CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Maastricht University Research Portal

Combinatorial Approximation Algorithms: Guaranteed Versus Experimental Performance

Citation for published version (APA):

Vredeveld, T. (2002). Combinatorial Approximation Algorithms: Guaranteed Versus Experimental Performance. Universiteit Eindhoven.

Document status and date:
Published: 01/01/2002

Document Version:
Publisher's PDF, also known as Version of record

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.umlib.nl/taverne-license

Take down policy
If you believe that this document breaches copyright please contact us at:

repository@maastrichtuniversity.nl

providing details and we will investigate your claim.

Download date: 04 Dec. 2019

Combinatorial Approximation Algorithms

Guaranteed Versus Experimental Performance

Tjark Vredeveld

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Vredeveld, Tjark

Combinatorial approximation algorithms : guaranteed versus experimental performance / by Tjark Vredeveld. - Eindhoven : Technische Universiteit Eindhoven, 2002.
Proefschrift. - ISBN 90-386-0532-3
NUGI 811
Subject headings: combinatorial optimisation / approximation algorithms
2000 Mathematics Subject Classification: 90C27, 90C59, 68W25, 68W40

Painting on the cover by Toon Jakobs (CLOSED UP, 1990)
Printed by Universiteitsdrukkerij Technische Universiteit Eindhoven

Copyright © 2002 by T. Vredeveld, Eindhoven, The Netherlands.

All rights are reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission of the author.

Combinatorial Approximation Algorithms

Guaranteed Versus Experimental Performance

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor
een commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 23 april 2002 om 16.00 uur

door

Tjark Vredeveld

geboren te Leiden

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. J.K. Lenstra
en
prof.dr.ir. C. Roos

Copromotor:
dr.ir. C.A.J. Hurkens

Contents

Preface

Five years ago, after my graduation, I decided to pursue my PhD. I started as a PhD student at Erasmus University in Rotterdam. The subject of my research was multi-agent systems.

Four years ago, I realized that my interest in multi-agent systems was too little to write a thesis about this subject. I therefore changed the subject to approximation algorithms for combinatorial optimization and started all over again. I became a PhD student in the combinatorial optimization group of Eindhoven University of Technology under the supervision of Leen Stougie and Jan Karel Lenstra. My research was part of the project "High Performance Methods for Mathematical Optimization" of the Netherlands Organization for Scientific Research (NWO).

Three years ago, there was again a change of subject, but a smaller one. I now started doing research on the empirical behavior of approximation algorithms and also looked into the worst-case behavior of local search methods. With this change of research Cor Hurkens was brought into the team of supervisors and Leen became less involved.

Two and a half years ago, like a true PhD student, I thought about stopping pursuing my PhD. Fortunately, my supervisors, colleagues, and friends helped me through this difficult period and after a few months I was back on track.

One and a half year ago, Dick den Hertog, Leen Stougie, and I organized the Fifth International Conference on High Performance Optimization Techniques. I enjoyed working with Dick and Leen.

Today my thesis is finished. I realize I could not have done this without the help of many people. Therefore, I would like to thank every person who made this possible. First of all, I would like to thank Jan Karel Lenstra for asking the right questions, giving good advice and helping to improve my writing. Cor Hurkens was valuable for his good advice and checking my proofs. I enjoyed writing two papers with him. I am grateful to Leen Stougie for his interest in my research, which kept me motivated. Kees Roos initiated the NWO project and I am grateful for that. I also would like to thank him for proofreading my thesis. Gerhard Woeginger and Emile Aarts are the other members of my reading committee. I thank them for their remarks.

Today, I also would like to thank Petra Schuurman for the pleasant collaboration, which resulted in my first refereed publication. I am grateful to Toon for allowing to use one of his paintings on the cover of this thesis, to Hans for making the picture, and to Eline and Marko for editing the electronic version of this picture.

Today, I thank my colleagues and former colleagues of the section operations

research and statistics of Eindhoven University of Technology for making Eindhoven a fun place to work at, Lavazza for making good coffee, and Dutch theater groups for making plays that occasionally took my mind of my research. Finally, I like to express my gratitude to my friends and family for their support. A special thanks goes to my two paranymphs Hugo and Oscar and to Eline for their support and encouragement at times when I had severe doubts and for the necessary relaxation.

Tjark Vredeveld
Eindhoven, February 2002

1

Introduction

For many combinatorial optimization problems it is hard to find an optimal solution. A lot of effort has therefore been put in the design and analysis of approximation algorithms for these types of problems. Such algorithms do not necessarily find an optimal solution, but attempt to find a good solution. Some of these approximation algorithms are designed from a theoretical point of view and give worst-case guarantees on the quality of the solution as well as on the running time for obtaining the solution. Others are designed from a practical point of view. They work well in practice, but we hardly know of any guarantees on the quality of the solution or on the running time. The latter group contains local search heuristics.

In this thesis we consider for several hard combinatorial optimization problems methods from one of both categories and judge them on the criteria of the other category. We have implemented some approximation algorithms, for which we know guarantees, and we compared their empirical performance to the performance of basic local search heuristics. In last two chapters of the thesis, we analyze from a worst-case perspective the quality of solutions obtained by the simplest form of local search. We also analyze the time needed for obtaining these solutions.

This introductory chapter continues with a description of combinatorial optimization problems in Section 1.1. We next describe what we mean by approximation algorithms with guarantees in Section 1.2. An introduction to local search is given in Section 1.3. This chapter concludes with an overview of the thesis in Section 1.4.

1.1 Combinatorial optimization

In combinatorial optimization one is asked to make the best choice among a number of alternatives. A *combinatorial optimization problem* is specified by a set of problem

instances and is either a *minimization* problem or a *maximization* problem.

Definition 1.1. *An* instance *of a combinatorial optimization problem* Π *consists of a set of feasible solutions,* S*, and an objective function,* $f : S \to \mathbb{R}$*, that assigns a real value to each solution. The problem is to find an optimal solution, i.e., a solution* $s^* \in S$ *such that*

$$f(s^*) \leq f(s) \quad \textit{for all} \quad s \in S$$

in the case that Π *is a minimization problem or*

$$f(s^*) \geq f(s) \quad \textit{for all} \quad s \in S$$

in the case that Π *is a maximization problem.*

Usually, the solution set S and the objective function f are given by an implicit description.

Some problems turn out to be easier to solve than others. We are interested in determining the time required to compute an optimal solution to a problem as a function of the length of the encoding of its input I, which we denote by $|I|$ and refer to as the *input size*. We say that a problem is *easy* to solve if there is a *polynomial-time algorithm* that solves it to optimality. A polynomial-time algorithm is one that runs in polynomial time, i.e., there is a polynomial p such that the algorithm applied to an input I always finds a solution within time $p(|I|)$. In this thesis we focus on *NP-hard* problems, which are not known to be easy and are even unlikely to be easy. We refer to Garey and Johnson (1979) for a thorough treatment of this subject. In the following examples, we present some NP-hard optimization problems.

Example 1.1. *The Santa Claus Sock Problem (SSP)*
At Christmas, Santa Claus wants to give presents to all children. For each family he has a bag filled with gifts, which he has to divide among the children of that family. Although he knows that it is rarely possible to give each child exactly the same value, Santa wants to avoid jealousy as well as he can. Knowing that a child always compares what it has got to the total value of the presents given to its sibling that has received most, Santa wants to minimize the value of gifts given to this fortunate brother or sister.

An instance of Santa's problem is given by the number of children, the bag with presents, and a value for each present. A solution is specified by the contents of the sock of each child. The weight of a solution is computed by taking the value of the gifts in the sock of the most fortunate child.

Example 1.2. *The Santa Claus Bag Problem (SBP)*
Santa has chosen not to divide the presents made in his factory at the North Pole directly to the socks of all children, because this problem is too large to handle. He therefore has to decide which present goes into which bag. That is, the gifts that are made in his factory have to be distributed over all bags. Again Santa wants to

avoid jealousy and he tries to divide the gifts such that the maximum ratio of the value in a bag divided by the number of children in the family is minimized.

An instance of this problem is specified by the number of families, the number of children in each family, the contents of the stock of his factory, and the value of each present in that stock. A solution is represented by a distribution of all presents over the bags, and the value of a solution is the maximum ratio over all bags of the value in the bag divided by the number of children who get their gifts from this bag.

Example 1.3. *The Traveling Santa Claus Problem (TSP)*
Once Santa has decided which child receives what presents, he still has to visit each family to deliver the gifts. He has to decide on a route that visits all families, starting and finishing at the North Pole. As Santa wants to enjoy his Christmas Eve with Mrs. Claus and his elves, he wishes to return to the North Pole as early as possible.

An instance of this problem is given by the number of families which Santa has to visit and a matrix of distances between each pair of chimneys. A solution is specified by a closed tour through the chimneys of all families and the North Pole. The value of a solution is the sum of all distances traversed in the solution.

Example 1.3 is a traveling salesman problem, see Lawler, Lenstra, Rinnooy Kan, and Shmoys (1985). Examples 1.1 and 1.2 are examples of multiprocessor scheduling problems that are discussed in Chapter 4. In these problems, we want to schedule jobs on multiple processors. Each job has a given processing time. The objective is to minimize the time at which the last job finishes. In Example 1.1, the processors are identical, i.e., each job has the same processing time on all processors. The processors in Example 1.2 are uniform: each processor has a given speed and the time required to complete a job is the job's processing requirement divided by the speed of the processor. In Chapter 4, we also consider unrelated processors: the time needed to process a job depends on the processor as well as the job.

Other combinatorial optimization problems considered in this thesis are the following. Chapter 2 is concerned with the scheduling of jobs on unrelated parallel processors. Each job has, besides given processing times, a given weight. The objective is to minimize the sum of the weighted job completion times. In Chapter 3 the max cut problem is considered. In this problem we want to partition the set of vertices of a graph into two disjoint subsets such that the total weight of the edges (interactions) between vertices in different subsets is maximized. In Chapter 5 we want to partition a set of items into k disjoint subsets, such that the total interaction within the subsets is minimized. We call this problem the generalized graph coloring problem. All these problems are NP-hard, as can be seen in the subsequent chapters.

One way to deal with NP-hard problems is finding near-optimal solutions within reasonable time. Algorithms that find such solutions are *approximation algorithms* or *heuristics*. In the following two sections, two types of approximation algorithms are discussed: algorithms with guarantees on the quality of the solution as well as the time needed to find them, and local search heuristics. These two classes of

heuristics are not disjoint, as we will see in Chapter 4.

1.2 Approximation algorithms with guarantees

The last decades, there has been a lot of interest in *polynomial-time ρ-approximation algorithms*. These are algorithms that run in polynomial time and give a guarantee on the quality of the solution.

Definition 1.2. *An algorithm A is called a* ρ-approximation algorithm *for a combinatorial optimization problem Π if it delivers a feasible solution with value between 1 and ρ times the optimal solution value. That is, for each instance I of a minimization problem Π, $A(I) \leq \rho OPT(I)$, and for each instance I of a maximization problem Π, $A(I) \geq \rho OPT(I)$, where $OPT(I)$ denotes the value of an optimal solution of I and $A(I)$ denotes the value of the solution obtained by algorithm A on input I. We refer to ρ as the* performance guarantee *or* worst-case ratio *of the algorithm.*

The value ρ can be viewed as the quality measure of an algorithm: the closer it is to 1, the better the algorithm is. Note that for minimization problems we have $\rho \geq 1$ and for maximization problems we have $\rho \leq 1$.

As an illustration we give a simple approximation algorithm for the Santa Claus Sock problem.

Example 1.4. *SSP continued: an approximation algorithm*
Santa has to divide the presents in a bag over the m children in the family. One way to do this is to take a gift from the bag and put it in the sock of the child that currently has the least. This algorithm is a polynomial-time $(2-\frac{1}{m})$-approximation algorithm, as can be seen as follows. Let I be an instance for this problem, let $A(I)$ denote the value of the solution obtained by the above algorithm, and let $OPT(I)$ denote the optimal solution value. Furthermore, let the value of the n presents be given by $v_1, \ldots, v_n$. The best Santa can hope for is that each child receives exactly the same value as its siblings. Hence $OPT(I) \geq \frac{1}{m}\sum_j v_j$. Also, $OPT(I) \geq v_k$ for any k, since a present may not be broken. We know that at the moment at which the most fortunate child receives its last present, having value v_k, this child was not having more than any its brothers and sisters. By an averaging argument, we know that $A(I) - v_k \leq \frac{1}{m}(\sum_j v_j - v_k)$. Hence $A(I) \leq \frac{1}{m}\sum_j v_j + \frac{m-1}{m}v_k \leq (2-\frac{1}{m})OPT(I)$. The last inequality is due to the above lower bounds on $OPT(I)$. This algorithm was first analyzed by Graham (1966). A variant of this algorithm, in which Santa takes the gifts from the bag in order of non-increasing value, has been tested by a.o. Finn and Horowitz (1979) and França, Gendrau, Laporte, and Müller (1994). They report that this so-called LPT algorithm finds solutions within a few percent of optimal.

SSP and SBP can be approximated very well: for every constant $\rho > 1$, there is a polynomial-time ρ-approximation algorithm, see Hochbaum and Shmoys (1987, 1988). Such a family of polynomial-time $(1+\epsilon)$-approximation algorithms for all $\epsilon > 0$ is called a *polynomial-time approximation scheme*, or PTAS for short. If the

time complexity of the PTAS is also polynomially bounded in $\frac{1}{\epsilon}$, it is called a *fully polynomial-time approximation scheme*, or FPTAS. For more details on approximation algorithms with guarantees for NP-hard problems we refer to Hochbaum (1997), Ausiello, Crescenzi, Gambosi, Kann, Marchetti-Spaccamela, and Protasi (1999), and Vazirani (2001).

In Chapters 2 and 3 of this thesis, we analyze polynomial-time approximation algorithms empirically and compare them to the experimental performance of local search heuristics. Previous work of this sort includes the following.

As mentioned above, Finn and Horowitz (1979) and França, Gendrau, Laporte, and Müller (1994) have tested the LPT algorithm for the multiprocessor scheduling problem with identical processors. The performance guarantee of this algorithm is $\frac{4}{3} - \frac{1}{3m}$. Finn and Horowitz report that the quality of the solutions found by this algorithm differs hardly from that of the solutions obtained through a simple local search heuristic. França et al. propose a more sophisticated local search algorithm, which performs better than the local search procedure of Finn and Horowitz and also better than the LPT algorithm. Haupt (1989) surveyed priority rule-based scheduling for the job shop scheduling problem, giving many references to empirical studies.

