 is also valid for the transformed instance. The tardiness in this schedule of each element from the transformed instance is shown in Figure 2-5 (ii).

From this example, all our earlier claims about the tardiness of the schedule for the transformed instance are easily verified including the fact that element a_9 in the last step has a tardiness of nine, which in turn equals its (the schedule's) makespan. It then follows from these observations that any minimum tardiness schedule for the transformed instance also has minimum makespan and as such, solving the tardiness minimization problem for the transformed instance tantamounts to solving the original makespan minimization problem.

$a_i \rightarrow$	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
$f_u(a_i)$	0	0	0	0	0	0	0	0	0

(i)



(ii)

Figure 2-5: (i). Transforming the instance of Figure 1-1 by adding the function f_u to it and (ii.) the tardiness of these elements in the schedule of Figure 1-2.

In this manner, these reductions or transformations allow us to cast the makespan minimization problem as a special case of the tardiness minimization problem with an initial 'pre-processing' step which involves adding an appropriate (uniform for example) cost-determining function as shown in Figure 2-4; a task which can be accomplished in time $O(n)$. This in turn allows us to cast both the tardiness and the makespan minimization versions of the scheduling problem as a single optimization problem in terms of sequences of hypergraph edges. An instance of this *optimization problem associated with hypergraphs* is represented by a six tuple $\langle P, C, \mathbf{l}, \mathbf{B}, k, f \rangle$ with the same components as an instance of the tardiness minimization problem. Given an instance of this problem, our goal is to find minimum cost sequences, where the *cost* of an element a_i with its last copy in edge E'_q in sequence S is $(l \cdot q - f(a_i))$ whenever $l \cdot q > f(a_i)$, and is zero otherwise, and the *cost* of a sequence is the maximum over all the costs of all the elements in it.

This allows us in the sequel to concern ourselves with only one type of optimization problem and cost which are associated with a sequence S, with the implicit understanding that if the sequence corresponds to an instance of the makespan minimization problem, we are actually referring to its transformed version. Also, in what follows, we will deal with sequences directly without always specifying the associated instance (of the optimization problem associated with hypergraphs) in an explicit fashion, although it is always understood that there is some underlying instance. In the same way, we will be referring to two sequences as being *equivalent* whenever they have the same associated instance for which they are defined, although such an instance might not always be explicitly described. We will see in the next two chapters that this approach of studying sequences without explicit reference to their associated instances helps us in focusing on the 'solution or output domain' of

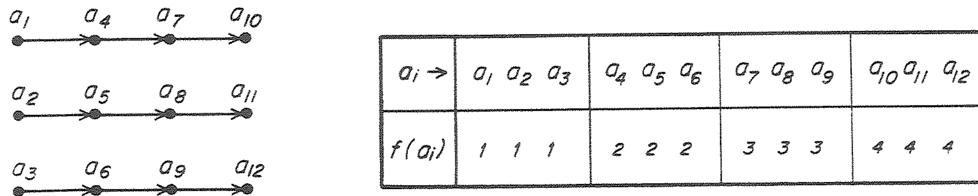
the problem directly, which in turn will prove to be of great value in understanding their behavior at optimality.

2.4 Factors Affecting the Feasibility of Sequences with Zero Cost

As a first step towards characterizing the behavior of minimum cost sequences or equivalently, the behavior of sequences at optimality, let us start understanding the issues which influence and thereby play a role in determining their cost. Since the cost of a sequence can assume any non-negative value, this problem can be quite complicated in general. In the interest of simplifying our task somewhat, let us start out by studying only specialized types of sequences and in particular, those with zero cost. In other words, we will first focus our attention on a special case of the problem on hand by characterizing the behavior and attributes of sequences with zero cost. Then, we can subsequently work towards extending these specialized characterizations of zero-cost sequences to encompass arbitrary optimum sequences which need not necessarily have an associated cost of zero.

2.4.1 The Role of the Bound Vector

The bound vector **B** clearly influences the cost of a sequence since it plays a role in determining as to how many elements can be packed into any of its edges. To illustrate this point in greater detail, consider the instance with the precedence graph and cost-determining function indicated in Figure 2-6. For this instance, let $m = 1$, $B_1 = 2$, $C_{i1} = 1$ for $1 \leq i \leq 9$ and $l = k = 1$. The interesting aspect of this instance is that the cost of any associated sequence is dependent only on the bound preserving constraint which in turn draws upon the values in the bound vector; this follows from the fact the cost of any sequence for this instance is at least two, even if we were to consider a sequence by ignoring all the precedence relationships or constraints.



$a_i \rightarrow$	$a_1\ a_2\ a_3$	$a_4\ a_5\ a_6$	$a_7\ a_8\ a_9$	$a_{10}\ a_{11}\ a_{12}$
$f(a_i)$	1 1 1	2 2 2	3 3 3	4 4 4

Figure 2-6: The precedence graph and the cost-determining function of an instance which shows the effect of the saturation indices on cost.

In order to quantitatively capture this influence of the bound vector on the cost of a sequence, we will distinguish different 'types' of elements where elements a_i and a_j are of the same *type* if and only if $\mathbf{C}(a_i) \underset{v}{=} \mathbf{C}(a_j)$ and $\underset{v}{=}$ represents vector equality. In this manner, the set of elements A is partitioned in M equivalence classes $\Pi_1, \ldots, \Pi_M$ where elements a_i and a_j belong to the same class Π_x for some x if and only if they are of the same type. Basically, the elements of a given type correspond to tasks with identical resource requirements. In the example instance of Figure 1-1, elements $\{a_1\ a_2\ a_3\ a_4\ a_7\ a_8\ a_9\}$, all of which belong to the same class say Π_1, are all of the same type, whereas elements $\{a_5\ a_6\}$ from class Π_2 are of a different type.

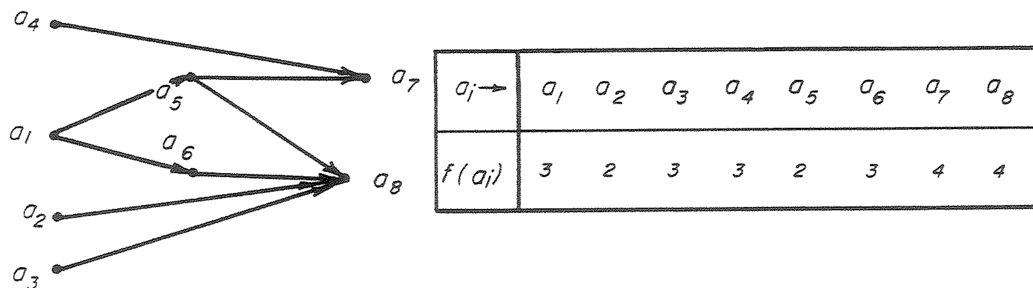
Using this, we quantify the maximum number of ηth (for $1 \leq \eta \leq k$) copies of type-x elements that can be 'packed' into in any edge E'_q in a sequence S through the *saturation index* SI_x of class Π_x (with respect to bound vector $\mathbf{B}$) as follows:

$$\min_{1 \leq v \leq m} \{ \lfloor B_v / C_{iv} \rfloor \ : a_i \in \Pi_x \}.$$

Note that for the example sequence of Figure 2-3, SI_1 and SI_2 both equal unity and consequently, no edge in this sequence contains more than one copy of an element of either type.

2.4.2 Predecessor-Successor Interactions and Cost

Let us now move on to understanding and formulating the effect of the precedence relationships or constraints on the costs of sequences. To do this, let us start with the example instance whose precedence graph and cost-determining function are shown in Figure 2-7. In addition, for this instance, let $m=1$, $\mathbf{B} = <2>$, $\mathbf{C}(a_i)=\langle 1 \rangle$ for $1 \leq i \leq 8$, and $l=k=1$. Two equivalent sequences for this instance S_1 and S_2, with respective costs of 1 and 0 are illustrated in Figure 2-8. Sequence S_1 has unit cost since element a_5 with $f(a_5)=2$ is in its third edge; intuitively, it is completed at a point in time (third time unit) which is later than its deadline (two) by one unit of time.



$a_i \rightarrow$	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
$f(a_i)$	3	2	3	3	2	3	4	4

Figure 2-7: The precedence graph and the cost-determining function of an instance.

We can view these two sequences S_1 and S_2 as being related through a series of swaps as shown in Figure 2-8. In this example, elements a_1 and a_5 are swapped in sequence S_1 respectively with elements a_3 and a_4 to get sequence S_2 with lower (in fact, optimum) cost. Also, note that only element a_5 is contributing to the cost of sequence S_1. Therefore, if the cost of sequence S_1 is to be lowered, any equivalent sequence which we might consider must have element a_5 in its first or second edge. Now, if we try to improve the cost of sequence S_1 by moving or swapping element a_5 by itself, it becomes clear

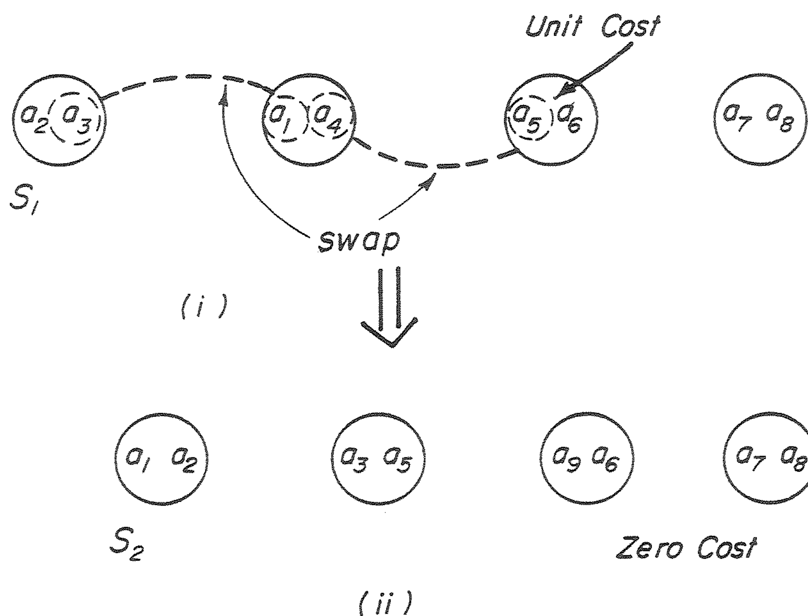


Figure 2-8: Two equivalent sequences for the instance from Figure 2-7 with a cost of : (i) one and (ii) Zero.

rather quickly that this approach is bound to fail. This is because element a_1 is in its second edge and therefore, moving element a_5 to either the first or the second edge without simultaneously moving element a_1 will yield a sequence which is not order preserving.

So, while element a_5 is indeed directly contributing to the cost of sequence S_1, element a_1 is also contributing to the same end, albeit indirectly. Thus, even though the direct influence of element a_1 on the cost of sequence S_1, owing to the value assigned to it by the cost-determining function, is not significant, the value which this (cost-determining) function assigns to element a_5 does affect element a_1 as well; this inherited constraint forces element a_1 to be included in the first edge of any sequence which is equivalent to sequence S_1, and which has a cost of zero.

This type of influence which an element has on the cost of a sequence, not due to its own constraints but due to those inherited through interactions determined by the precedence graph, represent another important factor which needs to be captured. Of course, in a general scenario, these precedence relationship based interactions between elements combine with the constraints drawn from the bound vectors (which we captured in the previous section through saturation indices) to play a joint role in influencing the costs of the corresponding sequences.

In order to capture the combined effect of the above mentioned two factors, which are rooted respectively in the bound vector and the precedence graph, we need to introduce the following additional definitions. An element a_i is a *predecessor* (*successor*) of element a_j in precedence graph P if and only if $\langle a_i, a_j \rangle \in \alpha^+$ ($\langle a_j, a_i \rangle \in \alpha^+$). *Immediate predecessors* (*successors*) are defined in a similar manner with α^+ replaced by the transitive reduction α. The *distance between* elements a_i and a_j (in $P = \langle A, \alpha \rangle$) whenever $i \neq j$ is the number of elements in the longest sequence $a'_1, a'_2, \ldots, a'_{i'}, a_j$ such that $\langle a_i, a'_1 \rangle$, $\langle a'_{i'}, a_j \rangle$, and $\langle a'_z, a'_{z+1} \rangle$ for $1 \leq z < i'$ are all in α. It is not defined when no such sequence exists between elements a_i and a_j for $i \neq j$, and equals zero by definition when $i = j$. The *level* of an element a_i with at least one successor in precedence graph P is the maximum of all the defined distances between element a_i and all the sink elements and the *sink elements* (those with no successors in P) are all at level zero by definition. The height h of a precedence graph P is the maximum over all the levels of all the elements $a_i \in A$.

Let us now return to our original goal of capturing the effect of predecessor-successor interactions on the cost of a sequence, especially as they

combine with the constraints drawn from the bound vector. To do this, we introduce the *modified cost-determining function* f' which is defined recursively as follows:

1. $f'(a_i) = f(a_i)$ whenever element a_i is a sink element in P.

2. $f'(a_i)$ of an element a_i at level $\lambda > 0$ is the minimum of $f(a_i)$ and

$$\min_{1 \leq x \leq M} \{ \min_{d':d' \in D_i} \{d' - l \cdot \{\lceil n(i,d',x)/SI_x \rceil + k - 1\}\}\}$$

where $n(i,d',x)$ equals the number of successors of element a_i in precedence graph P which belong to class Π_x, and whose modified cost-determining function values are bound above by d', and a $d' \in D_i$ if and only if there exists a successor a_j of element a_i in P such that $f'(a_j) = d'$. Also, for the sake of convenience let

$$f^*(a_i) = \min_{1 \leq x \leq M} \{ \min_{d':d' \in D_i} \{d' - l \cdot \Delta(i,d',x,k)\}\}$$

where

$$\Delta(i,d',x,k) = \{\lceil n(i,d',x)/SI_x \rceil + k - 1\}.$$

The modified cost-determining function simultaneously accounts for the effect which the cost-determining function values (deadlines) and the bound preserving (or resource) constraints of the successors of a given element have on its own urgency, which is reflected in its modified cost-determining function value. In summary, the saturation index and the modified cost-determining function play an obvious role in influencing the cost of a sequence and indeed, as we will see in the next couple of chapters, they play a surprisingly dominant role in helping us characterize the behavior of sequences at optimality.

2.5 On Characterizing the Feasible Set of Sequences and Some Properties

Before progressing further, let us recall that our present approach involves concentrating on the solution or output domain of the optimization problem on hand, rather than its input domain. To this end, let us formulate the solution domain associated with a given instance in terms of the *feasible set* $\mathbf{S}$, which is defined as the set of all equivalent sequences for the given instance. By doing this, we can start establishing properties of sequences in the light of this characterization of the solution domain. Specifically, what we are interested in accomplishing of course is to delineate non-trivial properties of subsets of $\mathbf{S}$; especially those subsets which include the minimum cost sequences. Now, in keeping with our strategy of focussing our attention initially on the behavior of sequences with zero cost, we will first look at $\mathbf{S}_\emptyset$ which is the (possibly empty) subset of feasible set $\mathbf{S}$ which includes exactly the zero-cost sequences.

As seen in the previous section, the modified cost-determining function gives us a much better estimate of the urgency, and hence the potential of each element a_i in determining whether or not an associated sequence S has zero cost. Moreover, the modified cost-determining function is directly related to the *modified cost* of a sequence S which is defined as the maximum of all the modified costs of its constituent elements, and the *modified cost* of an element a_i with its last copy in edge E'_q in sequence S equals $(l \cdot q - f'(a_i))$ whenever $(l \cdot q - f'(a_i))$ is greater than zero, and is zero otherwise. Then, since the modified cost-determining function of an element captures the urgency of an element better than the cost-determining function does, we will start our investigation into the behavior of zero-cost sequences by characterizing $\mathbf{S}_\emptyset^m$, the subset of $\mathbf{S}$ all of whose member sequences have a modified cost of

zero. Then, for any given feasible set $\mathbf{S}$, if we can appropriately relate its two important subsets $\mathbf{S}_\emptyset$ and $\mathbf{S}_\emptyset^m$, we can hope to extend the characterization of $\mathbf{S}_\emptyset^m$ which has the sequences with zero modified cost as its members, to a characterization of sequences with zero cost through the set $\mathbf{S}_\emptyset$.

Not coincidentally, a rather strong relationship exists between these two important subsets of a feasible set $\mathbf{S}$. In order to establish this fact, we first need to show that the following relationship exists between the modified costs of elements which are in a predecessor-successor relationship.

Lemma 2.1: *In any sequence S, if there exists an element a_i such that $f'(a_i) < l \cdot q$ and $f(a_i) \geq l \cdot q$ where the last copy of element a_i in sequence S is in edge E'_q, then there exists a successor element a_j of a_i with its last copy in edge $E'_{q'}$ such that $f'(a_j) < l \cdot q'$.*

Proof: If no such element a_i exists in any sequence, we are done trivially. On the other hand, let us suppose that such an element a_i exists in some sequence S. Then, since

$$f(a_i) > f'(a_i),$$

it follows that there exists some $\delta \in D_i$ and x without loss of generality for which $n(i,\delta,x) > 0$ and also,

$$f'(a_i) = \{\delta - l \cdot \Delta\,(i,\delta,x,k)\}.$$

Since sequence S is a weak k-matching and order preserving, it follows that all the k copies of each of these $n(i,\delta,x)$ successors of element a_i from class Π_x, whose modified cost-determining functions are bound above by δ, are in edges which occur after edge E'_q in sequence S. Since sequence S is also

bound preserving, it is easy to see that the last copy of at least one of these successors, say that of element a_j, must be in edge $E'_{q'}$, where

$$q' \geq \{\Delta(i,\delta,x,k) + q\} .$$

Therefore, multiplying both sides by l, we have

$$\begin{aligned} l \cdot q' &\geq l \cdot \{\Delta(i,\delta,x,k) + q\} \\ &> l \cdot \Delta(i,\delta,x,k) + f'(a_i) \\ &> l \cdot \Delta(i,\delta,x,k) + \{\delta - l \cdot \Delta(i,\delta,x,k)\} \\ &> \delta \\ &\geq f'(a_j) \end{aligned}$$

completing the proof. □

Based on this relationship, we can further show that given any sequence S, its cost and modified cost are strongly coupled in the following fashion:

Lemma 2.2: *In any sequence S, if element a_i is such that $f'(a_i) < l \cdot q$ where the last copy of element a_i is in edge E'_q, then there exists an element a_j in sequence S with its last copy in edge $E'_{q'}$ such that $f(a_j) < l \cdot q'$.*

Proof: Consider a sequence S without loss of generality and the largest q where $1 \leq q \leq L$ such that edge E'_q has the last copy of an element a_i for which $f'(a_i)$ is always strictly less than $l \cdot q$, for if no such sequence exists, we are done since the lemma is then trivially true. Now, if the lemma is false, there exists such a sequence S and moreover, this sequence must be such that $f(a_j)$ of any element a_j for $1 \leq j \leq n$ is never less than $l \cdot q'$, where the

last copy of element a_j in sequence S is in edge $E'_{q'}$. Then element a_i is such that

$$f(a_i) \not< l \cdot q$$

or equivalently,

$$f(a_i) \geq l \cdot q.$$

But since

$$f'(a_i) < l \cdot q$$

by hypothesis, it follows from Lemma 2.1 that element a_i has a successor $a_{i'}$ with its last copy in edge $E'_{q''}$ such that $f'(a_{i'}) < l \cdot q''$ which contradicts the maximality of q. □

This Lemma essentially confirms that if a given sequence S has a modified cost greater than zero, then so is its cost. This property is sufficient for us to relate the two subsets $\mathbf{S}_\emptyset$ and $\mathbf{S}_\emptyset^m$ as follows:

Corollary 2.1: *In any feasible set S,* $\mathbf{S}_\emptyset = \mathbf{S}_\emptyset^m$.

Proof: That $\mathbf{S}_\emptyset^m \subseteq \mathbf{S}_\emptyset$ is easily verified since $f'(a_i) \leq f(a_i)$ for all elements a_i is any sequence S. That $\mathbf{S}_\emptyset \subseteq \mathbf{S}_\emptyset^m$ follows from the contrapositive of Lemma 2.2, completing the proof. □

In the next chapter, we will start off by developing a characterization of the subset $\mathbf{S}_\emptyset^m$ of a feasible set $\mathbf{S}$. We will then use Corollary 2.1 (in Theorem 3.1) to establish that this characterization also extends to include the corresponding subset $\mathbf{S}_\emptyset$.

Chapter 3

On Projections and Affined Sequences

In this chapter, we will identify a class of sequences which have the special property that they have zero cost whenever an equivalent zero-cost sequence exists for the corresponding instance. To do this, we introduce some additional machinery in Section 3.1 in the form of the 'projection' operation on sequences. This operation allows us to study the essential parts of a given sequence which in turn leads us to identifying the important class of 'affined' sequences in Section 3.2. Subsequently, in Section 3.3, we use affined sequences to characterize a non-trivial subset of the feasible set $\mathbf{S}$ all of whose members satisfy the above mentioned 'special' property; equivalently, we use affined sequences to characterize the subset $\mathbf{S}_{\emptyset}$ of the feasible set $\mathbf{S}$. In Chapter 4, we will show that affined sequences actually provide us with a powerful characterization, not just of sequences with zero cost, but of minimum cost sequences as well. In doing so, we rely very heavily on some extremely interesting properties of affined sequences which follow from their inherent structure. In fact, we will see in the rest of this dissertation that this class of affined sequences plays an extremely central role in capturing the essence of the behavior of sequences at optimality.

3.1 Selecting Elements Through Projections

Given a sequence S, the question we are faced with in order to determine as to whether or not it or an equivalent sequence has a cost of zero, is the following:

- What *conditions* must its member *elements* satisfy so that we can argue that if the cost of sequence S is greater than zero, then in fact, there is no equivalent sequence that has zero cost?

For, if these conditions are well-defined to where a given element in the sequence either satisfies them or fails to do so, and if they are general enough, they can be used to delineate classes of sequences and thereby used to characterize subsets of $\mathbf{S}_{\emptyset}$ - this being our current goal. By posing the above question, we have essentially shifted the emphasis of our search for characteristics which capture the behavior of sequences with zero cost from the general level of looking at sequences, to focussing directly on its constituent elements instead.

To answer the above question, let us suppose for the sake of argument that a given sequence say S_1 has greater than zero cost; also, let sequence S_2 be an equivalent sequence with zero cost. Then, we can define a series of pairwise 'swaps' between elements of S_1 and thereby arrive at sequence S_2 with zero (or more generally improved) cost. This has already been illustrated in the example of Figure 2-8 in which we successively swap elements a_1 with a_3 and a_4 with a_5 (a two swap sequence) in sequence S_1 which has unit cost, to get sequence S_2 with zero cost. So, given a sequence S, the question posed in the previous paragraph can be refined and restated by using this notion of a series of element-swaps as follows:

- What *conditions* must its constituent *elements* satisfy in order to

rule out the possibility of a *series of swaps* which yield an equivalent sequence with zero (or more generally improved) cost?

In order to address the issue of whether or nor an element in a given sequence can be a part of a series of swaps with a concomitant cost improvement, we need to look closely at the way in which its (the sequence's) constituent elements are interacting. More importantly, we need a mechanism through which we can isolate and study specific groups of elements in the sequence ; especially those elements from a given sequence which are significant from the viewpoint of influencing its cost behavior, since not all the elements in a sequence play a significant role in influencing its cost.

In this connection, recall from Chapter 2 that an element's type (which in turn determines a saturation index) and the value assigned to this element by the modified cost-determining function, both play a role in its influence on the cost of a corresponding sequence. We will now draw upon this information to formulate the 'projection' operation on sequences; this operation will subsequently allow us to identify and select only those elements which impact upon the cost of a sequence. A *projected sequence* $\hat{S} = \langle E''_1, E''_2, ..., E''_L \rangle$ derived or *projected* from sequence $S = \langle E'_1, E'_2, ..., E'_L \rangle$ with respect to d' and x is defined such that an element a_i is in edge E''_q if and only if the first copy of element a_i is in edge E'_q of sequence S, it is of the xth type ($\in \Pi_x$), and $f'(a_i)$ is no greater than d'.

The projection operation helps us in selecting elements from sequences which satisfy certain special properties as shown in Figure 3-1. Specifically, given a d' and x, projections allow us to select those elements from and edge E'_q of a sequence S which lie in the intersection of the two sub-

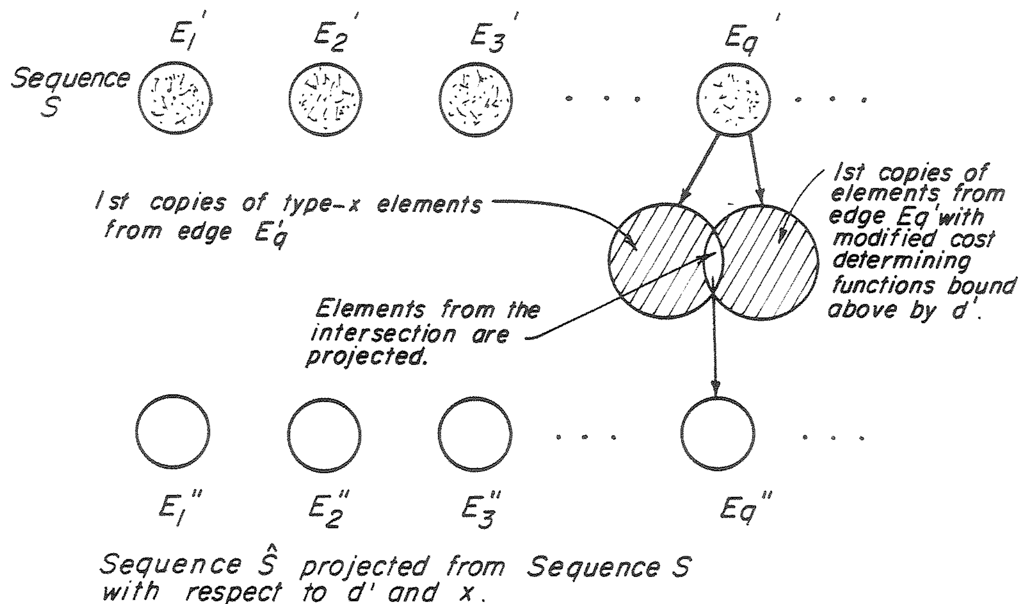


Figure 3-1: The elements selected by the projection operation.

sets of $E'_{q'}$; one subset which has exactly the elements from Π_x and another which has exactly those elements whose modified cost-determining functions are bound above by d'. Let us now use these projections to establish the following obvious but useful property of sequences:

Lemma 3.1: *In any sequence* $\hat{S} = \langle E''_1, E''_2, ..., E''_L \rangle$ *projected from sequence* S *with respect to some* d' *and* x, $|E''_q|$ *is always bound above by* SI_x.

