

*To my parents.
And especially to Anne.*

Prolog

According to Slack et al. (1998) a project is a unique task to be accomplished, which may consist of a hierarchy of individual subtasks, the smallest of which are called activities. Each of the activities requires a certain time for execution and a set of resources, which are taken to be scarce. Among these are most commonly raw materials, machines, human labor, money or time. While restricted by the limited amount of resources the project seeks to optimize a certain objective: In many cases this is the minimization of the project duration, albeit other objectives like meeting customer due dates, minimizing job lateness, maximizing resource utilization or maximizing the net present value of a project might be of relevance in different contexts (Russell and Taylor III 2000).

Furthermore often precedence relationships are given between some activities, implying an order of their execution. Because the execution time of each activity is fixed, precedence relationships can be modeled as a minimal time lag between the start of two activities. Likewise maximal time lags may also exist (Neumann, Schwindt, and Zimmermann 2002), meaning that an activity has to be released within a specified time window from the start of another activity: Take the processing of a chemical substance, for example, where each step must be carried out in a timely manner in order to prevent unintentional reactions with the environment.

Scheduling then, in its most simple understanding, implies the temporal shifting of activities, so that no (temporal or resource) constraints are violated and the objective function is optimized. Scheduling also involves uncertainty as projects are planned before they are executed. The risk that unexpected events might endanger the feasibility of the initial schedule is inevitable, but cannot be overcome. This is the nature of project management.

Due to the complexity of project scheduling itself most scheduling procedures are only concerned with an ex ante solution to the problem, rather than incorporating uncertainty. However, as early as 1959 Malcolm et al. proposed the use of three estimates for the activity durations (an optimistic, a most likely and a pessimistic estimate). Later works, as the pioneering one from

Zadeh (1965), recommend the use of fuzzy models. As uncertainty will be of no special interest in this text, the reader is referred to Demeulemeester and Herroelen (2002), who provide a comprehensive overview of the topic.

Each project is associated with a life cycle that consists of different phases: In the first phase the project is to be *defined* in its scale, scope, goals, performance measures, members and their work content. Once the project is properly defined a *structural analysis* can take place, which finds the hierarchical decomposition of the project in partial projects, sub projects, working packages and eventually activities. Notice that even an activity might be broken down into several substeps, whose order of execution is given exogenously, however, and therefore not subject to scheduling. The structural analysis also provides the (precedence) relationships between the activities and is followed by the *temporal analysis*. Here the durations of the activities are estimated and, if applicable, also the minimal and maximal time lags between them. In the case of limited resources the capacities and amounts of each resource have to be determined prior to the calculation of the feasible start and finish times for each activity in the *scheduling phase*. Then the theoretical considerations come to an end and the project has to be turned into reality. The *execution phase* starts with the project kick-off and includes, if necessary, monitoring and corrective actions. The life cycle concludes with the deliverance of the project results, i.e. the final product or service, in the *termination phase*.

Academics have been developing network-based scheduling methods since the late fifties (Kelley and Walker 1959) and with the increasing complexity of projects the interest in the field is booming. Bounds (1998) estimates that project management is an \$850 million industry that is expected to grow by as much as 20 percent per year. Hence the need for efficient and effective project management is evident. Since the first temporal-only considerations in the late fifties project scheduling has come a long way and now project managers can choose from a wide variety of software packages for resource-constraint scheduling problems (cf. Keßler and Trautmann 2004). Although this software substantially assists the planner, exact solutions of many of the problems remain to be computationally hard, even if calculated on far more advanced computers than available today.¹ Research on generic solution procedures, which do not incorporate the special structure of the unique problems arising in practice, is very unlikely to yield the boost in algorithmic performance needed for future applications. This book is

¹Computational intractability is dealt with in section 1.3.

therefore dedicated to one scheduling problem among the many, which is believed to be of high usefulness and relevance in practice.

Outline of Book

This paper focuses on the discussion of algorithms specifically designed to optimally solve non-preemptive scheduling problems involving independent activities with dedicated resources.² At first this class of scheduling problems is introduced by means of several practical examples, where key characteristics of the problem structure are identified. Next, chapter 1 locates the problem within the general resource constraint project scheduling framework and prepares the reader with some basic terminology and a brief introduction to the theory of computational complexity. Chapter 2 presents an overview of related problems and solution procedures. In particular, an integer linear program formulation is given and the problem is classified within the machine scheduling problems. Furthermore the branch-and-bound methodology is introduced while presenting the first exact solution procedure proposed in literature. The remainder of this paper will discuss graph-based solution approaches of the problem. Chapter 3 provides the therefore necessary foundation, introduces a graph representation of the problem and derives immediate results from it. Among these are two problem decompositions and the identification of a generic lower bound. Chapters 4 and 5 present two different graph algorithmic approaches in detail, which are believed to be the most powerful solution procedures to be known. A special emphasis will be put on the presentation of chapter 5, however, which includes several aspects new to this book. The paper concludes in chapter 6 with a thorough performance analysis of both of the latter solution procedures. To this extend both algorithms have been coded in a way that allows for comparison and are thereafter employed to derive experimental test data. This independent analysis of both solution procedures by means of computational analysis is also new to literature and reveals some aspects masked by previous contributions.

²All terms will be defined in following.

Contents

Introduction	1
1 Preliminaries	5
1.1 CSP in the RCPSP framework	5
1.2 Terminology	6
1.3 The Theory of NP-Completeness	9
2 Related Problems and Solution Procedures	13
2.1 Integer Linear Programming Approach	13
2.2 Machine Scheduling	15
2.2.1 Preemption vs. Non-Preemption	16
2.2.2 Single- vs. Multiprocessor	16
2.2.3 Dedication vs. Non-Dedication	17
2.3 A Pioneering Branch-and-Bound Approach	18
2.3.1 The General Branch-and-Bound Methodology	18
2.3.2 Bounding Techniques for the CPJS	21
2.3.3 The Branch-and-Bound Process	22
3 Graph Representation	27
3.1 Graph-Theoretic Foundations and Notation	27
3.1.1 Basic Definitions	28
3.1.2 (Data) Representation of Graphs	32
3.1.3 Vertex Sequences	35
3.2 Graph Representation of CSP and Immediate Conclusions	37
3.2.1 Constraint Graph	37
3.2.2 Complexity of CSP	38

3.2.3	Decomposition Graph and Trivial Decomposition Theorems	39
3.3	The Maximum Weighted Clique Problem	42
3.3.1	Sequential Greedy Heuristics	43
3.3.2	Local Search Heuristics	43
3.3.3	Randomized Heuristics	44
3.3.4	Tabu Search	44
4	Interval Coloring	47
4.1	Finding the chromatic number: A brief survey	47
4.1.1	Brown's Basic Algorithm	47
4.1.2	Improvements to Brown's Basic Algorithm	50
4.2	From Simple Coloring to Interval Coloring	52
4.2.1	Applying Simple Coloring to Weighted Graphs	52
4.2.2	Interval Coloring vs. Weighted Coloring	53
4.3	Upper Bounds of the Interval Chromatic Number	56
4.4	An Exact Interval Coloring Procedure	59
4.4.1	Example	62
5	Comparability Augmentation	65
5.1	Introduction to Comparability Graphs	65
5.1.1	Comparability Graphs, Posets and Transitivity	65
5.1.2	Implication Classes	66
5.1.3	G-Decompositions and Comparability Graph Recognition	68
5.1.4	The Knotting Graph and Comparability Graph Recognition	74
5.1.5	Comparability Graphs and Interval Graphs	77
5.2	Comparability Graphs and CSP	78
5.2.1	The Maximum Weighted Clique Problem Revised	78
5.2.2	Comparability Supergraphs	81
5.3	Decomposition	83
5.3.1	Comparability Decomposition Graphs	83
5.3.2	Edge Classes and the Canonical Decomposition	84
5.3.3	Algorithmic Aspects of Graph Decomposition	87
5.4	A BaB Comparability Augmentation Procedure	89

5.4.1	Example	94
5.5	Restricting Edge Choices	95
5.5.1	Triangular Chords	95
5.5.2	Triangular Chords and Golumbic’s Algorithm	96
5.5.3	Splitting the Knotting Graph	96
5.5.4	Autonomous Sets	98
5.6	Further Performance Enhancements	100
5.6.1	Ranking Eligible Edges	100
5.6.2	Adaptive MWC Heuristic	101
5.6.3	Eligible Edges of Subgraphs	102
5.6.4	Depth, Breadth and Scattered Search	102
6	Computational Analysis	105
6.1	Test Instances and Environment	105
6.2	Interval Coloring	106
6.2.1	Calculation Time and Goodness of Outcome	106
6.2.2	BaB Related Data	111
6.3	Comparability Augmentation	113
6.3.1	Calculation Time and Goodness of Outcome	113
6.3.2	BaB Related Data	118
6.3.3	Employing Decomposition	121
6.3.4	Employing Scattered Search	123
	Conclusion and Outlook	125
	References	127
	List of Symbols	137
	List of Abbreviations	141

Introduction

The subject of the matter of this paper is probably best introduced by the following informal example:

Suppose that you are to plan an important academic conference that usually enjoys lively interest. You are expecting many researchers to participate and many talks to be given. Unfortunately the great minds willing to come to your conference are very limited in time and therefore would like to have the conference as dense as possible, i.e. with the least amount of slack time in between their particular talks of interest. Even more importantly, however, none of the academics wants to miss out on a desired talk just because it overlapped with another speech they attended. In this case you would be blamed for the poorly scheduled conference. A disaster and shame you certainly seek to avoid! In order to prevent such conflicts in your schedule you have asked each researcher beforehand which talks he would like to attend and hence the duration and attendees of each talk are known. Obviously all speeches attended by a distinct participant cannot be held simultaneously. Two talks are therefore in conflict, if one researcher wants to attend both of them. Furthermore you know from experience that the participants have no preference on the order of the talks, which means that no precedence relationships exists between the talks.

Now that you have gathered this information, your challenge is to derive a timetable which is free of conflict, but yet concise.³ At first glance the problem does not seem too complicated: You know the length of each talk, which talks are in conflict and that the talks can be shifted freely in time. But as soon as you start working, you realize that your task is much harder than you thought...

Due to the above application the problem to be investigated in this paper has been named the *Conference Scheduling Problem (CSP)* (Roemer 2004a). However, the practical relevance is not limited to this scope. As will soon be understood, all of the following examples give rise to an equivalent problem structure:

³Remember, researchers are always under time pressure !

- Movie productions:
Scenes with the same actors have to be filmed sequentially.
- Formation of project teams:
Projects requiring the same highly specialized team member cannot be undertaken at the same time.
- Diagnosable microprocessor computer systems:
A job must be performed on parallel processors in order to detect fault (Krawczyk and Kubale 1985)
- Manufacturing:
Two jobs require the availability of the same unique resource.

In fact the last problem originally motivated extensive research in the area and is generally known as the *job shop scheduling problem (JSSP)*, which will be discussed in great detail later (see 2.2).

To address the commonalities among the examples one has to identify the smallest indivisible units (activities) first that are subject to the scheduling. Recall that an activity is indivisible to scheduling, but may itself involve a finite amount of executions steps which have to be carried out in a fixed order. At the conference, for instance, an activity is represented by the individual talks. It is easy to see that the talks may be scheduled arbitrarily, but the sentences within a talk certainly not.

Without further ado the following *key characteristics of CSP* can be identified and should be kept in mind throughout the text:

Connotation 0.1 (Characteristics of CSP). *The key characteristics of the class of scheduling problems represented by CSP are:*

1. *A priori information about the duration of each activity.*
2. *No precedence constraints are imposed between the activities: each can be executed independently.*
3. *Mutual conflicts between (some) activities limit the concurrence in processing.*
4. *All activities have be carried out without preemption.*

The non-preemption property (4.) has not been mentioned before because it evolves naturally from the context of CSP. Generally, *preemption* implies that an activity can be executed intermittently at no additional cost. Obviously this does not hold for CSP since the tasks cannot (should not!) be interrupted arbitrarily. Albeit this property seems almost not noteworthy, it will be very decisive in a later context. Knowing the characteristics of CSP, the reader may easily identify the 'activities' in the above examples and - presuming the non-preemption property holds - verify that they indeed share the same essential characteristics.