Johnson and McGeoch (1997) presented a case study in local optimization for the traveling salesman problem, in which they also experimentally analyzed several constructive heuristics with a worst-case or probabilistic performance guarantee.

Hariri and Potts (1991) examined the empirical behavior of the earliest completion time (ECT) heuristic and several two-phase heuristics for the problem of minimizing makespan on unrelated parallel processors. In the first phase of the two-phase algorithms, an LP-relaxation is solved to generate a partial schedule; the second phase consists of an exact or heuristic method to schedule the remaining jobs. The performance guarantees of the two-phase heuristics they examined are 2 and $(2 + \lfloor \log_2(m-1) \rfloor)$, the performance guarantee of the ECT heuristic is m. Their conclusion was that these constructive methods are quite unsatisfactory, as deviations of more than 10% from the optimal solution value are common. They also applied an improvement heuristic to the obtained solutions, which achieved a significant reduction in the makespan at very small computational costs. Glass, Potts, and Shade (1994) extended this research by analyzing the empirical performance of other local search heuristics for this problem. Although it is difficult to compare the results of these two papers, as the quality of the solution is measured as the relative deviation from the best obtained solution, which might not be the same in both papers, and the time spent on the local search heuristics is not equivalent to the time spent on the constructive heuristics, the results indicate that genetic descent, simulated annealing, and tabu search outperform the constructive heuristics; the local search heuristics have a deviation of about 1%.

Savelsbergh, Uma, and Wein (1998) studied the quality of lower bounds for the problem of minimizing total weighted completion time on a single processor with release dates for the jobs, obtained by LP-relaxations, and they also studied the quality of upper bounds delivered by a number of approximation algorithms based

on these relaxations. The best algorithms come within a few percent of the optimum. Although there are a few exceptions, they also concluded that the higher the quality of the solution to the LP-relaxation, the better the approximation algorithms performs. Uma and Wein (1998) made an analytical and empirical comparison of lower bounds for the same problem and also used some rounding techniques to obtain feasible solutions. They also applied simple local search heuristics to the solutions obtained by these approximation algorithms as well as to solutions from scratch. The best results were obtained by applying the local search heuristics to the solutions obtained by good approximation algorithms.

Van der Linden (2000) made a computational study on the empirical behavior of a convex quadratic approximation algorithm of Skutella (2001) for the problem of minimizing the total weighted completion time on unrelated parallel processors with release dates for the jobs. She compared the results to the solutions of an LP-based approximation algorithm by Schulz and Skutella (1997a). On all test instances, the LP-relaxation gave better lower bounds than the convex quadratic relaxation. In Chapter 2, we prove that for the case of equal release dates the LP-relaxation is guaranteed to give better bounds than the convex quadratic relaxation. Van der Linden also used the convex quadratic relaxation in a branch-and-bound method, which was able to solve problem instances with up to 20 jobs and 5 processors.

1.3 Local search

Local search in combinatorial optimization dates back to the late 1950s, when the first edge-exchange algorithms for the traveling salesman problem were introduced, see Bock (1958) and Croes (1958). We refer to Aarts and Lenstra (1997) for an overview of local search in combinatorial optimization.

Local search is a family of methods that iteratively search through the set of solutions. Starting from an initial solution, a local search procedure moves from one feasible solution to a neighboring solution until some stopping criteria are met. The choice of a suitable neighborhood function has an important influence on the performance of local search. These neighborhood functions define the set of solutions to which the local search procedure may move to in a single iteration. This is formalized in the following definition.

Definition 1.3. *For an instance of a combinatorial optimization problem with feasible solution set S, a* neighborhood function *is a mapping $\mathcal{N} : S \to 2^S$, which defines for each feasible solution $s \in S$ a set $\mathcal{N}(s) \subseteq S$ of feasible solutions that are in some sense close to s. The set $\mathcal{N}(s)$ is the* neighborhood *of the feasible solution s, and each $s' \in \mathcal{N}(s)$ is a* neighbor *of s.*

The simplest form of local search is *iterative improvement*. This method starts from an initial solution and repeatedly selects a neighbor as long as it improves on the current solution. The algorithm stops when it has found a *local optimum with respect to $\mathcal{N}$*.

Definition 1.4. *Let S be the solution set of an instance for a combinatorial opti-*

mization problem with weight function f and let $\mathcal{N}$ be a neighborhood function. A feasible solution $\hat{s} \in S$ is locally optimal with respect to $\mathcal{N}$ *if*

$$f(\hat{s}) \leq f(s) \quad \textit{for all} \quad s \in \mathcal{N}(\hat{s})$$

in case of a minimization problem and if

$$f(\hat{s}) \geq f(s) \quad \textit{for all} \quad s \in \mathcal{N}(\hat{s})$$

in case of a maximization problem.

Example 1.5. *SSP & SBP continued: a neighborhood function*
For SSP, once Santa has an assignment of the gifts to the children, he can try to improve upon it by moving a present from one child to another. For SBP, Santa could do a similar thing: move a present from one bag to another. The neighborhood of a solution is thus the set of all solutions in which exactly one present is in another sock or bag. Finn and Horowitz (1979) and Brucker, Hurink, and Werner (1997) show that there exists an iterative improvement procedure using this neighborhood function which is a polynomial-time $(2 - \frac{2}{m+1})$-approximation algorithm for SSP. Finn and Horowitz also showed that the experimental performance of this algorithm differs little from the experimental performance of the algorithm described in Example 1.4, where the presents are taken in non-increasing order of value from the bag. For SBP, there exists an iterative improvement procedure using this neighborhood function that is a polynomial-time $(1 + \sqrt{4m - 3})/2$-approximation algorithm (Cho and Sahni (1980) and Chapter 4).

The main drawback of iterative improvement is that it gets trapped in the first local optimum that it encounters, which may be poor. A way to overcome this drawback is by allowing moves to non-improving neighbors. Below, we discuss three popular types of local search heuristics that allow such moves: *tabu search*, *simulated annealing* and *variable-depth search*.

Tabu search, introduced by Glover (1989, 1990), always moves to the best neighbor, even when it is a deterioration. In this way the search can be directed away from local optima, such that other parts of the search space can be explored. To avoid cycling, i.e., returning to a solution encountered before, some information about solutions visited in past iterations is stored in a so-called *tabu list*. If a neighbor satisfies the properties of an entry in the tabu list, this solution is tabu and it may not be visited, unless it has value that is lower than the best found solution so far or satisfies some other 'aspiration criteria'. The tabu list contains a limited number of entries, and a limited number of forbidden properties.

Simulated annealing was introduced independently by Kirkpatrick, Gelatt, and Vecchi (1983) and Černy (1985). Like tabu search it may accept moves to non-improving neighbors. In simulated annealing, we choose randomly and uniformly a solution, s', in the neighborhood of the current solution, s. If this neighbor is a better solution, then it is accepted. If it is not an improving neighbor, then s' is accepted

with probability $e^{-|f(s)-f(s')|/T}$, where T is a positive control parameter, called the temperature. T is gradually decreased during the execution of the algorithm and thus the probability of accepting deteriorations is decreased. This lowering of the probability is determined by the *cooling schedule*.

Variable-depth search, introduced by Kernighan and Lin (1970), generates from an initial solution a sequence of subsequent neighbors. From this sequence, one solution is selected to serve as the initial solution for a next sequence. Variable-depth search can be viewed as a two-layered process that generates the sequence of neighbors in the first layer. In the second layer, it applies a search method, like iterative improvement, to the solutions selected in the fist one. Variable-depth search can also be seen as a neighborhood function that given a solution outputs the solution selected from the sequence of neighbors, see Chapter 4 and 5. The term variable-depth search is due to the fact that, on forehand, it is unknown which neighbor from the sequence is selected in the first layer.

The last two chapters of this thesis is concerned with the worst-case behavior of local search. Previous results on worst-case analysis of local search include the following.

Finn and Horowitz (1979) show that for the multiprocessor scheduling problem with identical processors (Example 1.1) a local optimum with respect to the neighborhood defined in Example 1.5 is guaranteed to have value at most $2 - \frac{2}{m+1}$ times the optimal solution. They also propose an iterative improvement procedure that finds a local optimum in a quadratic number of iterations (Brucker, Hurink, and Werner 1996, 1997).

Hurkens and Schrijver (1989) considered a packing problem in which we want to find the largest collection of pairwise disjoint sets among a given collection of k-sets. They gave a worst-case ratio guarantee on the quality of a local maximum and showed that an iterative improvement procedure runs in polynomial time. De Bontridder, Halldórsson, Halldórsson, Hurkens, Lenstra, Ravi, and Stougie (2001) extended this result to the problem in which we want to find the largest collection of disjoint paths of length 2 in a graph.

Korupolu, Plaxton, and Rajaraman (1998) show that iterative improvement for the metric uncapacitated facility location problem in polynomial time yields a solution with value no more than $5+\epsilon$ times the optimal solution value, for $\epsilon > 0$. They also show that for the metric uncapacitated k-median problem local search finds a solution using at most $(3 + 5/\epsilon)k$ facilities and having costs at most $1 + \epsilon$ times the optimal costs, for $\epsilon > 0$.

The metric labeling problem is the problem in which, given a graph, we want to assign labels to the vertices of a graph. The objective is to minimize the sum of the cost of assigning the label to a vertex and the interaction costs between each pair of adjacent vertices. The interaction cost is a function of the labels given to the vertices and it should be a metric. Boykov, Veksler, and Zabih (1999) showed that a local optimum is a 2-approximation when the interaction costs are all equal to 1. Gupta and Tardos (2000) show for a truncated metric, where the interaction cost

is the minimum of the distance between the two labels and some constant, that a local minimum is at most 4 times the optimal cost.

On the max-cut problem, it is easy to see that a local optimal solution with respect to the flip-neighborhood has value at least half the optimal cut value. Using the same neighborhood, Feige, Karpinski, and Langberg (2000) show that starting from a solution obtained by the approximation algorithm of Goemans and Williamson (1995) iterative improvement improves the performance guarantee of the starting solution, if the graph has bounded degree; in case of graphs with maximum degree three, a local optimum obtained in this way has a performance guarantee of 0.921, whereas the Goemans-Williamson algorithm is guaranteed to give a solution with value at least 0.878 times the optimal cut value.

Chandra, Karloff, and Tovey (1999) show that a 2-optimal tour for the TSP satisfying triangle inequalities is at most $4\sqrt{n}$ times the optimal value and that a lower bound on this guarantee is $\sqrt{n}/4$.

1.4 Overview

The outline of this thesis is as follows. Chapters 2 and 3 are empirical evaluations of approximation algorithms. Chapter 2, which is based on Vredeveld and Hurkens (2001), presents an empirical comparison of polynomial-time approximation algorithms and local search heuristics for the problem of scheduling unrelated parallel machines. The objective is to minimize the total weighted completion times of the jobs. In this chapter, besides the experimental comparison, we also investigate dominance relations among three lower bounds for this problem. These lower bounds are all obtained by solving a linear or nonlinear relaxation of a mixed integer programming formulation of the problem. The approximation algorithms with performance guarantees are based on rounding the solution to these relaxations. It turns out that the approximation algorithm that is based on the best lower bound yields, among the algorithms with a performance guarantee, the best results. This algorithm also performs better than the iterative improvement and tabu search algorithms. The best results are obtained by applying tabu search to the solution obtained by the algorithm based on the best relaxation.

In Chapter 3 an experimental evaluation of the max cut problem is made. Among the constructive heuristics, the one with the best performance guarantee, i.e., the celebrated algorithm of Goemans and Williamson (1995), yields the best cuts, but the running time is rather large. Simulated annealing applied to randomly generated solutions or greedily generated solutions yields on average the same performance as when it is applied to the solution obtained by the Goemans-Williamson algorithm. The time spent on the simulated annealing procedures, including obtaining the initial solutions, is the same. As the time needed to obtain the Goemans-Williamson solution is rather large, we have also used a smaller time bound for the local search procedures applied to the randomly and greedily generated starting solutions. The performance of these procedures is a little worse, about 0.1%, than that of the procedures using the larger time bound.

In Chapter 4 and 5, we investigate the worst-case behavior of local search. In

Chapter 4, which is based on Schuurman and Vredeveld (2001) and Hurkens and Vredeveld (2002), we consider multiprocessor scheduling problems. The objective is to minimize the makespan. We analyze the quality of local optima with respect to the jump, the swap, and the newly defined push neighborhood. In case of identical processors, an algorithm that finds a local optimum with respect to any of these three neighborhoods has a constant performance guarantee. For uniform processors, we show that the worst-case ratio of the value of a local optimum with respect to the push neighborhood to the optimal solution value is a constant, whereas for the jump and swap neighborhood this worst-case ratio grows with the square root of the number of processors. We also make some remarks about the running time of iterative improvement.

In Chapter 5, which is based on Vredeveld and Lenstra (2002), the generalized graph coloring problem is examined. We recall a result on the equivalence between the Karush-Kuhn-Tucker points of a quadratic programming formulation and FLIP-optimal solutions. We also show that the quality of local optima with respect to a large class of neighborhoods may be arbitrarily bad.

2

Scheduling unrelated parallel machines: an experimental investigation

2.1 Introduction

In this chapter, we make an experimental comparison of approximation algorithms for which we have constant performance bounds but no empirical evidence, and local search heuristics which exhibit good empirical behavior but for which we have no performance bounds. The former algorithms are polynomial-time ρ-approximation algorithms.

The problem under consideration is the problem of scheduling unrelated parallel machines so as to minimize the total weighted completion time. We are given a set of n jobs, $J_1, \ldots, J_n$, each of which has to be scheduled without interruption on one of m machines, $M_1, \ldots, M_m$, where m is part of the input. A machine can process at most one job at a time and all jobs and machines are available at time 0. If a job J_j is processed on a machine M_i, it will take a positive integral processing time p_{ij}. Furthermore, for each job we are given a nonnegative integral weight w_j. The objective is to schedule the jobs so that the sum of the weighted completion times of the jobs is minimized. Graham et al. (1979) denote this problem by $R\|\sum w_j C_j$. Bruno, Coffman, Jr., and Sethi (1974) and Lenstra, Rinnooy Kan, and Brucker (1977) showed that the problem of minimizing total weighted completion time on two identical parallel machines is NP-hard; $R\|\sum w_j C_j$ is NP-hard in the strong sense. In the case that the jobs have the same weight, the problem can be formulated as a bipartite matching problem, which can be solved in polynomial time; see Horn (1973) and Bruno et al. (1974). In case there is only one machine the problem is also

easy: sequence the jobs in order of non-increasing ratios $\frac{w_j}{p_{1j}}$ (Smith, 1956). Hence, the problem reduces to assigning the jobs appropriately to the machines; once the jobs have been assigned to the machines, we sequence the jobs on each machine by the ratio rule.