Proof: Immediate from the definition of SI_x and the fact that sequence S is interesting and hence bound preserving. □

This notion of projecting or selecting equivalence classes from sequences of sets is a natural extension of the set theoretic projection operation and similar generalizations of this operation have been used in the past, for example by Codd [8] in formulating the powerful and extremely effective

relational algebra. Finally, in our definition of projected sequences, it is not absolutely essential that we project only the first copies of the elements from the original sequence but nevertheless, we will continue to maintain this constraint since it helps us in simplifying some of the technical aspects of our further development.

3.2 Affined Sequences

Having defined projected sequences, let us now return to our original goal of using this operation to distinguish different types of edges and elements in them. An edge E''_q in a sequence $\hat{S}$, projected with respect to some d' and x from sequence S, is *saturated* (*unsaturated*) whenever $|E''_q|$ is equal to (less than) SI_x, the saturation index of class Π_x. An edge E''_q is a *primary tail* in a projected sequence $\hat{S}$ if and only if edge E''_q is unsaturated and every edge $E''_{q'}$ with $q' < q$ is saturated. Also, edge $E''_{q'}$ is in the *primary saturation subsequence* of some projected sequence $\hat{S}$ provided edge E''_q is its primary tail and $q' \leq q$.

Having identified these edges which constitute the saturated subsequence of a sequence, we are now in a position to build on this idea to delineate the elements which play a role in influencing its cost. An element a_i (of type-x with $f(a_i) = d'$) is *distinguished* in sequence S provided its first copy is in edge E'_q and edge E''_q is *not* in the primary saturation subsequence in the sequence $\hat{S}$ projected with respect to d' and x. As it turns out - a fact which will be verified in the next section - only distinguished elements need to be considered further in the sense of satisfying additional conditions which assure us that they cannot be a part of a series of swaps, as shown in Figure 2-8 for example, with an accompanying cost improvement. We will also see in the course of proving Theorem 3.1 that non-distinguished elements are already

constrained enough in this sense as a consequence of Lemma 3.1 and their membership in the primary saturation subsequence of the corresponding projected sequence $\hat{S}$.

Interestingly enough, even in the case of distinguished elements in a given sequence, it is sufficient if they satisfy a single condition in order to claim that they cannot be a part of a series of swaps with an accompanying cost improvement or equivalently, to assert that if the corresponding sequence has greater than zero cost, then so does every sequence equivalent to it. In order to precisely formulate this condition or constraint which distinguished elements need to satisfy, we need to identify the *critical distance* (Figure 3-2) of an element a_i with respect to d', and x as being equal to $\Delta(i,d',x,k)$. Also, the *distance between* elements a_i and a_j in sequence S is $|q - q'|$ where the first copies of elements a_i and a_j are respectively in edges E'_q and $E'_{q'}$, as shown in Figure 3-3.

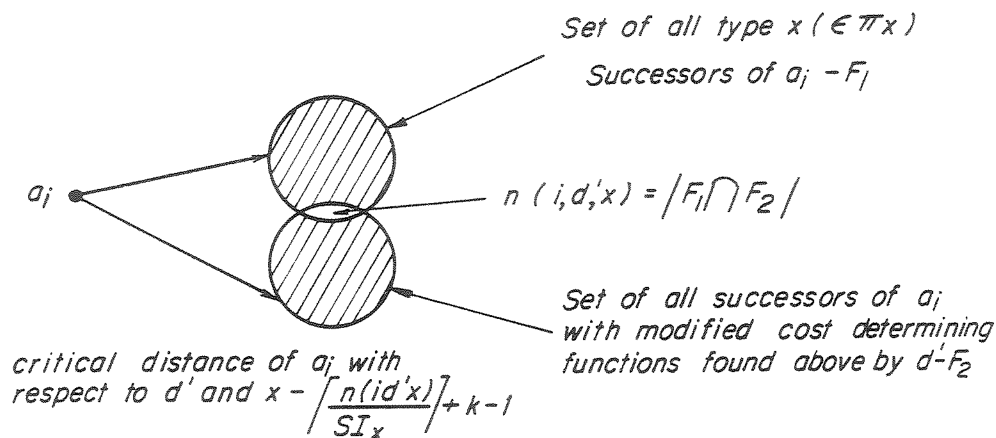


Figure 3-2: The critical distance of element a_i with respect to d' and x.

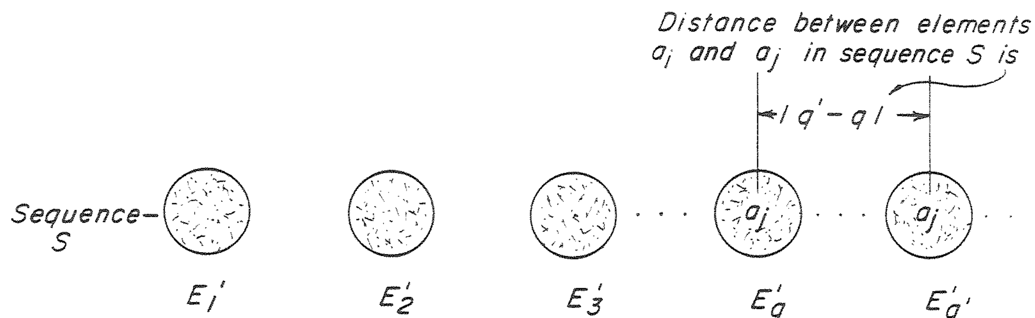


Figure 3-3: The distance between elements a_i and a_j in sequence S.

Then, the types of sequences in which distinguished elements satisfy the desired constraint which ensures that they cannot be a part of a series of swaps leading to an equivalent sequence with improved cost, will be referred to as *affined* sequences; we say that a sequence S is affined if and only if every distinguished element a_i has a predecessor element a_j such that the distance between elements a_i and a_j in sequence S is bound above by the critical distance of element a_j with respect to d' and x, where a_i is a type-x element and $f'(a_i) = d'$. This class of affined sequences lies at the very core of our problem on hand, and all our results will be built around this important class. They have the special property that in them, distinguished elements exhibit an affinity (hence the name affined) to their predecessors by trailing them by no more than a certain bounded distance - the critical distance of the predecessor. Because of this affinity, the modified costs of distinguished elements in an affined sequence are related to that of their predecessors in a very interesting way indeed, as stated below in Lemma 3.2.

Lemma 3.2: *In any affined sequence S, if a distinguished element a_i is such that $l \cdot q > f'(a_i)$ where the last copy of element a_i is in edge E'_q, then there exists a predecessor element a_j of a_i with its last copy in* edge $E'_{q'}$ *such that $l \cdot q' > f'(a_j)$.*

Proof: Since from the hypothesis of this lemma, we have an element a_i with its last copy in edge E'_q such that

$$f'(a_i) < l \cdot q,$$

and since from the definition of the modified cost-determining function, we know that for any predecessor element $a_{i'}$ of a_i,

$$f'(a_{i'}) \leq f'(a_i) - l \cdot \Delta(i',\delta,x,k)$$

where a_i is a type-x element with $f'(a_i) = \delta$, we can deduce that

$$f'(a_{i'}) < l \cdot q - l \cdot \Delta(i',\delta,x,k).$$

Now, since element a_i with its last copy in edge E'_q is distinguished and since sequence S is affined, it must have a predecessor element a_j with its last copy in edge $E'_{q'}$ such that

$$(q - q') \leq \Delta(j,\delta,x,k).$$

Also, since element a_j is a predecessor of element a_i, we know that

$$f'(a_j) < l \cdot q - l \cdot \Delta(j,\delta,x,k)$$

and therefore,

$$\begin{aligned} f'(a_j) &< l \cdot q - l \cdot (q - q') \\ &< l \cdot q' \end{aligned}$$

completing the proof. □

3.3 Affined Sequences and their Cost

Given a feasible set $\mathbf{S}$, let $\mathbf{S}_\alpha$ be its subset which contains exactly all the affined sequences in $\mathbf{S}$. Then, a powerful characterization of $\mathbf{S}_\emptyset$ is realized owing to the following inclusion relationship between $\mathbf{S}_\alpha$ and $\mathbf{S}_\emptyset$ which is stated below in Theorem 3.1 and is illustrated in Figure 3-4. From this inclusion relationship, we know that affined sequences have the desired property in that if they do not have zero cost, then in fact no equivalent zero-cost sequence exists for the corresponding instance.

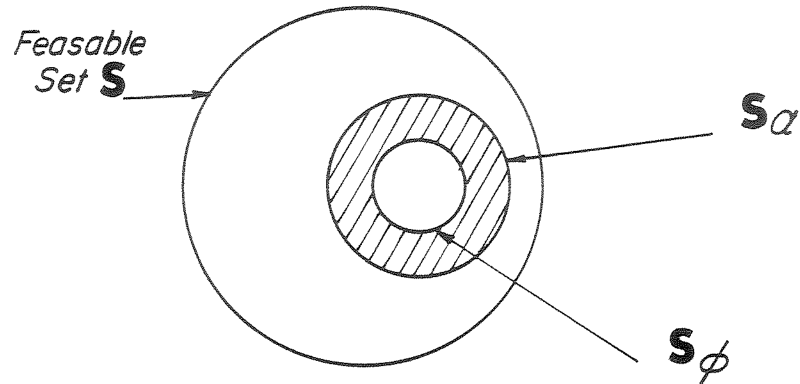


Figure 3-4: The relationship between subsets $\mathbf{S}_\emptyset$ and $\mathbf{S}_\alpha$ of a feasible set $\mathbf{S}$ whenever $\mathbf{S}_\emptyset$ is non-empty.

Theorem 3.1: *Given any feasible set* $\mathbf{S}$, $\mathbf{S}_\alpha \subseteq \mathbf{S}_\emptyset$ *whenever* $\mathbf{S}_\emptyset$ *is non-empty.*

Proof: Suppose that the theorem is false. Then, there exists an affined sequence $S \notin \mathbf{S}_\emptyset$ and yet $\mathbf{S}_\emptyset$ is not empty. Then it follows from Corollary 2.1 that $S \notin \mathbf{S}_\emptyset^m$ and that $\mathbf{S}_\emptyset^m$ is not empty, or that there exists a sequence S' equivalent to sequence S in $\mathbf{S}_\emptyset^m$. Since $S \notin \mathbf{S}_\emptyset^m$, there exists at least one element in it say a_i, with its last copy in edge E'_q such that $f'(a_i)$ is strictly less than $l \cdot q$. Among all such elements in sequence S, pick an element a_j with its last copy in edge $E'_{q'}$ such that any element $a_{j'}$ with its last copy in any edge E'_p for $q'' < q'$ is such that $f'(a_{j'})$ is not less than $l \cdot p$. That is, q' is the smallest index of an edge in sequence S which contains the last copy of an element - a_j in this case - such that $f'(a_j)$ is less than $l \cdot q'$. Without loss of generality, let a_j be a type-x element with $f'(a_j) = \delta$. Now if element a_j is distinguished, we are done since from the affined nature of sequence S and Lemma 3.2 we know that element a_j has a predecessor element $a_{j'}$ with its last

copy in edge $E'_{q''}$ with $q'' < q'$ such that $f'(a_{j'})$ is less than $l \cdot q''$, which contradicts the minimality of q'.

On the other hand if element a_j is not distinguished, consider sequence $\hat{S}$ projected from sequence S with respect to δ and x. Clearly, $\hat{S}$ has a primary saturation subsequence of $\gamma \geq 1$ edges $\langle E''_1 \ldots E''_\gamma \rangle$. Since sequence S is a weak k-matching and therefore proper, the first copy of element a_j must be in E'_r where $r = (q' - k + 1)$ and $1 \leq r \leq \gamma$. Now, since element a_j with its first copy in edge E'_r has a modified cost which is greater than zero, it follows that the first copy of any element $a_{j'}$ in any equivalent sequence S' with zero modified cost, must be in some edge $E'_{r'}$ where r' is less than r, whenever $f'(a_{j'})$ is no greater than $f'(a_j)$.

Now, if $\gamma = 1$ we are done since no sequence S' can have an edge $E_{r'}$ with $r' < 1$. On the other hand, if $\gamma > 1$, since each of the first $(\gamma - 1)$ edges in the primary saturated subsequence of sequence $\hat{S}$ is saturated, there are at least $\mathbf{r} = \{(\gamma - 1)\, SI_x + 1\}$ type-x elements in sequence S (or in any sequence equivalent to it) whose modified cost-determining functions are bound above by δ. Therefore, if there is a sequence S' with zero modified cost, then the first copy of each of these $\mathbf{r}$ elements must be in some edge edge $E'_{r'}$ for $r' < r$. Then, sequence $\hat{S}'$ projected from sequence S' with respect to δ and x has at least one edge $E''_{r'}$ with more than SI_x elements which contradicts Lemma 3.1 completing the proof. □

This completes our current goal for, given any feasible set $\mathbf{S}$, we have a characterization of its subset $\mathbf{S}_\emptyset$ whenever this subset is non-empty through the above property of the set of affined sequences $\mathbf{S}_\alpha$; the subset $\mathbf{S}_\alpha$ is itself characterized by means of identifying the special attributes of affined se-

quences. Therefore, our partial characterization of $\mathbf{S}_\emptyset$ - as stated in Theorem 3.1 whenever this subset is not empty or equivalently when the feasible set in question has member sequences with zero cost - is in fact accomplished through the attributes of affined sequences. This allows to claim at this point to have partially accomplished our current goal of characterizing the behavior of minimum cost sequences. In the next chapter, we will extend our present characterization to accomplish this goal in its entirety by considering the more general case of feasible sets in which the minimum cost sequences have greater than zero cost.

Chapter 4

On Relating Affined and Optimum Sequences

In this chapter, we provide a complete characterization of the behavior of sequences at optimality by building on the results of Chapter 3. To this end, we introduce a simple yet powerful proof technique in Section 4.1; powerful because we will see in Sections 4.2 and 4.3 that it can be used to establish the optimality of the rich class of affined sequences. In Section 4.2, we establish the surprising fact that the affined nature of a sequence is preserved under a certain kind of perturbation or transformation called 'shifting.' This property of affined sequences rests heavily on the Shifting Lemma (Lemma 4.2) which gives us extremely interesting and counterintuitive insight into the behavior of the modified cost determining function. Finally, in Section 4.3, all our earlier results from Chapters 2, 3, and 4, including our techniques from Section 4.1, culminate in the Characterization Theorem (Theorem 4.1) which is perhaps our most significant result. In this theorem, we will conclusively establish the important fact that affined sequences are always optimum (in the sense of having minimum cost). This property of affined sequences will enable us to capture the essence of the optimal behavior of sequences associated with a wide class of instances, in the subsequent chapters of this dissertation.

4.1 A Technique for Proving the Optimality of Affined Sequences

A couple of assumptions are implicit in our claim made in connection with Theorem 3.1 that the subset $\mathbf{S}_\alpha$ of a feasible set $\mathbf{S}$ provides us with a characterization of minimum cost sequences. Firstly, the given instance must have a corresponding affined sequence or else $\mathbf{S}_\alpha$ would be empty. If this were to be the case, that is if $\mathbf{S}_\alpha$ is empty for some feasible set $\mathbf{S}$, the consequence of the inclusion relationship between $\mathbf{S}_\alpha$ and $\mathbf{S}_\emptyset$ from Theorem 3.1 would be trivial when viewed from the standpoint of characterizing the behavior of the minimum cost sequences from $\mathbf{S}$, through its affined sequences. This situation does not however pose any problems since we will see in the next couple of sections that instances drawn from a variety of subdomains - specifically those from the subdomains listed in the tables of Chapter 1 and their generalizations - all have affined sequences at optimality.

Thus, even though this question regarding the existence of affined sequences is not of any concern, we are still faced with the potentially problematic situation that the given instance might not have any associated zero-cost sequences. If this were to be the case, any sequence from the given feasible set, whether it is affined or not, would have greater than zero cost. In this situation, Theorem 3.1 fails to provide us with a characterization of the behavior of minimum cost sequences even if the problem instance were to have affined sequences corresponding to it, since it is not immediately clear if affined sequences are also optimum even when the cost of any sequence for the given instance is greater than zero; in other words, we do not know if an analog of the inclusion relationship from Theorem 3.1 holds between the affined and the minimum cost sequences from the feasible set $\mathbf{S}$ in question, even when it does not have any members with a cost of zero.

4.1.1 The Proof Technique

Given that we wish to devise a technique through which we can build on the partial characterization provided by Theorem 3.1, we can initially assume that the inclusion relationship stated in this theorem between the subsets $\mathbf{S}_\alpha$ and $\mathbf{S}_\emptyset$ of a feasible set $\mathbf{S}$ is already know to us. Then, since $\mathbf{S}_{min} = \mathbf{S}_\emptyset$ whenever the feasible set $\mathbf{S}$ has at least one member sequence with a cost of zero, we can claim from Theorem 3.1 that in this situation :

$$(*) \quad \mathbf{S}_\alpha \subseteq \mathbf{S}_\emptyset = \mathbf{S}_{min}.$$

Now, let us extend our view to the more general and interesting situation where the feasible set $\mathbf{S}$ might not have any member sequences with zero cost. In particular, let us only consider such feasible sets $\mathbf{S}$ in which the cost of a sequence in $\mathbf{S}_{min}$ is always strictly greater than zero.

This brings us to the most crucial step of our technique where given a feasible set $\mathbf{S}$, we will *suppose* (hypothetically for now) that a 'corresponding' feasible set $\overline{\mathbf{S}}$ *exists* (for a related instance) in such a way that $\overline{\mathbf{S}}$ has member sequences with zero cost or equivalently, that $\overline{\mathbf{S}}_\emptyset$ is non-empty. In the same vein, let us also *suppose* that feasible sets $\mathbf{S}$ and $\overline{\mathbf{S}}$ are related through a total function or mapping '$\rightarrow$' which is defined from feasible set $\mathbf{S}$ to feasible set $\overline{\mathbf{S}}$; we will write $S(\in \mathbf{S}) \rightarrow \overline{S} \in (\overline{\mathbf{S}})$ to denote that sequence $\overline{S}$ is the image of sequence S under (the mapping) '$\rightarrow$'. For our technique to work, this mapping '$\rightarrow$' has to satisfy two specific properties which we will now proceed to describe.

The first of these two properties which this mapping has to satisfy is that if sequence $\overline{S}$ is in $\overline{\mathbf{S}}_{min}$ ($=\overline{\mathbf{S}}_\emptyset$) in feasible set $\overline{\mathbf{S}}$, then any sequence S of

which sequence $\overline{S}$ is an image under the mapping ' $\rightarrow$ ' must always be in the subset $\mathbf{S}_{min}$ of feasible set $\mathbf{S}$; let (**) be used to denote this property or condition which is illustrated in Figure 4-1.

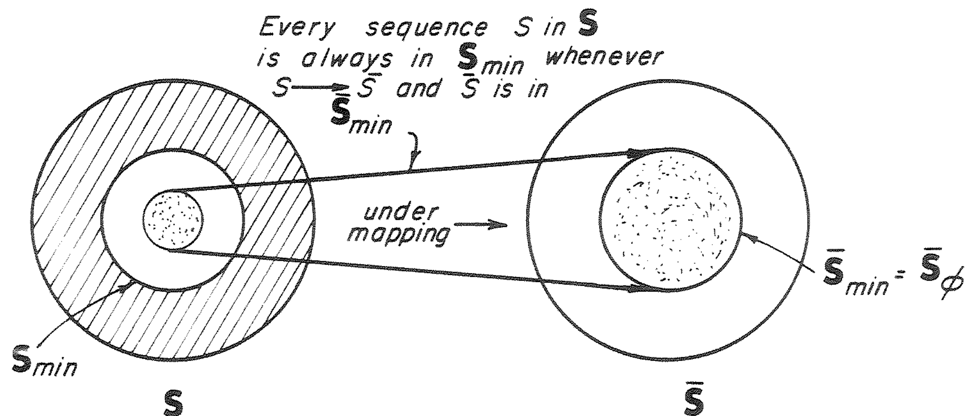


Figure 4-1: Property (**) satisfied by the mapping ' $\rightarrow$ '.

In addition to condition (**), we require that the mapping ' $\rightarrow$ ' satisfy one additional property which requires that the affined nature of a sequence be preserved under this mapping as shown in Figure 4-2, or equivalently

$$(***) \quad S \in \mathbf{S}_{\alpha} \Rightarrow \overline{S} \in \overline{\mathbf{S}}_{\alpha}$$

where $\Rightarrow$ represents logical implication. This condition is really quite interesting since if we consider the universe $\mathcal{S}$ of *all* sequences, which is essentially the union of all possible feasible sets, then condition (***) essentially requires that $\mathcal{S}_{\alpha}$ (which is the set of all possible affined sequences from $\mathcal{S}$) be *closed* under the mapping or transformation ' $\rightarrow$ ', or equivalently, condition (***) requires that a special subset $\mathcal{S}_{\alpha}$ of the universe of all sequences $\mathcal{S}$ satisfy a special closure property.

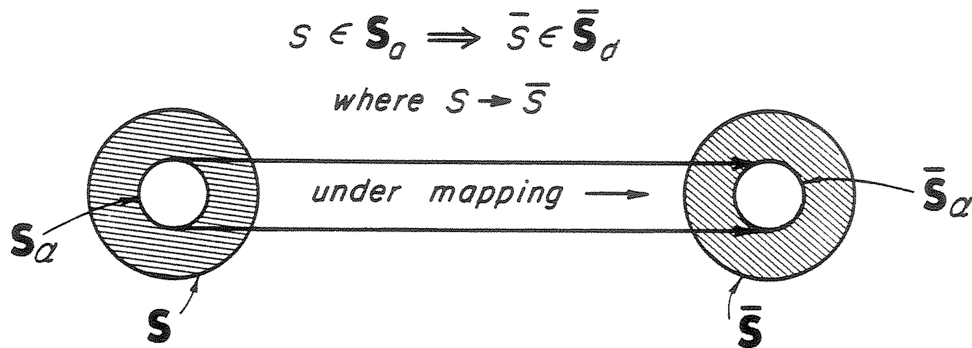


Figure 4-2: Property (***) satisfied by the mapping '$\rightarrow$'.

At this point, we have identified all the salient components and attributes which we need in order to start using our proof technique. Recall that our intention is to use this technique to show that the subset $\mathbf{S}_\alpha$ of a feasible set $\mathbf{S}$ is always included in $\mathbf{S}_{min}$, thereby establishing that affined sequences are always optimum for the underlying instance. Now, if the feasible set $\mathbf{S}$ has member sequences with zero cost, we are trivially done from condition (*); a fact which we have already established in Theorem 3.1. On the other hand, suppose that the feasible set $\mathbf{S}$ does not have any member sequences with zero cost. Now, if the desired inclusion relationship does not hold in this case between its subsets $\mathbf{S}_\alpha$ and $\mathbf{S}_{min}$, then

$$\mathbf{S}_\alpha \subsetneq \mathbf{S}_{min}.$$

In such a situation, we have a feasible set $\mathbf{S}$ with a non-empty subset $\mathbf{S}_\alpha$ which has an affined sequence S' as its member and moreover, since this sequence does not have minimum cost,

$$S' \in (\mathbf{S}_\alpha - \mathbf{S}_{min}).$$

We will now start invoking the various conditions and attributes which we developed earlier on in this section to show that this cannot be true.

As a first step, let us consider the 'corresponding' feasible set $\overline{\mathbf{S}}$ -

recall that we had hypothesized its existence earlier on - with a non-empty subset $\overline{\mathbf{S}}_{\emptyset}$. Also, feasible sets $\mathbf{S}$ and $\overline{\mathbf{S}}$ are related through the mapping '$\rightarrow$' and in particular, $S' \rightarrow \overline{S}'$. Then, from condition (**) which is satisfied by the mapping '$\rightarrow$' and the fact that this mapping is total, we can deduce that sequence $\overline{S}'$ in feasible set $\overline{\mathbf{S}}$ is such that,

$$\overline{S}' \notin \overline{\mathbf{S}}_{min} = \overline{\mathbf{S}}_{\emptyset}.$$

Also, from condition (***), we know that sequence $\overline{S}'$ is affined. Then we have a feasible set $\overline{\mathbf{S}}$ with an affined sequence $\overline{S}'$ as its member, and sequence $\overline{S}'$ does not have a cost of zero even though $\overline{\mathbf{S}}_{\emptyset}$ is not empty; this contradicts condition (*) which we have already established (Theorem 3.1) and as a consequence, we are done.

Note however that if we have to apply the above line of argument to extend the inclusion relationship stated in condition (*), to show that it holds even in the more general case when the feasible set $\mathbf{S}$ in question does not have any member sequences with zero cost - stated another way, if we wish to show through the above mentioned proof technique that affined sequences are always optimum for the given instance - the validity of this technique is conditioned on the (hypothesized) *existence* of a corresponding feasible set $\overline{\mathbf{S}}$ and a mapping '$\rightarrow$' which satisfies certain special attributes. As we will demonstrate in the next couple of sections, these 'existence questions' will not pose any problems in our case since we will be able to show that for any feasible set $\mathbf{S}$ of our optimization problem, a corresponding feasible set $\overline{\mathbf{S}}$, and a mapping with the desired attributes, both exist.

The principal advantage of this technique is that it aids us in simplifying the general problem of characterizing the behavior of minimum cost sequences, by allowing us to first tackle the relatively simpler and more specific problem of characterizing the optimal behavior of sequences at a single point in the range of the cost function. Affined sequences provided us with this specialized characterization, as stated in Theorem 3.1 and reiterated as condition (*) in this chapter; here, this characterization was shown to be valid at the single point in the range of the cost function which corresponds to an (optimum) cost of zero. Then, our technique gives us a definite series of steps through which we can extend this specialized characterization to a more complete one which encompasses all points in the range of the cost function. Essentially, given condition (*), the other components and attributes of our technique such as the mapping '$\rightarrow$', the properties or conditions (**) and (***) which it satisfies and so on, collectively aid us in deducing the following:

$$\overline{\mathbf{S}}_{\alpha} \subseteq \overline{\mathbf{S}}_{min} = \overline{\mathbf{S}}_{\emptyset} \Rightarrow \mathbf{S}_{\alpha} \subseteq \mathbf{S}_{min}.$$

In this way, our technique provides us with a mechanism through which we can *reduce* the general problem of showing that affined sequences are optimum, even when the underlying problem instances do not have any associated zero-cost sequences, to the relatively simpler problem of establishing a specialized version of this fact (which has already been established during the first step in Theorem 3.1 and) which is valid only in the event that the instances in question have associated zero-cost sequences.