Chapter 1

Preliminaries

1.1 CSP in the RCPSP framework

The well known *resource constraint project scheduling problem*, abbreviated as *RCPSP/max* (Brinkmann and Neumann 1996), is a very general approach to scheduling and has been studied extensively in literature (cf. Neumann, Schwindt, and Zimmermann 2002). In the three field project scheduling notation of Brucker et al. (1999) it is denoted by $PS|temp|C_{max}$, where PS refers to resource constraint project scheduling, $temp$ to temporal constraints given by minimum and maximum time lags and C_{max} to the objective of minimizing the project duration or makespan. One important difference between CSP and *RCPSP/max* is already evident from this notation: In *RCPSP* temporal constraints are assumed, whereas in CSP the activities can be executed independently. However, the reason why CSP is of particular interest among the variety of scheduling problems unified under the roof of *RCPSP* is yet due to another property of it, namely the mutual incompatibilities between activities. To understand how these incompatibilities can be modeled in the *RCPSP* framework some notation has to be introduced.

Let S be a schedule, i.e. a fixed assignment of start-times to each activity, then the *active set*, $A(S, t)$, contains all activities which are in progress at time instant $t = 1 \dots T$. Furthermore each activity $i = 1 \dots N$ is associated with a finite set of resources $\mathcal{R}_i \in \mathcal{P}(\mathcal{R})$, where $\mathcal{P}(\mathcal{R})$ is the power set of all resources. The resources are generally believed to be renewable, which means that $R_j \in \mathbb{N}$ units of resource $j = 1 \dots K$ are available at every t . R_j is referred to as the *capacity* of j . Conversely each activity i requires $r_{i,j}$ units of resource $j \in \mathcal{R}_i$ per time instant t

during its execution. Thus the schedule S is called resource-feasible if and only if

$$\sum_{i \in A(S,t)} r_{i,j} \leq R_j \quad \text{for all } j, t \quad (1.1)$$

In particular, any set of activities, Φ , relying on the same resource j , can possibly be run simultaneously, if their cumulative demand for j does not exceed the capacity R_j . A feasible schedule S must therefore yield

$$\Phi \in A(S,t) \quad \Rightarrow \quad \sum_{\phi \in \Phi} r_{\phi,j} \leq R_j \quad \text{for all } t, j \in \bigcap_{\phi \in \Phi} \mathcal{R}_\phi \quad (1.2)$$

With mutual incompatibilities, however, it is implied that every resource j has the smallest possible capacity of $R_j = 1$. Consequently (1.1) can never be fulfilled when $\bigcap_{i \in A(S,t)} \mathcal{R}_i \neq \emptyset$, i.e. activities running in parallel share a common resource. It is due to this property, that resources having a capacity of one unit only are called *unique* or *dedicated*. More generally, two activities α and β are said to be in *conflict* or *incompatible* iff $\mathcal{R}_\alpha \cap \mathcal{R}_\beta \neq \emptyset$; otherwise they are *compatible*. In analogy to (1.2) a feasible schedule S under unique resource constraints must obey

$$\Phi \in A(S,t) \quad \Rightarrow \quad \bigcap_{\phi \in \Phi} \mathcal{R}_\phi = \emptyset \quad \text{for all } t \quad (1.3)$$

Sometimes this formal description might appear awkward when applied in practice, but indeed it incorporates the notion of CSP's third key characteristic. When scheduling a conference, for example, the attendees are the unique 'resources' whose 'capacity' is limited to one conference at a time. Thus talks (corresponding to the activities) are only incompatible iff they share a participant.

In summary, CSP can be restated as 'non-preemptive scheduling of independent activities with dedicated resources', which is denoted by $PSm, 1, 1 | * | C_{max}$ in the previously introduced threefold notation. In particular, $m, 1, 1$ stands for an arbitrary number of (renewable) resources, the maximum capacity of any resource and the maximal resource demand of any activity, respectively, whereas $*$ signifies that no temporal constraints are imposed on the activities.

1.2 Terminology

To ensure a common understanding throughout this book it is necessary to define an unambiguous terminology. Unfortunately it has been inevitable to use some of the terms to be defined in

previous sections already. These terms have been left unspecified earlier for the sake of readability, as their broader understanding was thought to be sufficient for introductory purposes. The definitions to come have been adapted from Garey and Johnson (1979):

A *problem* is a general question to be answered, usually processing several *parameters* or free variables, whose values are left unspecified. Each problem is then defined through

1. a description of all its parameters and the range of their values
2. a property of what requirements the answer, called *solution*, must satisfy

An *instance* of a problem is given through proper specific values for all of the problem parameters, whereas a step-by-step solution procedure for a problem is referred to as an *algorithm*. One might think of an algorithm as a computer program being written in a precise computer language. For the reader's benefit, algorithms will be described in *pseudo-code* here, which outlines the general steps of an algorithm in a high-level programming language similar to BASIC, PASCAL or C, but does not include all (specialized) instructions that would be necessary in an actual executable program. An algorithm is said to *solve* a problem if it is guaranteed to produce a solution to any given problem instance. In particular this includes that the execution of the algorithm only involves a finite amount of steps for *all* instances.

Although 'efficiency' of an algorithm includes, in its broadest sense, all the various computing resources used for executing it, only a single resource is considered here, namely time.¹ It is assumed that the time required for the execution of an algorithm corresponds to the 'size' of a problem instance. To firm this notion, presume all input data has to be described by *strings*, formed from a finite symbol alphabet using a fixed *encoding scheme*.² Hence the *size* of an instance can be defined as the number of all symbols needed to describe its parameters.

A *time complexity function* of an algorithm expresses the time requirements for each size of a problem instance by providing an upper bound on the computation time needed to solve this instance. As the size of an instance depends on the specific alphabet and encoding scheme used, so does the time complexity function of course. However, this has little effect on the understanding of 'computational hardness', as will be seen later. A function $f(n)$ is element of $O(g(n))$, whenever there exist constants c and n_0 , so that $|f(n)| \leq c \cdot |g(n)|$ for all $n \geq n_0$. A *polynomial time algorithm* can then be defined as one whose time complexity function is

¹The only exception will be made when referring to pseudo-polynomial complexity, to be defined below.

²On the lowest level this alphabet is binary (base 2 alphabet), consisting of $\{0,1\}$ only. Bases must be fixed and not equal 1.

element of $O(p(n))$ with $p(n)$ being a polynomial function and n the size of the problem. If the time complexity function cannot be bound in such a way the corresponding algorithm has *exponential time complexity*. This distinction is especially relevant when considering the solution of large problem instances and was first discussed by Cobham (1964) and Edmonds (1965). In particular the latter referred to polynomial time algorithms as *good* algorithms and conjectured that certain integer programming problems might not be solvable in polynomial time.³ There is a wide agreement that a problem has not been 'well solved' until a polynomial time algorithm is known for it. Indeed, most exponential time algorithms are merely based on exhaustive search of the solution space, whereas polynomial time algorithms are generally only made possible through the gain of some deeper insight into the problem structure. It should also be noted that some algorithms are said to have *pseudo-polynomial* complexity. In this case the space (not time) requirements of the algorithm cannot be bounded by a polynomial. Problems of this kind have polynomial time complexity in time iff their input is polynomial in size. A *heuristic*, finally, is an algorithm that provides an approximate, but feasible solution to a problem having exponential time complexity. Heuristics are based on some rule of thumb to derive 'good' solutions in polynomial time, but cannot guarantee to find the optimal solution. A ε -performance-guarantee heuristic, however, has a worst case deviation of

$$\left| \frac{H(I) - O(I)}{O(I)} \right| \leq \varepsilon \quad \text{for all instances } I$$

where $H(I)$ denotes the approximate solution of an instance I , derived from the heuristic H , and $O(I)$ represents its optimal solution (Fisher 1980).

The simplex algorithm of Dantzig (cf. Dantzig 1990), for example, has exponential time complexity (Klee and Minty 1972), but is still regarded to be a good algorithm, because it has a reputation for running quickly in practice. Often this is due to the fact that the exponential time complexity of an algorithm stems from a worst-case analysis, whereas the majority of problem instances might be solvable in polynomial time. However, the simplex algorithm is regarded as an exception among the exponential time algorithms.

The problems within the *RCPSP* framework generally have exponential time complexity and are thus not considered well solved (Neumann, Schwindt, and Zimmermann 2002). In particular the same holds for CSP, which will be shown in section 3.2.2 after the reader has been familiarized with all relevant aspects of the proof. In fact, CSP belongs to a very stubborn class of exponential time problems, called *NP-hard*, which are explained in following.

³If this conjecture is true remains an open question today (cf. section 1.3).

1.3 The Theory of NP-Completeness

Although the theory of *NP*-completeness is of vital importance for understanding the computational hardness of scheduling problems, it is rarely introduced in literature, mostly due to limitations in space. In an effort to make this book self containing, this section is devoted to an (informal) synopsis of the fundamental book of Garey and Johnson (1979), instead of merely adding it to the list of references.

As a matter of convenience the theory of NP-completeness is build around *decision problems* only. Such problems are questions that can either be answered by 'yes' or 'no', thus simplifying formal treatment of problems.⁴ Where the optimization problem in a scheduling context would state: "Which feasible schedule has minimal makespan?" the decision problem asks "Is there a feasible schedule with makespan less than t ?". Generally decision problems can be derived from optimization problems by introducing a timespan, here t , and asking whether there exists a solution satisfying this particular bound. Trivially, knowing the answers to all decision problems, i.e. the subsets of 'yes' or 'no' answers for any given bound $t = 1 \dots T$, is equivalent to solving the optimization problem. Therefore, as long as the question to be asked can be evaluated relatively easily (i.e. in polynomial time), it is a valid approach to analyze computational complexity for decision problems only, since they can be no harder than their optimization version. Suppose, for example, that one is given an algorithm, which successfully determines an optimal solution of every instance of CSP. Certainly, this solution could be compared to the upper bound of the decision problem, thus solving it as well. Conversely, a definite 'yes' or 'no' answer for any fixed bound t of the decision problem can easily be translated into an optimal solution of the optimization problem.

Having justified the use of decision problems, the complexity class P can defined as:

Definition 1.1 (Complexity Class P). P is the equivalence class of all decision problems $\Pi_i \in \Pi$, such that there is a polynomial time function $f(x)$, where x is a string representing an instance of Π_i and $f(x) = true$ (i.e. 'yes') if and only if $\Pi_i(x) = true$.⁵

⁴This is because decision problems have a very natural formal counterpart, called a 'language'. All languages can later be classified by the type of computer-model (touring-machine) needed to proof their correctness. In this text touring-machines will not be introduced, since it is believed that the reader will understand the elementary difference between the complexity classes with the lack of this formality.

⁵Moreover P is formally defined as the set of languages Π that is accepted by a deterministic touring machine in polynomial time.

Informally P is the class of problems solvable by an algorithm having polynomial time complexity. The reader may notice that the polynomial nature of the function $f(x)$ relies on the input length of the string x and thus on the particular encoding scheme e used. Therefore e shall be restricted to 'reasonable' encoding schemes (cf. Garey and Johnson 1979), which essentially means (i) the encoding of an instance should be concise and not padded with unnecessary information and (ii) an instance should be easily decodable, that is being coded only using symbols from an alphabet with a fixed base. The resulting strings of any such reasonable encoding scheme will then differ at most polynomially in size and thus do not affect the membership in P .

Furthermore NP is the class of decision problems for which a given solution, s , can be *verified* in polynomial time. The vast majority of the decision problems encountered in scheduling belong to NP , because one can easily check the feasibility of a given schedule. However, a polynomial time algorithm for providing such a schedule may not be known.

Definition 1.2 (Complexity Class NP). *NP is the the class of decision problems $\Pi_j \in \Pi$, such that there is a polynomial time function $f(x,s)$ where x is a string, s is another string whose size is polynomial in the size of x , and $f(x,s) = true$ if and only if $\Pi_j(x,s) = true$.*⁶

Obviously any decision problem in P is also a member of NP , since a solution can be derived in polynomial time, which in turn must also yield a 'yes' answer to the particular decision problem. However, whether or not $P = NP$ is still an open question today, albeit there is strong evidence that this equality does not hold, which implies that some problems in NP cannot be solved in polynomial time. Therefore throughout this text it is presumed that the equality of P and NP does not hold in order to justify the exponential time complexity of some exact solution procedures to be presented.