The first polynomial-time approximation algorithm for minimizing the total weighted completion time on unrelated parallel machines was given by Phillips, Stein, and Wein (1997), who achieved a performance ratio $\mathcal{O}(\log^2 n)$. Hall, Schulz, Shmoys, and Wein (1997) presented a polynomial-time $\frac{16}{3}$-approximation algorithm that uses an LP-relaxation in time-indexed variables and relies on the rounding technique of Shmoys and Tardos (1993) for the generalized assignment problem. This result was improved by Schulz and Skutella (1997b, 1997a) to performance guarantees $2 + \epsilon$ and $\frac{3}{2} + \epsilon$, where ϵ is an arbitrary positive value. They also used LP-relaxations in time-indexed variables. Skutella (1998) presented a polynomial-time $\frac{3}{2}$-approximation algorithm that was based on a convex quadratic relaxation. On the negative side, Hoogeveen, Schuurman, and Woeginger (1998) showed that the problem is APX-hard, i.e., there exists an $\epsilon > 0$ such that there does not exist a polynomial-time $(1 + \epsilon)$-approximation for this problem, unless P = NP. For an overview of previous work on the experimental comparison of polynomial-time ρ-approximation algorithms, we refer to Chapter 1.

This chapter is organized as follows. In the following section, we discuss the heuristics for which we have a constant performance guarantee. In Section 2.3 we discuss the dominance relations of the lower bounds obtained by the relaxations discussed in Section 2.2, and in Section 2.4 we describe the local search heuristics. Next we make some remarks on implementation details and we discuss the results of our empirical evaluation. Finally, in Section 2.6 we make some concluding remarks.

2.2 Approximation algorithms with a constant guarantee

The heuristics for which we have a constant guarantee are all based on rounding a fractional solution to some relaxation. The rounding of the fractional solutions is done in the same way for all three relaxations. The main idea is to exploit the values of the relaxation as probabilities with which jobs are assigned to machines: each job is assigned to a machine using these probabilities. This way of randomized rounding can be derandomized using the method of conditional expectations, with no difference in performance guarantees, but at the cost of increased, but still polynomial, running times.

2.2.1 Convex quadratic programming relaxation

The first relaxation we consider is due to Skutella (1998). He introduced a convex quadratic programming relaxation that leads to a polynomial-time $\frac{3}{2}$-approximation algorithm. The basic observation is that the problem is reduced to an assignment problem. Therefore, the problem can be formulated as an integer quadratic program in nm assignment variables, z_{ij}. The integer quadratic program, which forms the basis of the relaxation, is given in (IQP), where $k \prec_i j$ means that according to the ratio rule job J_k precedes job J_j on machine M_i. Constraints (2.2) ensure that the

completion time of a job is its own processing time plus the processing times of its predecessors on the machine on which it is scheduled and constraints (2.1) ensure that each job is scheduled on exactly one machine.

$$
\begin{array}{lll}
 & \min & \sum_j w_j C_j \\
(IQP) & \text{s.t.} & \sum_i z_{ij} = 1 \quad \forall j \qquad (2.1) \\
 & & C_j = \sum_i z_{ij}(p_{ij} + \sum_{k \prec_i j} p_{ik} z_{ik}) \quad \forall j \qquad (2.2) \\
 & & z_{ij} \in \{0,1\} \quad \forall i,j \qquad (2.3)
\end{array}
$$

In (2.7), the quadratic objective function is convexified by carefully raising the diagonal entries of the matrix determining the quadratic term until it becomes positive semidefinite and the function becomes convex. Adding constraint (2.6), which ensures that the sum of weighted completion times is at least the sum of weighted processing times, results in the convex quadratic programming problem given in (CQP). Solving this problem and then applying the randomized rounding procedure is a polynomial-time $\frac{3}{2}$-approximation algorithm.

$$
\begin{array}{lll}
 & \min & Z \qquad (2.4) \\
 & \text{s.t.} & \sum_i z_{ij} = 1 \quad \forall j \qquad (2.5) \\
(CQP) & & Z \geq \sum_{i,j} w_j p_{ij} z_{ij} \qquad (2.6) \\
 & & Z \geq \sum_{i,j} (\frac{1}{2} w_j p_{ij} z_{ij} + \frac{1}{2} w_j p_{ij} z_{ij}^2) + \\
 & & \qquad \sum_{i,j} \sum_{k \prec_i j} w_j p_{ik} z_{ij} z_{ik} \qquad (2.7) \\
 & & z_{ij} \geq 0 \quad \forall i,j \qquad (2.8)
\end{array}
$$

2.2.2 Time-indexed variables on processing intervals

Schulz and Skutella (1997b, 1997a) generalized an LP-relaxation in time-indexed variables that was introduced by Dyer and Wolsey (1990) for the single-machine scheduling problem with release dates. It contains decision variables, y_{ijt}, indicating whether job J_j is being processed on machine M_i during the time interval $[t, t+1)$, for integral t, where t ranges from 0 to $T-1$, with T an upper bound on the length of a schedule. The resulting LP-relaxation is a $\frac{3}{2}$-relaxation of the scheduling problem under consideration, i.e., the value of an optimal schedule is within a factor $\frac{3}{2}$ of the optimum LP value. Moreover, solving this LP-relaxation and applying the randomized rounding procedure yields a solution with expected value no more than $\frac{3}{2}$ times the optimal value.

The LP-relaxation Schulz and Skutella used is the following:

$$
\begin{aligned}
(LPY) \qquad & \min \quad \sum_j w_j C_j && && (2.9)\\
& \text{s.t.} \quad \sum_{i,t} \frac{y_{ijt}}{p_{ij}} &= 1 \quad \forall j && (2.10)\\
& \sum_j y_{ijt} &\leq 1 \quad \forall i,t && (2.11)\\
& C_j &\geq \sum_{i,t} \left(\frac{y_{ijt}}{p_{ij}}(t+\frac{1}{2}) + \frac{1}{2}y_{ijt}\right) \quad \forall j && (2.12)\\
& C_j &\geq \sum_{i,t} y_{ijt} \quad \forall j && (2.13)\\
& y_{ijt} &\geq 0 \quad \forall i,j,t && (2.14)
\end{aligned}
$$

To see that this is indeed a relaxation of $R\|\sum w_j C_j$, consider an arbitrary feasible schedule, where job J_j is being continuously processed between time S_j and $S_j + p_{hj}$ on machine M_h. Then $y_{ijt} = 1$ if $i = h$ and $t \in \{S_j, \ldots, S_j + p_{hj} - 1\}$ and $y_{ijt} = 0$ otherwise. The right hand side in (2.12) corresponds for these values of y_{ijt} to the completion time of J_j in the schedule. This relaxation can be strengthened by adding the constraint that a job can be processed by at most one machine during each time interval:

$$\sum_i y_{ijt} \leq 1 \qquad \forall j,t. \tag{2.15}$$

As the time horizon, T, can be exponential in the input size, this relaxation may suffer from an exponential number of variables and constraints. One can overcome this drawback by turning to interval-indexed variables. The time intervals Schulz and Skutella used are of the form $I_0 = [0,1)$ and $I_l = [(1+\epsilon)^{l-1}, (1+\epsilon)^l)$ for $l = 1, \ldots, \frac{\log T}{\log(1+\epsilon)}$, where $\epsilon > 0$ can be chosen arbitrarily small. The LP-relaxation, including constraints (2.15), is then the following:

$$
\begin{aligned}
(LPY') \qquad & \min \quad \sum_j w_j C_j && && (2.16)\\
& \text{s.t.} \quad \sum_{i,l} \frac{y_{ijl}|I_l|}{p_{ij}} &= 1 \quad \forall j && (2.17)\\
& \sum_j y_{ijl} &\leq 1 \quad \forall i,l && (2.18)\\
& \sum_i y_{ijl} &\leq 1 \quad \forall j,l && (2.19)\\
& C_j &\geq \sum_i \left(\frac{1}{2p_{ij}} + \frac{1}{2}\right) y_{ij0} + && (2.20)\\
& && \sum_i \sum_{l\geq 1} \frac{y_{ijl}|I_l|}{p_{ij}} (1+\epsilon)^{l-1} +\\
& && \sum_i \sum_{l\geq 1} \frac{1}{2} y_{ijl}|I_l| \quad \forall j
\end{aligned}
$$

$$C_j \geq \sum_{i,l} y_{ijl}|I_l| \quad \forall j \tag{2.21}$$

$$y_{ijl} \geq 0 \quad \forall i,j,l \tag{2.22}$$

Any feasible solution, $\bar{y}$, of (LPY) plus constraints (2.15) can be transformed into a feasible solution, y, to (LPY') with the same or lower value. This is done by setting $y_{ijl} = \sum_t \frac{|[t,t+1)\cap I_l|}{|I_l|}\bar{y}_{ijt}$. Hence, the optimal value of (LPY') is at most equal to the optimal value of (LPY), including (2.15).

This leads to a $(\frac{3}{2}+\epsilon)$-relaxation of polynomial size and a polynomial-time $(\frac{3}{2}+\epsilon)$-approximation algorithm; in the subsequent sections, this method will be referred to as the *LPY approach*. Notice that the size of the relaxation still depends substantially on the time horizon and may be huge for small values of ϵ.

2.2.3 Time-indexed variables on starting times

The LP-relaxation used by Schulz and Skutella yields a poor lower bound, as even a $0-1$ solution to this relaxation does not necessarily correspond to a feasible schedule. Therefore, we have implemented another relaxation. This is a generalization of a second LP-relaxation introduced by Dyer and Wolsey (1990) for the single-machine scheduling problem with release dates. The problem is formulated as an integer program in time-indexed variables, x_{ijt}, denoting whether job J_j starts being processed on machine M_i at time t. This program is given in (IPX):

$$(IPX) \qquad \begin{aligned} \min \quad & \sum_j w_j C_j & & & \\ \text{s.t.} \quad & \sum_{i,t} x_{ijt} = 1 & \forall j & & (2.23)\\ & \sum_j \sum_{s=t-p_{ij}+1}^{t} x_{ijs} \leq 1 & \forall i,t & & (2.24)\\ & \sum_{i,t} (t+p_{ij}) x_{ijt} = C_j & \forall j & & (2.25)\\ & x_{ijt} \in \{0,1\} & \forall i,j,t & & (2.26) \end{aligned}$$

The LP-relaxation is obtained by relaxing the integrality constraints (2.26) to non-negativity constraints and will be denoted by (LPX).

As with LP problem (LPY), the time horizon, T, can be an exponential in the input size. Instead of trying to reduce the size of the LP problem, we solved this large problem by column generation, generalizing the work of Van den Akker, Hurkens, and Savelsbergh (2000) on the above mentioned single-machine problem.

To reduce the number of constraints we apply Dantzig-Wolfe decomposition. The constraints (2.24) and the non-negativity constraints describe a polytope P. This polytope can be written as the Cartesian product $P = P_1 \times \ldots \times P_m$, where $P_i = \{x \in \mathbb{R}_+^{nT} : \sum_j \sum_{s=t-p_{ij}+1}^{t} x_{jt} \leq 1,\ t = 1,\ldots,T\}$. Hence a point $x \in P$ can be written as $x = (x^{(1)},\ldots,x^{(m)})$, where $x^{(i)} \in P_i$ $(i = 1,\ldots,m)$. The polytopes

P_i have integral vertices (Van den Akker et al., 2000)) and these vertices can be considered as schedules on machine M_i in which jobs do not have to be processed exactly once, as they do not necessarily satisfy constraints (2.23). We will call such schedules *semi-schedules*.

Let $\xi_1^{(i)}, \ldots, \xi_{K_i}^{(i)}$ be the extreme points of P_i. Then any $x^{(i)} \in P_i$ can be written as a convex combination $x^{(i)} = \sum_{l=1}^{K_i} \lambda_l^{(i)} \xi_l^{(i)}$. The LP-relaxation can be reformulated using the variables $\lambda_l^{(i)}$ as follows:

$$
\begin{aligned}
& & \min \quad & \sum_i \sum_{l=1}^{K_i} \Big(\sum_{j,t} c_{ijt} \xi_{l,jt}^{(i)}\Big) \lambda_l^{(i)} & & & \\
(LPX') & & \text{s.t.} \quad & \sum_i \sum_{l=1}^{K_i} \Big(\sum_t \xi_{l,jt}^{(i)}\Big) \lambda_l^{(i)} & = & 1 \quad \forall j & (2.27) \\
& & & \sum_{l=1}^{K_i} \lambda_l^{(i)} & = & 1 \quad \forall i & (2.28) \\
& & & \lambda_l^{(i)} & \geq & 0 \quad \forall l, i & (2.29)
\end{aligned}
$$

Note that the solutions of (LPX) and (LPX') are in a one-to-one correspondence.

Observe that the jth element of the column corresponding to $\lambda_l^{(i)}$, i.e., $\sum_t (\xi_l^{(i)})_{jt}$, is equal to the number of times that job J_j occurs in the semi-schedule $\xi_l^{(i)}$. This means that the column corresponding to the semi-schedule $\xi_l^{(i)}$ only indicates how many times each job occurs in this schedule. The cost coefficient of $\lambda_l^{(i)}$ is equal to the cost of the semi-schedule $\xi_l^{(i)}$.

By reformulating this way, the number of constraints is decreased significantly from $n+mT$ to $n+m$. The number of variables, however, has increased to the total number of extreme points of the polytopes P_i. Fortunately, this does not matter, since the problem can be solved through column generation. To apply column generation, we have to find an efficient way to determine a column with minimal reduced cost, i.e., to solve a pricing problem. We determine for each machine such a minimal column in the same way as Van den Akker et al. (2000). The reduced cost of the variable $\lambda_l^{(i)}$ is given by

$$
\sum_{j,t} c_{ijt} \xi_{l,jt}^{(i)} - \sum_j \pi_j \Big(\sum_t \xi_{l,jt}^{(i)}\Big) - \alpha_i \tag{2.30}
$$

where π_j denotes the dual variable of constraint (2.27) for job J_j, and α_i denotes the dual variable of constraints (2.28) for machine M_i.