While we have been describing this technique in the specific context of the optimization problem associated with hypergraphs, it actually seems to be applicable in the broader context of combinatorial optimization problems in

general, beyond the specific problem with which we are concerned. In this more general scenario, what we need to identify as a first step is an appropriate property or collection of properties $\mathcal{P}$ (the affined property served this purpose in our case) which characterize optimal members of the appropriate feasible sets at some specific point (the point associated with zero cost in our case), or perhaps a 'few' points in the range of the corresponding cost function.

Then, in order to extend this specialized characterization provided by the collection of properties $\mathcal{P}$ to include all the points in the range of the cost function in question, we need to ensure that the analog of our requirements which include the existence of a 'corresponding' feasible set, and a mapping which satisfies certain special attributes (respectively the feasible set $\overline{\mathbf{S}}$, the mapping $\rightarrow$, and conditions (**) and (***)) are satisfied.

In this connection, a point worth recognizing is that all of the above mentioned 'requirements' and 'attributes' have obvious analogs in the context of any combinatorial optimization problem. Also, it is important to note that the line of reasoning - which, based on these requirements and attributes, will allow us, subsequently in this chapter, to *extend* the partial characterization of minimum cost sequences from Theorem 3.1 - is really quite independent of the fact that the feasible sets with which we are concerned are in fact sets of sequences. As such, it (this line of reasoning) can be applied consistently even if the feasible sets being considered are associated with some other combinatorial optimization problem. This in turn means that our technique is applicable in the more general context of characterizing the optimal behavior of solutions of a a wider class of optimization problems, so long as certain basic criteria

(which we have identified in the context of our problem) are met. Let us now proceed on to the issue of applying this proof technique to show that affined sequences are always optimum.

4.2 Perturbing Sequences Through Shifting

We have already seen in the previous section that in order to apply our basic proof technique, we need to ensure that certain basic criteria are met. To demonstrate that this is indeed true of our optimization problem, we will now introduce a mechanism through which instances and their feasible sets can be 'perturbed' or transformed. Given an arbitrary feasible set $\mathbf{S}$ such that any sequence in $\mathbf{S}_{min}$ has a cost of $K \geq 0$, we can essentially perturb the corresponding instance I, where intuitively this perturbation involves keeping everything about instance I the same except that we alter the values assigned by the cost determining function to each of its elements a_i, by shifting them all up uniformly by X or equivalently, by adding X to each of these values. It is easy to see that given any instance I and a $X \geq 0$, a corresponding perturbed instance $\overline{I}$ always exists.

In order to make this notion precise, we need to define the following. Given an instance I with precedence graph $P = \langle A, \alpha \rangle$, class function $\mathbf{C}$ and bound vector $\mathbf{B}$, we say that another instance say $\overline{I}$ with precedence graph $\overline{P} = \langle \overline{A}, \bar{\alpha} \rangle$, class function $\overline{\mathbf{C}}$ and bound vector $\overline{\mathbf{B}}$ is *isomorphic to it* if and only if there exists a (total) function $F: A \rightarrow \overline{A}$, which is also a bijection such that:

1. $\langle a_i, a_j \rangle \in \alpha$ if and only if $\langle F(a_i), F(a_j) \rangle \in \bar{\alpha}$,

2. $\mathbf{B} \underset{v}{=} \overline{\mathbf{B}}$ and,

3. $\mathbf{C}(a_i) \underset{v}{=} \overline{\mathbf{C}}(F(a_i))$.

Essentially, in a pair of instances I and $\overline{I}$ which are isomorphic to each other, the respective precedence graphs are isomorphic in the usual graph theoretic sense. In addition, the bound vectors of the two instances are equal and the two class functions are related (conditions 2 and 3 above).

Then, we say that an instance $\overline{I}$ is *isomorphic within shifting by X to* instance I provided instances I and $\overline{I}$ are isomorphic and for every element $a_i \in A$,

$$\overline{f}(F(a_i)) = f(a_i) + X$$

where f and $\overline{f}$ are respectively the cost-determining functions of instances I and $\overline{I}$. Viewed in the light of these definitions, perturbing an instance I by some quantity X creates another instance $\overline{I}$ which is isomorphic within shifting by X to the original instance I.

These perturbations also shift the corresponding feasible set in the same way in that for each sequence S in the original feasible set $\mathbf{S}$ corresponding to instance I, a sequence $\overline{S}$ always exists in feasible set $\overline{\mathbf{S}}$ of instance $\overline{I}$ which is derived by perturbing instance I by some $X \geq 0$, in such a way that for any element $a_i \in A$: if the kth copy of element a_i is in edge E'_q in sequence S, then the last copy of element $F(a_i)$ from $\overline{A}$ is in edge $\overline{E}'_q$ and vice-versa as shown in Figure 4-3; if this is true of a pair of sequences S and $\overline{S}$, then we say that sequence $\overline{S}$ is *isomorphic within shifting by X to* sequence S.

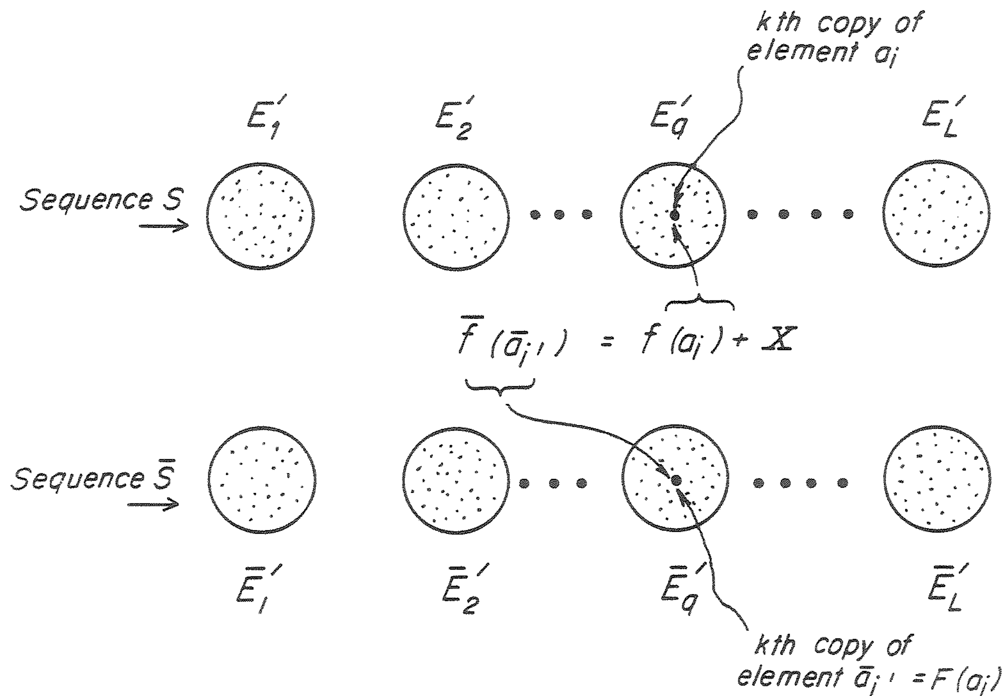


Figure 4-3: Sequence $\overline{S}$ is isomorphic within shifting by X to sequence S.

Basically, the relationship 'isomorphic within shifting by X to' relates pairs of sequences which are drawn from the feasible sets of a pair of instances which are in the same kind of relationship and in addition, elements from these two instances which are related through the mapping F are in edges which are the same distance (measured in terms of the number of edges) along in the respective sequences. We will now for the sake of convenience formalize this relationship between sequences through the binary relation $\mathcal{F}$ which, given a pair of feasible sets $\mathbf{S}$ and $\overline{\mathbf{S}}$, is a subset of $\mathbf{S} \times \overline{\mathbf{S}}$; a pair $\langle S, \overline{S} \rangle$ is in $\mathcal{F}$ if and only if sequence $\overline{S}$ from feasible set $\overline{\mathbf{S}}$, is isomorphic within shifting by X to sequence S from feasible set $\mathbf{S}$.

Let us now recall that in introducing this relationship $\mathcal{F}$ which formally captures the notion of pairs of instances and their sequences being related through a certain kind of perturbation of their respective cost-determining functions, we are motivated by the need to provide a mechanism which will allow us to use our proof technique from the previous section. This in turn will enable us to establish the fact that affined sequences are optimum, and that they can thereby provide us with the desired characterization of minimum cost sequences. In order to do this, we first need to establish that for any feasible set, a corresponding feasible set always exists in which the optimum sequences are guaranteed to have a cost of zero.

In this connection, it is easy and important to recognize that our perturbation method is such that there is exactly one sequence $\overline{S}$ in any feasible set $\overline{\mathbf{S}}$ which is isomorphic within shifting by X to a sequence S in feasible set $\mathbf{S}$, and for any $X \geq 0$. Also, any sequence S in feasible set $\mathbf{S}$ has exactly one sequence in feasible set $\overline{\mathbf{S}}$ of which it (sequence S) is 'isomorphic within shifting by X to'. As a consequence of this fact, the relation $\mathcal{F}$ is : *(i).* a total function and, *(ii).* a bijection from feasible set $\mathbf{S}$ to feasible set $\overline{\mathbf{S}}$. This allows us to write $\mathcal{F}(S) = \overline{S}$ in the usual notation to indicate that sequence $\overline{S}$ is the image of sequence S under the mapping $\mathcal{F}$. These points are illustrated in Figure 4-4.

Given these two facts about the function $\mathcal{F}$, it is easy to see that for any pair of feasible sets $\mathbf{S}$ and $\overline{\mathbf{S}}$:

Fact 1: Given any sequence S with a cost of K from feasible set $\mathbf{S}$, there exists a sequence $\overline{S}$ in feasible set $\overline{\mathbf{S}}$ such that $\mathcal{F}(S) = \overline{S}$.

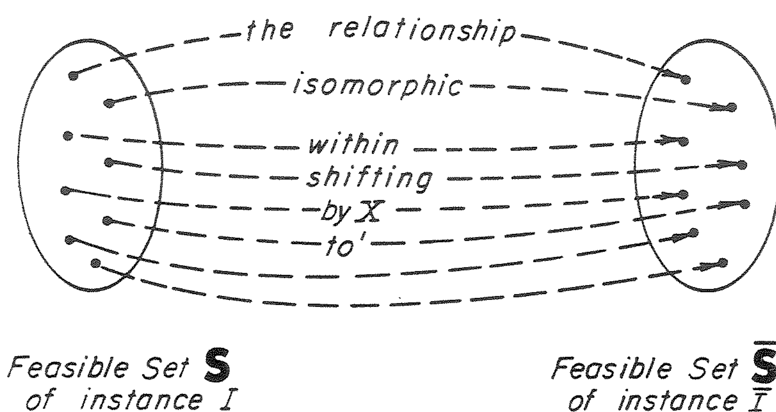


Figure 4-4: The relationship 'isomorphic within shifting by X to' captured by the mapping $\mathfrak{F}$.

Moreover, the cost of sequence $\overline{S}$ is exactly X less that that of sequence S for $0 \leq X \leq K$,

as shown in Figure 4-5.

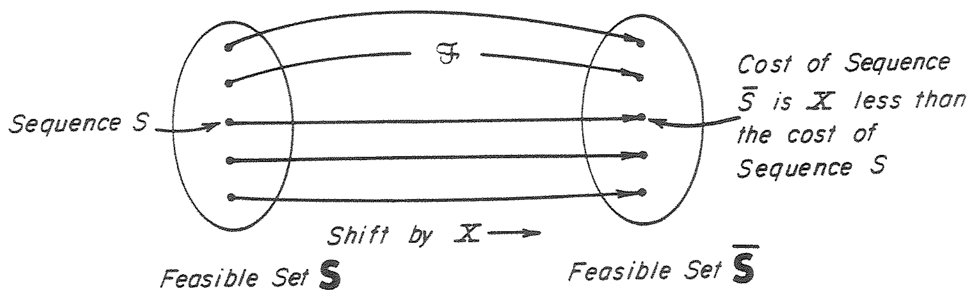


Figure 4-5: The effect of perturbation on cost.

In particular, Fact 1 tells us that given the feasible set **S** of instance I in which the minimum cost sequences have a cost of $K \geq 0$, the corresponding feasible set $\overline{\mathbf{S}}$ with an underlying instance $\overline{I}$, which in turn is isomorphic within shifting by K of instance I (clearly, such an instance always exists), will always have zero-cost member sequences. What this means of course is that

the first requirement of our proof technique is met, and in the rest of this chapter, we will be focussing our attention only on such feasible sets $\overline{\mathbf{S}}$ - those whose member sequences are isomorphic within shifting by K to the sequences in the given feasible set **S** whose minimum cost sequences have a cost of K.

Let us now address the next requirement of our proof technique namely, that of the existence of an appropriate mapping between feasible sets **S** and $\overline{\mathbf{S}}$ and which satisfies the analogs of conditions (**) and (***). It turns out that the mapping $\mathcal{F}$ meets all of these requirements rather well. To see that this function satisfies condition (**), we need to recollect that it is a bijection and that it is also total from **S** to $\overline{\mathbf{S}}$; this in conjunction with Fact 1 collectively imply that any minimum cost sequence in feasible set $\overline{\mathbf{S}}$ must always be the image under the mapping or perturbation $\mathcal{F}$ of a minimum cost sequence from feasible set **S**; this is equivalent to saying that the function $\mathcal{F}$ satisfies an analog of condition (**) from the previous section.

So far, we have been able to show that for any feasible set **S**, a corresponding set $\overline{\mathbf{S}}$ and a mapping $\mathcal{F}$ exist in such a manner as to satisfy all but one of the requirements of our proof technique from the previous section; in particular, we have not seen as yet that this mapping $\mathcal{F}$ satisfies condition (***) as well. Therefore, there is one more crucial issue yet to be resolved since we need to verify that the mapping $\mathcal{F}$ also satisfies condition (***). Recall that this condition requires that the affined nature of sequences be preserved under the perturbation which the mapping $\mathcal{F}$ represents. That is, given an affined sequence from a feasible set **S**, we need to show that its image under $\mathcal{F}$ in the corresponding feasible set $\overline{\mathbf{S}}$ is also affined. While this is in fact true of

the mapping $\mathcal{F}$, it is not as obvious, as was for example the fact that condition (**) is satisfied by it; actually, this is quite a counterintuitive fact and we will now proceed to establish it in the next section. In doing so, we will draw upon some extremely useful and interesting properties of affined sequences (to be established in Lemmas 4.1 and 4.2 in the next section) which will bear further testimony to the fact that they are indeed, a highly structured class of sequences.

4.3 Properties Preserved Under Shifting

In order to establish these properties of affined sequences, we will now introduce some additional notation, primarily to facilitate our present discussion. Firstly, given a pair of elements a_i and a_j from the set of elements A, which in turn is a part of instance I, we will denote their images under F in the set of elements $\overline{A}$ from instance $\overline{I}$ by $\bar{a}_{i'}$ and $\bar{a}_{j'}$ respectively. Also, the class Π_x of elements from the set A, and the class $\overline{\Pi}_y$ in the set of elements $\overline{A}$ which has *exactly* the images under F of all the elements from class Π_x, will be referred to as *corresponding* classes. We will generally use a 'bar' symbol to identify objects associated with an instance $\overline{I}$ and it is usually obvious as to what these objects represent; for example, $\overline{f}$ and $\overline{f}\,'$ respectively denote the cost-determining and modified cost-determining functions of instance $\overline{I}$. Also, let h_S denote the height of the precedence graph P corresponding to a sequence S and we will use d'' to denote $d' + X$ in what follows.

The first property which we will be establishing is the following conditional result which shows that any pair of elements a_i and $\bar{a}_{i'}$ from instances I and $\overline{I}$ have an equal number of successor elements which satisfy a given type

and modified cost-determining function constraints, as shown in Figure 4-6, or:

Lemma 4.1: *If sequence $\overline{S}$ is isomorphic within shifting by X to sequence S and if $\overline{f}'(\bar{a}_{i'}) = f'(a_i) + X$ for all elements a_i at a level no greater than λ for $0 \leq \lambda < h_s$, then for any d' and x and for any pair of elements a_j and $\bar{a}_{j'}$ at level $\lambda + 1$, $n(j,d',x)$ equals $\bar{n}(j',d'',y)$.*

Proof: Suppose that the lemma is not true. Then, a pair of sequences S and $\overline{S}$ with elements a_j and $\bar{a}_{j'}$ at level $(\lambda + 1) \geq 1$ in the respective precedence graphs, a δ, and a χ exist such that

$$n(i,\delta,\chi) = \omega \neq \bar{n}(j',(\delta + X),\psi) = \bar{\omega}$$

where Π_χ and $\overline{\Pi}_\psi$ are corresponding classes. Therefore, non-negative integers ω and $\bar{\omega}$ are such that either $\omega > \bar{\omega}$ or $\omega < \bar{\omega}$. Also, it must be the case that every element a_i at a level no greater than λ in precedence graph P is such that

$$\overline{f}'(\bar{a}_{i'}) = f'(a_i) + X.$$

Now, suppose that ω is greater than $\bar{\omega}$. Given that sequence $\overline{S}$ is isomorphic within shifting by X to sequence S, it follows that instance I and $\overline{I}$ are isomorphic to each other; from this fact, we can deduce that for every type-χ successor element which element a_j has, its (the type-χ successor element's) unique image under F is a type-ψ element and moreover, this element must be a successor of element $\bar{a}_{j'}$. This property in conjunction with the fact that $\omega > \bar{\omega}$ collectively imply that there exists at least one successor a' of element a_j in class Π_χ and at some level $\lambda' \leq \lambda$ such that

$$f'(a') \leq \delta$$

and,

$$\overline{f}'(F(a')) > \delta + X.$$

This contradicts the hypothesis that for all elements a_i at levels no greater than λ,

$$\overline{f}'(\bar{a}_{i'}) = f'(a_i) + X.$$

Now, if ω is less than $\bar{\omega}$, a similar argument can be used since in this case, we can show that element a_j must have a successor element a' at some level $\lambda' \leq \lambda$ such that,

$$f'(a') > \delta$$

and,

$$\overline{f}'(F(a')) \leq \delta + X$$

completing the proof. □

We will now use this result from Lemma 4.1 to establish the following extremely useful and interesting property of the modified cost-determining function; this property will play a central role in showing that the affined nature of a sequence is preserved under the special type of perturbation which we introduced earlier on in this chapter. Informally, this property states that if sequence $\overline{S}$ is isomorphic within shifting by X to sequence S, then the modified cost-determining function of (the instance $\overline{I}$ associated with) sequence $\overline{S}$ is perturbed in exactly the same way and more importantly, by the same amount X, by which the original cost-determining function was perturbed, as shown in Figure 4-7.

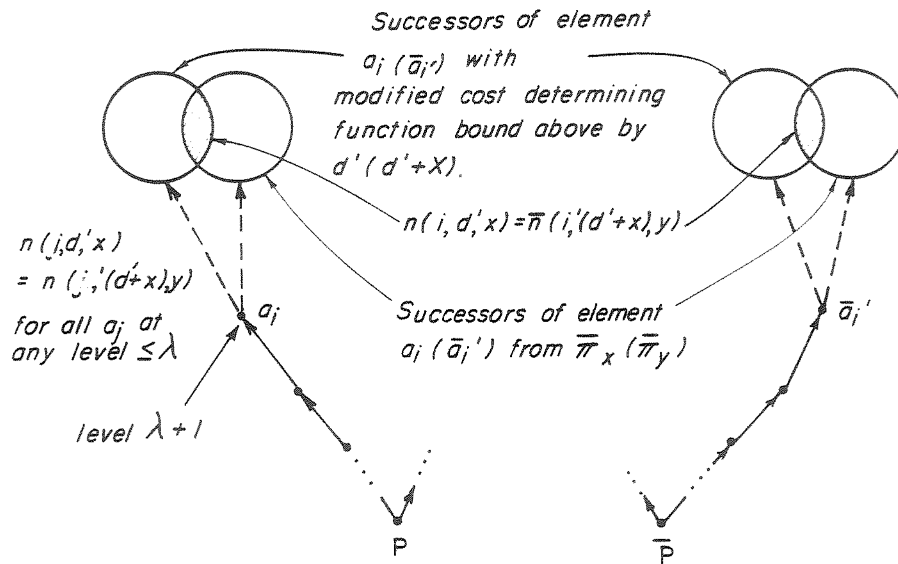


Figure 4-6: The relationship between $n(i,d',x)$ and $\bar{n}(i',d'',y)$ stated in Lemma 4.1.

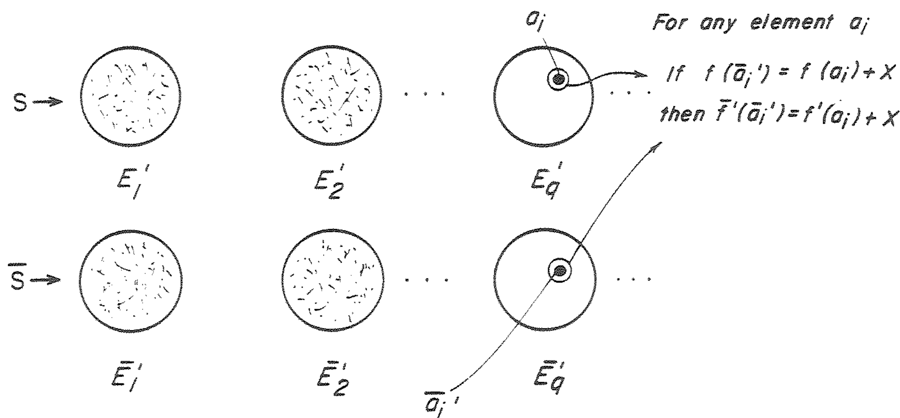


Figure 4-7: The property of sequences asserted in the Shifting Lemma.

More precisely, let us say that a sequence $\overline{S}$ is *isomorphic within shifting by X up to modification* to a sequence S whenever the corresponding instances I and $\overline{I}$ are isomorphic and for all elements a_i in A,

$$\overline{f}'(\bar{a}_{i'}) = f'(a_i) + X.$$

Then, surprisingly enough,

Lemma 4.2 (*The Shifting Lemma*): *If sequence $\overline{S}$ is isomorphic within shifting by X to sequence S, then it is also isomorphic within shifting by X up to modification to sequence S.*

Proof: We prove this Lemma by induction on the levels of precedence graph P.

For the base case, we simply need to recognize that

$$\overline{f}'(\bar{a}_{i'}) = f'(a_i) + X$$

for all elements a_i at level zero (the sink elements) in the precedence graph P since

$$f'(a_i) = f(a_i)$$

for these sink elements, and since we also know from the hypothesis of the lemma that sequence $\overline{S}$ is isomorphic within shifting by X to sequence S, which implies that

$$\overline{f}(\bar{a}_{i'}) = f(a_i) + X.$$

Let us now hypothesize that

$$\overline{f}'(\bar{a}_{i'}) = f'(a_i) + X$$

for any element a_i at a level no greater than λ, where $0 \leq \lambda < h_s$.

For the induction step, we will now show that

$$\overline{f}'(\bar{a}_{i'}) = f'(a_i) + X$$

for any element a_i at level $\lambda + 1$ in the precedence graph P. To see this, let us without loss of generality consider any element a_i at level $\lambda + 1$. Also, let

$$f^*(a_i) = \{\delta - l \cdot \Delta(i,\delta,\chi,k)\} = \Theta,$$

that is the expression

$$\{d' - l \cdot \Delta(i,d',x,k)\}$$

takes on a minimum for some $d' = \delta$ and $x = \chi$. Also, the corresponding function $\overline{f}^*$ for the instance $\overline{I}$ assigns a value to element $\bar{a}_{i'}$ as follows:

$$\overline{f}^*(\bar{a}_{i'}) = \{\overline{\delta} - l \cdot \overline{\Delta}(i',\overline{\delta},\psi,k)\} = \Theta'$$

for some $\overline{\delta} \in \overline{D}_{i'}$ and class $\overline{\Pi}_\psi$ in the set of elements $\overline{A}$.

We will first show that the function $\overline{f}^*$ is such that,

$$\Theta' = \Theta + X.$$

Suppose that this is not the case. Then, since Θ and Θ' are rationals, either

$$\Theta' < \Theta + X,$$

or

$$\Theta' > \Theta + X.$$

If

$$\Theta' < \Theta + X,$$

then

$$\{\overline{\delta} - l \cdot \overline{\Delta}(i',\overline{\delta},\psi,k)\} < \{\delta - l \cdot \Delta(i,\delta,\chi,k)\} + X$$

and therefore,

$$\{\overline{\delta} - X - l \cdot \overline{\Delta}(i',\overline{\delta},\psi,k)\} < \{\delta - l \cdot \Delta(i,\delta,\chi,k)\}.$$

Now, from Lemma 4.1 and the induction hypothesis, we know that

$$n(i,d',x) = \bar{n}(i',d'',y)$$

for any element a_i at level $\lambda + 1$, and for any d' and x; as usual, classes Π_x and $\overline{\Pi}_y$ are corresponding classes. From this fact and the definition of the function Δ, we can immediately deduce that for any $k > 0$,

$$\Delta\,(i,\delta',\chi',k) = \overline{\Delta}\,(i',\overline{\delta},\psi,k)$$

where $\delta' = \overline{\delta} - X$, and class $\Pi_{\chi'}$ from the set of elements A and class $\overline{\Pi}_{\psi}$ from the set of elements $\overline{A}$ are corresponding classes. Then, this fact in conjunction with our earlier deduction that

$$\{\overline{\delta} - X - l\cdot\overline{\Delta}\,(i',\overline{\delta},\psi,k)\} < \{\delta - l\cdot\Delta\,(i,\delta,\chi,k)\}$$

collectively imply that there exists a $\delta' \in D_i$ and a class $\Pi_{\chi'}$ where either $\delta \neq \delta'$, or $\chi \neq \chi'$, or both, such that

$$\{\delta' - l\cdot\Delta\,(i,\delta',\chi',k)\} < \{\delta - l\cdot\Delta\,(i,\delta,\chi,k)\} = \Theta$$

which contradicts the minimality of Θ.