The foundations of the theory of NP-completeness were laid by Cook (1971), who emphasized the significance of 'polynomial time reductability'. *Reducing* a problem to another is the principal technique for showing that they are related in terms of complexity. Essentially a 'reduction' is a constructive transformation of all instances x' of Π' to instances x'' of another problem Π'' , so that $\Pi'(x') = true$ if and only if $\Pi''(x'') = true$. Consequently, an algorithm that solves Π'' can be used to solve Π' as well and, since there is an additional transformation

⁶Formally Π_j is the class of languages that is accepted by a nondeterministic touring-machine in polynomial time. In this context the nondetermination of the computer model can be interpreted as 'guessing' a solution and then verifying it in polynomial time. Hence NP stands for 'nondeterministic polynomial'.

involved, Π' must be at least as 'hard' to compute as Π'' . A *polynomial time reduction* then is a reduction in which the transformation between the problem instances can be done in polynomial time.

Definition 1.3 (Polynomial Time Reductability). *A decision problem Π'' is polynomial time reducible to another decision problem Π' if there exists a polynomial time function $f : x'' \rightarrow x'$, so that $\Pi'(x') = \text{true}$ if and only if $\Pi''(x'') = \text{true}$.*

Cook proceeded by choosing one particular problem, called the 'satisfiability problem' out of NP and showed that every other problem in NP can be polynomially reduced to it. Hence the 'satisfiability problem' must be at least as hard as any other problem in NP , i.e. if it can be solved with a polynomial time algorithm, so can any other problem in NP . Cook conjectured that other problems might share the property of being the 'hardest' in NP , which was subsequently proven by Karp (1972). It was also due to Karp that the complexity class of these problems was named NP -complete.

Cook's initial proof of showing that the satisfiability problem is NP -complete, known as 'Cook's theorem', is far from simple and the amount of problems shown to be NP -complete would be rather small if such a proof would be necessary for every problem. However, since one problem is already known to be NP -complete, it suffices to show the following two properties: (i) the problem is in NP and (ii) the problem is NP -hard, i.e. polynomial time reducible to a known NP -complete problem. Since Cook's theorem a wide variety of problems have been shown to be NP -complete and the set of problems to choose from in step (ii) is therefore ever growing. These results are summarized by the following two definitions and illustrated by a view of the complexity classes, assumed $P \neq NP$.

Definition 1.4 (NP -hard). *A decision problem Π' is NP -hard if any other decision problem $\Pi'' \in NP$ can be polynomially reduced to it.*

Definition 1.5 (NP -complete). *A decision problem Π' is NP -complete iff it is both NP -hard and in NP .*

Note that a problem Π' does not have to be a member of NP in order to be NP -hard. In particular, the optimization versions of the scheduling problems, as dealt with in this book, are generally not in NP and therefore belong to the class of NP -hard problems. The decision versions of most scheduling problems, on the other side, belong to NP , and many of them are

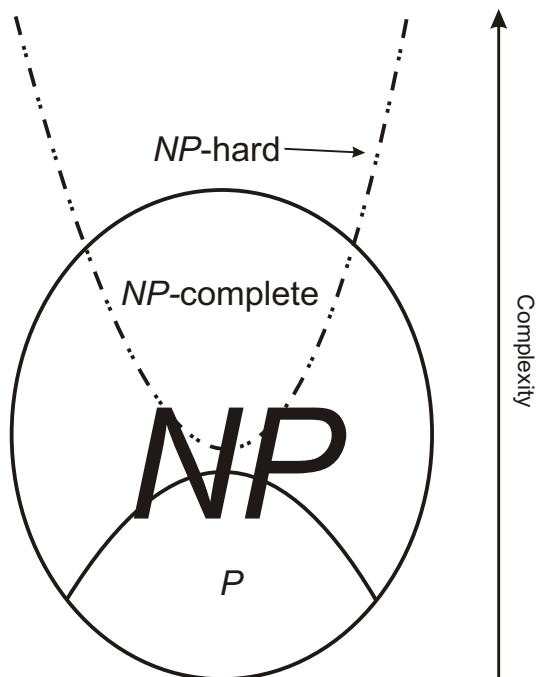


Figure 1.1: *The world of NP-completeness, given $P \neq NP$*

NP-complete. Furthermore Garey and Johnson (1979) introduced the notion of *strong NP-completeness*:

Definition 1.6 (Strong NP-completeness). *A decision problem Π' is NP-complete in the strong sense iff it cannot be solved by a pseudo-polynomial time algorithm, unless $P = NP$.*

Finally, it is mentioned that the decision version of CSP is *NP-complete* in the strong sense (cf. section 3.2.2).

Chapter 2

Related Problems and Solution Procedures

2.1 Integer Linear Programming Approach

A *combinatorial problem* is defined as a problem having a (i) discrete and (ii) bounded solution space (Neumann and Morlock 1993). Obviously CSP, and indeed any scheduling problem, is of combinatorial nature, because it deals with discrete points in time and w.l.o.g. an upper bound T of the project duration is given.¹ Dantzig (1960) was the first to show that a number of combinatorial problems can conceptually be modeled as an integer linear program. In following the formulation of Pritsker, Watters, and Wolfe (1969) is used to formally describe CSP. Let the binary variable $x_{i,t}$ be defined as

$$x_{i,t} := \begin{cases} 1 & : \text{if activity } i \text{ starts at time instant } t \\ 0 & : \text{otherwise} \end{cases}$$

Furthermore let d_i denote the duration of activity i and Z the project duration. The optimization problem can then be stated as

¹Note that the sum of the activity durations always constitutes such a proper upper bound.

$$\min \quad Z \tag{2.1}$$

s.t.

$$Z \geq \sum_{t=1}^T t \cdot x_{i,t} + d_i \quad , \quad i = 1 \dots N \tag{2.2}$$

$$\sum_{t=1}^T x_{i,t} = 1 \quad , \quad i = 1 \dots N \tag{2.3}$$

$$\sum_{i=1}^N \sum_{q=t}^{\min\{T, t+d_i-1\}} r_{i,j} \cdot x_{i,q} \leq 1 \quad , \quad j = 1 \dots K \text{ and } t = 1 \dots T \tag{2.4}$$

$$x_{i,t} \in \{0, 1\} \tag{2.5}$$

The project duration, Z , to be minimized is bounded from below by the completion time of the latest activity, as given by one of the inequalities (2.2). Equation (2.3) ensures that each activity is scheduled exactly once and inequality (2.4) corresponds to the unique resource constraints, where $r_{i,j} \in \{0, 1\}$ stands for the resource demand of activity i again. Here, in every time instant t and for every resource type j , the weighted sum of the resource demands $r_{i,j}$ of the activities in progress cannot exceed one, which is the capacity R_j of any resource. Finally equation (2.5) states the binary nature of the optimization variables. It is easy to see that this formulation requires $2N + MT$ restrictions.

Many OR problem solvers are available today that allow for input of integer linear programs (e.g. CPLEX, LAMPS, FortMP, MOSEK, OSL, SOPT, XA, Xpress). These usually employ generic solution methods based on enumeration or cutting planes (Gomory 1958) for a brute-force approach to solving the problem. Although no deeper insight in the problem structure is gained, the algorithms sometimes perform fairly well due to their highly efficient implementation by professional programmers.

The operational research community, however, is concerned with more sophisticated approaches, specifically designed for a certain problem. Exact solution procedures are then mainly based on *implicit enumeration*, also known as the *branch-and-bound (BaB)* technique. This method successively eliminates infeasible parts of the solution space with respect to upper and lower bounds of the objective function as opposed to performing an exhaustive search.² The details of the general technique will be explained in section 2.3.1 when being applied to a specific

²Some of the above mentioned solvers also provide a generic branch-and-bound approach. Here the infeasible parts of the solution space are identified rather inaptly, of course.

scheduling problem. In the scheduling literature linear programming approaches are merely considered for problem relaxations and with the intention of providing upper or lower bounds for a branch-and-bound procedure (e.g. Christofides, Alvarez-Valdes, and Tamarit 1987; Mingozzi et al. 1998; Brucker et al. 1998; Carlier and Néron 2000; Demassez, Artigues, and Michelon 2001; Möhring et al. 2003; Ralphs, Ladanyi, and Saltzman 2003).

2.2 Machine Scheduling

It has been previously mentioned that the *job-shop scheduling problem (JSSP)* evolved from a manufacturing environment and was one of the first scheduling problems to be considered in literature. Although the formulation of the problem is rather simple it remains to be one of the most difficult scheduling problems today, in spite of tremendous research efforts. According to Conway, Maxwell, and Miller (1967) the general JSSP is given through

1. a set of machines $M_j, j = 1 \dots K$
2. a set of jobs $J_i, i = 1 \dots N$,
each to be processed on a subset of the machines, and
3. a processing time $d_{i,j}$ for each job J_i on machine M_j .

The objective of JSSP is to minimize the time of all jobs in the shop. Furthermore it can be categorized into the following types:

- *Open-shop*: the machines required for a job can be used in any order
- *Job-shop (in the narrow sense)*: operations of a job are totally ordered
- *Flow-shop*: each job needs all machines for processing and all jobs go through all the machines in the same order

Many variants of the above models have been discussed and the reader may get easily lost in this diversity. Moreover different classification schemes have been used for machine scheduling and project scheduling problems, even when essentially referring to the same problem. The *RCPS* framework, for example, contains JSSP as a special case. The classification scheme by Brucker et al. (1999), as introduced earlier, has actually been a recent effort to standardize the notation. The ties between JSSP and CSP are evident when comparing JSSP with the key characteristics

of CSP: (i) each machine is associated with a duration (ii) the jobs (not necessarily machines in a job) can be scheduled independently (iii) mutual conflicts between jobs arise, when the same machine is needed for processing. The nature of unique resources is revealed through the fact that each machine can be available to one job at most, thus being a resource with capacity one. The careful reader will notice the general JSSP is not identical with CSP as a connection between mutual incompatibility of *jobs* as opposed to machines is still lacking. The remainder of this section is therefore meant to provide an overview of some machine scheduling problems, so that eventually CSP can be located within the machine scheduling problems.

2.2.1 Preemption vs. Non-Preemption

The non-preemption property of CSP has not been included in the above comparison because it is not part of the general JSSP. When *preemption* is allowed the processing of any machine can be interrupted and carried on later at no cost, thus freeing its resources for other jobs. This simplifies the calculation of an optimal schedule since resource conflicts can be resolved by simply interrupting the conflicting machines of one of the jobs. The problem can be solved in linear time if the number of machines is restricted to $K = 4$ (Bianco et al. 1994), but remains strongly *NP*-hard for an arbitrary number of machines (Krawczyk and Kubale 1985). Nevertheless, Amoura et al. (1997) present a heuristic for the non-preemptive case that is based on a preemptive relaxation of the problem, which can be solved by linear programming (Krämer 1995). Their basic idea is to adapt the schedule generated from the relaxation by scheduling all preempted jobs sequentially in the end. Because generally this procedure will yield a schedule being far from optimal, only 'short' jobs are allowed to be preempted in the relaxation.

2.2.2 Single- vs. Multiprocessor

For a long time research on JSSP has mainly assumed that each job requires one machine (or resource) at a time for processing only. These *single or uni-processor* models initially reflected the applications in industry, where a product was manufactured sequentially on different machines. A possible argumentation to the contrary might be, that the workers needed to operate the machines should have also been regarded as a resource, thus making it necessary to associate a job with the simultaneous need of at least two resources. The single processor model may still have been a good approximation, however, when considering scarce resources only: If no (or little) special skills are required from the workers operating these machines, then their

availability may not be a constraint worth considering.

Advances in technology, especially with respect to computer systems, have eventually diverted the research focus to *multiprocessor* models. Here each job demands for the simultaneous availability of two or more scarce resources for processing, which gives rise to the well known mutual incompatibilities of CSP: two jobs cannot be processed in parallel, if both require the same machine (i.e. unique resource). Whereas the general multiprocessor model with precedence constraints is *NP*-hard in the strong sense, even when restricted to two processors (Du and Leung 1989), the non-preemptive scheduling of independent jobs on multiprocessor systems with two or three processors has been shown to be solvable in pseudo-polynomial time. For more than four processors, however, the latter is also strongly *NP*-hard (Blazewicz et al. 1992). Obviously CSP is associated with a multiprocessor model having an arbitrary amount of processors.

2.2.3 Dedication vs. Non-Dedication

Due to further advances in technology researchers have been urged to diversify the model even more. Modern parallel computers, for example, consist of many *identical* microprocessors which are simultaneously available for computation. In the multiprocessor-world a job may still require more than one of these processors at a time, but is not *dedicated* to one, since all are identical. An application is mentioned by Krawczyk and Kubale (1985) where one microprocessor is used to test another one in a fault-tolerant multi-computer system. Likewise, the simultaneous availability of many identical machines can be interpreted as an increase in the capacity of that resource, contradicting its uniqueness. Clearly, CSP then reflects the *dedicated* variant of a multiprocessor system, where all jobs have to be performed on prespecified machines.