Recall that each extreme point $\xi_l^{(i)}$ represents a semi-schedule on machine M_i. These semi-schedules can be represented by paths in a network in the following way. The nodes of the network correspond to time points $0, 1, \ldots, T$. For each job J_j and each period s, with $s \leq T - p_{ij} + 1$, there is an arc from s to $s + p_{ij}$ that indicates the machine processes job J_j from time s to time $s + p_{ij}$; we say that this

arc corresponds to the variable $(x^{(i)})_{js}$. Furthermore, for each time point t there is an idle time arc from t to $t+1$ that indicates that the machine is idle in period $[t, t+1)$. Any directed path from 0 to T corresponds to a semi-schedule $\xi_l^{(i)}$ on machine M_i, and vice versa.

If we set the length of the arc corresponding to $(x^{(i)})_{jt}$ equal to $c_{ijt} - \pi_j$, for all j and t, and the length of all idle time arcs equal to 0, then the reduced cost of the variable $\lambda_l^{(i)}$ is equal to the length of path corresponding to $\xi_l^{(i)}$ minus the dual variable α_i. Therefore, finding a column with minimal reduced costs boils down to finding the shortest path in a directed acyclic network with arbitrary weights. As the network is directed and acyclic, the shortest path problem can be solved in $\mathcal{O}(nT)$ time.

As we do not know of any analytical bounds on the number of columns that have to be generated and T might not be a polynomial in the input size, we have no polynomial-time guarantee for solving the LP problem.

As a corollary to the work of Schulz and Skutella, solving the LP problem and then applying the randomized rounding technique yields a performance guarantee of $\frac{3}{2}$. This method will be called the *LPX approach*.

2.3 Dominance relations among lower bounds

The LP-relaxations and the CQP-relaxation described in the previous section provide us with lower bounds. In the following theorems, we discuss the dominance relations between the lower bounds obtained by (LPX), (LPY), and (CQP).

Theorem 2.1. *Let Z_{LPX} be the value of an optimal solution to (LPX) and (LPX') and let Z_{CQP} denote the value of an optimal solution to (CQP). Then $Z_{LPX} \geq Z_{CQP}$.*

Proof. Consider a feasible solution $\lambda = (\lambda_l^{(i)})$ for (LPX'), and define, for $i = 1, \ldots, m$, $z_i \in \mathbb{R}^n$ as

$$z_i = \sum_l \lambda_l n_l^{(i)},$$

where $n_l^{(i)}$ is the vector consisting of the first n elements of the column in (LPX') corresponding to variable $\lambda_l^{(i)}$, i.e., the jth element of $n_l^{(i)}$ is the number of copies of J_j occurring in semi-schedule $\xi_l^{(i)}$.

Let $c_i \in \mathbb{R}^n$ be given by $c_{ij} = w_j p_{ij}$ and $D_i = (d_{jk}^{(i)})_{jk} \in \mathbb{R}^{n \times n}$ be defined as $d_{jk}^{(i)} = \min(w_j\ p_{ik},\ w_k\ p_{ij})$. If we define Z_i^C as

$$Z_i^C(z) = \frac{1}{2} c_i^T z_i + \frac{1}{2} z_i^T D_i z_i,$$

and $Z_{CQP}(z)$ as

$$Z_{CQP}(z) = \max(\sum_i c_i^T z_i, \sum_i Z_i^C),$$

then $(z, Z) \in \mathbb{R}^{nm} \times \mathbb{R}$, with $z = (z_1, \ldots, z_m)$ and $Z \geq Z_{CQP}(z)$, is a feasible solution for (CQP), as $z_{ij} \geq 0$ and $\sum_i z_{ij} = \sum_i \sum_l \lambda_l n_{lj}^{(i)} \stackrel{(2.27)}{=} 1$, for all j.

The sum of the completion times of all copies of J_j in semi-schedule $\xi_l^{(i)}$ is

$$C_{lj}^{(i)} = \frac{1}{2} p_{ij} n_{lj}^{(i)} + \frac{1}{2} p_{ij} (n_{lj}^{(i)})^2 + \sum_{k \prec_i j} p_{ik} n_{lk}^{(i)} n_{lj}^{(i)}.$$

Thus the weighted sum of completion times for a machine M_i in (LPX') is

$$Z_i^X(\lambda) = \sum_{j,l} w_j \lambda_l^{(i)} C_{lj}^{(i)} = \sum_l \lambda_l^{(i)} (\frac{1}{2} c_i^T n_l^{(i)} + \frac{1}{2} (n_l^{(i)})^T D_i n_l^{(i)}).$$

As all elements of $n_l^{(i)}$ are integer, for all l and i, we know that

$$\begin{aligned} Z_i^X(\lambda) &= \sum_{j,l} w_j \lambda_l^{(i)} (\frac{1}{2} p_{ij} n_{lj}^{(i)} + \frac{1}{2} p_{ij} (n_{lj}^{(i)})^2 + \sum_{k \prec_i j} p_{ik} n_{lk}^{(i)} n_{lj}^{(i)}) \\ &\geq \sum_{j,l} w_j \lambda_l^{(i)} (\frac{1}{2} p_{ij} n_{lj}^{(i)} + \frac{1}{2} p_{ij} n_{lj}^{(i)}) \\ &= c_i^T (\sum_l \lambda_l^{(i)} n_l^{(i)}) = c_i^T z_i. \end{aligned} \tag{2.31}$$

Skutella (1998) proved that the matrix D_i is positive semi-definite, thus the function $f_i(x) = \frac{1}{2} c_i^T x + \frac{1}{2} x^T D_i x$ is convex. By constraint (2.28) $\sum_l \lambda_l^{(i)} = 1$ and thus

$$Z_i^X(\lambda) = \sum_l \lambda_l^{(i)} f_i(n_l^{(i)}) \geq f_i(\sum_l \lambda_l n_l^{(i)}) = Z_i^C(z). \tag{2.32}$$

By (2.31) we know that $\sum_i Z_i^X(\lambda) \geq \sum_i c_i^T z_i$ and by (2.32) we know that $\sum_i Z_i^X(\lambda) \geq \sum_i Z_i^C(z)$. Hence,

$$Z_{LPX}(\lambda) \geq \max(\sum_i c_i^T z_i, \sum_i Z_i^C(z)) = Z_{CQP}(z).$$

Thus, any feasible solution for (LPX') can be converted to a feasible solution for (CQP) with value at most equal to the value of the solution for (LPX'). Hence, $Z_{LPX} \geq Z_{CQP}$. □

Theorem 2.2. *Let Z_{LPX} be the value of an optimal solution to (LPX) and (LPX') and let Z_{LPY} denote the optimal value to (LPY) plus constraints (2.15). Then $Z_{LPX} \geq Z_{LPY}$.*

PROOF. Consider a feasible solution for (LPX) and construct a feasible solution, $(y, C^Y) \in \mathbb{R}^{nmT} \times \mathbb{R}^n$, for (LPY) as follows:

$$y_{ijt} = \sum_{t'=t-p_{ij}+1}^{t} x_{ijt'}, \tag{2.33}$$

$$C_j^Y = \max(\sum_{i,t} y_{ijt}, \sum_{i,t} \frac{y_{ijt}}{p_{ij}}(t+\frac{1}{2}) + \frac{1}{2}y_{ijt}). \tag{2.34}$$

It is easy to verify that y satisfies constraints (2.10), (2.11), and (2.15). The right hand side of constraint (2.13) is

$$\sum_{i,t} y_{ijt} = \sum_{i,t} \sum_{t'=t-p_{ij}+1}^{t} x_{ijt'} = \sum_{i,t} p_{ij} x_{ijt} \leq \sum_{i,t} (t+p_{ij}) x_{ijt},$$

and the right hand side of constraint (2.12) is

$$\sum_{i,t} (\frac{y_{ijt}}{p_{ij}}(t+\frac{1}{2}) + \frac{1}{2}y_{ijt}) = \sum_{i,t} \sum_{t'=t-p_{ij}+1}^{t} x_{ijt'} (\frac{t+\frac{1}{2}}{p_{ij}} + \frac{1}{2}) =$$

$$= \sum_{i,t'} x_{ijt'} \sum_{t=t'}^{t'+p_{ij}-1} (\frac{t+\frac{1}{2}}{p_{ij}} + \frac{1}{2}) = \sum_{i,t} (t+p_{ij}) x_{ijt}.$$

Hence, $C_j^Y = \max(\sum_{i,t} y_{ijt}, \sum_{i,t} \frac{y_{ijt}}{p_{ij}}(t+\frac{1}{2}) + \frac{1}{2}y_{ijt}) \leq \sum_{i,t}(t+p_{ij})x_{ijt}$ and the optimal value to (LPX) is at least as large as the optimal value to (LPY). □

To establish the relation between (LPY) and (CQP), we need the following lemma, that specifies an optimal solution to (LPY) without constraints (2.13) for a given fractional assignment of the jobs to the machines.

Lemma 2.3. *Let $z = (z_{ij}) \in \mathbb{R}_+^{nm}$ and let $\bar{y} \in \mathbb{R}_+^{nmT}$ be*

$$\bar{y}_{ijt} = |[t, t+1) \cap [S_{ij}, S_{ij} + p_{ij} z_{ij})|,$$

where $S_{ij} = \sum_{k \prec_i j} p_{ik} z_{ik}$. Then $\bar{y}$ minimizes

$$f(y) = \sum_j w_j \sum_{it} (\frac{y_{ijt}}{p_{ij}}(t+\frac{1}{2}) + \frac{1}{2}y_{ijt})$$

over all $y \in Y(z) = \{y \in \mathbb{R}_+^{nmT} : \sum_j y_{ijt} \leq 1, \sum_t \frac{y_{ijt}}{p_{ij}} = z_{ij}\}$.

PROOF. First note that, by construction, $\bar{y}$ satisfies $\sum_j \bar{y}_{ijt} \leq 1$ for all i, t, and $\sum_t \frac{y_{ijt}}{p_{ij}} = z_{ij}$, and so $\bar{y} \in Y(z)$. In addition $\bar{y}$ has the property that for all i there

exist an s_i and an $\alpha_i \in [0,1)$ such that

$$\sum_j \bar{y}_{ijt} = \begin{cases} 1 & \text{if} \quad t < s_i, \\ \alpha_i & \text{if} \quad t = s_i, \\ 0 & \text{if} \quad t > s_i. \end{cases}$$

Consider a solution $y \in Y(z)$, such that $y \neq \bar{y}$. Then there exists an triple (i,j,t) such that $y_{ijt} < \bar{y}_{ijt}$. Let (i,k,t') be such a triple, with minimal t'. Note that by construction of $\bar{y}$ and minimality of t', $y_{ijt} = \bar{y}_{ijt}$ for all $t < t'$, and thus there exists a $t'' > t'$ such that $y_{ikt''} > \bar{y}_{ikt''}$.

If $\sum_j y_{ijt'} < 1$, then we construct y' by adding $\beta = \min(1 - \sum_j y_{ijt'}, y_{ikt''})$ to $y_{ikt'}$ and subtracting the same value from $y_{ikt''}$. Clearly, $y' \in Y(z)$, and $f(y) - f(y') = \frac{w_k}{p_{ik}}\beta(t'' - t') > 0$.

If $\sum_j y_{ijt'} = 1$, then there is an l such that $y_{ilt'} > \bar{y}_{ilt'} \geq 0$. By construction of $\bar{y}$ we have $k \prec_i l$. Let $\beta = \min(y_{ilt'}, y_{ikt''}, \bar{y}_{ikt'} - y_{ikt'})$ and construct y' by adding β to $y_{ikt'}$ and $y_{ilt''}$ and subtracting it from $y_{ikt''}$ and $y_{ilt'}$. Then, clearly $y' \in Y(z)$ and

$$\begin{aligned} f(y) - f(y') &= \frac{w_k}{p_{ik}}\beta(t'' - t') + \frac{w_l}{p_{il}}\beta(t' - t'') \\ &= \beta(t'' - t')(\frac{w_k}{p_{ik}} - \frac{w_l}{p_{il}}) \geq 0. \end{aligned}$$

Hence, we have constructed a solution that has the same or lower value and is closer to $\bar{y}$. Setting $y = y'$ and repeating this procedure results in the solution $\bar{y}$, and it has the same or lower value than all intermediate solutions. That is, $f(\bar{y}) \leq f(y)$ for all $y \in Y(z)$. □

Theorem 2.4. *Let Z_{LPY} be the value of an optimal solution to (LPY) and let Z_{CQP} denote the optimal solution value to (CQP). Then $Z_{LPY} \geq Z_{CQP}$.*

PROOF. Let $(y, C^Y) \in \mathbb{R}^{nmT} \times \mathbb{R}^n$ be a feasible solution for (LPY) and let $z \in \mathbb{R}^{nm}$ be defined as $z_{ij} = \sum_t \frac{y_{ijt}}{p_{ij}}$, and $Z_{CQP}(z)$ as

$$Z_{CQP}(z) = \max(\sum_{i,j} w_j p_{ij} z_{ij}, \sum_{i,j} \frac{1}{2} w_j p_{ij} z_{ij} + \frac{1}{2} w_j p_{ij} z_{ij}^2 + \sum_{k \prec_i j} p_{ik} z_{ij} z_{ik}).$$

Then $(z, Z_{CQP}(z))$ is a feasible solution for (CQP), and

$$\sum_{i,j} w_j p_{ij} z_{ij} = \sum_j w_j \sum_{i,t} y_{ijt} \leq \sum_j w_j C_j^Y.$$

Below, we prove that

$$\sum_{i,j} \frac{1}{2} w_j p_{ij} z_{ij} + \frac{1}{2} w_j p_{ij} z_{ij}^2 + \sum_{k \prec_i j} p_{ik} z_{ij} z_{ik} \leq \sum_j w_j C_j^Y(y).$$