A similar argument can be used to establish that Θ' cannot be greater than $\Theta + X$ and therefore, given that Θ and Θ' are rationals, we conclude that

$$\overline{f}^{\,*}(\overline{a}_{i'}) = f^*(a_i) + X.$$

Now, from the definition of the modified cost-determining function, we know that

$$\overline{f}'(\overline{a}_{i'}) = min\{\overline{f}\,(\overline{a}_{i'})\,,\,\overline{f}^{\,*}(\overline{a}_{i'})\}.$$

Then, from our earlier deductions and the fact that sequence $\overline{S}$ is isomorphic within shifting by X to sequence S we have:

$$\overline{f}\,(\overline{a}_{i'}) = min\{f\,(a_i) + X,\, f^*(a_i) + X\},$$

and since

$$f'(a_i) = min\{f\,(a_i),\, f^*(a_i)\,\},$$

we have

$$\overline{f}'(\bar{a}_{i'}) = f'(a_i) + X$$

completing the proof. □

We are now in a position to use these two lemmas to show that affined sequences are very robust indeed, in that the method of perturbing sequences which we introduced earlier on does not destroy their essential structure or,

Lemma 4.3: *A sequence* $\overline{S}$ which is *isomorphic within shifting by* X *to an affined sequence* S*, is also affined.*

Proof: We will first show that if any element a_i is not distinguished in sequence S, then element $\bar{a}_{i'}$ is not distinguished in sequence $\overline{S}$ and vice-versa. Since sequence $\overline{S}$ is isomorphic within shifting by X to sequence S, if any element a_i has its first copy in edge E'_q in S, then element $\bar{a}_{i'}$ has its first copy in edge $\overline{E}'_q$ in sequence $\overline{S}$. Let element a_i be a type-x element and let $f'(a_i) = \delta$ without loss of generality. Now, if element a_i is not distinguished in sequence S, then it occurs in the primary saturation subsequence of $\hat{S}$ projected with respect to δ and x from sequence S.

Suppose that element $\bar{a}_{i'}$ is distinguished in sequence $\overline{S}$. Then, it cannot be in the primary saturation subsequence in sequence S^* projected with respect to $\overline{\delta}$ and y from sequence $\overline{S}$, where $\overline{f}'(\bar{a}_{i'}) = \overline{\delta}$; also, from the Shifting Lemma, $\overline{\delta} = \delta + X$. Then, if element $\bar{a}_{i'}$ is distinguished and if element a_i is not, it follows from the definition of a projected sequence that at

least one of the elements a_j in an edge in the primary saturated subsequence in $\hat{S}$ is such that element $\bar{a}_{j'}$ is not in any edge of S^*. Then, since $a_j \in \Pi_x$ implies that $\bar{a}_{j'} \in \overline{\Pi}_y$, it must be the case that

$$\overline{f}'(\bar{a}_{j'}) > \overline{\delta}$$

or

$$\overline{f}'(\bar{a}_{j'}) = f(a_j) + X'$$

for some $X' > X$ which contradicts Lemma 4.2. We can use a similar argument to show that if any element $\bar{a}_{i'}$ is not distinguished in sequence $\overline{S}$, then element a_i is not distinguished in sequence S and hence conclude that if element a_i is distinguished in sequence S, it follows that element $\bar{a}_{i'}$ is distinguished in sequence $\overline{S}$ and vice-versa.

Now, consider any distinguished type-x element $\bar{a}_{i'}$ with $\overline{f}'(\bar{a}_{i'}) = \overline{\delta}$, and with its first copy in edge $\overline{E}'_q$ in sequence $\overline{S}$. From the previous paragraph, we know that the type-x element a_i with $f'(a_i) = \delta$, and with its first copy in edge E'_q in sequence S, is also distinguished. Also, from the affinedness of sequence S, it follows that element a_i has a predecessor element a_j with its first copy in edge $E'_{q'}$ such that

$$\Delta\,(j,\delta,x,k) \geq q - q'.$$

Then, from Lemmas 4.1 and 4.2, we know that the critical distance of element $\bar{a}_{j'}$ with respect to $\overline{\delta}$ and y,

$$\begin{aligned}\overline{\Delta}(j',\overline{\delta},y,k) &= \Delta(i,\delta,x,k)\\ &\geq q - q'.\end{aligned}$$

where element $\bar{a}_{j'}$ is the image of element a_j under the mapping F.

Also, from the isomorphism of instances I and $\overline{I}$, we know that element $\bar{a}_{j'}$ has its first copy in edge $\overline{E}_{q'}$ in sequence $\overline{S}$, and that element $\bar{a}_{j'}$ is a predecessor of element $\bar{a}_{i'}$ in precedence graph $\overline{P}$; from Lemma 4.2, we know that

$$\overline{f}\,'(\bar{a}_{i'}) = \overline{\delta} = \delta + X.$$

So, element $\bar{a}_{i'}$ has a predecessor element $\bar{a}_{j'}$ such that the distance between these two elements in sequence $\overline{S}$ is bound above by the critical distance $\overline{\Delta}$ of element $\bar{a}_{j'}$ with respect to $\overline{\delta}$ and y and hence, sequence $\overline{S}$ is affined. □

4.4 On Affined and Optimum Sequences

Since Lemma 4.3 tells us that the mapping $\mathcal{F}$ satisfies condition (***) as well, we have satisfied all the requirements of our proof technique. This allows us to claim that affined sequences are always optimum; in other words, they characterize the behavior of minimum cost sequences at all points in the range of the cost function of our optimization problem, and not just when the given instance has associated zero-cost sequences. We are therefore in a position to completely characterize the behavior of minimum cost sequences in terms of affined sequences and thereby extend the partial characterization of minimum cost sequences provided by Theorem 3.1. This characterization of minimum cost sequences, provided by the inclusion relationship between the subset of affined sequences and the minimum cost sequences in a feasible set **S** shown in Figure 4-8, is stated in the following central theorem. This theorem basically follows from the fact that affined sequences are always optimum - a fact which we are already aware of - but we include its proof here merely for the sake of completeness.

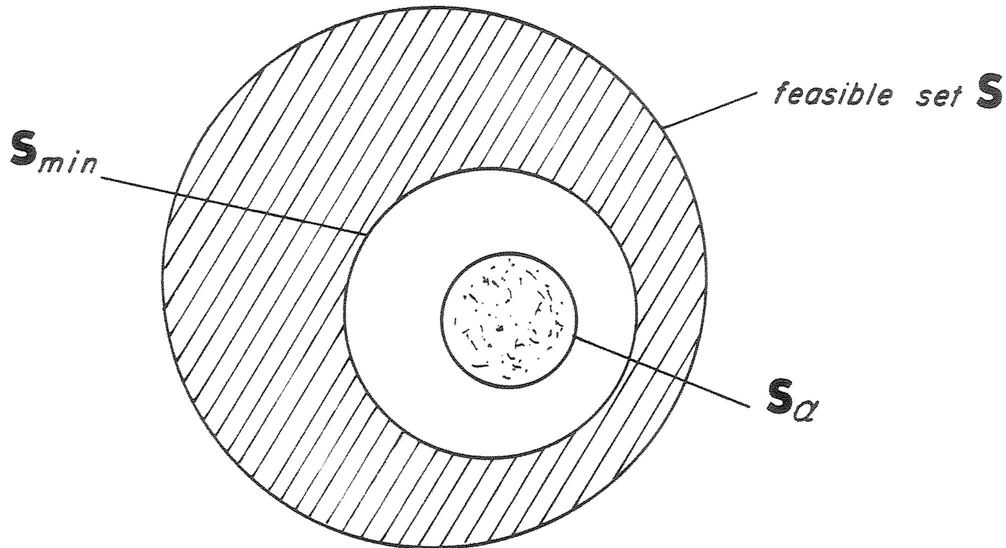


Figure 4-8: The relationship between $\mathbf{S}_\alpha$ and $\mathbf{S}_{min}$ in an arbitrary feasible set $\mathbf{S}$.

Theorem 4.1 (*The Characterization Theorem*): *For any feasible set* $\mathbf{S}$, $\mathbf{S}_\alpha \subseteq \mathbf{S}_{\min}$.

Proof: The proof is essentially an application of our technique from Section 4.1. Suppose that this theorem is false. Then, the sequences in $\mathbf{S}_{min}$ must have a cost $K' > 0$ or else we will be contradicting Theorem 3.1. Also, there must exists a sequence S in $\mathbf{S}_\alpha$ such that $S \notin \mathbf{S}_{min}$. Now, consider a sequence $T \in \mathbf{S}_{min}$ which is equivalent to sequence S, and let K and K' be the costs associated with sequences S and T respectively where $0 < K' < K$. Now, consider sequence $\overline{S}$ ($\overline{T}$) which is isomorphic within shifting by X to sequence S (T) such that:

$$\overline{f}\,(\bar{a}_{i'}) = f\,(a_i) + K'\,(\bar{g}\,(\overline{b}_{j'}) = g\,(b_j) + K')$$

(where g is the cost-determining function of the instance associated with se-

quence T and the b_j are its constituent elements). Then, sequences $\overline{S}$ and $\overline{T}$ have costs of $K - K' > 0$ and zero respectively. With this, we have shown that a sequence $\overline{S}$ exists which, from Lemma 4.3 is affined, and for which $|\overline{\mathbf{S}}_{\emptyset}|$ is non-empty (because $\overline{T} \in \overline{\mathbf{S}}_{\emptyset}$) and yet $\overline{S} \notin \overline{\mathbf{S}}_{\emptyset}$ which contradicts Theorem 3.1 completing the proof. □

The essence of our proof technique was originally used implicitly by Brucker, Garey and Johnson [2] in the context of finding minimum tardiness schedules for the restricted case of in-tree precedence graphs with a *single* type of *single-stage* resource (shown in the first row of Table 1-2). Besides our characterization of minimum cost sequences and other results, one of our main contributions has been in systematizing and generalizing this technique. We have also demonstrated the utility and scope of this technique by applying it in the context of the rich class of affined sequences; through this, we have been able to obtain a characterization of minimum cost sequences through the affined property. Finally, we are optimistic that this technique will be effective in providing similar characterizations of a wider class of scheduling problems and other types of combinatorial optimization problems as well.

Chapter 5

Obstructions and Permutation Lists

While we have been studying the behavior of sequences at optimality in the last few chapters, we have not addressed the issue, of the complexity of constructive methods for enumerating sequences from an instance of the optimization problem associated with hypergraphs. This approach (of ignoring the complexity of the enumeration question) was deliberate since we wished to study sequences independent of any explicit links to specific instances, or to algorithms for enumerating them. We now address this problem of enumerating sequences algorithmically in Section 5.1. First, we will describe an algorithm for enumerating sequences from instances of the optimization problem associated with hypergraphs. This algorithm plays a significant role in Section 5.2, where we identify a central condition which ensures polynomial solvability of our optimization problem. This condition or property will be used in Chapter 6 to accomplish our single most important goal; there, we will build a unifying framework around the subdomains from the tables of Chapter 1, and in some cases, their generalizations as well. To this end, in Section 5.3, we will establish some properties of sequences enumerated by our particular algorithmic method (from Section 5.1).

5.1 A Constructive Method for Generating Sequences

In this section, we will describe an algorithm for constructing sequences. In doing this, we are motivated by the fact that this algorithm can be used to formulate conditions which separate the tractable subdomains of the optimization problem associated with hypergraphs from the intractable ones; in order to better understand as to how an algorithm can play a role in such conditions, consider Figure 5-1 where he have shown the domain **D** of all instances of some optimization problem. Also, let us suppose that there is a constructively described algorithm **A** for solving this optimization problem, and which in addition has the following two special properties. Firstly, it always terminates in time proportional to a polynomial function of the input length for any instance from the input domain **D**. Secondly, it always solves the optimization problem for only a subset **S**$'$ of the input domain.

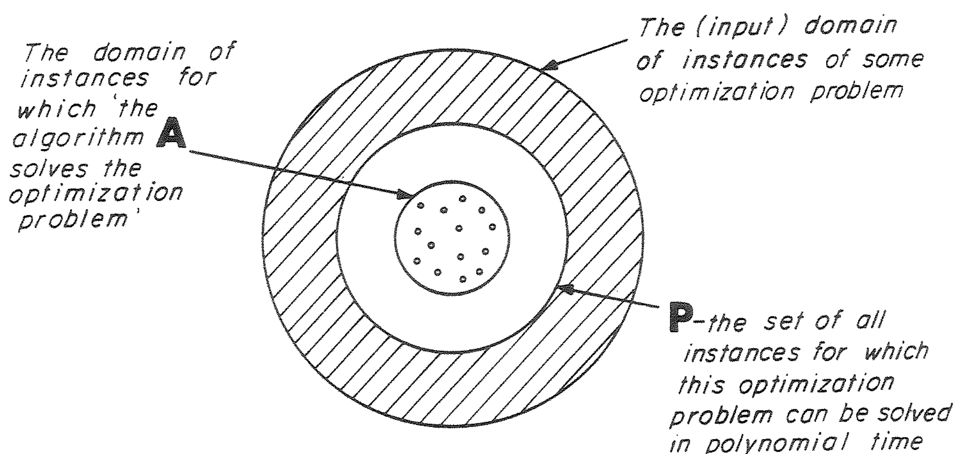


Figure 5-1: Identifying subdomains of an optimization problem through an algorithm **A**.

Let us for the sake of our present discussion, suppose that this optimization problem is NP-hard in general. Then, consider the subdomain D'

which is delineated in terms of algorithm **A** to be: *exactly that subset of the input domain for which algorithm* **A** *solves the optimization problem.* Given algorithm A and the second property which it satisfies, it is easy to see that this domain $\mathbf{D}'$ is well defined. Also, it follows from the first property of this algorithm that the optimization problem is polynomially solvable for instances from $\mathbf{D}'$. The obvious advantage of formulating complexity determining conditions in this fashion by basing them on an algorithm such as **A**, for separating polynomially solvable subdomains of an NP-hard input domain, is that in addition to identifying a polynomially solvable subdomain of the optimization problem being considered ($\mathbf{D}'$ in the above example), we would also have described an algorithm at the same time for solving the corresponding subproblem. We will see in this chapter and in Chapter 6 that this principle can in fact be used quite effectively in our present context so much so that the consequences have far reaching implications towards understanding the complexity of the optimization problem associated with hypergraphs.

5.1.1 The Generalized Schedule Construction Method

We will now describe a method of constructing sequences from instances of the optimization problem associated with hypergraphs. A list L which is a permutation $\{a'_1, a'_2, ..., a'_n\}$ of the elements in A plays an important role in formulating this method. Element a'_i *occurs before* (*after*) element a'_j in list L if and only if $i < j$ ($i > j$). In this manner, any list L totally orders the corresponding set of elements A; we will refer to this as the *list order* induced by L. Two lists for the instance of Figure 2-1 are illustrated in Figures 5-2 (i) and (ii) respectively.

We will refer to a list L together with the class and cost-determining functions, and the degree of slicing k, and the various other attributes of the

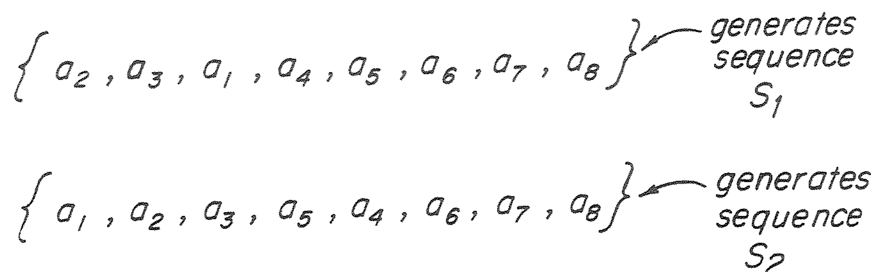


Figure 5-2: Alternate lists for the instance of Figure 2-1.

corresponding instance as the *representation with* list L (of that instance). Then, given a representation of an instance with list L, our method for constructing a sequence involves repeatedly 'scanning' list L. On the qth scan, we construct edge E'_q of the sequence (for $q \geq 1$) by selecting its constituent elements. We will say that scan q' is *earlier than* scan q whenever $q' < q$. Then, an element a'_i (from list L) is a *candidate for inclusion* on the qth scan (in edge E'_q that is) if and only if:

1. every predecessor of element a'_i, whenever it exists, is a member of at least k distinct edges constructed on earlier scans and,

2. there are no more than $k-1$ distinct edges which have been constructed on earlier scans, and which contain element a'_i.

Condition 1 ensures that the sequence constructed by our method is order preserving, while condition 2 constrains these sequences to be weak k-matchings.

We will now use these lists and the priority ordering which they induce on a set of elements, to guide our sequence construction method. For example, suppose that two elements a'_i and a'_j, where i is less than j, are both eligible (candidates) for inclusion in some edge E'_q as we are constructing the

sequence. But now, if the bound preserving property prevents us from including both of them in this edge - in other words, we are forced to choose between these two elements - then we will resolve this conflict by choosing the element with the higher property, that is one which occurs earlier in the permutation (element a'_i in this case).

To do this, we will now add an extra condition which will determine whether or not a candidate for inclusion is actually selected on the qth scan to be included in edge E'_q and define a function $Q(q, i)$ which represents the number of distinct edges which contain element a'_i and which have been constructed on scans earlier than q; also, $Q(1, i)$ is by definition zero for all i. Then, an element a'_i is *selected for inclusion* on the qth scan if and only if for each v where $1 \leq v \leq m$,

$$\sum_{j \in \mathbf{j}} C_{jv}$$

is bound above by B_v, where a $j \in \mathbf{j}$ if and only if element a'_j does not occur after element a'_i in list L and in addition, $Q(q, i) = Q(q, j)$. This condition influences the process of selecting from among all the contending elements which are candidates for inclusion on a given scan say q, since it simultaneously encodes the bound preserving requirement and the priority (order) induced by list L into our constructive method. In the interests of a simple and clear exposition, we have decided to define the conditions under which an element is a candidate or is selected for inclusion in a somewhat informal manner; there are obvious ways in which these notions can be stated more rigorously through recursion.

To see the manner in which these lists influence and determine the relative priorities of the elements of an instance, let us consider the example

lists L and L' in Figures 5-2 (i) and (ii) respectively. By altering the relative positions of elements a_1 and a_3 (or a_5 and a_4) in lists L and L', their relative positions in the constructed sequences respectively illustrated in Figure 2-8 (i) and (ii) are affected accordingly; note that even though these sequences were introduced earlier on in Chapter 2 without reference to any associated lists, it is easy to verify that they were in fact derived by applying our constructive method to the two lists in Figure 5-2. Specifically, in the former case, element a_3 (a_4) is given priority over element a_1 (a_5) if list L is used and these priorities are reversed in the case of the sequence of Figure 2-8 (ii) which is constructed from list L'. Note that it follows in an obvious way from the conditions which an element has to satisfy if it has to be a candidate for inclusion on a given scan and moreover, if it has to be selected for inclusion, that the resulting sequences are always interesting. Also, this procedure for constructing sequences terminates when there are no remaining candidates for inclusion in the corresponding list L.

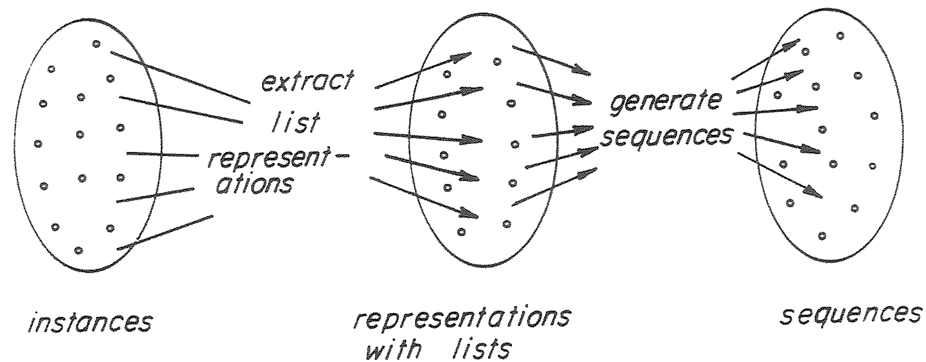


Figure 5-3: The two steps of the generalized sequence construction method.

Using this method, we will now specify a complete algorithm which will accept an instance of our optimization problem as its input and produce a

sequence as its output. Our algorithm or effective method for constructing sequences from instances, which will be referred to as the *Generalized Sequence Construction* (GSC) *Method*, is a two step process as illustrated in Figure 5-3 which involves:

1. extracting a representation with a list L from the given instance and,

2. applying the criteria described above for selecting the candidates for inclusion, to generate or construct a sequence from this (list) representation.

The generalized sequence construction method is a straightforward generalization of the well-known *list scheduling* method [14]; basically, we have extended list scheduling from [14] to where it can now accommodate many different types of resources as well as pipelining. The original list scheduling method was aimed at constructing sequences (schedules) from instances which only have a *single* type $(m = 1)$ of *single*-stage $(k = 1)$ resources.

5.2 A Sufficient Condition for the Optimization Problem to be Tractable

Let us now return to the issue of identifying unifying properties or conditions which determine the complexity of the optimization problem on hand, and in particular such properties which ensure that optimum sequences can be enumerated in polynomial time for instances satisfying these properties. As noted in Section 5.1, it will be especially advantageous if these unifying properties or conditions are based on constructive methods for enumerating sequences. For then, we would have produced an efficient algorithm for solving

the optimization problem hand in hand with providing a unifying complexity characterization through these properties.

It turns out that the GSC method by itself, without further refinement, is inadequate to provide such a unifying complexity characterization. To see this, we need to return to an earlier observation from Section 5.1 where we introduced the mechanism for identifying polynomially solvable subdomains based on constructively specified techniques or algorithms. In particular, recall that we expected the algorithm **A** to 'solve' the optimization problem, and moreover, it must do so in polynomial time for a subset $\mathbf{D}'$ of the input domain **D**. The GSC method fails to meet this requirement since it is *not* guaranteed to solve the optimization problem associated with hypergraphs for any non-trivial subdomain unless it is additionally constrained.

The problem lies in the first step of the GSC method (shown in Figure 5-3) which involves extracting 'a' (any) representation with (any) list L from the input instance. This allows us too much freedom during the first step, especially since not all of the $n!$ list representations associated with a given instance are guaranteed to produce optimum sequences; it is quite easy to find instances and list representations in such a way that the sequences constructed from them by subsequently applying step 2 of the GSC method, are *not* optimum. So, since the GSC method is not even guaranteed to always solve the optimization problem and therefore, it cannot yet play a role such as that played by the example algorithm **A** earlier on, in helping us identify complexity determining conditions.

To remedy this situation, let us start off by constraining the GSC method to where the resulting algorithm at least solves the optimization

problem. That is, given any input instance, our refined version of the GSC method should produce a corresponding minimum cost sequence as output. Naturally, we will focus our efforts on the first step of the GSC method since it is the source of our present problem. More specifically, we will modify this step by requiring that for any given input instance, the first step of our 'refined' method will produce a list representation in such a way that the sequence produced by a subsequent application of its second step - which is the same as the second step of the GSC method - is guaranteed to be optimum. By refining the GSC method in this fashion, we then have a new method for constructing sequences as shown in Figure 5-4, for solving the optimization problem associated with hypergraphs. Henceforth, we will refer to this as the *augmented generalized sequence construction* (AGSC) method.

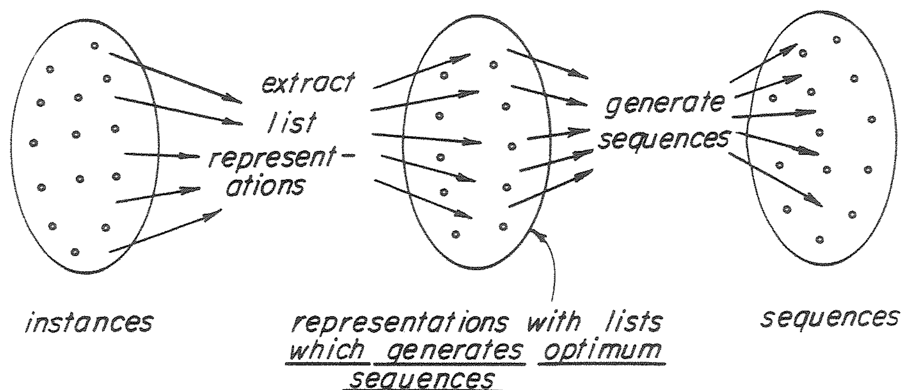


Figure 5-4: Refining the GSC method to ensure that the resulting method always solves the optimization problem.

However, even the AGSC method does not quite meet all the requirements which our hypothetical algorithm **A** satisfied. The problem with the GSC method was that we could not be assured of its ability to 'solve' our optimization problem. In contrast, the relatively refined AGSC method can always solve this optimization problem. But notice from Figure 5-4 that the

AGSC method is powerful enough to solve our optimization problem for any instance from its input domain. This in turn means that the scope of this AGSC method extends to the entire input domain of the optimization problem associated with hypergraphs. Now, since the general problem of finding optimum sequences is known to be NP-hard if we consider solving it for the entire input domain, we can conclude that there cannot be a polynomial time implementation of the AGSC method (unless P = NP) and therefore, it cannot be used in the manner in which algorithm **A** was used to separate a polynomially solvable subdomain of our example optimization problem with input domain **D**.

Given this shortcoming of the AGSC method, let us consider our options and especially, the possibility of refining it even further. In doing this, we are guided by the first of the two 'properties' which algorithms **A** satisfied : it always terminates in polynomial time. Specifically, it is this property which our AGSC method is violating, and which we need to enforce before we progress further.

Before we do this, let us first recognize the fact that if the time complexity of the AGSC method is to be kept low, we can once again restrict our attention to its first step. This is evident from the fact that its second step (shown in Figure 5-4) is always polynomially solvable and therefore, any difficulty in solving the problem through the AGSC method can be potentially encountered only during its first step. Equivalently, if the AGSC method is to run in polynomial time, the process of finding lists which are guaranteed to produce minimum cost sequences by subsequently applying step 2 of the AGSC method, must be 'easy' for the instances in question.

Clearly , if the first step of the refined version of the AGSC method runs in polynomial time for a given subdomain, we must be in a position to isolate such lists for a given instance from this subdomain which are guaranteed to produce optimum sequences, in an efficient manner. Then, what we really need to do is to identify what we can call *intractability obstructions*, which are conditions which constrain the search space of the first step of the AGSC method - if unconstrained, this search space potentially has all the $n!$ list representations of the instance - in such a way that a number of lists which generate suboptimal sequences can be ruled out, thereby converging upon one which does produce an optimum sequence.