In summary, CSP corresponds to non-preemptive scheduling of independent multiprocessor tasks on dedicated processors. In contrast to the more general project scheduling notation previously given, CSP can be stated as $P|fix|C_{max}$ in the machine scheduling notation of Veltman, Lageweg, and Lenstra (1990). Here P stands for multiprocessor scheduling, *fix* for prespecified processor assignments without preemption and C_{max} for the objective of minimizing the makespan.

2.3 A Pioneering Branch-and-Bound Approach

Bozoki and Richard (1970) were the first to tackle the special structure of the machine scheduling problem $P|fix|C_{max}$ with an exact solution procedure based on branch-and-bound. The authors refer to the non-preemptive scheduling of multiple tasks requiring the simultaneous availability of machines as the *continuous-process job-shop scheduling problem (CPJS)*, but essentially deal with the same problem as CSP, which is undermined by their following statement: "It is the competition among incompatible jobs which is at the origin of the scheduling difficulties."

Motivated by the general JSSP framework, Bozoki and Richard introduce a technological matrix $\tau_{i,j}$, where

$$\tau_{i,j} := \begin{cases} 1 & : \text{if job } J_i \text{ requires machine } M_j \\ 0 & : \text{otherwise} \end{cases}$$

Let $O_i := \{M_j \mid \tau_{i,j} = 1\}$ be the set of machines of job J_i that have to be available simultaneously for its processing, whereas $P_j := \{J_i \mid \tau_{i,j} = 1\}$ represents the set of jobs requiring a particular machine M_j . Because of the concurrence in processing the duration d_i of job J_i can be expressed by the maximum execution time of a single machine $M_j \in O_i$:

$$d_i = \max_{M_j \in O_i} d_{i,j}$$

Furthermore one can define an incompatibility matrix $h_{i,k}$ with

$$h_{i,k} := \begin{cases} 1 & : \text{if } O_i \cap O_k \neq \emptyset \\ 0 & : \text{otherwise} \end{cases}$$

i.e. two jobs J_i and J_k are incompatible if they require a common machine. Indeed, as machines are unique resources, the reader will find this definition very familiar (cf. equation (1.3)).

2.3.1 The General Branch-and-Bound Methodology

An elementary part of each branch-and-bound procedures is, as the name suggests, to identify proper bounds of the variable to be optimized; in this case, the makespan. Hence the determination of lower and upper bound is one of the essential and initial steps of each BaB procedure. Moreover, the difference between the value of both bounds shows how much can possibly be gained by searching for the exact solution and gives a rough estimate of the computational hardness of the particular problem instance. If the gap is sufficiently close, one might decide to go

with the feasible solution represented by the upper bound, and the BaB process must not be initiated after all. Otherwise the determination of tight bounds becomes crucial for the performance of the BaB algorithm, because they are used to eliminate suboptimal parts of the solution space. As implicit enumeration will be the principal exact solution technique for the combinatorial problems to be presented throughout the text, the general branch-and-bound methodology is described in more detail according to Agin (1966):

The set of all solutions, \mathcal{S} , of a combinatorial problem is called the *solution space*. Since \mathcal{S} is finite by definition, the optimal solution can always be found by enumerating all of its members. Implicit enumeration methods, however, do not search \mathcal{S} exhaustively, but rather successively identify partitions of it which cannot be optimal. Classically a *partition* of a set, here \mathcal{S} , is defined to be an exhaustive division into disjoint subsets $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_n$:

$$\mathcal{S}_1 \cup \mathcal{S}_2 \cup \dots \cup \mathcal{S}_n = \mathcal{S}$$

$$\mathcal{S}_i \cap \mathcal{S}_j = \emptyset \quad \text{for all } i \neq j$$

Some recent branch-and-bound techniques relax the second property and allow some overlap in the subsets (cf. Ibaraki 1977), although all BaB algorithms to be discussed in this paper will call for disjoint subsets.

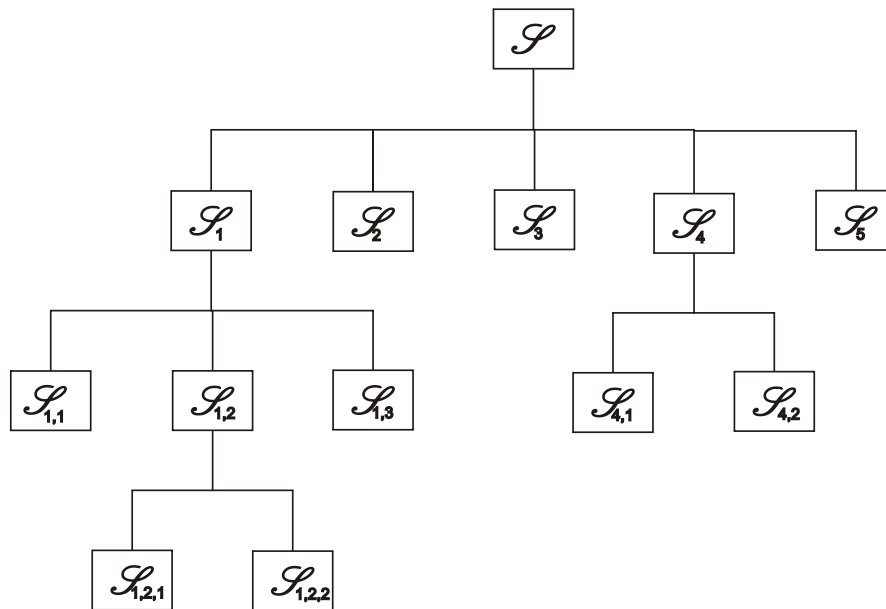


Figure 2.1: A typical branch-and-bound tree.

Branching is then defined to be the process of repetitively partitioning the solution space and evaluating the properties of the subsets in each step. Branching is often visualized by a *branch-*

and-bound tree, which has \mathcal{S} as its root node and the partitions as descendant nodes (see figure 2.1). Every level of the tree represents a refinement of each of the partitions until eventually \mathcal{S} is broken down into individual solutions. However, the BaB tree must not necessarily contain branches down to the solution level, because each node (subset) is judged for feasibility before it fathers children nodes through branching. The properties to be evaluated for each node in order to decide whether further branching takes place or not may be multidimensional, but is usually restricted to only one decisive dimension. For scheduling problems this is usually a lower bound of all schedules represented by the node under consideration. This lower bound is therefore referred to as a *local* lower bound, since it has been determined for a specific subset of the solution space only.³ Since the calculation of local lower bounds has to be done for every node of the BaB tree, this should be done by fast heuristics.⁴ If the local lower bound of a node exceeds the makespan of an already known feasible solution, i.e. an upper bound, then no solution within the corresponding subset can be optimal and the node and all descendant nodes can be omitted from further consideration. With respect to the BaB tree this process is figuratively called *pruning* or *fathoming*, and a pruned node is said to form a *leave* of the BaB tree. In the first level of the tree in figure 2.1, for example, the leaves are $\mathcal{S}_2, \mathcal{S}_3$ and \mathcal{S}_5 . After a node has been pruned, a new eligible node has to be found in the tree which is to be evaluated next and from which further branching might take place. This process is called *backtracking*, because from the current (fathomed) node, first one has to go at least one branch up in the hierarchy of the tree before another node is encountered which has not been explored yet. Thereby two general backtracking strategies can be differentiated: *depth-first* and *breadth-first*. In the former case the nodes in the lowest level of the tree are considered first, whereas in the latter case the nodes on the highest level are preferred. For instance, in the branch-and-bound tree of figure 2.1 a breadth-first strategy would consecutively consider the nodes $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_{1,1}, \dots, \mathcal{S}_{1,2,2}$, whereas a depth-first approach subsequently explores $\mathcal{S}_1, \mathcal{S}_{1,1}, \mathcal{S}_{1,2}, \mathcal{S}_{1,2,1}, \mathcal{S}_{1,2,2}, \mathcal{S}_{1,3}, \dots, \mathcal{S}_5$. Most commonly a depth-first strategy is employed, because it is more likely to find quickly new feasible solutions (upper bounds). For practical considerations it is even more important that a depth first approach has less memory demands, since it generates smaller sets of unexplored nodes than breadth-first search. However, the major disadvantage of depth first approaches is that they face the risk of thoroughly searching

³Conversely, the initial lower bound computed on \mathcal{S} is called a *global* lower bound.

⁴'Fast' means having polynomial time complexity of $O(n^k)$, where n is the problem size and k a 'small' integer (cf. Sahni 1976).

certain parts of the tree (determined by early node choices), whereas the optimal solution may be found elsewhere. This risk is especially high, when the current upper bound is (near) optimal. Klein and Scholl (1999) therefore suggest *scattered depth-first search*, where the search process alternates between phases of intensification (thorough search: depth-first) and diversification (determination of good candidate nodes).

2.3.2 Bounding Techniques for the CPJS

Lower Bounds

Bozoki and Richard identify two lower bounds for the CPJS. The first one (\underline{Z}_1) is constituted by the maximum amount of time a single machine in the shop needs to process all of its associated jobs. Clearly, all of these jobs must be in conflict and the optimal makespan cannot be smaller than the sum of the respective job durations. Formally

$$\underline{Z}_1 = \max_{M_j \in O_i} \left(\sum_{J_i \in P_j} d_i \right)$$

Another approach considers sets of mutually incompatible jobs $\mathcal{J}_r \in \mathcal{J}$. By nature all jobs $J_i \in \mathcal{J}_r$ have to be performed concurrently and hence a second lower bound (\underline{Z}_2) can be derived by finding the \mathcal{J}_r with jobs having the maximum sum of job durations:

$$\underline{Z}_2 = \max_{\mathcal{J}_r \in \mathcal{J}} \left(\sum_{J_i \in \mathcal{J}_r} d_i \right)$$

Indeed, \underline{Z}_2 is a tighter lower bound than \underline{Z}_1 and it holds that $\underline{Z}_1 \leq \underline{Z}_2 \leq Z$ since any set of jobs used to define \underline{Z}_1 constitutes a set of incompatible jobs and thus is contained in at least one set from \mathcal{J} . However, the calculation of a \underline{Z}_2 is *NP-hard*, because it corresponds to the problem of finding the maximum weighted clique in an undirected graph (cf. section 3.3). It is not advisable to solve a *NP-hard* problem by relaxing it to another *NP-hard* problem. Consequently \underline{Z}_2 should only be approximated from below by an heuristic. However, Bozoki and Richard note, that the \mathcal{J}_r correspond to the rows of the incompatibility matrix, which would in turn suggests the existence of a polynomial time algorithm for the computation of \underline{Z}_2 . This blunder probably occurred because the theory of *NP-completeness* had not yet been developed when the authors wrote their paper. The validity of \underline{Z}_2 as a good lower bound, however, is undisputed. Section 3.3 is dedicated to a brief survey on heuristics for the determination of \underline{Z}_2 .

Upper Bounds

Although the importance of lower bounds has been emphasized, the necessity for upper bounds should not be lost out of sight. When compared with a lower bound the feasible schedule corresponding to the upper bound may be judged to be sufficiently close and the tedious search for an optimal schedule may not be worthwhile undertaking. In the scheduling context upper bounds are usually derived through heuristics based on the *priority rule method (PRM)*. Simple PRM schedule the jobs sequentially, according to their previously assigned priority, i.e. each job is scheduled as soon as the production facilities (resources) it requires are available and only if there are no other jobs with higher priority that can be scheduled at this time. More sophisticated PRM approaches try to improve the initially generated schedule by later reassignment of starting times to already scheduled activities. An example of such a more sophisticated PRM procedure as well as an assessment of different priority rules is provided in chapter 4.3.

Bozoki and Richard restrict their attention to two priority rule heuristics of the simple kind. The first upper bound, \bar{Z}_1 , is derived by scheduling the jobs with the *shortest processing times (SPT)* first, whereas \bar{Z}_2 is computed using the *maximum degree of competition (MDC)* rule. The degree of competition (DC) is a measure of how many jobs are in conflict with the job J_i under consideration and is formally defined through the incompatibility matrix $h_{i,k}$ as:

$$DC(J_i) := \sum_{k=1}^N h_{i,k}$$

Jobs with a higher degree of competition are assigned a higher priority by the MDC rule, where ties are broken by the SPT rule. The authors state that no rule outperforms the other, but recommend the use of SPT when the variability among job required production times is high and the competition relatively low, whereas MDC should be used under opposite conditions.