Let $\bar{y}$ be the feasible solution for (LPY) as defined in Lemma 2.3, i.e.,

$$\bar{y}_{ijt} = |[t, t+1) \cap [S_{ij}, S_{ij} + p_{ij}z_{ij})|,$$

where $S_{ij} = \sum_{k \prec_i j} p_{ik}z_{ik}$. Let $\alpha_{ij} = S_{ij} - \lfloor S_{ij} \rfloor$ and $\beta_{ij} = \lceil S_{ij} + p_{ij}z_{ij} \rceil - (S_{ij} + p_{ij}z_{ij})$. Then

$$\bar{y}_{ijt} = \begin{cases} 1-\alpha_{ij} & \text{if} \quad t = \lfloor S_{ij} \rfloor < \lceil S_{ij} + p_{ij}z_{ij} \rceil - 1, \\ 1-\beta_{ij} & \text{if} \quad t = \lceil S_{ij} + p_{ij}z_{ij} \rceil - 1 > \lfloor S_{ij} \rfloor, \\ 1-\alpha_{ij}-\beta_{ij} & \text{if} \quad t = \lfloor S_{ij} \rfloor = \lceil S_{ij} + p_{ij}z_{ij} \rceil - 1, \\ 1 & \text{if} \quad \lfloor S_{ij} \rfloor < t < \lceil S_{ij} + p_{ij}z_{ij} \rceil - 1, \\ 0 & \text{otherwise.} \end{cases}$$

Let $C^Y_{ij}(\bar{y})$ be defined as

$$C^Y_{ij}(\bar{y}) = \sum_t \left(\frac{\bar{y}_{ijt}}{p_{ij}}(t + \frac{1}{2}) + \frac{1}{2}\bar{y}_{ijt}\right).$$

If $\lfloor S_{ij} \rfloor = \lceil S_{ij} + p_{ij}z_{ij} \rceil - 1$, then $1 - \alpha_{ij} - \beta_{ij} = p_{ij}z_{ij}$ and

$$\begin{aligned} C^Y_{ij}(\bar{y}) &= \frac{1-\alpha_{ij}-\beta_{ij}}{p_{ij}}(S_{ij} - \alpha_{ij} + \frac{1}{2}) + \frac{1}{2}p_{ij}z_{ij} \\ &= \frac{1}{2}p_{ij}z_{ij} + z_{ij}(S_{ij} + \frac{1}{2}p_{ij}z_{ij}) + \\ &\quad \frac{1-\alpha_{ij}-\beta_{ij}}{p_{ij}}(-\frac{1}{2}(1-\alpha_{ij}-\beta_{ij}) - \alpha_{ij} + \frac{1}{2}) \\ &= \frac{1}{2}p_{ij}z_{ij} + \frac{1}{2}p_{ij}z_{ij}^2 + z_{ij}S_{ij} + \\ &\quad \frac{1}{p_{ij}}\frac{1}{2}(\alpha_{ij} - \beta_{ij})(\alpha_{ij} + \beta_{ij} - 1). \end{aligned} \tag{2.35}$$

If $\lfloor S_{ij} \rfloor < \lceil S_{ij} + p_{ij}z_{ij} \rceil - 1$, then

$$\begin{aligned} C^Y_{ij}(\bar{y}) &= \frac{1-\alpha_{ij}}{p_{ij}}(S_{ij} - \alpha_{ij} + \frac{1}{2}) + \frac{1-\beta_{ij}}{p_{ij}}(S_{ij} + p_{ij}z_{ij} + \beta_{ij} - \frac{1}{2}) + \\ &\quad \frac{1}{p_{ij}}(p_{ij}z_{ij} - 2 + \alpha_{ij} + \beta_{ij})(S_{ij} + \frac{1}{2}p_{ij}z_{ij} - \frac{1}{2}(\alpha_{ij} - \beta_{ij})) + \\ &\quad \frac{1}{2}p_{ij}z_{ij} \\ &= \frac{1}{p_{ij}}(p_{ij}z_{ij} + \alpha_{ij} + \beta_{ij})(S_{ij} + \frac{1}{2}p_{ij}z_{ij} - \frac{1}{2}(\alpha_{ij} - \beta_{ij})) - \\ &\quad \frac{\alpha_{ij}}{p_{ij}}(S_{ij} - \alpha_{ij} + \frac{1}{2}) - \frac{\beta_{ij}}{p_{ij}}(S_{ij} + p_{ij}z_{ij} + \beta_{ij} - \frac{1}{2}) + \frac{1}{2}p_{ij}z_{ij} \\ &= \frac{1}{2}p_{ij}z_{ij} + \frac{1}{2}p_{ij}z_{ij}^2 + z_{ij}S_{ij} + \\ &\quad \frac{1}{p_{ij}}\frac{1}{2}(\alpha_{ij} - \beta_{ij})(\alpha_{ij} + \beta_{ij} - 1). \end{aligned} \tag{2.36}$$

Thus in both cases

$$C_{ij}^Y(\bar{y}) = \frac{1}{2}p_{ij}z_{ij} + \frac{1}{2}p_{ij}z_{ij}^2 + z_{ij}S_{ij} + \frac{1}{p_{ij}}\frac{1}{2}(\alpha_{ij} - \beta_{ij})(\alpha_{ij} + \beta_{ij} - 1).$$

Consider a machine M_i and assume w.l.o.g. that $\{j : p_{ij}z_{ij} > 0\} = \{1, \ldots, K\}$, and that $J_1 \prec_i \ldots \prec_i J_K$. Then $\alpha_{i1} = 0$ and, as $S_{i,j+1} = S_{ij} + p_{ij}z_{ij}$, $\alpha_{i,j+1} = 1 - \beta_{ij}$, for $j = 1, \ldots, K-1$.

Then

$$\begin{aligned}
\sum_j w_j C_{ij}^Y(\bar{y}) &= \sum_j (\frac{1}{2}w_j p_{ij} z_{ij} + \frac{1}{2}w_j p_{ij} z_{ij}^2 + \sum_{k \prec_i j} w_j p_{ik} z_{ij} z_{ik}) + \\
&\quad + \sum_j \frac{w_j}{p_{ij}} \frac{1}{2}(\alpha_{ij} - \beta_{ij})(\alpha_{ij} + \beta_{ij} - 1) \\
&= \sum_j (\frac{1}{2}w_j p_{ij} z_{ij} + \frac{1}{2}w_j p_{ij} z_{ij}^2 + \sum_{k \prec_i j} w_j p_{ik} z_{ij} z_{ik}) + \\
&\quad + \sum_{j=1}^{K-1} \frac{1}{2}(\beta_{ij} - \beta_{ij}^2)(\frac{w_j}{p_{ij}} - \frac{w_{j+1}}{p_{i,j+1}}) + \frac{1}{2}(\beta_{iK} - \beta_{iK}^2)\frac{w_K}{p_{iK}} \\
&\geq \sum_j (\frac{1}{2}w_j p_{ij} z_{ij} + \frac{1}{2}w_j p_{ij} z_{ij}^2 + \sum_{k \prec_i j} w_j p_{ik} z_{ij} z_{ik}). \qquad (2.37)
\end{aligned}$$

The last inequality is true as $0 \leq \beta_{ij} < 1$ and thus $\beta_{ij} - \beta_{ij}^2 \geq 0$ and by the ordering of the jobs, we have that $\frac{w_j}{p_{ij}} \geq \frac{w_{j+1}}{p_{i,j+1}}$.

Hence, we have that

$$\sum_{i,j} (\frac{1}{2}w_j p_{ij} z_{ij} + \frac{1}{2}w_j p_{ij} z_{ij}^2 + \sum_{k \prec_i j} w_j p_{ik} z_{ij} z_{ik}) \leq \sum_{i,j} w_j C_{ij}^Y(\bar{y}) \leq \sum_{i,j} w_j C_{ij}^Y(y).$$

The last inequality is due to Lemma 2.3. And thus $Z_{CQP}(z) \leq Z_{LPY}(y)$. □

(LPY) plus constraints (2.15) yields a higher lower bound than (LPY). However, as (LPY') is a underestimate of (LPY) plus constraints (2.15), it does not need to be the case that $Z_{LPY'} \geq Z_{CQP}$, where $Z_{LPY'}$ is the optimal solution to (LPY').

The inequalities in the above theorems are strict, as is shown in the following example.

Example 2.1. Consider an instance with three jobs and two machines. Job J_1 can only be processed on machine M_1 and has processing time $p_{11} = 3$ and weight $w_1 = 11$. J_2 can only be processed on machine M_2 and has weight $w_2 = 1$ and processing time $p_{22} = 1$. Job J_3 can be processed on both machines. Its processing times are $p_{13} = 2$ and $p_{23} = 6$ and its weight is $w_3 = 7$.

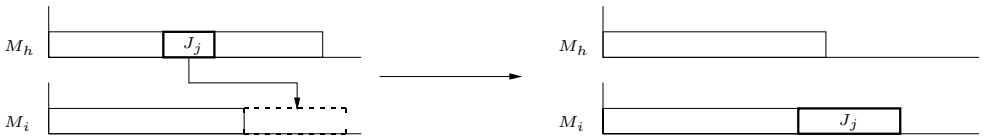


Figure 2.1: jump

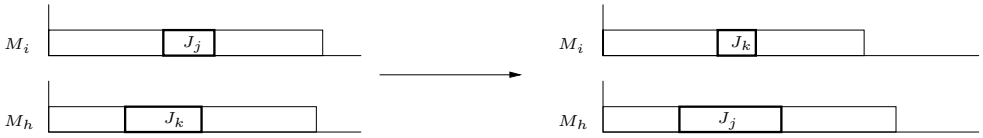


Figure 2.2: swap

The optimal solution to (LPX) is the optimal schedule: J_1 and J_3 are processed by M_1 and J_2 is scheduled on machine M_2. The optimal value to (LPX) is $Z_{LPX} = 69$. The optimal solution to (LPY) with or without constraints (2.15) has value $Z_{LPY} = 67\frac{1}{2}$ and in this solution J_1 is assigned to the time slots $[0,1)$, $[1,2)$, and $[2,3)$ on M_1, J_2 is assigned to the time slot $[2,3)$ on M_2, and J_3 is fully assigned to the time slots $[3,4)$ on M_1 and $[0,1)$ and $[1,2)$ on M_2 and for one third to time slot $[4,5)$ on M_1. Finally, the optimal solution to (CQP) assigns J_3 for $\frac{41}{56}$ to M_1 and for $\frac{15}{56}$ to M_2. The optimal value to (CQP) is $Z_{CQP} = \frac{7503}{112} = 66.991$.

2.4 Local search

We compare the algorithms described in the previous section to local search heuristics. As mentioned in Chapter 1, a local search procedure iteratively moves from a solution to a neighboring solution. Hence, we need to specify the neighborhood functions. Recall that the problem can be reduced to the problem of assigning the jobs appropriately to the machines. We, therefore, represent a schedule by the assignment of the jobs to the machines.

We consider two types of neighborhood functions. First, for the *jump* neighborhood, we select a job J_j and a machine M_i such that J_j is not scheduled on machine M_i. A neighbor is formed by moving job J_j to machine M_i (see Figure 2.1).

The second neighborhood function is called *swap*. For this neighborhood, we select two jobs J_j and J_k, assigned to different machines, and the neighbor is obtained by interchanging their machine allocation (see Figure 2.2).

Besides applying the local search methods to the solutions obtained by the heuristics described in the previous section, they are also applied to randomly generated

initial solutions. We have two strategies to generate these random solutions. The first strategy is the *completely random* strategy, in which we assign each job independently and uniformly to one of the m machines. In the second strategy, called *random greedy*, the jobs are, in random order, greedily assigned to the machines, that is, given a partial schedule and the first job that still has to be scheduled, the job is assigned to the machine for which the total weighted completion time of the new partial schedule is minimal.

We have implemented two local search heuristics: multi-start iterative improvement and tabu search.

2.4.1 Multi-start iterative improvement

There are several ways to select the neighbor to move to. First there is *first improvement*: we move to the first neighbor encountered that has lower cost. Another strategy for selecting the neighbor is *best improvement*: move to the neighbor that has lowest cost among all neighbors. Initial tests showed that the neighbor selecting methods perform equally well.

Multi-start iterative improvement refers to a repeated application of the iterative improvement procedure on multiple initial solutions. Note that we only apply multi-start iterative improvement on randomly generated starting solutions. To make a fair comparison with the other methods, the number of repetitions is chosen such that the total time spent on this procedure is about the same as that for the others.

2.4.2 Tabu search

Recall that in tabu search, we make use of a tabu list to avoid returning to the same solution. The information stored in the tabu list due to a jump move is the job that has changed its machine allocation. If the move is due to a swap, we look at the contribution of the two swapped jobs to the objective function before and after the swap. The swapped job that has lowest decrease or highest increase in its contribution is stored in the tabu list. Hence, after each move, jump or swap, we store one job in the tabu list. This job has to remain on its new machine for a number of iterations, unless moving it yields a better solution than found so far. The number of iterations during which a job has a fixed machine allocation is equal to the length of the tabu list, which, after some initial experiments, we have chosen to be $n/2$ whenever $n < 40$ and 20 otherwise.

In our tabu search procedure, we use *best improvement* for finding the neighbor to move to. However, if a move is made to a non-improving neighbor, we only allow jumps. The reason for this is that a jump creates more space on the machine from which the job is leaving than a swap and thus allows for better improvements in the subsequent iterations. Initial tests show indeed that tabu search with this feature gives better results than tabu search where non-improving swaps are also accepted.

Another feature of our tabu search procedure is the *backjump*, cf. the tabu search procedure of Nowicki and Smutnicki (1996) for the job-shop scheduling problem: if we have made 500 non-improving moves without improving the best found solution, we return to this solution and move to a neighbor that has not been visited yet directly from this solution.

In the case that tabu search is applied to the solutions obtained by the heuristics with constant performance guarantee, the procedure terminates when there have been too many, that is 20, backjumps to the same solution. In the case of randomly generated start solutions, we repeat the procedure of applying tabu search to a start solution until the total time spent is about the same as for the other procedures.

2.5 Computational experience

2.5.1 Test instances

Our heuristics have been tested on instances with size varying from 10 jobs and 5 machines to 100 jobs and 50 machines. The solution quality may depend on the structure of the test instances. To allow for possible variations in performance, three classes of test instances were considered, each of which is based on a different method of generating the processing time p_{ij} of job J_j on machine M_i.

- *No correlation*: all processing times p_{ij} are independently drawn from the uniform distribution over $[10, 100]$; w_j is an integer from the uniform distribution over $[1, 100]$.
- *Machine correlation*: p_{ij} is an integer from the uniform distribution over $[\alpha_i, \alpha_i + 10]$ where α_i is an integer from the uniform distribution over $[1, 100]$; w_j is an integer from the uniform distribution over $[1, 100]$.
- *Favorite machines*: each job J_j has two favorite machines, $M_{i_1(j)}$ and $M_{i_2(j)}$, which are randomly selected; $p_{i_1(j),j}$ and $p_{i_2(j),j}$ are drawn from the uniform distribution over $[\beta_j, \beta_j + 4]$, where β_j is an integer from the uniform distribution over $[15, 25]$, and p_{ij} $(i \neq i_1(j), i_2(j))$ is drawn from the uniform distribution over $[70, 90]$; w_j is an integer from the uniform distribution over $[1, 100]$.