Generally speaking, an intractability obstruction is a condition (or conditions), possibly formulated in terms of an algorithm for solving the optimization problem in question, in such a way that this optimization problem can be solved efficiently for instances which satisfy this obstruction. Actually, this notion of an intractability obstruction is rooted in that of an *obstruction* which Lenstra introduced in [63]. In Lenstra's formulation, an obstruction is a forbidden structure, such as the two well-known subgraphs $K_{3,3}$ and K_5 which were discovered by Kuratowski [58], and whose presence (up to homeomorphism) obstructs or rules out the possibility of the desired solution - graph planarity being the desired solution in this case. The important thing about obstructions such as these is that while they do signify deep and insightful work into the problem on hand, it is quite often the case that they cannot be easily related to complexity issues, or incorporated into a computational framework for solving this problem.

Therefore, while obstructions (in the Lenstra sense) can provide us with insight into the existence of good algorithms, the problem of finding ex-

plicit links between such obstructions and good algorithms is not always straightforward, and as such tends to be quite a challenging and fascinating endeavor in its own right. The problem of efficiently detecting graph planarity, and finding such an embedding whenever possible, serves to illustrate this point well. For this problem, Hopcroft and Tarjan [47] have discovered an extremely efficient algorithm which does not use Kuratowski's obstructions. Instead, it relies on a certain technique for decomposing the graph into paths, combined with efficient backtracking methods. The reason behind not using conditions such as $K_{3,3}$ or K_5 in this algorithm stems from the fact that these obstructions do not seem to lend themselves to efficient algorithmic implementation and moreover, they do not address the issue of constructively finding embeddings, even after we determine that the given graph is planar. This has motivated us to integrate the complexity issue explicitly into the notion of obstructions. As we will see in the sequel, by basing the formulation of intractability obstructions on algorithms, we will not only have an explicit characterization and thereby direct insight into the complexity (tractability) of the optimization problem on hand, but we would also have synthesized an algorithm at the same time for solving this problem.

Let us now proceed to identify such an intractability obstruction in the context of our problem, by defining a *non-decreasing ordered* list L as one in which $f'(a'_i) \leq f'(a'_j)$ whenever element a'_i occurs before element a'_j in list L. Also, a *representation* with a *non-decreasing ordered* list L for a given instance is defined accordingly. Our interest in such representations stems from the fact that a non-decreasing ordered list representation can always be constructed in polynomial time for a given instance; first by computing the modified cost determining function f' and subsequently sorting and constructing the list based on the values assigned by f' to the various elements, to get

the desired non-decreasing order. In this manner, non-decreasing ordered lists provide us with an easily computable representation of instances since we can claim that for any instance:

Proposition 5.1: *A representation with a non-decreasing ordered list L can be enumerated in polynomial time.*

These non-decreasing ordered lists play an important role in formulating our intractability obstruction. To see this, let us denote the (unique) sequence, constructed by applying the procedure which constitutes the second step of the AGSC (or for that matter, the GSC) method to a representation with list L, by S_L. We will say that sequence S_L is *generated* from the representation with list L. Now, suppose that there is a collection of instances in the input domain for which any non-decreasing ordered representation is guaranteed to produce an optimum sequence. Better yet, let us also suppose that the sequences generated from these representations are affined. Finally, let us change the first step of the AGSC method, as shown in Figure 5-5, by replacing it with a step which produces a non-decreasing ordered list representation of the input instance.

Then, it is easy to see that the resulting method (Figure 5-5) solves the optimization problem for the above mentioned subdomain of instances - those for which non-decreasing ordered list representations generate affined sequences. An interesting point to note is that the condition: 'all non-decreasing ordered representations generate affined sequences' , is formulated with the algorithm from Figure 5-5 as its basis. Finally, since both the steps of this algorithm can in fact be implemented to run in polynomial time (as we will see in Chapter 7), the optimization problem is actually polynomially solvable for the

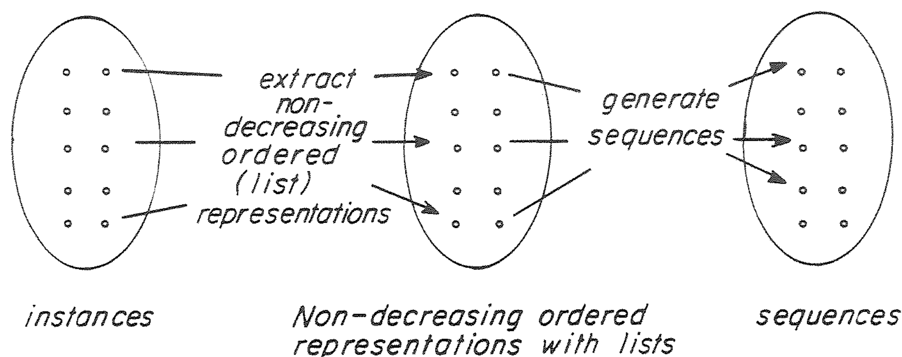


Figure 5-5: Refining the AGSC method to run in polynomial time.

set of instances which satisfy this condition. In essence, this condition is actually an intractability obstruction which satisfies all our requirements.

From the above discussion, we can conclude that to formulate our intractability obstruction, we need to draw upon our knowledge of affined sequences and of non-decreasing ordered lists. This obstruction which has to be satisfied by instances of the optimization problem associated with hypergraphs is stated as follows:

> **Intrinsically Ordered (IO) Convergence:** *Every representation with a non-decreasing ordered list L (for the corresponding instance) generates an affined sequence.*

Then, from the characterization theorem, the conditional truth (to be verified in Chapter 7) of Proposition 5.1, and the fact that the process of enumerating a sequence from a representation with list L is always polynomial (an obvious fact which will also be verified in Chapter 7), we have:

Theorem 5.1: *If an instance of the optimization problem (associated with hypergraphs) is IO convergent, then an optimum sequence can be enumerated for it in polynomial time.*

While most of our development so far in this chapter has been quite straightforward, it turns out that intrinsically ordered convergence, which we can verify from Theorem 5.1 to be an intractability obstruction, has some very intriguing consequences. We will see this in the next chapter where we show that instances drawn from *any* of the subdomains listed in Chapter 1 and in fact, from a few other *more general* subdomains as well, all satisfy this unifying property. In this manner, this property, through Theorem 5.1, forms a basis for a unified explanation of the like complexity characteristics of these seemingly different subdomains. We would also like to point out in passing that our approach towards identifying intractability obstructions draws upon the behavior of sequences, which are objects in the output domain of the optimization problem. This is in contrast with conventional approaches where such obstructions are typically formulated based on properties drawn from the input domain; in-trees, interval orders or cyclic forests from Chapter 1 are all examples of the latter approach.

5.3 Some Properties of Generated Sequences

We will now establish some properties of sequences generated from non-decreasing ordered lists which will prove to be useful later on. In establishing these results, we will consider instances which have the following special types of class functions. We say that the class function $\mathbf{C}$ is *perfect* if and only if for any pair of elements a_i and a_j in A, if an integer v' exists such that $C_{iv'}$ and $C_{jv'}$ are both greater than zero, then for any v where $1 \leq v \leq m$, C_{iv} equals C_{jv}. An instance is *perfect* whenever the corresponding class function is and in an analogous manner, we have *perfect* sequences as well. Basically, the set of elements in a perfect instance is partitioned in such a way that the elements in one of its classes say Π_x, do not require *any* resources which are needed by the elements from a different class say Π_y and vice-versa.

Our interest in perfect instances and sequences follows from the fact that those drawn from any of the subdomains listed in Chapter 1, are all perfect. In what follows, we will be focussing our attention on perfect instances and sequences only, and unless otherwise specified, any instance or sequence which we will be looking at is assumed to be perfect.

The first property of generated sequences which we are interested in establishing relates the position of the first copy of an element in a generated sequence S_L, to the 'state' of the list when it was selected for 'primary' inclusion; an element a_i is a *candidate (selected) for primary inclusion* on the qth scan if and only if it is a candidate (selected) for inclusion on this scan, and in addition, it must not have been selected for inclusion on any earlier scan. Also, the state of a list on some scan say q is determined by the set of all elements which are candidates for primary inclusion on this scan, and by the relative positions of these elements in the list.

Lemma 5.1: *In any sequence S_L generated from a representation with list L, a type-x element a_i has its first copy in edge E'_q if and only if:*

1. *it was a candidate for primary inclusion on the qth scan and*

2. *there were less than SI_x type-x elements which were also candidates for primary inclusion on the qth scan and which occurred before element a_i in L.*

That this property is true follows trivially from the greedy nature of our sequence construction method, the definitions of an element being a candidate and being selected for primary inclusion, and the fact that the instance is perfect.

In what follows, let S_ν always denote a sequence generated from a representation with a non-decreasing ordered list ν. In the next two lemmas, we will establish certain relationships between distinguished elements in such sequences and the state of the list when these elements were selected for primary inclusion. The first of these two relationships can be formally stated as follows:

Lemma 5.2: *If edge E''_q in any sequence $\hat{S}_\nu$, projected with respect to some d' and x from some sequence S_ν, is unsaturated, then the first copy of any type-x element a_i with $f'(a_i) \leq d'$ is in edge E'_q if and only if it was a candidate for primary inclusion on the qth scan.*

Proof: It is straightforward to verify the 'only if' part since the set of elements with their first copies in any edge E'_q in a sequence S_L is always a subset of the set of elements which were candidates for primary inclusion on the qth scan. So we will move on to establishing the 'if' part of this lemma. Suppose that it is not true. Then, it follows that there is a sequence $\hat{S}_\nu$ projected from sequence S_ν with respect to some d' and x without loss of generality, and a type-x element a_i with $f'(a_i)$ bound above by d' in such a way that element a_i was a candidate for primary inclusion on the qth scan and yet, it was not selected for primary inclusion. Among all such elements, let element $a_{i'}$ occur earliest in the corresponding list L; that is, any type-x element a_j with $f'(a_j)$ bound above by d', which occurred before element $a_{i'}$ in list L, and which was a candidate for primary inclusion on the qth scan, was also selected for inclusion.

Now, since edge E''_q is unsaturated or equivalently, since $|E''_q|$ is less than SI_x, it follows that there were less than SI_x type-x elements with

modified cost-determining functions bound above by d', and which were selected for primary inclusion on the qth scan. Therefore, there were less than SI_x such elements which occurred before element $a_{i'}$ in list L. Then, since element $a_{i'}$ is the earliest type-x element with $f'(a_{i'}) \leq d'$ which was a candidate for primary inclusion on the qth scan, and which was not selected for inclusion, and since list L is a non-decreasing order from the hypothesis of the lemma, we can conclude that there were less than SI_x type-x elements which were candidates for primary inclusion on the qth scan and which occurred before element $a_{i'}$ in this list; yet, it was not selected for inclusion on the qth scan which contradicts Lemma 5.1 completing the proof. □

The above lemma essentially gives us a means for working backwards from a given sequence S_ν and deducing a relationship between certain of its distinguished elements which were candidates for inclusion on a given scan during the process of constructing sequence S_ν, and whether or not they were actually selected for primary inclusion on this scan. Using this property, we can now show that in any sequence S_ν, certain distinguished elements always have immediate predecessors in such a way that the distance between these two elements (the predecessor-successor pair) is always k, or more precisely:

Lemma 5.3: *Any element a_i in edge E''_q for $q > 1$ in a projected sequence $\hat{S}_\nu$ has an immediate predecessor with its last copy in edge $E'_{(q-1)}$ in sequence S_ν whenever edge $E''_{(q-1)}$ is unsaturated.*

Proof: If the lemma is false, there exists a sequence $\hat{S}_\nu$ projected from some sequence S_ν with respect to d' and x, with an element a_i in edge E''_q for $q > 1$. Also, the corresponding edge $E''_{(q-1)}$ is unsaturated and no immediate predecessor element a_j of element a_i has its last copy in edge E'_{q-1}.

Since $a_i \in E''_q$, its first copy in sequence S_ν must be in edge E'_q and so, we can deduce from Lemma 5.1 that element a_i must have been a candidate for primary inclusion on the qth scan. From this, it follows that if element a_i has no immediate predecessor with its last copy in edge $E'_{(q-1)}$ and if it was a candidate for primary inclusion on the qth scan, either it has no predecessors in the corresponding precedence graph or the last copy of any of its predecessor element a_j must be in some edge $E'_{q'}$ for $q' < (q-1)$. Then, the type-x element a_i with $f'(a_i)$ bound above by d', was a candidate for primary inclusion on the $(q-1)$th scan, edge $E''_{(q-1)}$ is unsaturated, and yet it was not selected for primary inclusion on the $(q-1)$th scan which contradicts Lemma 5.2 □

The interesting thing about these specialized, and sometimes tedious to establish properties is that they depend heavily upon two aspects of our constructive method; the first of which we will refer to as the *greedy* nature of each scan of a list. Recall that the greedy algorithm solves the combinatorial optimization problem associated with a matroid $M = \langle E, \mathcal{I} \rangle$ [20]. Specifically, on each step, it chooses an element with the largest (smallest in the case of a minimization problem) weight from the corresponding set E; this element is an edge with maximum weight when M is a graphic matroid in which case, we are solving the MWF (or an edge with minimum weight in the case of the MST) problem. The element so chosen in then added to the solution provided, upon doing this, the partially constructed solution remains feasible; in the case of the MWF (or MST) problem, the solution remains feasible if the edge being added does not form a cycle with the partially constructed subgraph.

Then, the analogy between this method and *each* scan of the second

step of our procedure for generating sequences from lists is obvious since on each scan, we try to include as many elements as possible by choosing the 'candidate' elements; we are greedy in the sense that we choose the elements with the with lower list indices which are our analogs of the 'weights' from the matroidal optimization problem, where i is the list index of element a'_i. There are however important differences between the two algorithms, for example in the way that the cost of the solutions - which are minimum cost bases in the case of the greedy algorithm, and minimum cost sequences in our case - are formulated as a function of the weights in the respective cases. The second important aspect upon which our results of this section rest, is the non-decreasing ordered nature of the lists.

Chapter 6

Unifying the Polynomially Solvable Subdomains

In the previous chapters, we studied the behavior of sequences at optimality which culminated in the Characterization Theorem. We were also able to provide a complete characterization of a class of polynomially solvable instances through the IO convergence property. We now move on to showing the somewhat surprising result that this property is in fact shared by instances drawn from any of the subdomains identified in Chapter 1, as well as from other subdomains which we will introduce in this chapter. In Section 6.1, we will establish some important properties of distinguished elements which will prove to be of value in the subsequent sections of this chapter. In Section 6.2, we introduce the domain β_{11} of instances and show that any instance from this subdomain is IO convergent. We then show that subdomains β_1, β_3, β_5, and β_7 (of Chapter 1) are all properly included in subdomain β_{11} - therefore, instances from any of these subdomains are also IO convergent. In Section 6.2, we continue to do this for the remaining subdomains from the first chapter. Once again, in the context of subdomain β_3, we do this by showing that it is included in a more general subdomain β_{12} which we introduce here; any instance from β_{12} satisfies our unifying property. In this manner, we show that the complexity of many of these seemingly different subdomains follows eventually from the intrinsically ordered convergence property.

6.1 Saturated and Unsaturated Subsequences

In order to show that an instance is IO convergent, we need to first establish that the sequences generated by their non-decreasing ordered representations are always affined. Recall that in an affined sequence, any distinguished type-x element a_i with $f'(a_i) = d'$, must have a predecessor element such that the distance between this predecessor-successor pair is no greater than a special quantity : the critical distance of the predecessor element with respect to x and d'. Because of this, the problem of showing that only affined sequences are generated by our constructive method (Figure 5-5) from non-decreasing ordered (list) representations of instances drawn from the various subdomains of interest to us, translates to one of establishing the above mentioned bound on the distances between distinguished elements and their predecessors in the sequences which are generated from these instances.

But, in order to do this, it is very helpful if we recognize that distinguished elements fall into two broad categories; those which are members of *saturated subsequences*, and those which are members of *unsaturated subsequences* in the corresponding projected sequences. To identify these two categories of distinguished elements, let us consider a sequence S_ν and some sequence $\hat{S}_\nu$ projected from it with respect to d' and x. A continuous subsequence $\{E''_p, E''_{(p+1)}, \ldots, E''_q\}$ is said to be a saturated (unsaturated) subsequence of sequence $\hat{S}_\nu$ if and only if:

1. every edge $E''_{q'}$ for $p \leq q' < q$ is saturated (unsaturated),

2. edges $E''_{(p-1)}$ and $E''_{(p-2)}$ exist, and are respectively saturated (unsaturated) and unsaturated (saturated, unless $p = 1$ in which case edge $E''_{(p-2)}$ need not exist) respectively and,

3. unless edge E''_q is the last edge in sequence $\hat{S}_\nu$, that is unless $q = L$, it is unsaturated (saturated)

Basically, saturated (unsaturated) subsequences represent a continuous sequence of saturated (unsaturated) edges (Condition 1), with special types of edges at their end points (Conditions 2 and 3 above) as illustrated in Figures 6-1 (*i*) and (*ii*).

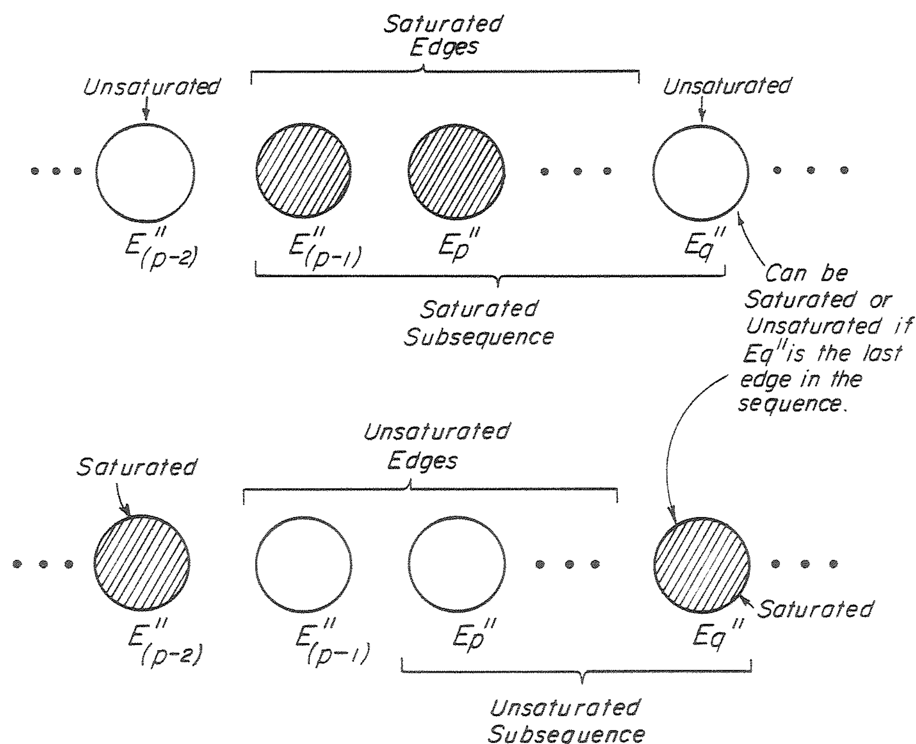


Figure 6-1: The edges of a (*i*) saturated and (*ii*) unsaturated subsequence of a projected sequence.

Note that primary saturated subsequences (of Chapter 2) do not qualify to be saturated subsequences in the above sense owing to the requirement on edge $E''_{(p-2)}$ in Condition 2. Then, since distinguished elements are those which do not belong to an edge in the primary saturated subsequence,

they are guaranteed to be always in an edge in a saturated subsequence or in an unsaturated subsequence of the corresponding projected sequence. Also, owing to our present interest in the IO convergence property and thereby in the behavior of distinguished elements - especially in those which occur in affined sequences - we wish to consider their behavior in these two types of subsequences separately. In particular, it is easy to see that any distinguished element in an edge of an unsaturated subsequence of any sequence, projected from one which is generated by our constructive method from Figure 5-5, automatically satisfies the distance bounds on these distinguished elements in an affined sequence, owing to Lemma 5.3; therefore, we need not concern ourselves with such distinguished elements anymore. We will see this fact being used repeatedly in the rest of this chapter. Also, this allows us to concern ourselves only with distinguished elements which occur in edges of the saturated subsequences.

To start with, we will establish a couple of very interesting properties of these distinguished elements which occur in edges in the saturated subsequences of the corresponding projected sequences. Consider a saturated subsequence $\{E''_p, ..., E''_q\}$ of a sequence $\hat{S}_\nu$ and an element a_i in some edge $E''_{q'}$ of this subsequence. In the first of these properties, we are interested in certain special kinds of predecessors which elements such as a_i might have. More precisely, given an element a_i in an edge of a saturated subsequence, we identify its predecessor element a_j to be the *earliest predecessor relative* to this saturated subsequence $\{E''_p, ..., E''_q\}$ if and only if the last copy of element a_j is in edge $E'_{p'}$ for $(p-2) \leq p' \leq (q-1)$, and the last copy of any other predecessor element $a_{j'}$ of element a_i, is not in any edge $E'_{p''}$ for $(p-2) \leq p'' < p'$. Then, the first of our properties tells us that such an earliest predecessor element a_j always exists for element a_i. This property is actually quite straightforward to establish as follows:

Lemma 6.1: *Let* $\{E''_p, ..., E''_q\}$ *be a saturated subsequence of sequence* $\hat{S}_\nu$ *projected from* S_ν *with respect to* d' *and* x. *Then, any element* a_i *in edge* $E''_{q'}$ *in* sequence $\hat{S}_\nu$ *for* $p \leq q' \leq q$ *has an earliest predecessor* a_j *relative to this saturated subsequence.*

Proof: That such an element a_i has at least one predecessor with its last copy in some edge $E'_{p'}$ for $(p-2) \leq p' \leq (q'-1)$ is immediate from Lemma 5.2. Otherwise, element a_i would have been a candidate for primary inclusion on the $(p-2)$th scan and even though edge $E''_{(p-2)}$ is unsaturated, it was not selected for primary inclusion on this scan - a contradiction. That one of these predecessors has to be the earliest relative to subsequence $\{E''_p, ..., E''_q\}$ follows from the definition of the earliest predecessor relative to a given subsequence and the fact that sequence S_ν has a finite number of edges. □

The next property of the elements in a saturated subsequence is much more interesting. To state this property, once again consider an element a_i in some edge $E''_{q'}$ which is in a saturated subsequence $\{E''_p, ..., E''_q\}$ of sequence $\hat{S}_\nu$, which in turn is projected with respect to some d' and x. Then, let us suppose that there is some element a_j which is a predecessor of any element in an edge of this saturated subsequence $\{E''_p, ..., E''_q\}$. Then, the following relationship stated in Lemma 6.2 exists between the critical distance of element a_j with respect to d' and x, and the distance between elements a_i and a_j is sequence S_ν.

Lemma 6.2: *Any element* a_i *in a saturated subsequence* $\{E''_p, ..., E''_q\}$ *of sequence* $\hat{S}_\nu$ *projected from some sequence* S_ν *with respect to* d' *and* x, *has a predecessor* a_j *such that the distance between elements* a_i *and*

a_j in sequence S_ν is bound above by the critical distance of element a_j with respect to d' and x whenever:

1. *the last copy of element a_j is in edge $E'_{(p-2)}$* in sequence $\hat{S}_\nu$ *and,*

2. *every element in the saturated subsequence is its successor.*

Proof: From the definition of a saturated subsequence, it follows that for $(p-1) \leq q' < q$,

$$|E''_{q'}| = SI_x$$

and

$$|E''_q| = e \leq SI_x.$$

Consider an element a_j which satisfies the hypothesis of the lemma. Then, element a_j is such that it has

$$n(j,d',x) = (q-p+1) \cdot SI_x + e$$

type-x successors whose modified cost-determining functions are bound above by d'. Then, the critical distance of element a_j with respect to d' and x, say CD is such that

$$\begin{aligned} CD &\geq \lceil n(j,d',x)/SI_x \rceil + k - 1 \\ &= \lceil (q-p) + 1 + (e/SI_x) \rceil + k - 1. \end{aligned}$$

Then, if $e > 0$, we have

$$CD_a = q + k + 1 - p$$

and, if $e = 0$,

$$CD_b = q + k - p.$$

Now, since S_ν is proper, it follows that the first copy of element a_j is

in edge $E'_{p'}$ of sequence S_ν where $p' = p - (k+1)$. Then, the distance δ_i between any element a_i in an edge of the saturated subsequence $\{E''_{p'}, ..., E''_q\}$, and element a_j, in sequence S_ν, is such that when $e > 0$,

$$\delta_i \leq q + k + 1 - p = CD_a$$

and when $e = 0$,

$$\delta_i \leq q + k - p = CD_b$$

and hence the result. □

In unifying the polynomially solvable subdomains in the next two sections, when we show that instances from these subdomains are always IO convergent, we will be actually be working withing the framework of the general optimization problem associated with hypergraphs where arbitrary cost determining functions are allowed; these subdomains of this general optimization problem therefore correspond to the tardiness minimization version of the scheduling problem. Also, we have already shown in Chapter 2 that the makespan minimization version of the scheduling problem can be reduced to its tardiness minimization version. As we pointed out earlier on, this allows us to view the makespan minimization version of the scheduling problem as a special case of its tardiness minimization version and consequently, as a special case of our optimization problem associated with hypergraphs. As a result, when we are dealing with subdomains for which the original results were oriented towards the makespan minimization version of the scheduling problem, as is the case with Hu's [48] subdomain β_1 for example, we will be actually referring to the subdomains which correspond to the 'reduced' or 'transformed' versions from the original scheduling problem.

6.2 Unification Results

In this section, we start unifying the polynomially solvable subdomains through IO convergence. To start with, we introduce the new subdomain β_{11}. Our motives behind doing this and its relationship to the subdomains from Chapter 1 will become clear as we progress further. The precedence graphs of instances from β_{11} are restricted to be in-forests where a precedence graph $P = \langle A, \alpha \rangle$ is an *inforest* if and only if every element a_i in the set of elements A has no more than one immediate successor. In addition, for an instance to be in the subdomain β_{11}, we require certain special relationships to hold between the class function, the manner in which it partitions the set A into classes, and the interactions of the member elements of this instance through the precedence graph.