2.3.3 The Branch-and-Bound Process

The branching procedure considers all possible arrangements of the jobs that may yield a feasible schedule. Branching will only take place at *transition times* t^l ($l = 0, 1, 2, \dots, L$), which are successive and distinct points in time at which at least one job is completed in a particular schedule S . By definition, $t^0 < t^1 < \dots < t^L$ and $L \leq N$ since two or more jobs can terminate at the same time and $t^0 = 0$ and $t^L = Z$. Moreover, let x_i ($i = 1, 2, \dots, N$) be the time at which Job J_i is released for production. The corresponding transition time of job J_i under schedule S

is then given through the equation

$$t^l = x_i + d_i \quad , \text{ for exactly one } l \in \{1, 2, \dots, L\}$$

A *partial schedule* $S(t^l)$ is a schedule which is only defined between the times 0 and t^l . Consequently three different sets of jobs can be distinguished at the transition time t^l :

1. Jobs currently in process, denoted by

$$J^1(t^l) : \{J_i, J_k | O_i \cap O_k = \emptyset; x_i < t^l; x_k < t^l; x_i + d_i > t^l; x_k + d_k > t^l\}$$

2. Jobs not yet released, denoted by

$$J^2(t^l) : \{J_i | x_i \geq t^l\}$$

3. Jobs completed, denoted by

$$J^3(t^l) : \{J_i | x_i + d_i \leq t^l\}$$

The branching will then take place over all *acceptable subsets* $Q(t_\varphi^l)$ of the set of jobs remaining to be processed ($J^2(t_\varphi^l)$), where t_φ^l is the transition time of a partial schedule $S_\varphi(t_\varphi^l)$. The authors define such an acceptable subset as having the following properties:

- Jobs of the subset are mutually compatible

$$O_i \cap O_k = \emptyset, \quad \forall J_i, J_k \in Q(t_\varphi^l)$$

- Each job of the subset must be compatible with each job currently in progress

$$O_i \cap O_k = \emptyset, \quad \forall J_i \in Q(t_\varphi^l), \forall J_k \in J^1(t_\varphi^l)$$

- Each job of the subset has to be incompatible with at least one job, which terminates at t_φ^l

$$\exists J_i \in J^3(t_\varphi^l), \exists J_k \in Q(t_\varphi^l), \text{ such that } O_i \cap O_k \neq \emptyset$$

All of the above conditions are meant either to ensure optimality or feasibility of the partial schedules generated. Clearly with each choice of a job from $Q(t_\varphi^l)$ a new partial schedule $S_\varphi(t_\varphi^{l+1})$ has been constructed. Consequently a new lower bound for this schedule can be calculated, denoted by $\underline{Z}(S_\varphi(t_\varphi^{l+1}))$. In particular $\underline{Z}(S_\varphi(t_\varphi^l))$ is equal to the makespan of $S_\varphi(t_\varphi^l)$, plus a lower bound of the jobs which have not yet terminated ($J^1(t_\varphi^l)$ and $J^2(t_\varphi^l)$). In this calculation the jobs $J_i \in J^1(t_\varphi^l)$ are associated with their remaining production time $d_i^l = d_i - (t_\varphi^l - x_i)$ instead of their total production time (d_i). Recalling the previously introduced lower bounds $\underline{Z}_\alpha, \alpha \in \{1, 2\}$, the lower bound of a partial schedule can be summarized to:

$$\underline{Z}(S_\varphi(t_\varphi^l)) = t_\varphi^l + \underline{Z}_\alpha(J^1(t_\varphi^l) \cup J^2(t_\varphi^l))$$

Input : An instance of CPJS given through a set of jobs $J_i, i = 1 \dots N$, a set of machines $M_j, j = 1 \dots K$, job durations d_i and the technological matrix δ

Output: A schedule S_ϕ with minimal makespan

```

1 foreach  $J_i$  do
2   | assign priority according to some priority rule (e.g. STP or MDC);
3   | derive the incompatibility matrix from the technological matrix ;
4   | generate a feasible schedule with a priority rule heuristic and compute an initial upper
   | bound  $\bar{Z}$ ;
5   | determine a lower bound  $Z_\alpha, \alpha \in \{1, 2\}$  of the optimal makespan;
6   | choose the partial schedule  $S_\phi(t_\phi^l)$  with the smallest lower bound ;
7   | compute the acceptable subset  $Q(t_\phi^l) \subset J^2(t_\phi^l)$ ;
8   | if  $Q(t_\phi^l) = \emptyset$  and  $J^1(t_\phi^l) \cup J^2(t_\phi^l) \neq \emptyset$  then
9     |   goto line 6;
10  | branch over  $S_\phi(t_\phi^l)$  by selecting the jobs from  $Q(t_\phi^l)$ ;
11  | determine lower bounds  $Z(S_\gamma(t_\gamma^{l+1}))$  of newly generated schedules  $S_\gamma(t_\gamma^{l+1})$ ;
12  | if  $Z(S_\gamma(t_\gamma^{l+1})) \geq \bar{Z}$  then
13    |   fathom  $S_\gamma(t_\gamma^{l+1})$  ;
14    |   goto line 6;
15  | else if  $Z(S_\gamma(t_\gamma^{l+1})) > \min_\phi Z(S_\phi(t_\phi^l))$  then
16    |   goto line 6;
17  | else if  $Z(S_\gamma(t_\gamma^{l+1})) \leq \min_\phi Z(S_\phi(t_\phi^l))$  then
18    |   switch  $S_\gamma(t_\gamma^{l+1})$  complete? do
19      |     case no
20        |       | goto line 7 and branch on  $S_\gamma(t_\gamma^{l+1})$ ;
21      |     case yes
22        |       | return  $S_\gamma(t_\gamma^{l+1})$ ;

```

Algorithm 2.1: BaB algorithm for the CPJS by Bozoki and Richard (1970)

Obviously any partial schedule $S^\phi(t_\phi^l)$ generated, which corresponds to a node in the BaB tree, can be fathomed iff its lower bound exceeds a known upper bound ($Z(S_\phi(t_\phi^l)) \geq \bar{Z}$). Furthermore, the node with the smallest lower bound will be considered first when branching, until eventually a (partial) schedule evolves, which fixes the starting times for all jobs. Such a schedule is called *complete* and since the schedules are generated in order of their lower bounds the first complete schedule will be optimal.⁵ More specifically, branching on schedule $S_\phi(t_\phi^l)$ occurs if and only if $Z(S_\phi(t_\phi^l)) \leq \min_\phi Z(S_\phi(t_\phi^l))$, where ϕ is the set of subscripts of all previously

⁵Recall that by definition of the lower bound of a schedule $S_\phi(t_\phi^l)$, this bound will eventually be realized when no more jobs are left unscheduled ($J^2(S_\phi(t_\phi^l)) = \emptyset$).

generated partial schedules. The complete algorithm according to Bozoki and Richard (1970) is summarized by algorithm 2.1.

Chapter 3

Graph Representation

The reader will agree that the representation of scheduling problem instances by schedules, as used by Bozoki and Richard, is very cumbersome and tends to be overly loaded with sub- and superscripts. In an effort to facilitate the formal description in a more intuitive way several graph representations have been proposed in literature (e.g. Roy and Sussmann 1964; Blazewicz, Pesch, and Sterna 2000; Fondahl 1962; Dibon 1970). Graph representations are usually tailored to the characteristics of a special problem class and allow not only for mere descriptive purposes, but also for the exploitation of graph-theoretic properties. In this chapter the so-called constraint graph is introduced, which is capable of representing every distinct instance of CSP and serves as foundation for all further considerations. The remainder of this book will therefore focus on graph-based solution procedures, which are presented in chapters 4 and 5, respectively. Besides introducing the constraint graph, this chapter also provides a complexity proof for CSP as well as two problem decompositions. It concludes with an thorough investigation of the maximum weighted clique problem, which constitutes a lower bound of the optimal makespan of CSP. At first, however, the reader is familiarized with graph-theoretic notation and basic definitions.

3.1 Graph-Theoretic Foundations and Notation

It is noted, that the definitions and notation introduced in following will not all be of immediate use. The reader may therefore use this section as a reference and return at a later time. The following definitions are adapted from Golubic (1980) unless stated differently.

3.1.1 Basic Definitions

Binary Relations

Let X and Y be (finite) sets, then $x \in X$ indicates, that x is a member of X , whereas $Y \subset X$ means that Y is a proper subset of X . In addition to the usual set union ($X \cup Y$), set intersection ($X \cap Y$), and set subtraction ($X - Y$), the $+$ operator shall indicate the union of mutually disjoint sets:

$$X + Y := \{X \cup Y \mid X \cap Y = \emptyset\}$$

The *cardinality* or *size* of a set X reflects the amount of members $x \in X$ and is denoted by $|X|$. $\mathcal{P}(X)$ is the *power set* of X and represents the collection of all subsets of X , including the empty set \emptyset and X itself. Thus the cardinality of $\mathcal{P}(X)$ is $2^{|X|}$, because $|X|$ binary variables are necessary to indicate whether a specific $x \in X$ is a member of the subset or not.

Definition 3.1 (Binary Relation). *A binary relation R on X is defined as a function from X to the power set of X .*

$$R : X \rightarrow \mathcal{P}(X)$$

Binary relations are often given by a collection of *ordered pairs* $\mathcal{R} \subseteq X \times X$, where

$$\langle x, x' \rangle \in \mathcal{R} \quad \text{if and only if} \quad x' \in R(x)$$

Since the pairs are ordered, x and x' can generally not be interchanged. However, a relation that would satisfy this property is called *symmetric*, formally defined as

$$x' \in R(x) \Rightarrow x \in R(x') \quad \text{for all } x, x' \in X \text{ and } x \neq x'$$

Conversely, R is *antisymmetric* iff

$$x' \in R(x) \Rightarrow x \notin R(x') \quad \text{for all } x, x' \in X \text{ and } x \neq x'$$

Furthermore a relation is called *reflexive* if and only if

$$x \in R(x) \quad \text{for all } x \in X$$

whereas an *irreflexive* relation fulfills

$$x \notin R(x) \quad \text{for all } x \in X$$

Finally, a *transitive* relation satisfies

$$z \in R(y) \text{ and } y \in R(x) \Rightarrow z \in R(x) \quad \text{for all } x, y, z \in X \text{ and } x \neq y \neq z$$

Note that a relation can satisfy more than one of the properties. In particular, an *equivalence* relation (\Leftrightarrow) is reflexive, symmetric, and transitive, whereas a *strict partial order* (\prec) is irreflexive, antisymmetric and transitive.

Graphs

Definition 3.2 (Graph). *A graph G consists of a finite set of vertices V and an irreflexive binary relation on V .*

The binary relation on V defines the *adjacency* between the vertices and is referred to as Adj . The adjacency may either be represented by a collection E of ordered pairs or as a function from V to its power set

$$Adj : V \rightarrow \mathcal{P}(V)$$

Moreover $Adj(v)$ is called the *adjacency set* of vertex $v \in V$ and the ordered pair $\langle u, v \rangle \in E$ an *edge* of G , where u and v represent endpoints of the edge. A graph $G = (V, E)$ is said to be *undirected* if Adj is symmetric, i.e.

$$\langle u, v \rangle \in E \Rightarrow \langle v, u \rangle \in E \quad \text{for all } u, v \in V, u \neq v$$

In this case the mutually adjacent vertices u, v can be represented by the *unordered* pair (u, v) in the edge set E . If the symmetry for Adj does not hold, the graph is *directed*, and even more specifically, if Adj is antisymmetric, then the graph is *oriented*. Therefore throughout the text an undirected edge with endpoints u, v will be referenced by (u, v) , whereas a directed edge will be denoted by $\langle u, v \rangle$.¹ To emphasize the difference between undirected and directed graphs (short digraphs), the latter will sometimes be denoted by $\vec{G} = (V, \vec{E})$. However, in all cases the irreflexivity of Adj implies that

$$v \notin Adj(v) \quad \text{for all } v \in V$$

Coherently the *neighborhood* of vertex v is defined as

$$Nei(v) := \{v\} \cup Adj(v)$$

Let $\bar{G} = (V, \bar{E})$ be the *complement graph* of graph $G = (V, E)$, where

$$\bar{E} := \{\langle u, v \rangle \in V \times V \mid u \neq v \text{ and } \langle u, v \rangle \notin E\}$$

¹Note that at the same time $(u, v) \in E$ means that both $\langle u, v \rangle \in E$ and $\langle v, u \rangle \in E$. Since only undirected and oriented graphs will be considered later, no confusion can arise. This notation deviates from the reference of Golumbic (1980).