For each value of m and n, 50 instances have been generated for each of the three instance classes.

2.5.2 Implementational details

Multi-start iterative improvement, tabu search and both LP approaches have been implemented in C, using CPLEX for solving the LP problems. The CQP approach has been coded in MATLAB, and we use SEDUMI Sturm (1999) for solving the CQP problem. The reason for implementing the CQP approach in MATLAB instead of C is that SEDUMI is only available in MATLAB. The tests have been run on a Sun Ultra-1, 140 MHz, with 256 MB memory.

2.5.3 Computational results

Before looking how good the schedules, obtained by the several algorithms, are, we first look at the quality of the lower bounds. In Table 2.1, we show the average relative deviation of the lower bounds obtained by the CQP and LPY'-relaxation from the best obtained lower bounds, which are all obtained by the LPX-relaxation as shown in Section 2.3. The average and maximum are taken over all instances.

$n \times m$	10×10	20×20	50×50	10×5	20×10	50×20	100×50	average
CQP	-5.13	-5.17	-4.35	-3.69	-3.67	-2.76	-3.35	-4.02
	(-11.82)	(-10.23)	(-9.02)	(-8.02)	(-8.46)	(-4.98)	(-5.69)	
LPY'	-4.48	-4.41	-3.85	-5.01	-5.10	-4.95	-5.00	-4.68
	(-10.28)	(-9.20)	(-8.65)	(-9.08)	(-8.65)	(-7.14)	(-7.81)	

Table 2.1: Lower bounds: average (maximum) relative deviation in % from best lower bound

$n \times m$	10×10	20×20	50×50	10×5	20×10	50×20	100×50	average
CQP	-6.27	-6.04	-5.27	-4.12	-4.57	-3.68	-3.83	-4.83
LPY'	-7.17	-7.31	-7.12	-6.26	-6.61	-6.25	-6.52	-6.75

Table 2.2: Lower bounds: average relative deviation in % from best lower bound - machine correlated instances

We see that for the square instances, where the number of jobs is equal to the number of machines, the LPY'-relaxation yields somewhat better lower bounds, whereas the CQP-relaxation yields the better lower bounds for the rectangular instances. The reason can be found in the fact that the schedule length for square instances is on average shorter than for the rectangular instances with the same number of machines and therefore the LPY'-relaxation is closer to the LPY-relaxation. On average, the CQP and LPY'-relaxation are about 4% to 5% from the lower bounds obtained by the LPX-relaxation. For the uncorrelated and favorite machine instances, there is the same behavior, whereas for the machine correlated instances the CQP-relaxation is better for the square as well as the rectangular instances; these results are found in Table 2.2. In the machine correlated instances all jobs have the same favored machine, which results in a somewhat higher schedule length. Because of this higher schedule length, the LPY'-relaxation has a larger deviation from the LPY-relaxation, resulting in a worse lower bound.

The upper bounds, i.e., values of schedules, obtained by the CQP, LPY, and LPX approach are given in Table 2.3.

The LPX approach yields the best upper bounds of these three methods, on average around 0.13% from the best lower bound. The reason that the LPX approach yields better solutions is that the solution to the LPX-relaxation is much closer to a real schedule than the solutions to the CQP and LPY'-relaxations, that is, the number of jobs J_j for which there is a machine M_i such that $\sum_t x_{ijt} = 1$ is much larger than the number of jobs for which there is a machine M_i such that $\sum_t \frac{y_{ijt}}{p_{ij}} = 1$ or $z_{ij} = 1$, respectively.

In Table 2.4 we report on the average relative deviation from the best lower bound obtained by the CQP, LPY and LPX approach for the machine correlated instances. Although the lower bounds for the LPY'-relaxation are worse than those

$n \times m$	10×10	20×20	50×50	10×5	20×10	50×20	100×50	average
CQP	1.20	1.23	1.48	1.02	1.19	1.19	1.37	1.24
	(9.38)	(6.57)	(3.56)	(5.61)	(4.55)	(3.46)	(3.26)	
LPY	0.69	0.77	0.81	0.84	0.96	1.05	1.12	0.89
	(7.05)	(2.99)	(2.68)	(4.77)	(4.10)	(3.07)	(3.01)	
LPX	0.07	0.08	0.06	0.15	0.16	0.20	0.17	0.13
	(1.51)	(1.25)	(0.71)	(3.48)	(2.65)	(0.91)	(1.24)	

Table 2.3: Heuristics with constant guarantee: average (maximum) relative deviation in % from best lower bound

$n \times m$	10×10	20×20	50×50	10×5	20×10	50×20	100×50	average
CQP	1.54	1.56	1.69	1.16	1.63	1.75	1.68	1.57
LPY	0.94	1.30	1.43	1.15	1.36	1.63	1.61	1.34
LPX	0.05	0.09	0.11	0.07	0.04	0.06	0.07	0.07

Table 2.4: Heuristics with constant guarantee: average relative deviation in % from best lower bound - machine correlated instances

for the CQP-relaxation, we see that the values of the schedules obtained by the LPY approach are not worse than those obtained by the CQP approach. The reason for this is that the average number of fractional assignments obtained by the CQP-relaxation is not less than the number of fractional assignment obtained by the LPY'-relaxation.

Table 2.5 shows the results of applying iterative improvement, denoted by II, and tabu search, denoted by TS, to the solutions of the heuristics with a constant guarantee. We see that applying a simple iterative improvement procedure to the CQP and LPY solutions, although improving the solutions significantly, still yields results that are not better than the solution of the LPX approach. Applying tabu search to the solutions of the CQP and LPY approach yields results that are on average as good as the LPX approach; note that the tabu search solutions are all very close to optimal as their values are only about 0.1% away from the best lower bound. Tabu search applied to the LPX solution yields the best results: on average less than 0.1% from the best lower bound.

In Table 2.6, the results of the local search heuristics are given. We have used two different time bounds as stopping criteria for the local search heuristics. The first one is that the time is equal to the time used by the LPX approach; the second time bound is that the time equal to the time used by the LPX approach and applying tabu search to these solutions. The latter will be denoted by the extension "long". We see that tabu search applied to a good starting solution yields better results than applying it to random start solutions and the multi-start iterative improvement procedures. The multi-start iterative improvement procedures perform better than the tabu search applied to the randomly generated start solutions. This implies

$n \times m$	10×10	20×20	50×50	10×5	20×10	50×20	100×50	average
CQP	1.20	1.23	1.48	1.02	1.19	1.19	1.37	1.24
$CQP + II$	0.50	0.45	0.68	0.34	0.40	0.44	0.59	0.48
$CQP + TS$	0.06	0.06	0.18	0.12	0.11	0.18	0.27	0.14
LPY	0.69	0.77	0.81	0.84	0.96	1.05	1.12	0.89
$LPY + II$	0.29	0.23	0.36	0.25	0.35	0.40	0.47	0.34
$LPY + TS$	0.06	0.07	0.15	0.12	0.11	0.16	0.23	0.13
LPX	0.07	0.08	0.06	0.15	0.16	0.20	0.17	0.13
$LPX + II$	0.06	0.07	0.04	0.13	0.13	0.14	0.11	0.10
$LPX + TS$	0.04	0.05	0.03	0.12	0.09	0.11	0.08	0.08

Table 2.5: Heuristics with constant guarantee plus local search: average relative deviation in % from best lower bound

$n \times m$	10×10	20×20	50×50	10×5	20×10	50×20	100×50	average
LPX	0.07	0.08	0.06	0.15	0.16	0.20	0.17	0.13
$LPX + TS$	0.04	0.05	0.03	0.12	0.09	0.11	0.08	0.08
II	0.06	0.07	0.27	0.12	0.09	0.10	0.38	0.16
II long	0.05	0.05	0.22	0.12	0.09	0.09	0.37	0.14
TS	0.09	0.30	0.56	0.13	0.19	0.30	0.39	0.28
TS long	0.08	0.19	0.45	0.12	0.12	0.20	0.35	0.22

Table 2.6: LPX versus local search: average relative deviation in % from best lower bound

that it is hard to get out of the local optima and it is better to have many start solutions and do a simple improvement procedure than to do a more sophisticated improvement procedure on few start solutions.

In Table 2.7, we report on the performance of the several heuristics on the hardest instances for the LPX approach. For each size, we have chosen from all the instances the one for which the LPX approach has the worst performance ratio. We see that for these instances the tabu search procedure applied to the schedule obtained by the LPX approach yields a schedule that is the best found or close to the best found schedule. Moreover, we see that the CQP and LPY approach on these hard instances for the LPX approach do not perform much better, except for the CQP approach on the instance of size 10×10.

Finally, Table 2.8 reports on the time it takes to find the solutions for the heuristics with guarantees. As the local search procedures that are applied to the randomly generated start solutions take about the same time as the LPX approach, we do not report on the time usage of these heuristics. Applying iterative improvement to the solutions obtained by the good start solutions takes about 0.02 seconds for the largest instances. Therefore, the time for obtaining the good start solution hardly differs from the time of obtaining it and then applying iterative improvement to it. Although solving the LPX-relaxation has no polynomial-time guarantee, we see

$n \times m$	10×10	20×20	50×50	10×5	20×10	50×20	100×50
CQP	0.03	0.62	1.51	4.11	2.25	1.52	1.43
LPY	1.51	0.62	2.21	4.11	1.38	1.49	1.40
LPX	1.51	1.25	0.71	3.48	2.65	0.91	1.24
$CQP + TS$	0.03	0.33	0.34	3.48	1.14	0.46	0.61
$LPY + TS$	0.03	0.33	0.64	3.48	1.14	0.46	0.55
$LPX + TS$	0.03	0.33	0.26	3.48	1.14	0.52	0.60
II	0.03	0.33	0.04	3.48	1.19	0.48	0.48
II long	0.03	0.33	0.03	3.48	1.14	0.48	0.48
TS	0.77	0.79	0.24	3.48	1.57	0.52	0.57
TS long	0.77	0.33	0.24	3.48	1.35	0.52	0.57

Table 2.7: Hardest instances for LPX approach

$n \times m$	10×10	20×20	50×50	10×5	20×10	50×20	100×50
CQP	1.66	6.72	153.92	1.12	3.46	41.22	985.98
$CQP + TS$	1.75	8.92	225.79	1.21	5.25	95.66	1278.72
LPY	2.00	13.33	442.15	0.81	5.93	68.58	2407.76
$LPY + TS$	2.08	15.40	507.83	0.90	7.70	122.55	2684.35
LPX	0.05	0.41	14.36	0.08	0.59	28.56	684.89
$LPX + TS$	0.12	2.04	42.47	0.16	1.98	66.57	883.09

Table 2.8: Time usage in seconds: average over all instances

that this method is fastest. The smallest instances are solved in a few seconds. The CQP approach needs on average more than 15 minutes and the LPY approach even needs around 40 minutes for solving instances of size 100×50. The LPX approach is much faster: it takes about 10 minutes. The application of tabu search takes about another 3 to 5 minutes for the largest instances.

2.6 Concluding remarks

Our main goal in this chapter was to make an empirical comparison of approximation algorithms with performance guarantees and local search heuristics for $R\|\sum w_j C_j$. The algorithms with worst-case performance guarantees are based on rounding solutions to relaxations of the scheduling problem; these relaxations also provide us with lower bounds. In Section 2.3, we proved that the best lower bounds are obtained by the LPX-relaxation. In Section 2.5, we saw that rounding the solution to this relaxation and then applying tabu search on this feasible schedule also yields the best upper bounds.

Comparing our results with the work of Savelsbergh et al. (1998) and Uma and Wein (1998) on $1|r_j|\sum w_j C_j$, we come to the same conclusions: rounding a better relaxation yields a better schedule and the best schedules are obtained by combining a heuristic based on a good relaxation and local search.

With the work of Hariri and Potts (1991) and Glass et al. (1994) on $R\|C_{\max}$ there is some difference. In an empirical evaluation of heuristics with a constant performance guarantee, Hariri and Potts concluded that the performance of these heuristics is unsatisfactory as a deviation from the optimal solution of more than 10% was normal. Applying a simple improvement procedure resulted in a significant improvement of the schedules. Glass et al. observed that some good local search heuristics usually yield schedules within 1% of the best found solution. The procedures with a constant performance guarantee that we evaluated resulted in solutions that were within 1 or 2% of optimal and for the LPX approach even within 0.2%. The reason that the heuristics with constant performance guarantee we considered are much better can be found in the objective function. We considered the sum of weighted completion times and each job contributes to the objective function. Some jobs will have a smaller completion time compared to theirs in an optimal schedule and other jobs will have a higher one. In $R\|C_{\max}$ the objective is makespan minimization, so we only look at the last job to be completed. Also the objective value for the sum of weighted completion times is much larger than for makespan minimization and the same absolute difference yields a lower relative difference for the problem considered in this chapter than for makespan minimization problems.

3

Max cut: an empirical evaluation

3.1 Introduction

Consider an undirected graph $G = (V, E)$ and a weight function $w : E \to \mathbb{Z}$. We denote the number of vertices $|V|$ by n and the number of edges $|E|$ by m. We assume that $V = \{1, \ldots, n\}$. A set $S \subseteq V$ defines the cut $\delta(S)$ consisting of all edges with one end point in S and the other in $V \backslash S$. The weight of this cut is the sum of the weights of the edges in the cut, i.e., $w(S) = \sum_{e \in \delta(S)} w_e$. The max cut problem is the problem of finding a set S that maximizes the weight of the cut. It is one of the original NP-hard problems in Karp (1972). Even when the graph is unweighted, i.e., $w_e = 1$ for all $e \in E$, it is NP-hard, see Garey, Johnson, and Stockmeyer (1976). Besides its theoretical importance, max cut has applications in, among others, VLSI design and statistical physics.