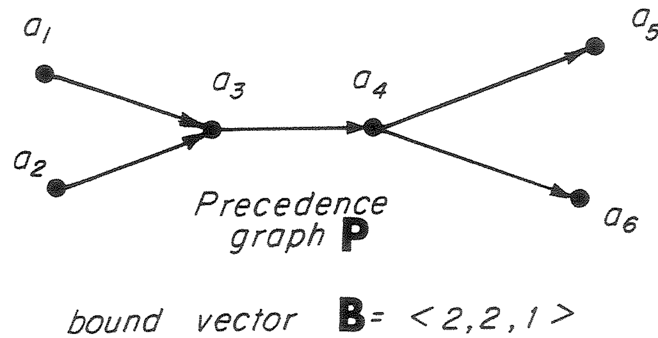
In order to formulate these relationships in a more precise manner, we define the *type (interaction) graph*, which is a directed graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ associated with the precedence graph $P = \langle A, \alpha \rangle$. Intuitively, each vertex $v_x \in \mathcal{V}$ of the type graph is associated with one of the classes of elements Π_x in the set of elements A. An edge $\langle v_x, v_y \rangle$ is in the edge set $\mathcal{E}$ if and only if there is at least one pair of elements a_i and a_j in A such that:

1. elements a_i and a_j are respectively type-x and type-y elements and,

2. element a_i is an immediate predecessor of element a_j.

An example instance and its type graph are shown in Figures 6-2 and 6-3, respectively. In the type graph of Figure 6-3, vertex v_1, which corresponds to class Π_1 of the set A (indicated parenthetically in the figure) has an edge directed towards vertex v_2 which corresponds to class Π_2 since: (i).

elements a_1 and a_3 are respectively type-1 and type-2 elements and (*ii*). since element a_1 is an immediate predecessor of element a_2 in the precedence graph of this instance (Figure 6-2).



a_i	**C** (a_i)	Element Type
a_1	< 1,0,0 >	1
a_2	< 1,0,0 >	1
a_3	< 0,1,0 >	2
a_4	< 1,0,0 >	1
a_5	< 0,0,1 >	3
a_6	< 0,0,1 >	3

Figure 6-2: The relevant components of an instance including a classification of the elements into various 'types'.

Using type graphs, we are now in a position to identify the sub-

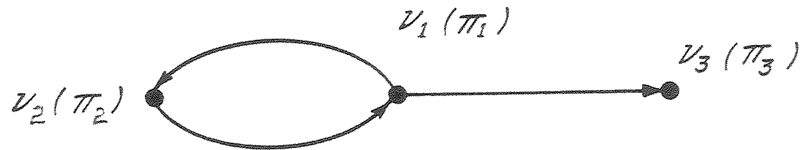


Figure 6-3: The type graph of the instance from Figure 6-2.

domain β_{11} of instances. In addition to having in-tree (or in-forest) precedence graphs, the type graphs of these instances have to be conservative, where an instance of the optimization problem associated with hypergraphs is said to have a *conservative* type graph provided for any y where $1 \leq y \leq M$,

$$\sum_{x\,:\,\langle v_x, v_y \rangle \in \mathcal{E}} SI_x \leq SI_y.$$

Conservative type graphs require a special relationship to hold between the saturation indices of the elements which are interacting through the relations derived from the corresponding precedence graph. It is easy to see that the instance indicated in Figure 6-2 is not conservative since $SI_3 = 1$, and since it is less than $SI_1 = 2$, even though edge $\langle v_1, v_3 \rangle$ is defined in the corresponding type graph. We will now prove the following result for instances from this subdomain.

Theorem 6.1: *Any instance from subdomain β_{11} of the optimization problem associated with hypergraphs is IO convergent.*

Proof: Consider any type-x element a_i in some sequence S_ν without loss of generality. Moreover, let element a_i be a distinguished element is sequence S_ν. If no such element exists, we can conclude trivially that sequence S_ν is affined. Then, the edge E''_q, to which element a_i belongs must either be in a saturated or in an unsaturated subsequence of the sequence $\hat{S}_\nu$; sequence $\hat{S}_\nu$ is projected from sequence S_ν with respect to d' and x where $f'(a_i) = d'$.

We will now show that in a sequence S_ν corresponding to any instance from subdomain β_{11}, any distinguished element a_i always has a predecessor element a_j with its last copy in edge $E'_{(q'-1)}$.

Suppose that edge $E''_{q'}$ is in a saturated subsequence $\{E''_p, ..., E''_q\}$ of sequence $\hat{S}_\nu$, and that it has a member element a'_i such that this is not true. From Lemmas 5.3 and 6.1, we can conclude that any element in an edge $E''_{p'}$ for $(p-1) \leq p' \leq q$ has at least one predecessor with its last copy in some edge E'_r of sequence S_ν, where $(p-2) \leq r < q$. Then, from this and the fact that every one of the $E''_{p''}$ edges for $(p-1) \leq p'' < q'$ is saturated, we can conclude that there must be at least

$$\delta = (q' - p + 1) \cdot SI_x$$

type-x elements which became candidates for primary inclusion for the first time, in the interval starting with scan $(p-1)$ and ending with scan $(q'-1)$. In fact, if element a'_i in edge $E''_{q'}$ does not have any predecessor with its last copy in edge $E''_{(q'-1)}$ and since it was selected for primary inclusion on the q'th scan, we can conclude that it must have been a candidate for primary inclusion on the $(q'-1)$th scan as well. Therefore, there were in fact at least $(\delta + 1)$ elements which became candidates in the interval starting with scan $(p-1)$ and ending with scan $(q'-1)$. From this, it follows that there exists a q'' where $(p-1) \leq q'' < q'$ such that more than SI_x type-x elements - say δ' of them - with their modified cost-determining function values bound above by d' became candidates for primary inclusion on the q''th scan for the first time; in other words, every one of these $\delta' > SI_x$ elements have immediate predecessors with their last copies in edge $E'_{(q''-1)}$.

Now, since the instances from subdomain β_{11} have conservative type graphs, it follows that there can be no more than SI_x of these immediate

predecessors (of the δ' type-x elements which became candidates for primary inclusion on the q''th scan for the first time) with their last copies in edge $E'_{(q''-1)}$ in sequence S_ν. This implies that no more than SI_x elements with their last copies in edge $E'_{(q''-1)}$ are immediate predecessors of $\delta' > SI_x$ type-x elements which became candidates for primary inclusion of the q''th scan for the first time. From this, we conclude that at least one of these predecessor elements with its last copy in edge $E'_{(q''-1)}$ has more than one immediate successor, which contradicts the fact that the precedence graph of this instance from subdomain β_{11} is an in-tree.

On the other hand, if edge $E''_{q'}$ is in an unsaturated subsequence of sequence $\hat{S}_\nu$, then we are done since the fact that element a_i has an immediate predecessor with its last copy in edge $E'_{(q'-1)}$ of sequence S_ν follows immediately from Lemma 5.3.

Therefore, we conclude that any distinguished element a_i with its first copy in edge $E''_{q'}$ has a predecessor a_j with its last copy in edge $E'_{(q'-1)}$. Also, since sequence S_ν is proper, the first copy of this element a_j must be in edge $E'_{(q'-k)}$ and so the distance between elements a_i and a_j equals k. Since element a_j has at least one type-x element a_i as a successor for which $f'(a_i) = d'$, the critical distance of element a_j with respect to d' and x must be bound below by k. From this, we can conclude that sequence S_ν is affined completing the proof. □

We will now show that a number of subdomains from Chapter 1 are all essentially special cases of subdomain β_{11}. We start off with subdomains β_5 [2] and β_9 [4]. It is clear from the very definition of subdomain β_9 (from Table 1-3) that it is a proper subset of β_{11} for which the following additional restrictions hold:

1. $m = 1$ (there is a single 'processor-type' resource),

2. $\mathbf{C}(a_i) = \langle 1 \rangle$ for all elements a_i (each task needs only one unit of this resource) and,

3. the length attribute $\mathbf{l}$ equals the degree of slicing k (task lengths equal the number of stages in the pipelines).

Also, we get subdomain β_5 by restricting the instances from subdomain β_9 even further by requiring that their degree of slicing k, equal unity. Also, since we know from our earlier comments that the makespan minimization version of the scheduling problem can be reduced to its tardiness minimization version, we can conclude that subdomains β_7 [4] and β_1 [48] are essentially special cases of subdomains β_9 and β_5 respectively. From these observations and Theorem 6.1, we have the following result for these subdomains:

Corollary 6.1: *Any instance from subdomains β_1, β_7, β_5,* or *β_9 is IO convergent.*

We now consider the subdomain β_4 for which Goyal [39] discovered a polynomial time algorithm for solving (the makespan minimization version of the) the optimization problem; his goal was to extend the basic scheduling model to include multiprocessor systems with many different types of 'specialized processors'. It turns out that even subdomain β_4 is also a very special case of subdomain β_{11}; the instances from subdomain β_4 satisfy the following additional constraints, over and above those satisfied by instances from β_{11}:

1. the elements B_i of the bound vector are all unity that is, $B_i = 1$

for $1 \leq i \leq m$ (translates to the fact that there is only a single unit of any resource),

2. for each i, C_{ij} equals one for only a single value of j and is zero everywhere else or equivalently, for each i, there exists a unique j' such that $C_{ij'} = 1$ and $C_{ij} = 0$ wherever $j \neq j'$ (each task requires exactly *one* unit of *one* of the m resources types in the system),

3. the degree of slicing k equals one (only 'single-stage processors' are allowed),

In addition to these three constraints, for an instance to be in Goyal's subdomain β_4, its precedence graph has to be a *cyclic forest*. Stated in the light of our framework, a precedence graph is a cyclic forest provided it is a forest and in addition, if the type graph of the associated instance $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ is exactly a hamiltonian cycle, or a hamiltonian path. Intuitively, all the leaf elements - that is those at level $(h-1)$ in a cyclic forest, which from its definition is also an in-forest - are all of the same type, in the sense that they all need the same type of 'processors' from the resource set. Furthermore, the elements at any level $h-i$ for $1 < i \leq m$ all need the same type of 'processor', which is different from the type of 'processor' needed by the 'tasks' at any of the higher levels, and this pattern repeats from level $h-m-1$ on, as we progress towards the roots of the precedence graph. This idea is illustrated in Figure 6-4 below.

Since we have seen that subdomain β_4 is a subdomain β_{11}, derived by adding the above mentioned (four) conditions, we can deduce that it (subdomain β_4) also fits into our unifying framework and so,

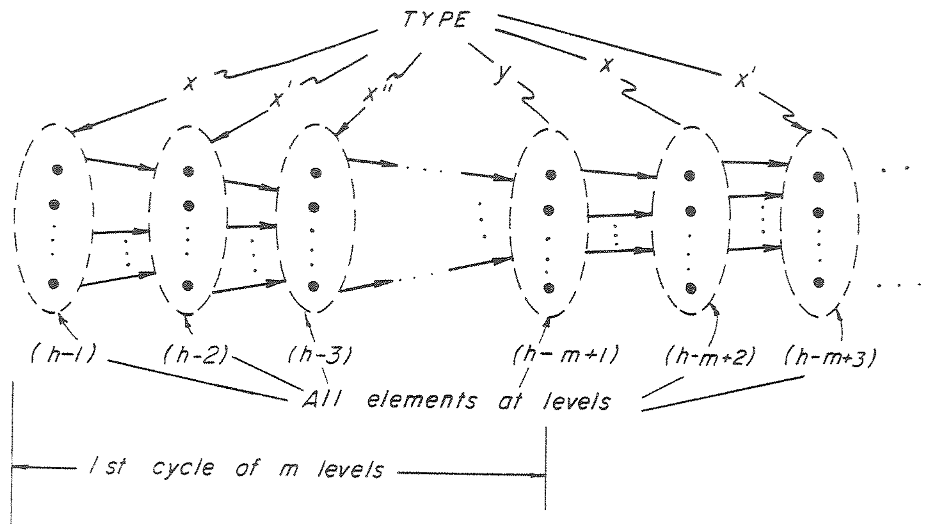


Figure 6-4: The relationship between the elements at the various levels of a *cyclic forest.*

Corollary 6.2: *Any instance of* β_4 *is IO convergent.*

As an aside, we would like to point out that the subdomain β_{11} is significantly more general than any of those which we have shown thus far to be properly included in it. By identifying it, we have not only progressed in the direction of synthesizing a significant part of the intended unifying framework, but we have also contributed along another quite independent dimension of this problem by discovering this rich and hitherto unknown class of polynomially solvable instances of the optimization problem on hand.

6.3 More Unification Results

We now move on to the subdomain β_6 discovered by Garey and Johnson [33]. Instances in this subdomain are allowed to have arbitrary precedence graphs. However, they need to satisfy the following additional constraints, mostly on the types of 'resources' that they are allowed to have :

1. m equal 1 (a single 'processor-type' resource),
2. the bound vector $\mathbf{B} = \langle 2 \rangle$ (only two of these 'processors'),
3. the degree of slicing $k = 1$ (these 'processors' have a single stage) and,
4. for any i, $\mathbf{C}(a_i) = \langle 1 \rangle$ (each 'task' needs exactly one of these processors).

Once again, it turns out that instances which satisfy these constraints always satisfy our unifying IO convergence property. The special property of elements from Lemma 6.2 plays a central role in proving this result.

Theorem 6.2: *Any instance from subdomain β_6 of the optimization problem associated with hypergraphs is also IO convergent.*

Proof: Without loss of generality, let us consider some instance from this subdomain and its associated sequence S_ν. We now need to show that this sequence is affined. From the definition of subdomain β_6, we have $B_1 = 2$ and $m = 1$ for any corresponding instance. Also, each element a_i in any such instance is such that $\mathbf{C}\,(a_i) = 1$. Therefore, the saturation index SI_1 of the (only) class Π_1 of elements, is always two. Consider any distinguished element

a_i with $f'(a_i) = d'$; since there is only a single class or type of elements, it must be (trivially) from class Π_1. Also, let the first and only (since $k = 1$) copy of element a_i in sequence S_ν be in edge E'_q. Consider sequence $\hat{S}_\nu$ projected with respect to d' (where the type parameter is not specified since there can only be a single element type for instances from this subdomain).

Now, if edge E''_q is in an unsaturated subsequence of sequence $\hat{S}_\nu$, it follows from the definition of an unsaturated subsequence that edge $E''_{(q-1)}$ is unsaturated. We can then deduce from Lemma 5.3 that element a_i has a predecessor element a_j with its last (and only) copy in edge $E'_{(q-1)}$ in sequence S_ν. Also, element a_j has at least one successor element a_i with its modified cost-determining function bound above by d'; therefore, the critical distance of element a_j with respect to d' (and the single element type of course) is bound below by unity. Also, since $k = 1$ the first copies of elements a_i and a_j are respectively in edges E'_q and $E'_{(q-1)}$ and therefore, the distance between these elements in sequence S_ν equals unity. This allows us to conclude that element a_i meets the requirements of a distinguished task in an affined sequence.

On the other hand, suppose that edge E''_q is in a saturated subsequence $\{E''_p, ..., E''_q\}$ in sequence $\hat{S}_\nu$. From the definition of a saturated subsequence, edge $E''_{(p-2)}$ must be unsaturated and furthermore since $B_1 = 2$ and $SI_1 = 1$, $|E''_{(p-2)}| \in \{0,1\}$. Since there is only a single element type, it follows that there is no more than one element a_j with its first copy in edge $E'_{(p-2)}$ in the original sequence S_ν that $f'(a_j) \leq d'$. We will first show that edge $E'_{(p-2)}$ in fact has exactly one such element a_j and in addition, that this element a_j is the predecessor of any element from any edge of the saturated subsequence in question.

To do this, let us without loss of generality consider some element $a_{i'}$ in edge $E''_{q''}$ of the saturated subsequence. From Lemma 6.1, element $a_{i'}$ must have an earliest predecessor a' relative to this saturated subsequence of sequence $\hat{S}_{\nu}$. This in turn implies that the last (and only) copy of element a' must be in some edge $E'_{p'}$ for $(p-2) \leq p' < q''$. We will first see that the last copy of element a' must be in edge $E'_{(p-2)}$.

Suppose p' is not less than $(p-1)$. Since element a' is the earliest predecessor of element a_i relative to the saturated subsequence $\{E''_p, ..., E''_q\}$, the last copy of any predecessor element a'' of element a' - if such an element a'' exists - must be in some edge E'_r for $r < (p-2)$. Whether element a' has a predecessor or not, the above fact implies that element a' was a candidate for primary inclusion of the $(p-2)$th scan; moreover, $f'(a') \leq d'$ and there is a single element type. Despite all this and the fact that edge $E''_{(p-2)}$ in sequence $\hat{S}_{\nu}$ is unsaturated, the first copy of element a' in sequence S_{ν} is not in edge $E'_{(p-2)}$ which contradicts Lemma 5.2. Therefore, p' must equal $(p-2)$.

Then, since $f'(a') \leq d'$, and since there can be at most one such element a' with its first copy in edge $E'_{(p-2)}$ which is an unsaturated edge, it follows that this element a' must in fact be the predecessor of any element in the saturated subsequence $\{E''_p, ..., E''_q\}$; in other words, element a' is actually the desired element a_j whose existence was in question. Then, by invoking Lemma 6.2, we conclude that the critical distance of this element a_j with respect to d' (and the single type), is no less than the distance between the distinguished element a_i (with its only copy in edge E'_q) and its predecessor element a_j in sequence S_{ν}, completing the proof. □

We can show that the same result is true of instances from sub-

domain β_{10} which is very similar to β_6 except that we change restrictions (2) and (3) above from requiring that the bound vector $\mathbf{B} = \langle 2 \rangle$ and the degree of slicing k equal unity, to respectively require that the bound vector $\mathbf{B} = \langle 1 \rangle$ and the degree of slicing k equal two (two 'single-stage processors' in the former case are replaced by a single 'two-stage pipelined processor' in the latter. Then given any instance from subdomain β_{10}, we have:

Theorem 6.3: *Any instance from subdomain β_{10} of the optimization problem associated with hypergraphs is IO convergent.*

Proof: The proof of this theorem is analogous to that of Theorem 6.1 and we will only outline the important points here. Once again, we consider an arbitrary sequence S_ν corresponding to an instance from subdomain β_{10} and a distinguished element a_i in it with $f'(a_i) = d'$. If element a_i with its first copy in edge E'_q is such that E''_q is in an unsaturated subsequence of sequence $\hat{S}_\nu$ projected with respect to d' (and the single type), then the argument is a trivial variant of that used in the context of this case in the proof of Theorem 6.2. On the contrary, edge E''_q could be in a saturated subsequence of sequence $\hat{S}_\nu$. If this is the case, an argument similar to that used in the proof of Theorem 6.2 shows us that there is an element a_j with its first copy in edge $E'_{(p-3)}$ in sequence S_ν such that it is a predecessor of every element from this (saturated) subsequence. This allows us to subsequently invoke Lemma 6.2 to complete the proof. □

At this point, we would like to point out that the algorithms from [4] and [33], which were designed with the intention of solving the optimization problem for instances from the above mentioned two subdomains, found optimum sequences (schedules) by first constructing non-decreasing ordered

list representations and so on, much in the same manner as we do. However, in both these cases, the optimality of the sequences so determined was not recognized except when their (the sequences') cost was zero. By extending this result through the machinery developed in Chapter 4, we arrived at the Characterization Theorem, which now allows us to immediately recognize that these sequences which are constructed from non-decreasing ordered (list) representations of instances from subdomains β_6 and β_{11} are optimum, not only when their cost is zero, but also at all other points in the range of the cost function as well. As a consequence, we can now claim that a single pass of our constructive method yields affined and therefore optimum sequences for instances from these subdomains.

Since this fact was not recognized earlier on, the approach in [4] and [33] towards finding minimum cost sequences involved using a subroutine which constructs sequences (schedules) form non-decreasing ordered representations of the given instance; this subroutine is invoked $log(n)$ times in a binary search mode, each time incrementing or decrementing the values assigned by the cost determining function (deadlines in the case of the original results) until a sequence (schedule) with zero cost (meeting all the deadlines) was found. Then, by combining their knowledge of the analog of Theorem 3.1 for sequences which correspond to instances from subdomains β_6 and β_{10}, Garey and Johnson [33] and Bruno, Jones and So [4] conclude that this binary search procedure will eventually find a minimum cost sequence for an instance from either of these subdomains. However, owing to the characterization theorem, we now know that this binary search phase is really quite unnecessary even when the sequences with minimum cost for the given instances have greater than zero cost.

Finally, we consider subdomain β_2 for which progressively faster algorithms were designed by Fujii et al. [23], Coffman and Graham [9] and Gabow [29], and subdomain β_8, both of which are subdomains of the makespan minimization version of the scheduling problem. It is easy to see from the tables in Chapter 1 that the constraints which the instances from subdomains β_2 and β_8 obey are respectively identical to those which are satisfied by instances from subdomains β_6 and β_{10}. From this and the fact that the makespan minimization version of the scheduling problem is essentially a special case of its tardiness minimization version, we have the following consequence of Theorems 6.2 and 6.3 :

Corollary 6.3: *Any instance from subdomains β_2 or β_8 is IO convergent.*

This leaves us with subdomain β_3 due to Papadimitriou and Yannakakis [75] which has not been shown to be a part of our unifying framework thus far. Our approach in including this subdomain into our unifying framework is similar to that taken in the previous section in connection with subdomain β_1 in that we first introduce a new subdomain β_{12}. The only restriction which instances from this subdomain need to satisfy is that the corresponding precedence graphs need to be interval-orders, where a partial order $P = \langle A, \alpha \rangle$, each of whose elements $a_i \in A$ is a closed interval in the real line, is an *interval order* provided an edge $\langle a_i, a_j \rangle \in \alpha$ if and only if for any pair of reals γ and γ', if $\gamma \in a_i$ and $\gamma' \in a_j$, then $\gamma < \gamma'$.

Interval orders have the following interesting relationship between the set of successors $\mathcal{A}(a_i)$ and $\mathcal{A}(a_j)$ of any pair of elements a_i and a_j from A, which we now paraphrase from [75]:

Lemma 6.3: *If the precedence graph $P = \langle A, \alpha \rangle$ is an interval order, then for any pair of elements a_i and a_j, either $\mathcal{A}(a_i) \subseteq \mathcal{A}(a_j)$ or $\mathcal{A}(a_j) \subseteq \mathcal{A}(a_i)$.*

It turns our that the property stated in the above lemma has very positive algorithmic consequences as we will now see during the process of including subdomain β_{12} in our unifying framework. Also, it is interesting to note that this property of interval-ordered graphs will be used in our arguments in a quite different way, when compared to the way in which it is used in Papadimitriou and Yannakakis's proofs from [75].

Theorem 6.4: *Any instance from subdomain β_{12} of the optimization problem associated with hypergraphs is IO convergent.*

Proof: We need to show that any sequence S_ν corresponding to an instance from subdomain β_{12} is affined. To do this, let us consider some type-x element a_i which is distinguished in sequence S_ν and with its first copy in edge E'_q without loss of generality. Also, let us suppose that $f'(a_i) = d'$. Once again, let $\hat{S}_\nu$ represent the sequence projected with respect to d' and x from the original sequence S_ν.

If edge E''_q is in an unsaturated subsequence of sequence $\hat{S}_\nu$, we can deduce from Lemma 5.3 that element a_i has a predecessor element a_j with its last copy in edge $E'_{(q-1)}$ in sequence S_ν. Now, since the distance between elements a_i and a_j in sequence S_ν equals k, and since the critical distance of element a_j with respect to d' and x is at least k (since it has at least one type-x element a_i as its successor and since $f'(a_i) = d'$), we are done.

It is much more interesting to consider the case where edge E''_q is in a saturated subsequence $\{E''_p, ..., E''_q\}$ of sequence $\hat{S}_\nu$. Once again, we will proceed to show that there exists an element a_j with its last copy in edge $E'_{(p-2)}$ such that any element in an edge of this saturated subsequence is its successor. In particular, we will show that this desired element a_j is any maximal element with its last copy in edge $E'_{(p-2)}$; element a_j is a maximal element with its last copy in edge $E'_{(p-2)}$ if and only if for any element $a_{j'}$ with its last copy in edge $E'_{(p-2)}$, $\mathcal{A}(a_{j'}) \subseteq \mathcal{A}(a_j)$. From Lemma 6.3 and the transitive nature of the subset inclusion relationship, we know that such a maximal element a_j exists.

Now, suppose that an element $a_{i'}$ exists in edge $E''_{p'}$ in the saturated subsequence such that it is not a successor of element a_j or equivalently, $a_{i'} \notin \mathcal{A}(a_j)$. From Lemma 6.1, we know that element $a_{i'}$ has an earliest predecessor element a' in sequence S_ν with its last copy in some edge E'_r for $(p-2) \leq r < p'$. Clearly, r must be strictly greater that $(p-2)$ or else, $\mathcal{A}(a') \subsetneq \mathcal{A}(a_j)$ which contradicts the maximality of element a_j; therefore, r must be such that in fact $(p-2) < r < p'$.

Now, consider any element a'' in edge $E''_{(p-1)}$ which is saturated by definition. Since edge $E''_{(p-2)}$ is unsaturated, we can deduce from Lemma 5.3 that this element has a predecessor with its last copy in edge $E'_{(p-2)}$, and therefore from the maximality of element a_j, it follows that $a'' \in \mathcal{A}(a_j)$. Now, given that the last copy of element a' is in some edge E''_r where $r > (p-2)$, it must follow that $a'' \notin \mathcal{A}(a')$ or else sequence S_ν would not be order preserving. Therefore, elements a_j and a' are such that $\mathcal{A}(a_j) \subsetneq \mathcal{A}(a')$ and $\mathcal{A}(a') \subsetneq \mathcal{A}(a_j)$ which contradicts Lemma 6.3 and so, element $a_{i'}$ must in fact be a successor of element a_j.

Then, from Lemma 6.2, it follows that any distinguished element a_i in some edge E''_q in the saturated subsequence $\{E''_p, ..., E''_q\}$ of sequence $\hat{S}_\nu$, must have a predecessor element a_j in such a way that the corresponding sequence $\hat{S}_\nu$ is affined. □

From this theorem, we conclude that our framework also includes subdomain β_3 since it is a proper subset of β_{12}, derived by imposing the following additional constraints:

1. $m = 1$ (only a single 'processor-type resource'),

2. the degree of slicing $k = 1$ ('single-stage processors') and,

3. $\mathbf{C}(a_i) = \langle 1 \rangle$ for all elements a_i (each 'task' needs any one of these 'processors') .

Moreover, in [75], the primary goal was to solve the makespan minimization version of the (scheduling) problem and since we have already seen that this version is a special case of the corresponding tardiness minimization version, and since subdomain β_3 is a subset of β_{12}, we can deduce from Theorem 6.3 that:

Corollary 6.4: *Any instance from subdomain β_3 is IO convergent.*

With this, we have accomplished most of our goals since we have discovered the intrinsically ordered convergence property, from which the polynomial solvability of our optimization problem, for instances from many of the apparently unrelated subdomains which we have been considering thus far, follows. A very intriguing aspect of this result is that these apparently dif-

ferent subdomains which were discovered at varying points in time by different groups of researchers, all share this underlying structure at a deeper level. Specifically, through IO convergence, we now know that instances from any of these subdomains which seem to satisfy intrinsically different constraints at the outset, all have affined sequences (at optimality) and in addition, that for these instances, we can also find these sequences efficiently (in polynomial time). In the next chapter, we will address the issue of implementing these algorithms for finding such sequences in greater detail.