Furthermore the complementary adjacency \overline{Adj} can be represented through \overline{E} :

$$u \in \overline{Adj}(v) \quad \text{if and only if} \quad \langle u, v \rangle \in \overline{E} \quad \text{for all } u, v \in V$$

The *complementary neighborhood* of vertex v is then defined as

$$\overline{Nei}(v) := \{v\} \cup \overline{Adj}(v) \quad \text{for all } v \in V$$

whereas the *reversal* of graph G is denoted by $G^{-1} = (V, E^{-1})$, with

$$E^{-1} := \{\langle u, v \rangle \in V \times V \mid \langle v, u \rangle \in E\}$$

Note that G , \overline{G} and G^{-1} all have the same set of vertices and differ only in the edge set E . However, a graph $H = (V', E')$ that satisfies $V' \subseteq V$ and $E' \subseteq E$ is called a *subgraph* of $G = (V, E)$. Among all subgraphs two specific types of interest can be identified:

The subgraph $H = (V_S, S)$ *spanned* by the edge set $S \subseteq E$ contains all edges of S and only those vertices, which are endpoints of these edges. Formally

$$V_S := \{v \in V \mid \exists \langle u, v \rangle \in S \quad \text{or} \quad \exists \langle v, u \rangle \in S, u \in V\}$$

Secondly, a subgraph $H = (A, E_A)$ is said to be *induced* by a subset of the vertices $A \subseteq V$, when

$$E_A := \{\langle u, v \rangle \in E \mid u \in A \quad \text{and} \quad v \in A\}$$

For conciseness the notation of Gallai (1967) is adopted, in which $[A]_G$ refers to the subgraph of G that is induced by the vertex set A .

A *vertex weighted graph* $G = (V, E, W)$ is a graph extended by the vector $W = (w_1, w_2, \dots, w_{|V|})$, $w_i \in \mathbb{N}, w_i > 0$ containing the weights. Each integer w_i is associated with the vertex $v_i \in V$ and called the *vertex chromaticity* or simply *weight* of v_i . If no vertex chromaticities are given, it will be implicitly assumed that each vertex has weight one. Moreover a graph is said to be *composite* if its chromaticities are not all equal. Finally it is noted, that the edges of a graph can be assigned weights analogously. *Edge weighted graphs* are commonly encountered in the scheduling literature, but will not be of interest in this book. Here a *weighted graph* is always understood to be a vertex weighted graph.

Special (Sub-)Graphs and Vertex Sets

Consider the *undirected* graph $G = (V, E)$.

Graph G is *complete*, iff every pair of distinct vertices is adjacent, i.e. $(u, v) \in E, \forall u, v \in$

V , $u \neq v$. Consequently each graph $G^x = (V, E \cup \bar{E})$, derived by merging G and its complement \bar{G} , is a complete graph. By definition, each undirected complete graph satisfies²

$$|E| = \frac{|V|(|V|-1)}{2}$$

Coherently the *density* $\rho(G)$ of an undirected graph $G = (V, E)$ can be defined as³

$$\rho(G) := \frac{|E|}{\frac{|V|(|V|-1)}{2}}$$

Conversely, graph G is *degenerate* iff $|E| = 0$.

Definition 3.3 (Clique). *A subset $A \subseteq V$, $|A| = r$ of the vertices is a r -clique if it induces a complete subgraph.*

Trivially, each single vertex is a 1-clique. A clique is *maximal* if it is not properly contained in another clique of G and a clique is *maximum* if there is no clique of G with greater cardinality. The number of vertices in a maximum clique is called the *clique number* of G , denoted by $\omega(G)$.

Definition 3.4 (Stable Set). *A stable set is a subset $X \subseteq V$ of vertices, so that no two of them are adjacent. Formally*

$$X = \{u, v \in V \mid (u, v) \notin E\}$$

The *stability number* $\alpha(G)$ is the number of vertices in a stable set of maximum cardinality.

Definition 3.5 (Autonomous Set). *An autonomous set or module, $AS(G)$, of graph G is a subset $A \subseteq V$ of the vertices, such that*

$$\text{for every } v \in V - AS(G) : \exists a \in AS(G) \text{ with } (a, v) \in E \quad \text{or}$$

$$\text{for every } a \in AS(G) : \nexists (a, v) \in E \text{ with } v \in V - AS(G)$$

A graph G is *prime* if it contains only the following three *trivial autonomous sets*:

$$AS_1(G) = \emptyset, \quad AS_2(G) = \{v\}, v \in V, \quad AS_3(G) = V$$

Definition 3.6 (Proper Simple Coloring). *A proper simple c -coloring is a partition $\{X_i\}_{i=1, \dots, c}$ of the vertices V , such that each X_i is a stable set.*

²For complete directed graphs the amount of edges is doubled, resulting in $|\vec{E}| = |V|(|V|-1)$.

³Obviously $\rho(G) := \frac{|\vec{E}|}{|V|(|V|-1)}$ for a directed graph $\vec{G} = (V, \vec{E})$.

Each of the stable sets $X_i, i = 1, \dots, c$ is associated ("painted") with the color i and thus any two adjacent vertices have a different color. The *chromatic number* $\chi(G)$ is the smallest possible c for which a proper simple c -coloring of G exists. Furthermore, a graph G is called *bipartite* iff it is 2-colorable, i.e. $\chi(G) = 2$.

In a clique any two vertices are adjacent and have to be painted with a different color. Thus the following inequality holds

$$\omega(G) \leq \chi(G)$$

Moreover, it is easy to see that the maximum clique of G is a maximum stable set in the complement of G . Hence

$$\omega(G) = \alpha(\overline{G})$$

Consequently finding stable sets and cliques is essentially the same task.

Now consider the *directed* graph $\vec{G} = (V, \vec{E})$.

The *in-degree* $\delta^-(v)$ of a vertex $v \in V$ is defined as

$$\delta^-(v) := |\{u \in V \mid v \in \text{Adj}(u)\}|$$

i.e. the number of edges ending in v . Conversely the *out-degree* $\delta^+(v)$ is the number of edges originating in v : $\delta^+(v) = |\text{Adj}(v)|$. A vertex whose out-degree or in-degree equals zero is called *source* or *sink*, respectively. If both $\delta^+(v) = 0$ and $\delta^-(v) = 0$ hold, the vertex is said to be a *singleton* or *isolated*. Note that undirected graphs neither have a source nor a sink, because $\delta(v) = \delta^+(v) = \delta^-(v)$. The maximum (in- or out-) degree of any vertex is $\delta_{max} = |V| - 1$.

3.1.2 (Data) Representation of Graphs

It is often convenient to draw a 'picture' of a graph $G = (V, E, W)$. Since this can be done in many ways, a consistent pattern is proposed for the course of this text:

Let each vertex $v_i \in V, i \in \{1, \dots, |V|\}$ be represented by a circle. Then, a directed arrow is drawn from v_i to $v_j \in V, j \in \{1, \dots, |V|\}, j \neq i$ whenever $\langle v_i, v_j \rangle \in E$. Since any vertex v_i in an undirected graph will always have a forward and a backward arc to its adjacent vertices v_j , in this case the pair of arrows resulting from $\langle v_i, v_j \rangle$ and $\langle v_j, v_i \rangle$ is replaced by a single solid line between the circles representing v_i and v_j .

Furthermore, for weighted graphs, each circle is separated in an upper and a lower half, where the upper half is reserved to denote the *label* i of vertex v_i and the lower half shall contain the

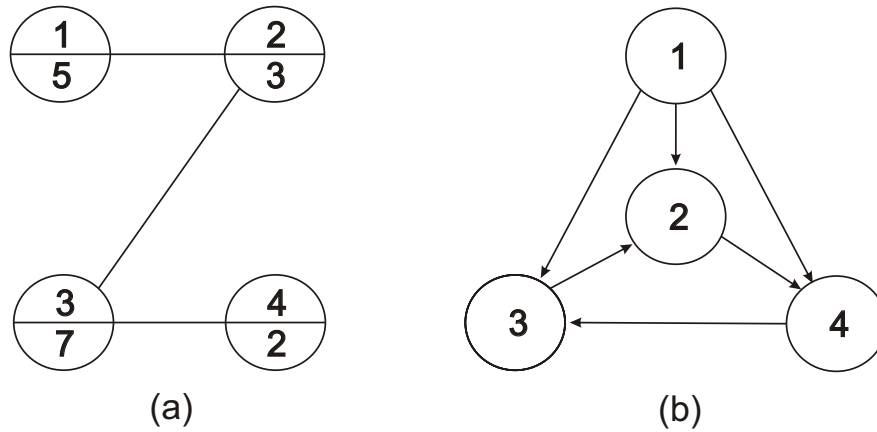


Figure 3.1: Two sample graphs: (a) undirected weighted graph (b) directed unweighted graph

weight w_i . If the graph is unweighted, the circle holds the label only. Two example graphs are drawn in figure 3.1.

Although the pictures derived from the above procedure might be very intuitive for the human, they are not an appropriate representation as input for computer algorithms. One of the more data-oriented representations is given through the *adjacency matrix* $\sigma_{i,j}$. The elements of this $|V| \times |V|$ matrix are defined as

$$\sigma_{i,j} := \begin{cases} 1 & : \text{if } \langle v_i, v_j \rangle \in E \\ 0 & : \text{otherwise} \end{cases}$$

Unfortunately the weights of G have to be stored in a separate $1 \times |V|$ matrix if needed. Consequently this representation has a space requirement of $|V|^2 + |V|$, independent of the cardinality of the edge set. This is especially unsatisfactory for sparse graphs ($|E| \ll |V|^2$), of course. In addition, for undirected graphs more than half of the data in $\sigma_{i,j}$ is redundant, because then the relation Adj on V is not only irreflexive ($\sigma_{i,i} = 0$), but also symmetric ($\sigma_{i,j} = \sigma_{j,i}$).

Holding a (sorted)⁴ *adjacency list* $Adj(v_i)$ for each vertex v_i is slightly more concise, because it stores only information of those edges, which are actually contained in the graph. However, the redundancy problem of undirected graphs still persists. Adjacency lists also have no natural way of storing information on the vertex chromaticities, such that a separate matrix is still needed for this purpose. Thus the storage requirements are at most $\sum_i \delta^+(v_i) + |V| \leq |V|^2$,

⁴Sorting refers to the labels of the vertices.

which reduces the space demand by at least $|V|$ units, formerly used by the $\sigma_{i,i} = 0$ entries. Moreover, adjacency lists can be implemented very efficiently by means of linked lists. A *list* is a data structure which consists of homogeneous *records* that are linked together through *pointers*. Each of these records has at least one data field and one pointer to the next record, but may store much more information, if necessary. The main advantage of linked lists is that the records of a list can be scattered throughout memory, whereas the data in an array, for example, must be stored sequentially in memory. The pointers maintain law and order, which allows for more flexibility when inserting or deleting items, because this is accomplished by some simple pointer operations rather than having to shift large blocks of data. In particular, each record represents an edge of the graph and each linked list corresponds to the adjacency list of a distinct vertex. Consequently $|V|$ such lists are needed, each of which again is accessible through a master pointer (see figure 3.2).

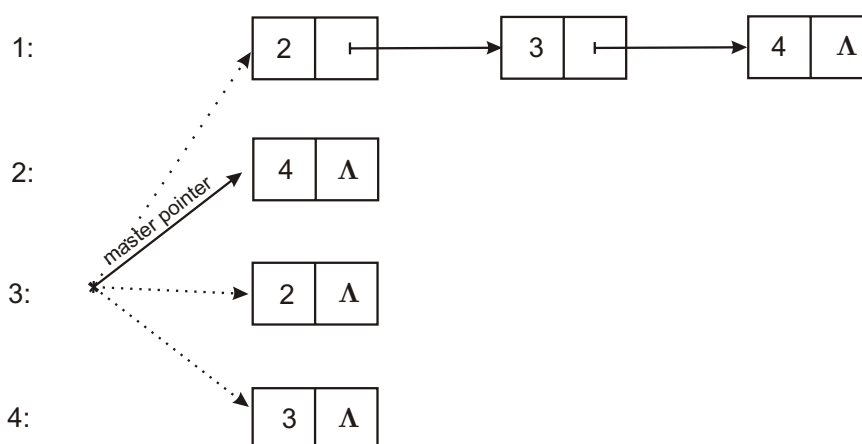


Figure 3.2: A sorted adjacency list representation of the sample (unweighted) graph from figure 3.1(b), implemented by linked lists. (The symbols \dashrightarrow and Λ denote a pointer and NULL, respectively)

Adjacency lists are not unconditionally the better choice for all operations, however. For instance, consider the task in which one seeks to determine whether the vertices v_i and v_j are adjacent. If the data was stored in an adjacency matrix, the result could be obtained by checking the two entries $\sigma_{i,j}$ and $\sigma_{j,i}$ only, whereas in an adjacency list one would have to browse through both $Adj(v_i)$ and $Adj(v_j)$. Thus the data representation has to be chosen in accordance with the specific operations of the algorithm to be run on it.