In this chapter, we make an empirical comparison of *local search heuristics* and *polynomial-time ρ-approximation algorithms*. The first polynomial-time ρ-approximation algorithm for max cut is due to Sahni and Gonzalez (1976). The performance guarantee of their algorithm is $\frac{1}{2}$. In the twenty years thereafter, polynomial-time algorithms were presented with performance guarantees of $\frac{1}{2} + \frac{1}{2m}$ (Vitányi, 1981), $\frac{1}{2} + \frac{n-1}{4m}$ (Poljak and Turzik, 1986), $\frac{1}{2} + \frac{1}{2n}$ (Haglin and Venkatesan, 1991), and $\frac{1}{2} + \frac{1}{2\Delta}$ (Hofmeister and Lefmann, 1996), where Δ denotes the maximum degree in the graph G. No progress was made in improving the constant in the performance guarantee beyond that of Sahni and Gonzalez's straightforward algorithm, until in 1995 Goemans and Williamson (1995) presented a polynomial-time algorithm with performance guarantee 0.878. Their algorithm is based on a semidefinite relaxation of max cut. Feige, Karpinski, and Langberg (2000) showed that applying

a simple local search procedure to the solution obtained by the Goemans-Williamson algorithm improves the performance guarantee to $0.878 + \epsilon_\Delta$, where ϵ_Δ is a positive constant depending on the maximum degree of the graph. For $\Delta = 3$, the performance guarantee is 0.921, and for 3-regular graphs, the performance guarantee is even 0.924. In general, $\epsilon_\Delta = 1/(2^{33}\Delta^4)$. On the negative side, Håstad (1997) showed that, unless P = NP, there cannot exist a polynomial-time ρ-approximation algorithm for any $\rho > \frac{16}{17}$.

In their paper, Goemans and Williamson also made a small computational evaluation of their algorithm. For the instances they used the cuts obtained were usually within 4% of the semidefinite upper bound and never lower than 91% of this upper bound. Homer and Peinado (1997) made a parallel implementation of the Goemans-Williamson algorithm and also implemented a simulated annealing procedure to compare the quality of the cut. They concluded that simulated annealing in most of their test instances yields the largest cut and both methods yield cuts that differ at most 5% from the upper bound.

Poljak and Rendl (1994) tried to obtain good upper bounds for max cut through eigenvalue optimization and as a by-product they also obtained some cuts which were typically about 2–6% and never more than 9% away from the upper bound. Helmberg and Rendl (2000) also used eigenvalue optimization to find good upper bounds. Experiments using polyhedral approaches are reported by Barahona, Grötschel, Jünger, and Reinelt (1988), Barahona, Jünger, and Reinelt (1989) and De Simone, Diehl, Jünger, Mutzel, Reinelt, and Rinaldi (1995), who tested cutting plane algorithms. Branch-and-bound algorithms are proposed by Carter (1984) and Pardalos and Rodgers (1990).

There is not much literature on computational experiments with local search methods for max cut. Empirical evaluations of local search methods of some graph partitioning problems include the work of Lang and Rao (1993), who considered the minimum quotient cut problem, i.e., the problem of minimizing the value of the cut divided by the size of the smallest of the two partitions, and compared a flow based algorithm with the Fiduccia and Mattheyses (1982) variant of the neighborhood proposed by Kernighan and Lin (1970) for graph partitioning. Berry and Goldberg (1999) compared their so-called *path optimization* algorithm to the Kernighan-Lin method and a simulated annealing heuristic for the min-quotient-cut-problem as well as max cut. The path optimization heuristic is a variant of the Kernighan-Lin method in which not necessarily all vertices are examined. For max cut, the simulated annealing procedure yielded slightly better cuts. Johnson, Aragon, McGeoch, and Schevon (1989) report on an experimental study of simulated annealing for the graph partitioning problem, that is, the problem of partitioning the vertex set in two equal sized parts such that the weight of the cut edges is minimized. For more references on max cut, we refer to the survey by Poljak and Tuza (1995) and the annotated bibliography by Laurent (1997).

The quality of the obtained cuts are given relative to an upper bound. In the following section, we discuss some methods to obtain upper bounds. In Section 3.3, we present the polynomial-time ρ-approximation algorithms. The solutions obtained

by these methods can be improved by using the local search heuristics described in Section 3.4. Then we make some remarks on the implementational details and the test instances, and we present the results of our empirical evaluation. Finally, we make some concluding remarks.

3.2 Upper bounds

Polyhedral bound. The max cut problem can be formulated as a linear program over the *cut polytope*. This polytope is the convex hull of the incidence vectors of the edges of the cuts of the graph. More formally, let $\chi(S) \in \{0,1\}^m$ denote the characteristic vector of the cut induced by $S \subseteq V$, i.e.,

$$\chi_e(S) = \begin{cases} 1 & \text{if } e \in \delta(S), \\ 0 & \text{otherwise.} \end{cases}$$

Then the cut polytope of the graph G, $P_C(G)$, is defined by $P_C(G) = \text{conv}\{\chi(S) : S \subseteq V\}$. Using this polytope, max cut can be formulated as

$$(P) \qquad \begin{array}{ll} \max & \sum_{e \in E} w_e y_e \\ \text{s.t.} & y \in P_C(G). \end{array}$$

Note that $P_C(G)$ is an integral polyhedron. Thus there exists an optimal solution to (P) that is integral. In such a solution, the binary variable y_e denotes whether edge $e \in E$ is in the cut or not.

Obviously, we do not know how to optimize over $P_C(G)$ in polynomial time. Therefore, we optimize over a relaxation of the cut polytope. The relaxation we consider is the metric polytope, which consists of all vectors $y \in \mathbb{R}^m$ satisfying

$$\sum_{e \in F} y_e - \sum_{e \in C \setminus F} y_e \leq |F| - 1 \qquad \text{for all cycles } C \text{ and all } F \subseteq C, |F| \text{ odd}, \quad (3.1)$$

$$0 \leq y_e \leq 1 \qquad \text{for all } e \in E. \quad (3.2)$$

These inequalities are clearly valid for $P_C(G)$ since a cycle can only contain an even number of edges in the cut. Moreover, inequalities (3.1) define a facet for $P_C(G)$ if and only if C is a chordless cycle (Barahona and Mahjoub, 1986). Inequalities (3.2) define a facet if and only if e is not an edge of a triangle in G (Barahona and Mahjoub, 1986). Barahona and Mahjoub (1986) showed that the separation problem for the metric polytope can be solved in polynomial time. That is, given a point y it can be verified in polynomial time whether or not y is in the polytope and if not, a hyperplane is found in polynomial time that separates the point from the polytope. Hence, we can optimize a linear function over the metric polytope in polynomial time using the ellipsoid algorithm, see Grötschel, Lovász, and Schrijver (1981, 1984). Using an LP solver like CPLEX, we cannot add all inequalities a priori to the LP, as the number of constraints may be an exponential in the input size, e.g., when G is a cycle on a odd number of vertices. Hence, we solve a smaller LP, e.g. the one with only constraints (3.2), and we iteratively add a number of violated constraints until there are no violated constraints.

Another way to deal with a possibly large number of constraints is to extend G to the complete graph by setting $w_{ij} = 0$ whenever $\{i, j\} \notin E$. The only chordless cycles in the complete graph are triangles and the metric polytope is thereby completely defined by the inequalities corresponding to the triangles. The number of inequalities in this case is $\mathcal{O}(n^3)$. However, this number of inequalities is still too large to add directly to an LP. Therefore, also in this case, we have to iteratively add violated constraints. In practice this method performs not as well as the previously mentioned method, as we cannot add more than 5,000 triangle constraints due to memory limitations.

Eigenvalue bound This bound was introduced by Mohar and Poljak (1990) and is defined in terms of the maximum eigenvalue of the Laplacian matrix L of the weighted graph (G, w). This matrix has elements l_{ij}, for $i, j \in V$, defined by $l_{ij} = -w_{ij}$ if $i \neq j$ and $l_{ii} = \sum_j w_{ij}$. For $S \subseteq V$, we define the associated node incidence vector $x(S) \in \{-1, 1\}^n$ by $x_i(S) = 1$ if and only if $i \in S$. The weight of the cut induced by S is $w(S) = \frac{1}{4}x^T(S)Lx(S)$. The max cut problem can be formulated as

$$\begin{array}{ll} \max & \frac{1}{4}x^T Lx \\ \text{s.t.} & x \in \{-1, 1\}^n. \end{array}$$

Clearly, for all $u \in \mathbb{R}^n$ satisfying $\sum_i u_i = 0$, we have that $x^T \text{diag}(u)x = 0$ for all vectors $x \in \{-1, 1\}^n$, where $\text{diag}(u)$ denotes the diagonal matrix with entries u_i on the diagonal. Define $f(u) = \max\{\frac{n}{4}x^T(L + \text{diag}(u))x : x^T x = 1\}$. Since, for all $u \in \mathbb{R}^n$ with $\sum_i u_i = 0$, $f(u)$ is an upper bound on the value of the maximum cut, the upper bound can be defined as $\min\{f(u) : u \in \mathbb{R}^n, \sum_i u_i = 0\}$. Since $f(u) = \frac{n}{4}\lambda_{\max}(L+\text{diag}(u))$, where $\lambda_{\max}(A)$ is the maximum eigenvalue of the matrix A, the function can be computed efficiently for all u and the eigenvalue bound is the following:

$$(EV) \qquad \begin{array}{ll} \min & \frac{n}{4}\lambda_{\max}(L + \text{diag}(u)) \\ \text{s.t.} & \sum_i u_i = 0. \end{array}$$

Delorme and Poljak (1993) showed that this program can be solved up to an additive error $\epsilon > 0$ in time polynomial in the input size and $\log \frac{1}{\epsilon}$.

Semidefinite bound. For the semidefinite bound, we also use the node incidence vector $x(S) \in \{-1, 1\}^n$ associated with a subset $S \subseteq V$. If an edge $\{i, j\}$ is in the cut induced by S, then $x_i(S)x_j(S) = -1$ and otherwise $x_i(S)x_j(S) = 1$. Hence, $\frac{1}{2}(1-x_i(S)x_j(S)) = 1$ if $\{i, j\}$ is in the cut and 0 otherwise. Using the node incidence variables, we can formulate the problem as

$$(IQP) \qquad \begin{array}{lll} \max & \sum_{i<j} w_{ij}\frac{1-x_i x_j}{2} & \\ \text{s.t.} & x_i \in \{-1, 1\} & i \in V, \end{array}$$

where we set $w_{ij} = 0$ whenever $\{i, j\}$ is not an edge.

For a vector $x \in \{-1, 1\}^n$ the matrix X defined by $X = xx^T$ is a positive semidefinite matrix which has diagonal entries equal to 1. The rank of this matrix

is 1. Hence (IQP) can be reformulated as

$$(IQP') \qquad \begin{array}{lll} \max & \sum_{i<j} w_{ij} \frac{1-X_{ij}}{2} & \\ \text{s.t.} & X_{ii} = 1 & i \in V, \\ & X \succeq 0, & \\ & \text{rank}(X) = 1, & \end{array}$$

where $X \succeq 0$ denotes that X is positive semidefinite. Given a solution X to (IQP'), the corresponding solution $x \in \{-1, 1\}^n$ to (IQP) is equal to the first column of the rank-one matrix X. The semidefinite bound is obtained by relaxing the rank-one-condition:

$$(SDP) \qquad \begin{array}{lll} \max & \sum_{i<j} w_{ij} \frac{1-X_{ij}}{2} & \\ \text{s.t.} & X_{ii} = 1 & i \in V, \\ & X \succeq 0. & \end{array}$$

This semidefinite program can be solved up to an additive error $\epsilon > 0$ in time polynomial in the input size and $\log \frac{1}{\epsilon}$, see e.g. Alizadeh (1995). Feige and Schechtman (2002) showed that the integrality gap for the semidefinite bound is $\alpha = \min_{0<\theta<\pi} \frac{2}{\pi} \frac{\theta}{1-\cos\theta} > 0.878$.

Strengthened semidefinite bound. The semidefinite bound (SDP) can be strengthened by adding some valid inequalities, such as the triangle inequalities. These inequalities are given by

$$\begin{array}{rcrcrcll} X_{ij} & + & X_{ik} & + & X_{jk} & \geq & -1, & \forall 1 \leq i < j < k \leq n, \\ X_{ij} & - & X_{ik} & - & X_{jk} & \geq & -1, & \forall 1 \leq i < j < k \leq n, \\ -X_{ij} & + & X_{ik} & - & X_{jk} & \geq & -1, & \forall 1 \leq i < j < k \leq n, \\ -X_{ij} & - & X_{ik} & + & X_{jk} & \geq & -1, & \forall 1 \leq i < j < k \leq n. \end{array}$$

To see that these inequalities are valid, note that at most two edges of a triangle can be in the cut. Adding these inequalities to (SDP), we obtain a semidefinite program, denoted by $(SDP+)$, which can be solved in polynomial time up to an additive error $\epsilon > 0$. The integrality gap for this bound is bounded from above by 0.891 (Feige and Schechtman, 2002).

The number of additional constraints is too large to be handled by an SDP solver. Hence, we iteratively add some of the most violated constraints to our SDP problem. After each iteration, we can round the obtained solution to a feasible solution, in the way that will be described in the following section. If this lower bound is equal to the upper bound, we have found an optimal solution and we stop. Due to memory limitations, we could not add more than 2,000 triangle constraints.

Dominance relations. Poljak and Rendl (1995) showed that the eigenvalue bound and the semidefinite bound are equivalent. As any feasible solution to $(SDP+)$ is a feasible solution to (SDP), the strengthened semidefinite bound dominates the semidefinite bound. The semidefinite bound does not dominate the polyhedral bound, and the polyhedral bound does not dominate the semidefinite bound, as can be seen by the following two examples. For the 5-cycle with unit weights

on the edges, the polyhedral bound yields the optimal cut value, 4, whereas the semidefinite bound has value 4.52. On the other hand, the complete graph on five vertices yields a polyhedral bound of $6\frac{2}{3}$, whereas the semidefinite is equal to 6.25. Finally, the strengthened semidefinite bound dominates the polyhedral bound. Given a solution, X, to $(SDP+)$, we can construct a solution, y, to (LP) by setting $y_{ij} = (1 - X_{ij})/2$, whenever $\{i, j\}$ is an edge. It is easy to verify that this solution y satisfies the odd cycle inequalities of the polyhedral bound and it yields the same objective value as the strengthened semidefinite bound.

3.3 Constructive heuristics

We have considered three randomized polynomial-time ρ-approximation algorithms. The first one, called *greedy*, is due to Sahni and Gonzalez (1976). This algorithm iterates through the vertex set and based on the assignment of vertices $1, \ldots, i$ it assigns vertex $i + 1$ to S if the cut induced by $S \cup \{i + 1\}$ has a larger weight than the cut induced by S in the induced subgraph on $\{1, \ldots, i + 1\}$. In the case that all edge weights are nonnegative, this algorithm has performance guarantee $\frac{1}{2}$. If there are also negative edge weights, the weight of the cut found by greedy minus the sum of the negative edge weights is at least half of the optimal cut value minus the sum of the negative edge weights. Note that the order in which the vertices are processed may influence the obtained cut and thus we can get different cuts by labeling the vertices in different ways.