Chapter 7

The Algorithm and its Complexity

In this chapter, we return to the problem of constructing optimum sequences from instances of the optimization problem associated with hypergraphs. Basically, the constructive method which we described in Chapter 5 consists of three major steps. The first step involves determining the modified cost-determining function for the given instance. We can then enumerate a non-decreasing ordered list for the instance by appropriately sorting the elements and finally, we apply our generalization of list scheduling to get the corresponding sequence. In Section 7.1, we describe our algorithmic notation and related issues. In Section 7.2, we describe an algorithm for computing the modified cost-determining function. In Section 7.2.1, we identify the data structures which play a role in this algorithm. Subsequently, in Section 7.2.2, we specify the algorithm using the notation from Section 7.1. In Section 7.2.3, we analyze the time complexity of this algorithm. Finally, we outline our generalized list scheduling method in Section 7.3.

7.1 Algorithmic Notation

To express our algorithms, we will use a programing language which is based to a significant extent on that used by Tarjan in [89]; this language combines Dijkstra's guarded command language [18] and SETL [57]. All our algorithms are of course *sequential* and *deterministic* in nature.

We will generally count each operation as one step (the *uniform cost measure*) as opposed to charging for each operation, a time proportional to the number of bits in the operands (the *logarithmic cost measure*). As we will see, this approach helps us in getting a clearer understanding of the computationally intensive parts of the algorithm.

In keeping with standard practice, we measure the running time of the algorithms as a function of input length. This *dynamic* measure of time complexity is more relevant in our context than say, a *static* measure which is essentially independent of the size of the inputs. Also, as usual, we ignore constant factors and assume that the reader is familiar with the $O(n), \Omega(n), \Theta(n)$ notation for expressing the appropriate asymptotic bounds on the running times of algorithms. We will also measure the running time as a function of the *worst-case* input data as opposed to different types of *average-case* methods [51], [56], [76], [86].

7.1.1 Data Structures

The basic types of data or data types which we will be dealing with include *integers*, *reals* and *elements*. An element a_i is a representation of a member of set A from an instance of the optimization problem, and it will be represented by just storing the index i. By allowing an element data type, our algorithms can be expressed directly in terms of elements and as such, they will be very easy to read and comprehend. In addition, we also allow *lists*, *sets*, and *maps*, all of which are essentially more complex objects made up of one of the previously described data types.

A list $\mathcal{L} = \{\mathbb{l}_1, \mathbb{l}_2, \dots \mathbb{l}_n\}$ is any sequence of n arbitrary integers, reals, elements or one of the other more complex objects such as other lists for ex-

ample; of course, the members $\llbracket_i$ of a given list $\mathcal{L}$ are all of the same type. Typically, we will use lists when there is a specific order associated with its elements. For example, if list $\mathcal{L}$ is a list of integers, they will be listed in non-decreasing or non-increasing order, and so on. While the members of a list can be repeated in principle, we will never have occasion to use this feature. Also, we will use $\in$ to denote membership in a list.

We define two basic operations on lists:

- *Access*: Given a list $\mathcal{L}$ and an integer i, return the ith element $\llbracket_i$ of list $\mathcal{L}$; if element $\llbracket_i$ is not in the list that is $i < 1$ or $i > n$, then access returns the value *null*.

- *Replace*: The inverse of access where given a list $\mathcal{L}$, an i, and a "value" (in that order), we replace the ith element in list $\mathcal{L}$ by "value"; defined only when the data type of "value" matches that of the elements of list $\mathcal{L}$ and also element $\llbracket_i$ must exist.

The size of a list $\mathcal{L}$ is the number of elements (n) in it. It can stay fixed throughout each run of the computation, or it can be changed during the course of the computation by adding new elements to it. In particular, we will refer to the former type of lists as *random access lists* since there are obvious ways of implementing them such that any member can be accessed or replaced in time O (1).

A set $\mathcal{S}$ is a list of n members where no specific order is associated with its elements. We define a single operation, *set-union*, denoted by the symbol $\cup$; this operation takes time O (1). A *map* f is also a specialized kind of list which is defined relative to another list $\mathcal{L}$ (which is not necessarily a

map). In this case, list $\mathcal{L}$ is referred to as the *domain* of map f. Map f has exactly one member corresponding to each $\llbracket_i$ in its domain. We can apply access and replace operations to maps in much the same way as before, with the exception that now, an element $\llbracket_i$ from the domain is specified as the input parameter; the access operation in a map f is denoted by *access* $(f[\llbracket_i])$ and a replace operation can be specific likewise. In particular, we do not provide a mechanism for accessing the members of a map (type list) directly; they are always accessed implicitly by specifying the members of the corresponding domain. It is easy to implement maps in such a way that arbitrary members can be accessed in time $O(1)$ whenever they have random access lists as their domains.

7.1.2 Notation

Assignment is denoted by " := " and ";" is used as a statement separator. We use a single control structure for iterations namely, the **for** ... **rof** statement which is of the form:

for *iterator* → *statement list* **rof**.

The *statement list* is executed exactly once for each value of the iterator; a typical *iterator* is specified through a list $\mathcal{L}$ (or a set), in which case the iteration proceed for each $\llbracket_i \in \mathcal{L}$, in the order of the elements of list $\mathcal{L}$.

We also have procedures which are specified as follows:

procedure *name* (*parameters*); *statements* **end**.

When the parameters are complex objects such as lists, we assume that only a pointer to this object is passed, and the same is true of assignment statements as well. Also, only a constant number of parameters are always passed between procedures. Therefore, independent of the type of objects being passed as parameters, a single procedures call or return can be executed in constant time. This is also true of assignment statements when the arguments are lists, sets and so on.

Finally, variables can be declared to be local to a procedure and if this is done, the same variable name can be used locally in many different procedures in a consistent way. Also, new data types can be constructed from the primitive types such as integers, elements and others. We will be somewhat informal in doing this and in particular, we will not concern ourselves with specific mechanisms for doing this. In some cases, we will declare some of the new data types and explain their structure and role in our implementations during the course of our explanation. In other cases where the structure and role of these new data types is obvious, we will declare them directly in the appropriate procedures without extensive explanation.

7.2 Details of the Algorithm

7.2.1 The Data Structures

The main data structure with which we will be working is a list called *elements*. Each member a'_i of this list is of data type element, and corresponds to a member of the set A from the input instance. We assume that some straightforward preprocessing steps are already done before we start the main algorithm. The first of these steps involves determining the level of each element in the precedence graph P, which is specified as part of the input. This 'level' information is then used during the second preprocessing step to sort *elements* to make sure that its members are arranged in the non-decreasing order of levels. That is, given any two members a'_i and a'_j of *elements*, we wish to make sure that whenever $i < j$, the level of element a'_i is no greater than the level of element a'_j in the corresponding precedence graph P.

We also expect that for each member a'_i in *elements*, the set of all of

its successors is know, or else it is determined as a part of our preprocessing step. This information is stored in *successors*, which is a map; each of its members is of type *set of elements*, and it has *elements* as its domain. We also have a map f' whose members are rational numbers. This map also has *elements* as its domain and therefore, its elements are also accessed through *elements*. Each member of map f' will be used to maintain the value assigned by the modified cost-determining function to the members of elements. To start with, for each member a'_i of *elements*, $f'[a'_i]$ (in the map f') is initialized to the value assigned by the cost-determining function $f(a_i)$, which of course is specified as a part of the input. There are straightforward ways in which all of the above mentioned preprocessing steps can be implemented to run in time $O(max(|\alpha|, n))$ if the precedence graph is specified in its *transitively reduced* form. If not, the transitive reduction (which requires the same time as transitive closure [1]) of the precedence graph can be found in time $O(min(|\alpha| \cdot n,\ n^{2.61}))$, either through depth-first search or by reducing it to matrix multiplication and using an algorithm from [72].

We also define a map, *type*, whose members are integers which also has *elements* as its domain. Given any member of *elements* say a'_i,$type[a'_i]$ represents the 'type' - determined by the class Π_x to which element a'_i belongs in the corresponding set of elements A. We assume that this map is also initialized during the preprocessing step; this step can be implemented in a straightforward way to run in time $O(n^2 m)$. Finally, we use a list *saturation*, whose members are integers and its xth member is used to record the saturation index of class Π_x in the set of elements A. This part of the preprocessing can be done in conjunction with the previous step during which the various elements were grouped into the respective classes Π_x, and it is easy to see - especially since the latter step which involves computing the saturation indices

adds only a constant factor to the overall time complexity of the former - that both these steps can be completed in time $O(n^2 \cdot m)$. All these time complexity figures are based on the uniform cost measure of course, where each action is charged unit cost.

Finally, we assume that the maximum value of $f(a'_i)$ in the input instance is never more than n. If some of these values are initially greater than n for a given instance, we simply replace such values by n; it is very easy to see that this change does not affect the solution to the problem in any way, and on the other hand, it actually results in a much improved algorithm by aiding us in the design of efficient data structures. We also assume that the cost-determining function ranges only over the non-negative integers, whereas we had originally allowed it to range over the non-negative rationals. Even this assumption is aimed at helping us realize an efficient implementation of our algorithm, and given that our concerns are currently restricted to uniformly k-sliced systems, it is not difficult to see that this assumption does not entail any loss in generality.

Let us now start using these data structures towards designing an algorithm which calculates the modified cost-determining function of a given input instance. This calculation is done iteratively, where on the ith iteration, we calculate the value of $f'[a'_i]$. Since we are progressing in list order, and since the members of *elements* at the lower levels in the corresponding precedence graph occur earlier on in this list, by the time we reach element a'_i, we would have already determined the modified cost-determining function values for all elements which occur at lower levels in the given precedence graph. Therefore, on the ith iteration, we merely need to count the number of successors (through map successors) of element a'_i - these 'counts' are grouped

according to the type and d' values where we continue to interpret d' in the same way as we did earlier on. Then, from these count values and given *saturation* from our preprocessing step as well as the degree of slicing k, we can determine $f^*(a_i)$ (see the definition of the modified cost determining function in Chapter 2 for f^*) and subsequently bind $f'[a'_i]$ in a straightforward manner.

Note that an important part of this computation involves maintaining a count of successor elements a'_j of element a'_i, and classifying these 'counts' based on their type and d' values. To this end, we introduce an additional list, *memory* which is a list of M other lists. Each member in this list is a list in itself - in fact a random access list of $(2n + 1)$ integer variables, and whose list indices range from $-n$ to $+n$ (as opposed to from 0 to 2n). Basically, as shown in Figure 7-1, there are as many members in *memory* as there are types of elements and in particular, its xth member list corresponds to the xth element type. Also, each of these $2 . n + 1$ locations of this xth member list correspond to a d' value. Recall that we restricted the cost-determining function to range over the non-negative integers and in addition, we also expect the maximum value which it can assign to any element to be never greater than n. From this, it follows that for any instance, the modified cost-determining function ranges over the integers which are bound above and below respectively by $+n$ and $-n$. This in turn allows us to use the d'th location in the xth member list of *memory* for maintaining a count of all the type-x successors of element a'_i whose modified cost-determining function values are no greater than d'.

Now, on the ith iteration, we consider each element a'_j in *successors* a'_i. From our preprocessing steps, we already know the type, say x, of element

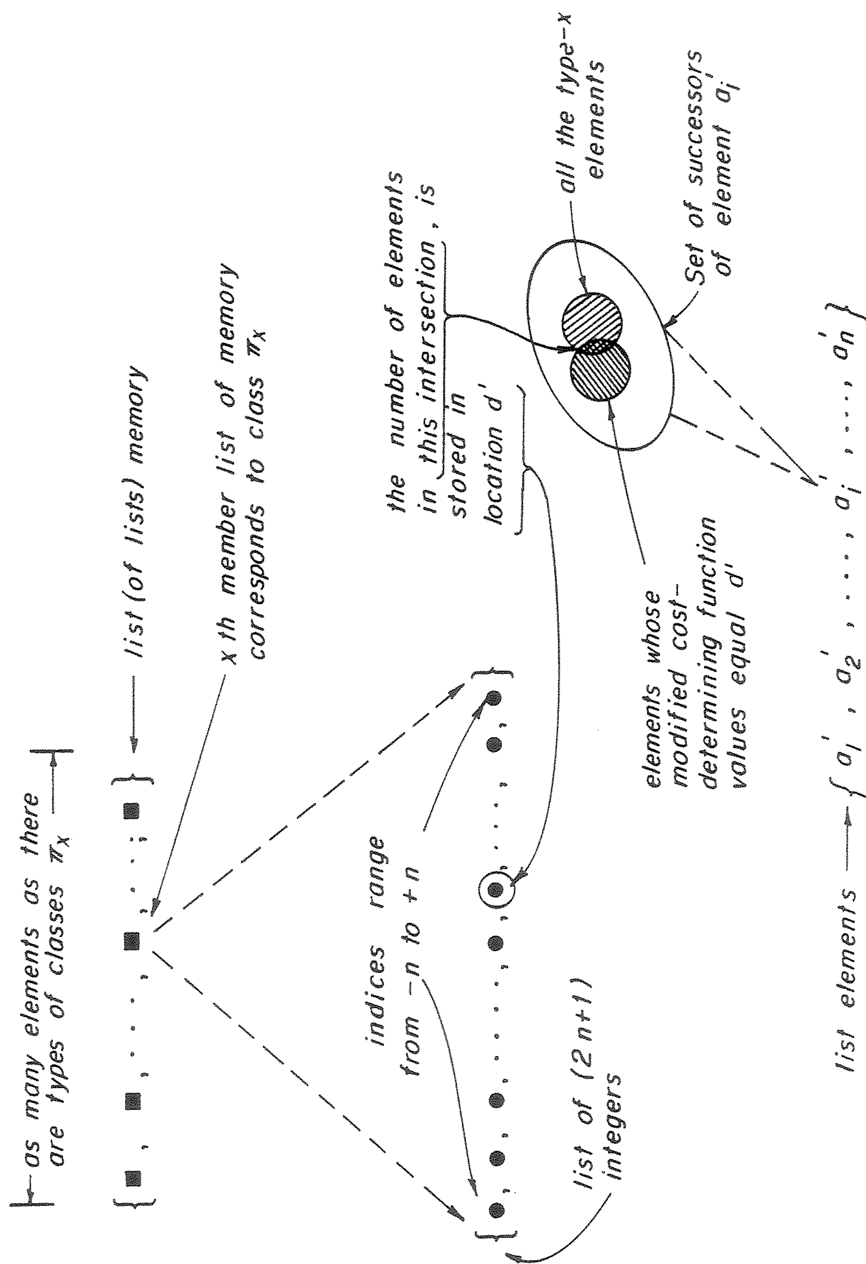


Figure 7-1: The list (of lists) *memory*.

a'_j; also, note that $f'[a'_j] = d'$ must have been computed on a previous iteration if such an element a'_j exists. With this information, we can now record or 'count' element a'_j by simply incrementing the d'th member in the xth member list of *memory*. Once we finish doing this for all the elements in *successors* $[a'_i]$, we are ready to compute $f^*(a_i)$.

To do this, we have to consider each member (list) of *memory* in turn. Let us suppose that we are currently inspecting the xth member of *memory*, which we will denote by *aux* for purposes of our present discussion. Then, for the d'th entry in *aux*, we can easily calculate

$$E = \{d' - (\lceil aux[d']/saturation[x]\rceil) + k - 1\}$$

and therefore the smallest such E, say E_x, computed over all values of d' in this xth member list of *memory*. Then, we repeat this step for each of the M possible members of *memory* to calculate $f^*(a'_i)$, which is essentially the smallest such E_x. Having determined $f^*(a'_i)$, calculating $f'(a'_i)$ only involves an additional comparison and we will be done. Let us now consider this algorithm in greater detail.

7.2.2 The Algorithm

This implementation of the above described algorithm is essentially a precise specification presented in the setting of the notation from Section 7.1. There are certain parts of this implementation which have been added here in the interests of its efficiency, and whose role will be explained subsequently. Basically, there is one major procedure *modify*, which we will now specify. This procedure accesses three other procedures : *update, remember*, and *calculate*, which we will also specify in this section.

```
procedure modify (elements, f', k, saturation, successors, type)
    set of elements current successors; integer δ, SI, E and x;
    for a'_i ∈ elements →
    current successors := successors [a'_i];
    Sort (current successors by f');
        A. for a'_j ∈ current successors →
            1. x := access (type,a'_j); δ := access(f', a'_j);
            2. update (δ, memory, x);
            3. remember (active types, δ, D, x)
          rof;
        B. for x ∈ active types →
            4. active d' := access (D, x);
            5. aux := access (memory, x);
            6. SI := access (saturation, x);
            7. for d' ∈ actived' →
                E := access (aux, d');
                calculate (a'_i, d', elements, f', k, SI, E)
              rof
          rof
        /Reinitialize relevant lists and variables/
    rof
end modify;
```

The first part of this procedure indicated by statements 1 and 2 in loop A essentially represents the 'counting' process. *Current successors* is a set which is used locally in this procedure and on the ith iteration of the outermost **for** loop, it has all the successors of element a'_i from *elements*, as its members. To start with, we sort *current successors* by rearranging its members in the non-decreasing order of their modified cost-determining function values. All these values must have been computed on an earlier iteration of the outermost **for** loop. This reordering of *current successors* will prove to be useful subsequently, in procedure *update*. In statement 1 loop A, we successively assign for each member a'_j in *current successors*, its type and $f'[a_j]$ values respectively to variables x and δ. Then, in procedure *update* which we will

now specify, we increment the appropriate location in *memory* which corresponds to the type and $f'[a'_j]$ values of element a'_j.

procedure update (δ, *memory*, x)
  list of integers *aux, counter*; **integer** δ';
  (i.) $aux := access(memory, x)$; $\delta' := access(aux, \delta)$;
  (ii.) If $\delta' = 0 \rightarrow \delta' := counter[x]; counter[x] := counter[x] + 1$;
  (iii.) $\delta' := \delta' + 1$; $replace(aux, \delta, \delta')$;
  (iv.) $replace(memory, x, aux)$;
end update;

Here, *counter* is a list of n integers which is local to *update*. The xth member of this list is associated with element type x; it will be used to maintain a running count of the number of type-x successors of element a'_i whose modified cost-determining function values are *less* than δ, and which were encountered on earlier iterations of **for** loop A. Then, in statement line *(ii.)* of *update*, we use the information from *counter*, in conjunction with the non-decreasing order which the members of *current successors* obey, to ensure that $aux[\delta]$ (where *aux* is currently assigned the xth member of *memory*) is appropriately updated. Through this, we ensure that when we execute **for** loop B subsequently, the d'th location in the xth member list of *memory* contains an accurate count of the number of type-x successors of element a'_i whose modified cost-determining function values are *no greater* than d'; recall that this value was denoted by the function $n(i,d',x)$ in Chapter 2 in connection with the definition of the modified cost-determining function.

In statement 3, as we consider each member a'_j in loop A, we add its type (say x) to *active types*, which is a set of integers. We also have a list D, each of whose members is a set of integers. The xth member set of D is associated with element type x, and it will be used to store the value: $f'[a'_j]$. Both these actions are executed in procedure *remember* which is specified

below. Through this procedure, we maintain a record of exactly those locations in *memory* which have been updated on the current iteration of the outermost **for** loop. This information will prove to be of great value subsequently since it will help us in limiting our search, when we return to *memory* and compute f^*. In procedure *remember*, *active* d' is a set of integers which is local to it.

```
procedure remember (active types, δ, D, x)
    set of integers active d';
    active types := active types ∪ x;
    active d' := access (D, x);
    active d' := active d' ∪ δ;
    replace (D, x, active d');
end remember;
```

In statements 4 through 7, we scan each of the 'count' values in list *memory* and invoke procedure *calculate* to evaluate f^*, and subsequently the modified cost-determining function f'. In doing this, we are guided by the information we carry in *active types* and the list D. The former serves as an iterator for the **for** loop B which consists of statements 4 through 7. Note that for each member x in *active types*, there is at least one type-x successor of element a'_i whose modified cost-determining function value is being determined currently. In the same way, the information in list D guides us in the **for** loop of statement 7 by ensuring that for each $d' \in D[x]$, there is at least one type-x successor a'_j of element a'_i such that $f'[a'_j] = d'$. Therefore, by using this information form *active types* and D, we avoid unnecessarily searching all the n^2 locations in *memory* for each element a'_i but instead, we visit only those locations which correspond to the type and modified cost-determining function values of at least one of the successors of a'_i. In the next section, we will see that this selective searching of *memory* will improve the worst-case performance of our implementation, when compared to one in which all the locations in *memory* are searched each time.

```
procedure calculate (a'_i, d', elements, f', k, SI, E)
    integer ε and ε';
    ε := {d' − (⌈E/SI⌉ + k − 1)};
    ε' := access (f', a'_i); ε = min (ε, ε');
    replace (f', a'_i, ε)
end calculate;
```

Statements 5 and 6 are respectively accesses of the appropriate members of *memory* and *saturation*. The information from these accesses is passed to procedure *calculate* which is invoked in the **for** loop of statement 7. In this loop, E is the current 'count' from *memory*, which indicates the number of type-x successors of element a'_i whose modified cost-determining function values are bound above by d'. Subsequently, E is used to determine the modified cost-determining function value of the element being currently processed in procedure *calculate*; this value is returned in the map f'. It is easy to see that upon finishing **for** loop B on the ith iteration of the outermost **for** loop (with list *elements* as its iterator), we would have bound $f'[a'_i]$. Also, if element a'_i is a sink element (at level zero), neither loop A nor loop B is executed and the value of $f'[a'_i]$ remains unchanged from its original value. Finally, upon completing loop B, we reinitialize all the relevant variables and data structures such as set *active elements*, list D and so on, in preparation for the next iteration during which we will compute $f'[a'_{(i+1)}]$.

Since it is not difficult to verify that this procedure correctly calculates the modified cost-determining function which is returned in the map f', we will not pursue the issue of proving its correctness at this point. The procedures which we described above are actually straightforward implementations of the recurrence relationship through which the modified cost-determining function was defined in Chapter 2. However, an interesting and

significant aspect of this implementation is the use of *active types* and D, both of which help us in realizing an efficient implementation of this algorithm. We will now address this issue of efficiency and analyze the time complexity of these procedures.

7.2.3 Complexity of Procedure Modify

To analyze procedure modify, let us denote the total cost, which is measured in terms of the number of instructions executed (under the uniform cost measure), by some quantity say $\mathcal{C}$. Let $\mathcal{C}_1$ denote the total cost due to instructions other than those in loops A or B. The sorting step which precedes the execution of **for** loop A dominates this cost since only a constant number of all the other instructions which are neither in loop A nor in loop B are executed for each element, and since each of these instructions cost a constant amount of time, their total cost is no greater than $c_1 \cdot n$ for some positive constant c_1. This means that the total cost due to all the instructions which are neither in loop A nor in loop B, even when we include the 'reinitialization' part at the end of procedure *modify*, is determined by the sorting step where we rearrange the members of *current successors*. Finally, since this step costs us $O(nlogn)$ time each time it is executed, and since it can be executed no more than n times, $\mathcal{C}_1$ is $O(n^2 \cdot logn)$.

Let us now analyze the cost due to the instructions form loops A and B. Let $\mathcal{C}_2$ and $\mathcal{C}_3$, respectively denote the total contributions to the final cost due to the execution of loops A and B. Since transferring control to and from a procedure is a constant time function even when complicated objects are passed as parameters, and since only a constant number of steps are executed during each invocation of procedures *update* or *remember* - a fact which follows from their non-iterative nature - we can conclude that statements 1, 2, or

3 in procedure *modify*, each cost us only a constant amount of time, each time they are executed. Therefore, C_2 is determined by the number of elements in the map successors which is easily seen to be bound above by n^2, or equivalently, C_2 is $O(n^2)$.

Coming to loop B, we see that the cost C_3 is within a constant multiplicative factor of the number of times procedure *calculate* is invoked. This is where the information carried in *active types* and D plays a role. For, without this information, we would have to potentially search each of the $O(n^2)$ locations in *memory* for each element a'_i, whenever this element has successors in the underlying precedence graph. If this is the case, it is easy to construct examples where C_3 grows as $\Theta(n^3)$.

However, by trading off space for time through these two data structures (namely set *active types* and list D), we can improve C_3 by a factor of n. To see this, consider the case when loop B is executed during say the ith iteration of the outermost **for** loop when we are calculating the modified cost-determining function value of element a'_i. In this case, owing to the information available from *active types* and D, the number of times procedure calculate is invoked is no more than the number of successors of element a'_i in the precedence graph. This follows from the following two basic facts. Firstly, in our scheme, we visit a location of *memory* only if it corresponds to at least one of the successors of element a'_i. Secondly, each successor of element a'_i contributes towards updating exactly one of the $O(n^2)$ locations of this list. Therefore, the total number of times that procedure *calculate* is invoked, is no more than $|\alpha^+|$ for a given instance. From this, we can deduce that C_3 is also within a constant multiplicative factor of n^2.

What is important to note is that this latter situation can arise quite often since an instance is not always guaranteed to have a corresponding sequence with zero cost. In fact, it is easy to find instances such that no corresponding sequence has a cost of zero. For example, it was seen that the instance describe in Chapter 2, whose precedence graph and cost determining function were illustrated in Figure 2-6, and for which $\mathbf{l} = k = 1$, $\mathbf{B} = \langle 2 \rangle$, and $\mathbf{C}(a_i) = 1$ for $1 \leq i \leq 12$, does not have any corresponding sequence with zero cost. So, to reiterate an earlier observation, Theorem 3.1 only provides us with a partial characterization of the minimum cost sequences in a feasible set $\mathbf{S}$; for the special case where this minimum cost is zero.

Therefore, to accomplish our goal of characterizing minimum cost sequences, we need to remedy this situation by extending the somewhat specialized inclusion relationship between $\mathbf{S}_\alpha$ and $\mathbf{S}_\emptyset$ which we established in Theorem 3.1. More specifically, we would like to establish a similar relationship between $\mathbf{S}_\alpha$ and $\mathbf{S}_{min}$ where $\mathbf{S}_{min}$ is the subset of the feasible set $\mathbf{S}$ which contains exactly its minimum cost sequences. Moreover, in doing this, it will be very convenient if we can actually build on the partial characterization provided by Theorem 3.1 to arrive at this more general and extended result. We will now introduce a technique which allows us to do just this, and in addition we will also see that it (this technique) is likely to be powerful enough to establish similar characterizations of the optimal behavior of the solutions of a wider class of optimization problems beyond scheduling.