The above example shows that linked lists are a very general means of storing data and especially well suited for dynamic space requirements. Later in the text two specific types of

linked lists will be of importance. In a *queue* new records are always added to the end (tail) of the list, whereas only the first record (head) can be accessed or deleted from the list. Thus a queue represents a first-in-first-out (FiFo) storage strategy. Conversely, a *stack* mimics a last-in-first-out (LiFo) strategy, because it allows the addition, deletion and access of records at the head of the list only.

3.1.3 Vertex Sequences

Let $G = (V, E)$ be an arbitrary (directed or undirected) graph.

Definition 3.7 (Chain). *A finite sequence of vertices $\zeta = [v_1, v_2, \dots, v_l]$ in G , satisfying*

$$\langle v_i, v_{i+1} \rangle \in E \quad \text{or} \quad \langle v_{i+1}, v_i \rangle \in E, \quad \text{for all } i \in \{1, \dots, l-1\}$$

is called a chain of length l .

Chains can occur in both directed and undirected graphs.⁵ However, some authors restrict their attention to digraphs and refer to chains as *semi-paths* then. A *path* in \vec{G} is defined as

Definition 3.8 (Path). *A path from v_1 to v_l is a sequence of vertices $\zeta = [v_1, v_2, \dots, v_l]$ in \vec{G} , with*

$$\langle v_i, v_{i+1} \rangle \in \vec{E}, \quad \text{for all } i \in \{1, \dots, l-1\}$$

Thus in the case of undirected graphs the notion of chain and path coincide. A path or chain $[v_1, v_2, \dots, v_l]$ in G is called *closed* if $v_1 = v_l$, *simple* if no vertices other than $v_1 = v_l$ occur more than once in the sequence ($v_i \neq v_j, \forall i, j \in \{2, \dots, l-1\}$) and referred to as *trivial* if $l = 1$.

Definition 3.9 (Component). *A component $[C]_G$ is the subgraph of G induced by a maximal vertex set $C \subseteq V$, such that any two members of C are joined by a chain.*

The members of a component are said to be *connected*. If G has only one component $[C]_G$ with $C = V$, then G is connected.

The next definition is taken from Gilmore and Hoffman (1964):

Definition 3.10 (Cycle). *An l -cycle of graph $G = (V, E)$ is a finite sequence of vertices $\zeta = [v_1, v_2, \dots, v_l, v_1]$ of V , such that all of the edges $\langle v_i, v_{i+1} \rangle$, $1 \leq i \leq l-1$, and the edge $\langle v_l, v_1 \rangle$ are in E , and for no vertices u, v and integers $i, j < l, i \neq j$, is $u = u_i = u_j$, $v = v_{i+1} = v_{j+1}$ or $u = u_i = u_l$, $v = v_{i+1} = v_1$.*

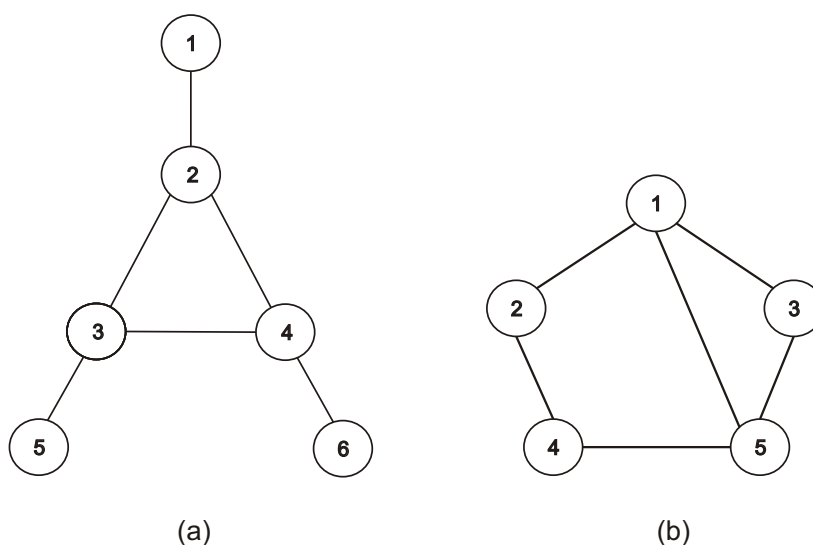


Figure 3.3: Two graphs having cycles

More figuratively speaking, a cycle is a closed path which may not contain the same edge more than once. In particular, for undirected graphs this means that no edge can be traversed twice in the same direction. To some extent this definition deviates from the narrow understanding of a cycle, which is usually considered to be a simple and closed sequence of vertices. The cycle defined here is closed, but not necessarily simple. Figure 3.3 illustrates this difference: The sequence $\zeta_a = [v_1, v_2, v_3, v_5, v_3, v_4, v_6, v_4, v_2, v_1]$ forms a cycle in graph (a) which is closed at vertex v_1 , but not simple. Sequence $\zeta_b = [v_1, v_2, v_4, v_5, v_3, v_1]$ in graph (b) is both closed and simple.

Gilmore and Hoffman further define a cycle $[v_1, v_2, \dots, v_l, v_1]$ to be *odd* if and only if l is an odd number. Finally, a *triangular chord* is introduced as

Definition 3.11 (Triangular Chord). A *triangular chord* of a cycle $[v_1, v_2, \dots, v_l]$ is any one of the edges $\langle v_i, v_{i+2} \rangle, 1 \leq i \leq l-2$, or $\langle v_{l-1}, v_1 \rangle$ or $\langle v_l, v_2 \rangle$.

A cycle is *chordless* if it has no triangular chord. Of the sequences ζ_a and ζ_b in graphs 3.3(a) and (b) respectively, only ζ_a is chordless.

⁵Recall that an undirected graph having edge $(u, v) \in E$ actually contains both $\langle u, v \rangle \in E$ and $\langle v, u \rangle \in E$.

3.2 Graph Representation of CSP and Immediate Conclusions

3.2.1 Constraint Graph

By now the graph representation of CSP - the *constraint graph* - can easily be established. First, the reader is again reminded to recall the special characteristics of CSP which introduced this book. Certainly the same properties must be reflected by the graph representation. Furthermore, the constraint graph is only then a legitimate input for graph-based solution procedures, if each instance of CSP is uniquely representable by it, and if each constraint graph is unambiguously transformable back into a schedule.

Thus let $G = (V, E, W)$ be an undirected weighted graph. The individual activities are modeled through the members of the vertex set $v_i \in V$, where w.l.o.g. i is a unique and consecutive label associated with each activity/vertex ($i = 1, \dots, |V|$). However, the labels do not imply any order of the vertices, since all activities can be executed independently. The activity duration is expressed in terms of the vertex weight w_i , and finally, an edge (v_i, v_j) is added to the edge set E , whenever the activities i and j are mutually incompatible ($\mathcal{R}_i \cap \mathcal{R}_j \neq \emptyset$).

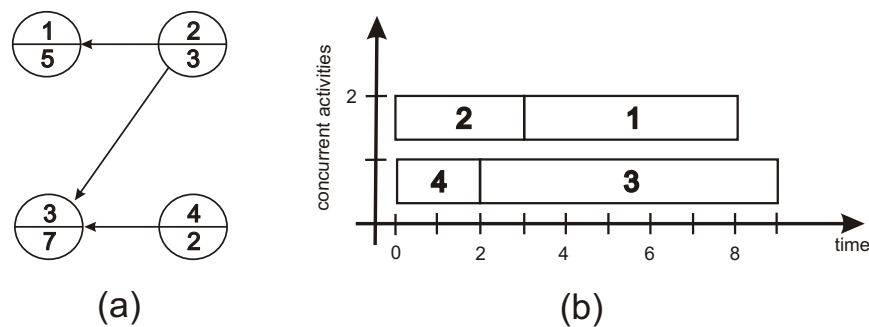


Figure 3.4: *Constraint Graph: (a) acyclic orientation (b) corresponding schedule represented by a Gantt-chart*

It is easy to see that the constraint graph satisfies all of the above requirements. A specific instance of CSP will therefore be denoted by $CSP(G)$ from now on. This gives also rise to an interchangeable usage of the terms 'vertex' and 'activity' as well as 'weight' and 'duration', when referring to the constraint graph. Furthermore, G facilitates the solution procedure tremendously compared to the pioneering approach of Bozoki and Richard, whose algorithm was based on adding single activities to a partial schedule, requiring complex calculations to be

performed in every iteration. A basic iteration of a constraint-graph-based solution procedure, however, is very simple. The idea is to direct the edges of G , such that the resulting orientation is acyclic. By directing an edge $\langle u, v \rangle$ of G , one implies a precedence relationship between the vertices, where u precedes v . Consequently every acyclic orientation corresponds to a partial order of the activities, which in turn may be translated back into a uniquely determined feasible schedule. Conversely, any feasible schedule can be expressed through a uniquely determined acyclic orientation. Figure 3.4 illustrates the equivalence of the two representations by means of the sample constraint graph from figure 3.1(a). Because of this duality, at least one acyclic orientation must exist which represents a feasible schedule with minimum makespan.

Theorem 3.12 (CSP and Acyclic Orientation). *Let $Z^*(G)$ denote the makespan of an optimal schedule of $CSP(G)$. Then $CSP(G)$ is equivalent to finding an optimal acyclic orientation \vec{F} of G , i.e.:*

$$Z^*(G) = \min_{\vec{F}} Z^*(\vec{G}) = \min_{\vec{F}} \left(\max_P \sum_{v_i \in P} w_i \right)$$

,where P denotes the set of nodes on a critical path of $\vec{G} = (V, \vec{F}, W)$

Proof: Notice, that any acyclic oriented graph $\vec{G} = (V, \vec{F}, W)$ must contain a path $\zeta = [v_i \mid v_i \in P]$, because it has both a source and a sink. Let $w(\zeta) = \sum_{v_i \in P} w_i$ be the weight of such path, then the critical path of \vec{G} is defined as the path in \vec{G} whose weight is maximum. The critical path represents the longest sequence of activities in the corresponding schedule and therefore $Z^*(\vec{G}) = \max_P w(\zeta)$. Since CSP does not prescribe any precedence relationships, an arbitrary acyclic orientation can be assigned to G . Because of the duality between acyclic orientations and feasible schedules, CSP can be solved optimally by finding the acyclic orientation with the minimum weighted critical path. ■

3.2.2 Complexity of CSP

Eventually all necessary concepts have been introduced in order to undertake the long promised complexity proof of CSP:

Let $G_1 = (V, E, W), w_i = 1, \forall i$ be the constraint graph representing the instances $CSP_1 = CSP(G_1)$, where each activity requires exactly one unit of processing time. Obviously CSP_1 is a relaxation of CSP and therefore CSP_1 must be at least as hard as the general CSP. Furthermore the following property holds:

Lemma 3.13. *CSP₁ is equivalent to finding a simple coloring of G_1 .*

Proof: (Roemer 2004b) Let S be a schedule with makespan k . All vertices in G_1 which represent activities that are performed concurrently under S can be assigned the same color, because all activities require one unit processing time only. The number of colors needed is therefore equal to the length of the schedule and thus G_1 is k -colorable. Conversely, given a k -coloring of G_1 , all activities which correspond to vertices having the same color can be executed concurrently, and there must be a schedule with makespan k . ■

Consequently CSP₁ can be restated as: "Find the chromatic number $\chi(G_1)$ ", and its decision version would ask: "Is G_1 k -colorable?"

In particular, the latter question can be evaluated in polynomial time, if provided with a k -coloring. One would simply have to check for every edge $(v_i, v_j) \in E$ if v_i and v_j have different colors. Therefore at most $O(|E|)$ steps are needed to verify or falsify the answer. Thus the decision version of CSP₁ is clearly in *NP*.