Then, since the total cost C is simply the sum of costs C_1, C_2, and C_3, we conclude that procedure modify runs in time $O(n^2 \cdot log n)$. If the input instance is not given to us in transitively reduced form and if the elements from the input instance are not classified into their various types, we need to factor the complexity of the preprocessing steps into our analysis; in this case the overall complexity of our implementation would be $O(max(min(|\alpha| \cdot n, n^{2.61}), m \cdot n^2))$.

If we use the logarithmic cost measure instead of the uniform cost measure in our analysis, parameters such as the degree of slicing k and others, will enter our complexity figures in an obvious way. In specifying the above implementation of this algorithm, our emphasis was on the clarity of the exposition and as a consequence, some of the data structures and other aspects of our design were motivated from this standpoint. As such, some of our decisions might reflect features which are not absolutely essential in an implementation of an algorithm for computing the modified cost-determining function of a given instance, and by taking advantage of this fact, it is conceivable that alternate implementations can be realized, which have lower space requirements and also with time complexity figures involving smaller multiplicative constants.

7.3 Enumerating Sequences from Lists

Having designed an algorithm for computing the modified cost-determining function, we now move on to the next step of constructing non-decreasing ordered list representations and sequences for the given instance. Given *elements* and f', we can construct an appropriate list representation by reordering the members of *elements* to conform to the non-decreasing ordered requirements of Chapter 5. Recall that *elements* will satisfy this requirement

provided, given any two of its members a'_i and a'_j, if i is less than j, then, $f'[a'_i]$ is no greater than $f'[a'_j]$. This can be accomplished easily by sorting *elements* with the members of map f' as the keys - a task which can be easily implemented to run in time $O(nlogn)$. Since sorting is a well understood problem, we will not go into the details of implementing this step.

The list which is output by the sorting step can then be repeatedly scanned by our 'iterative greedy method' which was described in Chapter 5. We implement this step by maintaining a *count* for each member a'_i; this *count* indicates the number of predecessors which this element has. As each one of these predecessors get selected for inclusion, this count is decremented by one. Finally, when the last of the predecessors of element a'_i is selected on say the qth scan, the count goes down to zero. Then, it follows that element a'_i will be a candidate for the first time on the q'th scan where $q' = (q + k)$; in other words, $E'_{q'}$ is the earliest edge in which this element can be included. This value of q' is recorded in the map *earliest*. Initially, *earliest* $[a'_i]$ is assigned the value zero if element a'_i has no predecessors, and it is assigned $+\infty$ (or some value greater than n) if it does.

Then, we start off by constructing the first edge in the sequence. This is done by picking those members of *elements* which have no predecessors, subject to the bound preserving (or resource) constraints which are derived from the list, *saturation*. We then update the count values of the successors of these members of *elements* which have just been selected for inclusion in the first edge. If the count of some element say a'_i goes down to zero, we update the corresponding location in *earliest* to reflect the fact that this element is a candidate for primary inclusion in any edge which occurs after edge E'_k in the sequence which we are constructing, and move on to construct

the second edge. We construct edges contiguously in this fashion until at some point, the index q of the next edge E'_q to be constructed is less than the smallest value *earliest*, say q', (which indicates that none of the remaining unselected elements can be candidates for primary inclusion in the interval starting with scan q and ending with scan $(q'-1)$. In this situation, we 'jump' ahead and start constructing the sequence from edge $E'_{q'}$ on, by choosing from the elements which are candidates for primary inclusion on the q'th scan, subject to bound preserving constraints. This entire procedure can be implemented to run in time $O(n^2)$.

Note that the sequences (schedules) produced by the above described method represent only a partial specification of the entire sequence when k is greater than unity, since we only select and include the first copies of the various elements. In some situations, we even 'jump' ahead and skip constructing certain intermediate edges which do not have the first copy of any of the members of *elements*. Since all of our sequences are proper (corresponding to non-preemptive schedules), this approach does not pose any problems since under these circumstances, a partial specification of the sequence also tantamounts to a complete specification, albeit implicitly. A more important aspect of this approach towards constructing sequences partially is that through it, we circumvent a serious complexity related problem.

This problem arises because the degree of slicing k is a parameter which is quite independent of n (which denotes the number of elements in the input instance). Therefore, in a given subdomain, k can grow much faster than n; for example, it could be proportional to 2^n for the instances from a given subdomain. As a consequence, the size of a complete specification of the sequence, and consequently the time required to produce it might well grow at a

much faster rate than any polynomial function of n. Therefore, if k grows as an exponential (or faster growing) function of n, then the size of the complete sequence would also grow as an exponential (or faster growing) function in n. Note that in any case, the size of a complete specification of the output is at least proportional to $2^{k'}$ where k' is the number of bits in the binary encoding of k, since the degree of slicing k typically undergoes a logarithmic encoding when it is specified as an input. In this situation, if we assume that a standard binary encoding scheme is used for specifying the input, and if k' is $\Omega(p(n))$ for some polynomial function $p(n)$ of n, it follows trivially that specifying the output completely would require time $\Omega(2^{|input-size|})$ (or more generally, $\Omega(k^{|input-size|})$ if a k-ary encoding scheme is used). This problem of being faced with the potential exponential explosion in the running time is avoided by specifying the sequence partially - as we have outlined in the previous paragraphs - by only constructing the edges which contain the first copies of the various elements.

Chapter 8

Concluding Remarks

Even though much is known about the complexity of various types of scheduling problems, most of this knowledge takes the form of disparate and often incomparable collections of properties which 'map' out the boundary between the polynomially solvable and the NP-hard. This is in stark contrast with other well known classical areas in combinatorial optimization, such as matching theory for example, which are often characterized by deep and unifying theories; despite its obvious importance, this is unfortunately not true of scheduling theory. In this dissertation, we remedy this situation by studying the factors which really influence the complexity of a number of known polynomially solvable subdomains of the important class of precedence constrained scheduling problems. We show that surprisingly enough, the factors which determine the complexity of this problem in many cases - even though they may seem to be quite different at the outset, especially in light of many of the existing results in this field - are really influenced and determined by the single unifying intrinsically ordered convergence property (which we formulated in Chapter 5).

Rather than looking at the influence of the 'natural parameters' of the scheduling problem on its complexity (which seems to indicate that there are intrinsic differences among the reasons at to why individual subdomains of the scheduling problem are polynomially solvable), we look into the factors in-

fluencing the optimal behavior of schedules. This approach revealed that at a deeper level, instances from many of the seemingly different subdomains of the scheduling problem in fact share a striking characteristic; they all have the same 'type' of schedules at optimality.

To establish this fact, we formulated the precedence constrained scheduling problem in the setting of hypergraphs, where a schedule is essentially an appropriately constrained sequence of hypergraph edges. In other words, each edge of the sequence corresponds to a step of the schedule. In this setting, we are able to characterize the above mentioned 'types' of schedules which are common to instances from the various subdomains of the scheduling problem in terms of affined sequences (of hypergraph edges). We do this in Chapters 3 and 4 where we also established several interesting properties of affined sequences.

We build on the optimality and the structure of affined sequences in Chapter 5 by incorporating them into the formulation of IO convergence. This property is essentially a sufficient condition for the polynomial solvability of the precedence constrained scheduling problem. It embodies a set of conditions which ensure that if an instance satisfies this property, then it is not only the case that it (the instance) has associated affined sequences, but it is also guaranteed that we will be able to construct these (affined) sequences in polynomial time. In Chapter 6, we complete our unifying framework by showing that indeed, the apparently unrelated conditions which seem to be determining the polynomial nature of many of the subdomains of the precedence constrained scheduling problem which are of interest to us, are all really specific variants of IO convergence.

Another interesting aspect of intrinsically ordered convergence is that it has an in-built constructive or algorithmic component. The practical implication of this aspect of our unifying property is that if the instances are drawn from a subdomain which satisfies it, we not only know that the optimization problem is polynomially solvable for this instance but in addition, our constructive component immediately gives us an algorithm for doing this. We outline an efficient implementation of this algorithm in some detail in Chapter 7.

If we digress for a moment from polynomially solvable scheduling problems and consider their NP-hard counterparts instead, it turns out that IO convergence gives us some pretty interesting insight in this case as well. Specifically, it is the case that there is no finite upper bound on the size of instances from many of the known NP-hard subdomains [5, 38] which violate this property. So, in addition to giving us insight into the polynomial nature of certain scheduling problems, intrinsically ordered convergence also gives us a definitive and unified basis for understanding the structure that is being lost, as we go from the polynomially solvable to the NP-hard; the former type of scheduling problem is IO convergent whereas the latter explicitly violates it.

While our results give us a unified view of the factors which influence the complexity of many well known precedence constrained scheduling problems, they also yield several interesting open questions. An obvious question that arises in this respect is the following: is it easy to test if a given instance is IO convergent ? This question is important because, while we know that we have an efficient algorithm for determining optimum schedules for the class of instances satisfying our unifying property, the utility of this algorithm in the context of a given instance depends on whether we can efficiently ascer-

tain if it (the instance) actually falls into this class. This problem of testing whether a given instance is IO convergent, is an open question, and based on our experience, we conjecture that it is an (NP-)hard problem. Even if this were to be the case, IO convergence can still guide us towards discovering new and useful polynomially solvable subdomains, which are more general in many senses than the ones that were hitherto known. Note that this happened in our own results in the context of subdomains β_{11} and β_{12} (Chapter 6).

At another level, we can consider extending our unifying scheme to include other subdomains of the scheduling problem as well; the subdomain of *opposing forests* for which a polynomial time algorithm was discovered by Garey, Johnson, Tarjan and Yannakakis [36] is a notable example of a case that does not currently fit into our framework. This seems to be an important direction to pursue in order to resolve the long standing open question - which has interesting theoretical as well as immensely practical ramifications - about the complexity of scheduling arbitrary precedence constrained task systems on a constant number (say three) of processors [35, 36]. Many related and similar open questions arise, which include extending our unifying framework to include more general scheduling models which allow specific *release-times* to be associated with the tasks [34], or even those which allow multiple *release-time deadline* intervals [85, 87].

Another direction that has not been explored much involves the 'pipelined generalization' of the basic scheduling problem. Pipelined systems are interesting and important since they are practical and are also widely used. It has been our experience that this generalization adds significantly to the complexity of the basic scheduling problem and gives rise to useful, intriguing, and non-trivial algorithmic, combinatorial and complexity results. For ex-

ample, we have shown that [60] the tardiness minimization problem goes from being polynomially solvable to NP-hard if we take the subdomain β_5, and rather than having instances with single stage processors, we now have in-tree task systems to be scheduled on just two identical (multistage) pipelined processors, with a polynomial number of stages each.

Another important and challenging problem area that is worth exploring is that of finding fast parallel algorithms, especially for the domain of instances which are IO convergent; this issue includes such questions as the membership of this domain in R-NC or even in NC. A solution to a very specialized version of this question was provided by Vazirani and Vazirani in [93] where they show that the subdomain β_2 is in R-NC, although their results do not draw upon our unifying property but rely on novel algebraic techniques instead.

Finally, we wish to point out that given the explosive growth in activity and in the number of results, both in combinatorial optimization and complexity theory, we were confronted with the problem of having to be selective in choosing the papers which we could directly cite in this dissertation. Primarily, we chose such papers which were significant from a historical perspective within the context of our exposition, and which were most relevant to the scheduling problem. In the process, we could not directly include many recent and beautiful results from both of the above mentioned fields.

However, we have tried to rectify this situation by citing several survey papers and books which collectively give a more complete picture of the activity in combinatorial optimization and computational complexity theory. Also, the recent book entitled 'Combinatorial Optimization - Annotated

Bibliographies' [71] is an excellent addition to this list. It includes broad coverage of a range of recent developments in several exciting areas, some of which are: polyhedral combinatorics, duality for integer optimization, probabilistic analysis and randomized algorithms, and several current topics from computational complexity.

Bibliography

1. A. V. Aho, M. R. Garey, and J. D. Ullman. "The Transitive Reduction of a Directed Graph". *SIAM J. Comp. 1* (1972), 131-137.

2. P. Brucker, M. R. Garey and D. S. Johnson. "Scheduling Equal-Length Tasks Under Treelike Precedence Constraints to Minimize Maximum Lateness". *Math.ofOper.Res. 2* (August 1977), 275-284.

3. G. S. Boolos and C. Jeffrey. *Computability and Logic.* CAmbridge University Press, New York, N. Y., 1980.

4. J.Bruno, J.Jones and K.So. "Deterministic Scheduling with Pipelined Processors". *IEEETrans.Comp. C-29* (April 1980), 308-316.

5. J. Blazewicz, J. K. Lenstra and A. H. G. Rinnooy Kan. "Scheduling Subject to Resource Constraints: Classification and Complexity". *Disc.Appl.Math. 5* (1983), 11-24.

6. R. E. Burkard. Travelling Salesman and Assignment Problems: A Survey. In *Ann. Disc. Math.*, North-Holland, Amsterdam, 1974, pp. 193-215.

7. A. Cobham. The Intrinsic Computational Difficulty of Functions. In *Proc. 1964 International Congress for Logic Methodology and Philosophy of Science*, North-Holland, Amsterdam, 1964, pp. 24-30.

8. E. F. Codd. Relational Completeness of Database Sublanguages. In *Data Base Systems*, R. Rustin, Ed., Prentice Hall, Englewood Cliffs, NJ, 1972, pp. 65-98.

9. E. G. Coffman Jr. and R. Graham. "Optimal Scheduling for Two-processor Systems". *ActaInform. 1* (1972), 200-213.

10. N. Christofides. *Graph Theory: An Algorithmic Approach.* Academic Press, New York, 1975.

11. S. A. Cook. The Complexity of Theorem-proving Procedures. In *Proc. Ann. ACM Symp. on Theory of Computing*, Association for Computation Machinery, New York, 1971, pp. 151-158.

12. S. A. Cook. "A Hierarchy for Nondeterministic Time Complexity". *J.Comput.SystemSci.* *7* (1973), 343-353.

13. R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling.* Addison-Wesley, Reading, MA, 1967.

14. E. G. Coffman Jr. (Ed.). *Computer and Job/Shop Scheduling Theory.* John Wiley, New York, 1976.

15. N. J. Cutland. *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press, New York, 1980.

16. M. Davis (Ed.). *The Undecidable.* Raven Press, Hewlett, N. Y., 1965.

17. R. P. Dilworth. "A Decomposition Theorem for Partially Ordered Sets". *Ann.Math.* *51* (1950), 161.

18. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

19. J. Edmonds. "Paths, Trees and Flowers". *Canad.J.Math.* *17* (1965), 449-467.

20. J. Edmonds. "Matroids and The Greedy Algorithm". *Math.Prog.* *1* (1971), 127-136.

21. J. Edmonds and E. L. Johnson. Matching: A Well Solved Class of Integer Linear Programs. In *Combinatorial Structures and Their Applications*, Gordon and Breach, New York, NY, 1970, pp. 89-92.

22. S. Even. *Graph Algorithms.* Computer Science Press, Potomac, MD, 1979.

23. M. Fujii, T. Kasami and K. Ninomiya. "Optimal Sequencing of Two Equivalent Processors". *SIAMJ.Appl.Math.* *17* (1969), 784-789. Erratum, SIAM J. Appl. Math., 20 (1971) p. 141.

24. M. Florian, W. Maxwell, and H. D. Ratliff (eds.). "Scheduling". *Oper. Res.* *26* (1978).

25. M. J. Fischer and M. O. Rabin. Super-exponential Complexity of Presburger Arithmetic. In *Complexity of Computation*, American Mathematical Society, Providence, R. I., 1974, pp. 27-41.

26. G. N. Fredrickson and M. A. Srinivas. Data Structures for On-line Updating of Matroid Intersection Problems. In *Proc. Sixteenth ACM Symp. Theo. Comp.*, IEEE Computer Society, Washington, DC, 1984, pp. 383-390.

27. G. N. Fredrickson and M. A. Srinivas. On-line Updating of Solutions to A Class of Matroid Intersection Problems. to appear.

28. L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

29. H. Gabow. "An Almost Linear Algorithm for Two-processor Scheduling". *J.Assoc.Comput.Mach. 29* (1982), 766-780.

30. D. Gale. *The Theory of Linear Economic Modes*. McGraw-Hill, New York, NY, 1960.

31. B. Carre. *Graphs and Networks*. Clarendon Press, Oxford, 1979.

32. H. N. Gabow and R. E. Tarjan. A Linear-time Algorithm for a Special Case of Disjoint Set Union. Proc. Fifteenth Ann. Symp. Theo. Comp., 1983, pp. 246-251.

33. M. R. Garey and D. S. Johnson. "Scheduling Tasks with Nonuniform Deadlines on Two Processors". *J.Assoc.Comput.Mach. 23* (July 1976), 461-467.

34. M. R. Garey and D. S. Johnson. "Two-processor Scheduling with Start Times and Deadlines". *SIAMJ.Comput. 6* (1977), 416-426.

35. M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman &Co., San Francisco, 1979.

36. M. R. Garey, D. S. Johnson, R. E. Tarjan and M. Yannakakis. "Scheduling Opposing Forests". *SIAMJ.Alg.Disc.Meth. 4* (March 1983), 72-93.

37. D. Gale, H. W. Kuhn, and A. W. Tucker. On Symmetric Games. In *Ann. Math. Stud.*, Princeton University Press, Princeton, NJ, 1950.

38. R. L. Graham, E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling : A Survey. In *Ann. Disc. Math.*, North-Holland, Amsterdam, 1979, pp. 287-326.

39. D. K. Goyal. Scheduling Processor Bound Systems. CS-76-036, Computer Science Department, Washington State University, Pullman, Washington, 1976.

40. R. E. Gomory. "Outline of an Algorithm for Integer Solution to Linear Programs". *Bull.Amer.Math.Soc. 64* (1958).

41. M. Grotschel and M. W. Padberg. Linear Characterization of the Symmetric Travelling Salesman Polytope. 7417, Institut fur Okonometrie und Operations Reseatrch, University of Bonn, 1974.

42. M. Grotschel and M. W. Padberg. On the symmetric travelling salesman problem I: Inequalities, II: Lifting Theorems and Facets. RC6820, 6821, IBM Thomas J. Watson Research Center, 1977.

43. H. N. Gabow and R. E. Tarjan. "Matroidal Intersection Problems". *J.Algorithms 5* (1984), 80-131.

44. D. Hilbert and S. Cohn-Vossen. *Geometry and the Imagination.* Chelsea Publishing Company, New York, NY, 1952.

45. J. Hartmanis, P. M. Lewis, and R. E. Stearns. Classification of Computations by Time and Memory Requirements. In *Proc. IFIP Congress 1965*, IFIP, Spartan, NY, 1965, pp. 31-35.

46. J. Hartmanis and R. E. Stearns. "On the Computational Complexity of Algorithms". *Trans.Amer.Math.Soc. 11* (1965), 285-306.

47. J. T. Hopcroft and R. E. Tarjan. "Efficient Planarity Testing". *J. Assoc. Comp. Mach. 21* (1974), 549-568.

48. T.C.Hu. "Parallel Sequencing and Assembly Line Problems". *Oper.Res. 9* (1961), 841-848.

49. D. S. Johnson. "The NP-completeness Column: An Ongoing Guide". *J.Algorithms 4* (1983), 189-203.

50. R. M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85-103.

51. R. M. Karp. The Probabilistic Analysis of Some Combinatorial Search Algorithms. In *Algorithms and Complexity: New Directions and Recent Results*, J. Traub (ed.), Academic Press, New York, 1976, pp. 1-19.

52. R. M. Karp, and C. H. Papadimitriou. On Linear Characterizations of Combinatorial Optimization Problems. Proc. Twenty First Ann. Symp. Foundations Comp. Sci., 1980, pp. 1-9.

53. P. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, New York, NY, 1981.

54. V. Klee. "Combinatorial Optimization: What is the State of the Art". *Math.Oper.Res. 5* (1980), 1-26.

55. J. Krarup and P. M. Pruzan. Selected Families of Discrete Location Problems. In *Discrete Optimization*, Ann. Discrete Math., 1978, pp. 327-387.

56. R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-matching Algorithms. to appear.

57. K. Kennedey and J. Schwartz. "An Introduction to the Set Theoretical Language SETL". *Comp. Math. Appl. 1* (1975), 97-119.

58. K. Kuratowski. "Sur le Probleme des Courbes Gauches en Topologie". *Fund. Math. 15* (1930), 271-283.

59. R. M. Karp, E. Upfal and A. Wigderson. Constructing a Perfect Matching is in Random NC. In *Proc. Seventeenth Ann. ACM Symp. on Theo. Comp.*, ACM, Providence, RI, 1985, pp. 22-32.

60. K. V. Palem. Complexity of Scheduling Problems. Unpublished results.

61. E. L. Lawler. *Combinatorial Optimization Networks and Matroids.* Holt, Rinehart, and Winston, New York, 1976.

62. D.W.Leinbaugh. "Guaranteed Response Times in a Hard-real-time Environment". *IEEETrans.Soft.Engr. SE-6* (1980), 85-91.

63. J. K. Lenstra (Chairman). Report of the Session on : Algorithms for a Special Class of Combinatorial Optimization Problems. In *Ann. Disc. Math.*, North-Holland, Amsterdam, 1979, pp. 295-299.

64. R. Lipton. The Reachability Problem Requires Exponential Space. 62, Yale University, 1976.

65. B. J. Lageweg, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Computer Aided Complexity Classification of Deterministic Scheduling Problems. BW 138/81, Mathematisch Centrum, Amsterdam, The Netherlands, 1981.

66. B. J. Lageweg, E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy kan. "Computer Aided Complexity Classification of Combinatorial Problems". *Comm.ACM 25* (1982), 817-822.

67. E. Minieka. *Optimization Algorithms for Networks and Graphs.* Marcel Dekker, New York, 1978.

68. C. L. Monma. "Linear-time Algorithms for Scheduling on Parallel Processors". *Oper. Res. 30* (1982), 116-124.

69. A. R. Meyer and L. J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Time. In *Proc. 13th Ann. Symp. on Switching and Automata Theory*, IEEE Computer Society, Long Beach, CA, 1972, pp. 125-129.

70. S. Micali and V. V. Vazirani. An $0(\sqrt{|V|} \cdot |E|)$ Algorithm for Finding Maximum Matchings in General Graphs. In *Proc. Twenty-first Ann. Symp. Found. Comp. Sci.*, IEEE Computer Society, Long Beach, CA, 1980, pp. 17-27.

71. M. O'hEigeartaigh, J. K. Lenstra and A. H. G. Rinnooy Kan (Ed.). *Combinatorial Optimization - Annotated Bibliographies.* John Wiley, New York, 1985.

72. V. Y. Pan. Field extension and trilinear aggregating, uniting and cancelling for the acceleration of matrix multiplications. In *Proc. 20th Ann. IEEE Symp. Found Comp. Sci.*, IEEE Computer Society, San Juan, Puerto Rico, 1979, pp. 28-38.

73. C. H. Papadimitriou and K. Steiglitz. "On the Complexity of Local Search for the Traveling Salesman Problem". *SIAMJ.Comp. 6* (1977), 76-83.

74. C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

75. C. Papadimitriou and M. Yannakakis. "Scheduling Interval-ordered Tasks". *SIAMJ.Comput. 8* (August 1979), 405-409.

76. M.O.Rabin. Probabilistic algorithms. In *Algorithms and complexity: New Directions and Recent Results*, J. Traub (ed.), Academic Press, New York, 1976, pp. 21-39.

77. C. V. Ramamoorthy and H. F. Li. "Pipeline Architecture". *ACMComp.Sur. 9* (1977), 61-102.

78. C.V.Ramamoorthy and H.Li. Sequencing Control in Multifunctional Pipeline Systems. Proc. Sagamore Comp. Conf. on Parl. Proc., 1975, pp. 78-79.

79. H. Rogers, Jr.. *Theory of Recursive Functions and Effective Computability*. Mc-Graw Hill, New York, N. Y., 1967.

80. J. R. Robacker. Min-max Theorems on Shortest Chains and Disjoint Cuts of a Network. RM-1660-PR, The Rand Corporation, 1956.

81. S.Sahni. "Scheduling Multipipeline and Multiprocessor Computers". *IEEETrans.Comp. C-33* (1984), 637-645.

82. R. Sethi. "Scheduling Graphs on Two Processors". *SIAM J. Comp. 5* (1976), 73-82.

83. J. I. Seiferas, M. J. Fischer, and A. R. Meyer. "Separating Nondeterministic Time Complexity Classes". *J.Assoc.Comp.Mach. 25* (1978), 146-167.

84. S.Su and K.Hwang. Multiple Pipeline Scheduling in Vector Supercomputers. In *Proc. 1982 International Conf. Parallel Processing*, IEEE Computer Society, Los Angeles, California, 1982, pp. 226-231.

85. B. Simons. A Fast Algorithm for Multiprocessor Scheduling. In *Proc. 21st Ann. Symp. on Foundations of Comp. Sci.*, IEEE Computer Society, Los Angeles, California, 1980, pp. 50-53.

86. R. Solovay and V. Strassen. "A Fast Monte-Carlo Test for Primality". *SIAM J. Comput. 6* (1977), 84-85.

87. B. Simons and M. Sipser. On Scheduling Unit-time Jobs with Multiple Release Time/Deadline Intervals. RJ3236, IBM Research Laboratory, San Jose, California, 1981.

88. M. N. S. Swamy and K. Thulasiraman. *Graphs, Networks, and Algorithms*. John Wiley, New York, 1981.

89. R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1983.

90. R. E. Tarjan. "Efficiency of a Good but Not Linear Set Union Algorithm". *J. Assoc. Comp. Mach. 22* (1975), 215-225.

91. A. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". *Proc. Lond. Math. Soc. Ser. 2 42 and 43* (1936), 230-265 and 544-546.

92. J. D. Ullman. "NP-Complete Scheduling Problems". *J.Comp.Sys.Sci. 10* (1975), 384-393.

93. U. V. Vazirani and V. V. Vazirani. The Two Processor Scheduling Problems in R-NC. In *Proc. Seventeenth Ann. Symp. Theo. Comp.*, ACM, Providence, RI, 1985, pp. 11-21.

94. J. Von Neumann. The Duality Theorem. Private Notes, 1947.

95. H. Whitney. "On the Abstract Properties of Linear Dependence". *Amer. J. Math. 57* (1935), 509-533.