Karp (1972) proves that the decision version of CSP₁ is solvable in polynomial time for $k = 2$, but *NP*-hard for an arbitrary fixed $k \geq 3$. Of course, if the decision version of CSP₁ is *NP*-hard, then its optimization version is even more so. Furthermore the decision version of CSP₁ is *NP*-complete, since it is both *NP*-hard and in *NP*.

Finally, to show the strong *NP*-completeness of CSP₁, one has to prove that some polynomially sized instances of it cannot be solved in polynomial time. However, in fact *all* instances of CSP₁ have a space requirement of $O(|V|^2)$ and its strong *NP*-completeness follows directly.

Theorem 3.14 (Complexity of CSP). *CSP is NP-hard, whereas the decision version of CSP is NP-complete in the strong sense.*

Proof: Above ■

3.2.3 Decomposition Graph and Trivial Decomposition Theorems

The constraint graph also allows for the deduction of some trivial problem decompositions. In order to address this topic in detail, some definitions and theorems from the fundamental paper of Gallai (1967) have to be introduced. Kelly (1985) gives a very good exposition of Gallai's results and Maffray and Preissmann (2001) provide an English translation of his article.

Consider the constraint graph $G = (V, E, W)$. The *canonical decomposition* is defined as

Definition 3.15 (Canonical Decomposition). *A canonical decomposition is a unique proper partition $\Omega = \{AS_i(G)\}_{i \in \{1,2,\dots,n\}}$ into nonempty autonomous sets:*

1. *If $|V| = 1$ then $\Omega = \{V\}$*
2. *If G is disconnected into the components $[C_i]_G$, then $AS_i(G) = C_i, \forall i$.*
3. *If \bar{G} is disconnected into the components $[C_i]_{\bar{G}}$, then $AS_i(G) = C_i, \forall i$.*
4. *If both G and \bar{G} are connected, then Ω is the partition into disjoint autonomous sets $AS_i(G)$ of V , such that each $AS_i(G) \subset V$ is maximal.*

Gallai (1967) refers the above autonomous sets as *quasimaximal strong* autonomous sets ('quasimaximal stark geschlossen'). 'Quasimaximal' reflects the fact, that $AS_i(G) \subset V$ for $|V| \geq 2$, since trivially V itself is always the maximal autonomous set of G . 'Strong' means that either $AS_i(G) \cap AS_j(G) = \emptyset$ or $AS_i(G) \subset AS_j(G)$ or $AS_j(G) \subset AS_i(G)$, for $i \neq j$. A quasimaximal strong autonomous set $AS_i(G)$ is called *trivial* iff $|AS_i(G)| = 1$. The canonical decomposition is trivial iff all $AS_i(G)$ are. This is obviously only the case if G is prime. The canonical decomposition is uniquely defined for every graph and used to derive the *decomposition graph* G/Ω :

Definition 3.16 (Decomposition Graph). *The decomposition graph G/Ω is obtained by representing each autonomous set $AS_i(G)$ of the canonical decomposition by a single vertex v_i and adding edge (v_i, v_j) if and only if $AS_i(G)$ and $AS_j(G)$ are connected in G .*

Gallai states the following properties of decomposition graphs:

Theorem 3.17 (Properties of the Decomposition Graph). *Let G/Ω be the decomposition graph of graph $G = (V, E)$, then*

1. *The quasimaximal strong autonomous sets of the decomposition graph have cardinality one.*
2. *If G has one vertex only, so has G/Ω .*
3. *The decomposition graph G/Ω is degenerate if and only if G is disconnected.*
4. *The decomposition graph G/Ω is complete if and only if \bar{G} is disconnected.*

5. *The decomposition graph G/Ω is connected and has more than one vertex if and only if G and \bar{G} are connected and have more than one vertex. In this case G/Ω has at least four vertices and is prime.*

Proof: see Gallai (1967) ■

The first property stems directly from the quasimaximality of the autonomous sets: If G/Ω would contain a non-trivial autonomous set, then one $AS_i(G)$ of G cannot have been quasimaximal. The last four properties are trivially derived from the definition of the canonical decomposition and the decomposition graph. Recall that the vertices belonging to an autonomous set are either not connected to vertices outside of the set or connected to all of them. Therefore, in the latter case for example, the autonomous set is a component of the complement graph and thus the decomposition graph is complete, since each of the completely connected autonomous sets has been 'shrunk' into one vertex. It is pointed out that any decomposition graph must either be complete, degenerate or prime.

With the help of the decomposition graph, two trivial decompositions of CSP into smaller subproblems can directly be identified (Roemer 2004a):

Theorem 3.18 (Degenerate Decomposition Graph). *Let $Z^*(G)$ be the optimal makespan of $CSP(G)$. If the decomposition graph G/Ω , $\Omega = \{C_q\}_{q=1..n}$ is degenerate, then $Z^*(G) = \max_q \{Z^*([C_q]_G)\}$, where $Z^*([C_q]_G)$ is the optimal makespan associated with component $[C_q]_G$.*

Proof: Because the decomposition graph G/Ω is degenerate, the constraint graph $G = (V, E, W)$ must be disconnected and has components $[C_q]_G, q = 1, \dots, n$. Obviously, since there exists no chain between any vertex $u \in C_q$ and each $v \in C_r, r \neq q$, all components can be scheduled concurrently. Consequently $Z^*(G)$ coincides with the optimal makespan $Z^*([C_q]_G)$ of the component $[C_q]_G$, having the longest optimal schedule. Hence it suffices to solve CSP on every subgraph $[C_q]_G$ to determine an optimal schedule of $CSP(G)$. ■

Furthermore the logic behind theorem 3.18 can be reversed to yield another decomposition theorem:

Theorem 3.19 (Complete Decomposition Graph). *Let $Z^*(G)$ be the optimal makespan of $CSP(G)$. If the decomposition graph G/Ω , $\Omega = \{C_q\}_{q=1..n}$ is complete, then $Z^*(G) = \sum_{q=1}^n Z^*([C_q]_G)$, where $Z^*([C_q]_G)$ is the optimal makespan associated with component $[C_q]_G$.*

Proof: Because the decomposition graph G/Ω is complete, the complement $\bar{G} = (V, \bar{E}, W)$ of $G = (V, E, W)$ must be disconnected into the components $[C_q]_{\bar{G}}, q = 1, \dots, n$. Since there

is an edge $(u, v) \in E$ between every $u \in C_q$ and each $v \in C_r, r \neq q$ any two activities, which are not represented by vertices of the same autonomous set, are in conflict and thus cannot be processed simultaneously. Consequently each autonomous set (component of \overline{G}) has to be scheduled sequentially. ■

Considering the exponential time complexity of CSP any decomposition into non-trivial subproblems contributes substantially to the performance of a solution procedure. As shown in this section, problem decompositions can be easily derived from the graph representation of CSP. For completeness it is noted that yet another decomposition possibility will be presented in section 5.3 which may also be applied to prime decomposition graphs.

3.3 The Maximum Weighted Clique Problem

Inspired by the proof of theorem 3.19, one can derive another immediate result from the constraint graph $G = (V, E, W)$. The concepts of maximal and maximum cliques are not sufficient for the notion of weighted graphs, however, and thus a *maximum weighted clique (MWC)* is introduced as follows

Definition 3.20 (Maximum Weighted Clique). *A maximum weighted clique is a maximal clique, whose sum of vertex chromaticities is maximum.*

The sum of vertex chromaticities in a MWC of graph G is called the *weighted clique number*, denoted by $\omega_w(G)$.

Obviously any clique represents a set of mutually conflicting activities which must be performed concurrently. This is the same notion that Bozoki and Richard (1970) describe when referring to the lower bound \underline{Z}_2 (cf. section 2.3.2). Hence the weighted clique number, $\omega_w(G)$, constitutes a lower bound of the project duration and is valuable information when applying a branch-and-bound procedure. Unfortunately Karp (1972) proofed the maximum weighted clique problem, i.e. the computation of $\omega_w(G)$, to be *NP-hard*, like CSP itself. Consequently exact solution procedures are not suitable for the determination of the lower bound $\underline{Z}^w = \omega_w(G)$. Therefore this section will provide an overview of the most important heuristic solution procedures for the MWC problem, as found in Pardalos and Xue (1994).

3.3.1 Sequential Greedy Heuristics

In general *sequential greedy heuristics (SGH)* approximate a solution by consecutively computing locally best decisions. Sequential greedy heuristics for the MWC problem generate a clique through the repeated addition of a vertex into a partial clique, or through the repeated deletion of a vertex from a set that is not a clique. Due to their sequential nature SGH have a relatively small memory demand and perform quickly in practice, making them a preferred choice for large problems (graphs). Thus it is not surprising that sequential greedy heuristics form the majority of the approximation algorithms found in the MWC literature.

SGH which sequentially add vertices to a clique are referred to as *best-in*, whereas those that operate with the deletion of vertices from a non-clique are called *worst-out* (Kopf and Ruhe 1987). Both methods use an indicator to rank the candidate vertices to be moved in or out. For a best-in SGH such an indicator may be the degree of a vertex or the weight of neighboring nodes, for example. Kopf and Ruhe (1987) differentiate each class (best-in and worst-out) further into *new* and *old*, respectively describing whether the indicator is updated with every iteration of the algorithm or not. Algorithm 3.1 describes a new, best-in sequential greedy heuristic for the maximum weighted clique problem. The indicator used here is the sum of the vertex chromaticities of the neighborhood of a candidate node (line 3). In order to evaluate the indicator at most δ_{max} neighbors have to be identified for each vertex, resulting in a time complexity of $O(\delta_{max}|V|)$. Since the while-loop is executed at most $O(|V|)$ times, the complexity of algorithm 3.1 amounts to $O(\delta_{max}|V|^2)$.

Input : An undirected graph $G = (V, E, W)$
Output: An approximation for a maximum weighted clique

- 1 *Initialize*: $R = V, C = \emptyset$;
- 2 **while** $R \neq \emptyset$ **do**
- 3 select vertex $v_i \in R$, such that $\sum_{v_j \in Nei(v_i)} w_j$ is maximum ;
- 4 $C = C \cup v_i$;
- 5 $R = R \cap Nei(v_i)$;
- 6 **return** C ;

Algorithm 3.1: A new, best-in heuristic for the maximum weighted clique problem.

3.3.2 Local Search Heuristics

The main defect of sequential greedy heuristics is, that they find exactly one maximal clique only. More precisely, let \mathcal{C} be the space consisting of all maximal cliques, one of which has

maximum weight. Then a SGH identifies one point $p \in \mathcal{C}$, which may (or may not) be (close to) the optimal point. *Local search heuristics*, on the contrary, make an attempt to improve this initial solution by expanding the search to the neighbors of p . A *k-interchange* heuristic, for example, explores all points q , which are within a distance of $|p - q| \leq k$ from the initial point p , measured by an appropriate metric defined on \mathcal{C} . The improvement comes at the cost of increased computational effort, of course, which grows for a k-interchange heuristic roughly with $O(|V|^k)$.

3.3.3 Randomized Heuristics

It is easy to see that the quality of a local search heuristic rises and falls with the quality of the starting point p . One possible solution to overcome a poor initial clique is to examine an even wider neighborhood. However, this is a very costly choice as the computational complexity increases polynomially with the neighborhood. As a consequence *randomized heuristics* have been designed to search the neighborhood of various points in \mathcal{C} . The modifications necessary to turn a local search heuristic into a randomized heuristic are very small indeed. In essence, only the sequential greedy heuristic, which derives the initial point, has to be extended by random factors, so that a different output is computed on each run. Feo, Resende, and Smith (1994) have obtained encouraging results from such a randomized heuristic having only the three basic components of SGH, probabilistic selection procedure and local search technique. In particular, a new, best-in greedy heuristic similar to the one described in algorithm 3.1 is employed to derive an initial clique.⁶ In order to achieve a random outcome, however, not necessarily the 'best' vertex will be added to the partial clique, but rather any one of the top ranked vertices as determined by the probabilistic component.

3.3.4 Tabu Search

Finally, one last approach, also based on local search, shall briefly be sketched due to its relevance in practice: *Tabu search* performs a local search in the most promising direction of \mathcal{C} , determined by some indicator. To prevent the procedure from circling between local minima, a list of points already visited is maintained, which are 'tabu' for future examination. A sam-

⁶The indicator proposed here is the vertex degree.