The second method is called *coin flip*, which flips, independently, an unbiased coin for each vertex to decide whether or not to assign it to S. The randomized performance guarantee of this algorithm for graphs with nonnegative edge weights is also $\frac{1}{2}$, i.e., the expected weight of the cut is at least half of the optimal cut. If there are also negative edge weights, then the expected weight of the cut minus the sum of the negative edge weights is at least half of the optimal cut minus the sum of the negative edge weights. Note that a derandomized version of coin flip is the greedy algorithm.

The last constructive heuristic that we consider is the celebrated algorithm of Goemans and Williamson (1995). First, they solve the semidefinite program (SDP) presented in the previous section. Using Cholesky decomposition, the matrix X obtained from this relaxation can be decomposed into vectors $v_1, \ldots, v_n \in \mathbb{R}^n$ such that $X_{ij} = v_i^T v_j$. A vector v_i represents the vertex $i \in V$. This vector solution can be rounded to a feasible solution using the so-called random hyperplane method: we choose randomly and uniformly a vector r on the unit sphere $B_n = \{x \in \mathbb{R}^n : \|x\| = 1\}$ and we set $S = \{i : r^T v_i \geq 0\}$. In other words, we choose a random hyperplane through the origin with r as its normal and partition the vertices into those vectors that lie above or in it ($r^T v_i \geq 0$) and those that lie below the hyperplane. If Z_{SDP} denotes the value of the semidefinite bound and all edge weights are nonnegative, then the expected weight of the cut obtained in this manner is bounded from below by αZ_{SDP}, where $\alpha = \min_{0<\theta<\pi} \frac{2}{\pi} \frac{\theta}{1-\cos\theta} > 0.878$. Karloff (1999) showed that the performance ratio does not improve when triangle inequalities are added to the semidefinite relaxation. As a Cholesky decomposition can be done in polynomial

time and (SDP) can be solved in polynomial time up to any additive error $\epsilon > 0$, this algorithm is a randomized polynomial-time $(\alpha - \epsilon)$-algorithm. In the case that there are also negative edge weights, we have that $E[w(S)] - W^- \geq \alpha(Z_{SDP} - W^-)$, where W^- is the sum of all negative edge weights. In the following sections, this algorithm will be denoted by GW. The randomized way of rounding the vectors v_i can be derandomized, as was shown by Mahajan and Ramesh (1999).

3.4 Improvement heuristics

We compare the algorithms described in the previous section to local search heuristics. A local search method starts from some initial solution. Our starting solutions are obtained by the three constructive heuristics from the previous section. In the case that the initial solutions are obtained by the greedy and coin flip procedures, we use multi-start local search, i.e., we start the local search procedure from several initial solutions. For the coin flip procedure, we used multiple random assignments; for the greedy algorithm we generated several random orders of the vertices. The number of starting solutions generated by the coin flip and the greedy method is chosen such that the total time spent on a local search heuristic is about equal to the time for obtaining the cut from the GW algorithm.

We mentioned that in local search, a move is made to a neighboring solution. We have chosen to use two basic neighborhoods. The first one, called *flip*, selects a single vertex and moves it to the other side of the partition, i.e., if the vertex is in S it is outside of S after the flip and vice versa. The other neighborhood function is called *swap*. In this neighborhood, we select an edge that is in the cut and flip both vertices that form this edge.

In the following subsections, we describe the implemented local search heuristics: iterative improvement, tabu search, simulated annealing, and variable-depth flip.

3.4.1 Iterative improvement

As mentioned in Chapter 1, iterative improvement is the simplest form of local search. We have implemented this search strategy using a neighborhood consisting of all flip and swap neighbors.

There are several ways to select the neighbor to move to. In *first improvement*, we move to the first better neighbor that we encounter. In *best improvement*, we move to the neighbor with highest cut value among all neighbors.

For this simple form of local search, some effort has been put in the time complexity of finding local optimal solutions. Schäffer and Yannakakis (1991) have shown that max cut with the flip neighborhood is PLS-complete, see Johnson, Papadimitriou, and Yannakakis (1988). Schäffer and Yannakakis also showed that there exist instances of max cut and starting solutions such that the number of flips needed to find a local optimum is exponential in the input size. For cubic graphs Poljak (1995) showed that a flip optimal solution can be found using $\mathcal{O}(n^2)$ flips.

3.4.2 Tabu search

Recall that in tabu search we make use of a tabu list to avoid cycling. The information we store in the tabu list is the move, i.e., the vertex that has been flipped or the edge that has been swapped. Storing a move in the tabu list means that we cannot undo this move in the next few iterations. The number of iterations during which this is not allowed is equal to the length of the tabu list, which, after some initial experiments, we have set to 10.

Another feature of our tabu search procedure is the so-called *backjump*, an idea that Nowicki and Smutnicki (1996) applied to the job shop scheduling problem: if we have made 100 non-improving moves without improving the best found solution, we return to this best found solution and move to a neighbor that has not been visited yet directly from this solution.

3.4.3 Simulated annealing

As mentioned in Chapter 1, simulated annealing uses a control parameter, called the temperature, for determining the probability for accepting deteriorations. We need to specify some parameters for simulated annealing: an initial temperature, the cooling schedule, and the stopping criteria. The initial temperature is chosen such that in the beginning approximately 80% of the neighboring solutions are accepted. Most of the simulated annealing procedures perform a number of iterations with the same temperature. These iterations form a Markov chain. When the temperature is high, many solutions will be accepted. Therefore, we let the length of the Markov chain be small when the temperature is high. For the initial temperature this length is set to the number of vertices. Each time the temperature decreases, we increase the length by n, until it reaches its maximum value of 25 times the number of vertices. Finally, after some initial tests, we have decided to use the cooling schedule proposed by Aarts and Van Laarhoven (1985a,b). The decrement of the temperature depends on a parameter δ. Let T_k be the temperature in the kth Markov chain. Then the temperature in the $(k+1)$st Markov chain, T_{k+1}, is given by

$$T_{k+1} = \frac{T_k}{1 + (\log(1+\delta)/3\sigma_k)},$$

where σ_k is the standard deviation of the cut values of the solutions obtained during the iterations where the temperature was T_k. After some initial test, δ was set equal to 0.1.

We stop when the temperature has dropped below 0.05, or when the process seems *frozen*, i.e., when for the last six temperatures less than 2% of the number of neighbors examined, has been accepted.

As we do not want to spend much time on choosing our random neighbor, we have decided only to use the flip neighborhood, for which we can generate a random neighbor in constant time, whereas to generate a swap neighbor, we might need time proportional to the number of edges.

3.4.4 Variable-depth flip

This search strategy is a variant of the neighborhood of Kernighan and Lin (1970) for graph partitioning. We generate a sequence of n solutions using the following procedure. We start by labeling all vertices as unflipped. As long as there are vertices labeled unflipped, we select among these vertices the one for which flipping it yields the highest increase or lowest decrease in the cut value. We flip this vertex and label it flipped. After n iterations all vertices are labeled flipped and we have constructed n different solutions. Note that the last solution yields the same cut as the original one. We move to the best of these n cuts and repeat the above procedure, until among the sequence of n solutions, no better cut is found than the solution at the beginning of that iteration.

3.5 Computational experience

3.5.1 Test instances

We have tested the described heuristics on graphs with sizes varying from 50 vertices to 1000 vertices. The first set of instances is an extension of the instances used by Helmberg and Rendl (2000). They generated a set of 21 graphs with 800 vertices using `rudy`, a machine independent graph generator written by Rinaldi. We generated graphs with $50, 100, 200, 300 \ldots, 1000$ vertices in the same manner. The command line arguments for `rudy` specifying these graphs are given in Vredeveld (2001). For each of these values of n we have generated the following types of graphs:

1. unweighted random graphs with a density of 6%: five instances;
2. same graphs as in 1, with edge weights uniformly drawn from $\{-1, 1\}$;
3. toroidal grids with random edge weights uniformly drawn from $\{-1, 1\}$: three instances;
4. unweighted almost planar graphs, having as edge set the union of two planar graphs: four instances;
5. same graphs as in 4, with edge weights uniformly drawn from $\{-1, 1\}$.

We also tested our methods on some instances from the Ising model of spin glasses. These instances are obtained from the web-site of the seventh DIMACS Implementation Challenge, see Pataki and Schmieta (1999). There are two instances with 512 vertices and 1536 edges, toruspm8-50 and torusg3-8. Toruspm8-50 has edge weights in $\{-1, 1\}$, torusg3-8 has arbitrary edge weights, which were made integer by multiplying them by $100,000$. These instances will be called *torus instances*. There are two other instances on the web-site, toruspm3-15-50 and torusg3-15, with 3375 vertices and 10125 edges, but these instances could not be solved by the SDP solver, and are therefore excluded from our tests.

3.5.2 Implementational details

All algorithms have been coded in C. For solving the linear program, we used CPLEX 6.6.1; for solving the semidefinite programs, we used CUTSDP by Karish (1998). The

SDP and LP relaxations have been run on a Sun Ultra-1, 140 MHz with 256 MB and the other methods have been run on faster machines but with less memory. The time usage has been corrected to the Sun Ultra-1.

3.5.3 Computational results

We start by comparing the performance of the obtained upper bounds. These results are given in Table 3.1 and Figures 3.1 and 3.2. In the table the average ratio of the upper bound to the best lower bound and its standard deviation are given; the figures denote the minimum, maximum, and average ratio. As the torus instances behave similarly to the weighted instances with 500 vertices, we have included their results in those of the weighted graphs of 500 vertices. Note that for each other number of vertices, there are 9 unweighted instances and 12 weighted ones.

		LP	SDP	$SDP+$
50	unweighted	1.005 (0.008)	1.028 (0.009)	1.000 (0.000)
	weighted	1.000 (0.000)	1.078 (0.041)	1.000 (0.000)
100	unweighted	1.014 (0.010)	1.044 (0.009)	1.002 (0.002)
	weighted	1.006 (0.010)	1.121 (0.023)	1.000 (0.000)
200	unweighted	1.201 (0.079)	1.045 (0.009)	1.018 (0.005)
	weighted	1.507 (0.385)	1.149 (0.032)	1.036 (0.027)
300	unweighted	1.323 (0.106)	1.046 (0.006)	1.026 (0.005)
	weighted	1.775 (0.644)	1.181 (0.036)	1.075 (0.051)
400–800	unweighted	1.501 (0.080)	1.043 (0.003)	1.032 (0.004)
	weighted	2.491 (1.066)	1.182 (0.045)	1.115 (0.065)
900–1000	unweighted	1.568 (0.099)	1.041 (0.002)	
	weighted	3.077 (1.562)	1.191 (0.047)	
all	unweighted	1.381 (0.219)	1.042 (0.007)	1.024 (0.013)
	weighted	2.195 (1.219)	1.167 (0.053)	1.079 (0.071)

Table 3.1: Upper bounds: average ratio to best lower bound (standard deviation)

The SDP+ relaxation could not be solved for instances with 900 or more vertices and for the unweighted almost planar graphs on 100 vertices. The LP relaxation yields good upper bounds for the smaller instances, i.e., those with 50 and 100 vertices. On unweighted graphs with 50 vertices the bound is 0.5% above the best lower bound and for the weighted instances of this size it yields optimal bounds. On graphs with 100 vertices the LP bound is better than the SDP bound. On instances of 200 or more vertices the LP relaxation performs rather poorly: on average, for unweighted instances it is almost 50% above the best lower bound and for the weighted graphs it is even more than twice the best lower bound. The SDP

method yields reasonable bounds: on average, the ratio of this bound to the best lower bound is 1.04 for the unweighted instances and for the weighted instances this ratio is 1.17. For instances on which it could be solved, the SDP+ relaxation gives good bounds: 2.4% above the best lower bound for the unweighted instances and 8% for the weighted ones.

The time required to solve the relaxation is given in Table 3.2, which reports on the average time usage and its standard deviation in seconds, and Figures 3.3 and 3.4, which report on the minimum, maximum, and average time usage in seconds. The LP relaxation could be solved in reasonable time: for graphs with 1000 vertices it needs about 6 minutes for the almost planar graphs and the time required for the other instances varies from half a minute to 2 hours. The SDP relaxation can be solved fast for the smaller instances: within a second for graphs with 50 vertices. However, the running time grows to more than 3 hours for graphs with 1000 vertices. Adding triangle constraints to the SDP relaxation results in huge running times.

We now look at the heuristics for obtaining cuts. The results for the three described constructive methods are given in Table 3.3, which report on the average ratio of the cut value to the best upper bound and its standard deviation, and in Figures 3.5 and 3.6, which report on the minimum, maximum, and average ratio for unweighted and weighted graphs. Recall that in the GW procedure, we round the solution to the SDP relaxation by choosing a random hyperplane through the origin. We have tested this procedure using 50 and 100 random hyperplanes. The quality of the obtained cuts when using 100 random hyperplanes is on average .03% better than the cuts obtained when using 50 random hyperplanes. Therefore, we report on the cuts obtained by using 100 random hyperplanes. The time needed for this rounding procedure varies from 0.24 seconds for the graphs with 50 vertices to 106 seconds for instances with 1000 vertices. When using 50 random hyperplanes, the time required for rounding is halved. Note that the time needed for the GW heuristic is dominated by the time usage of solving the SDP relaxation: on average the rounding procedure requires less than 3% of the total time of the procedure. We have repeated the Greedy and Coin Flip methods several times using random orders on the vertices for the Greedy heuristic. The number of repetitions is chosen such that the time spent on either procedures is the same as the time spent on the GW heuristic.

The GW procedure yields the best cuts and the Greedy heuristic outperforms the Coin Flip method. For the smallest instances the GW heuristic produces almost optimal cuts. On average the cuts are 4% below the best upper bound for the unweighted graphs and 14% for the weighted ones. These values for the Greedy procedure are 6% and 25% respectively. The Coin Flip procedure performs poorly, especially when the weights may be negative.

As the amount of time used for the GW procedure is rather large, we also ran the Greedy and Coin Flip heuristics with a time bound of 450 seconds, for graphs with 400 or more vertices. For the greedy heuristic this resulted in cuts that are about 0.1% worse than the cuts using the large time bound for graphs with 400 vertices up

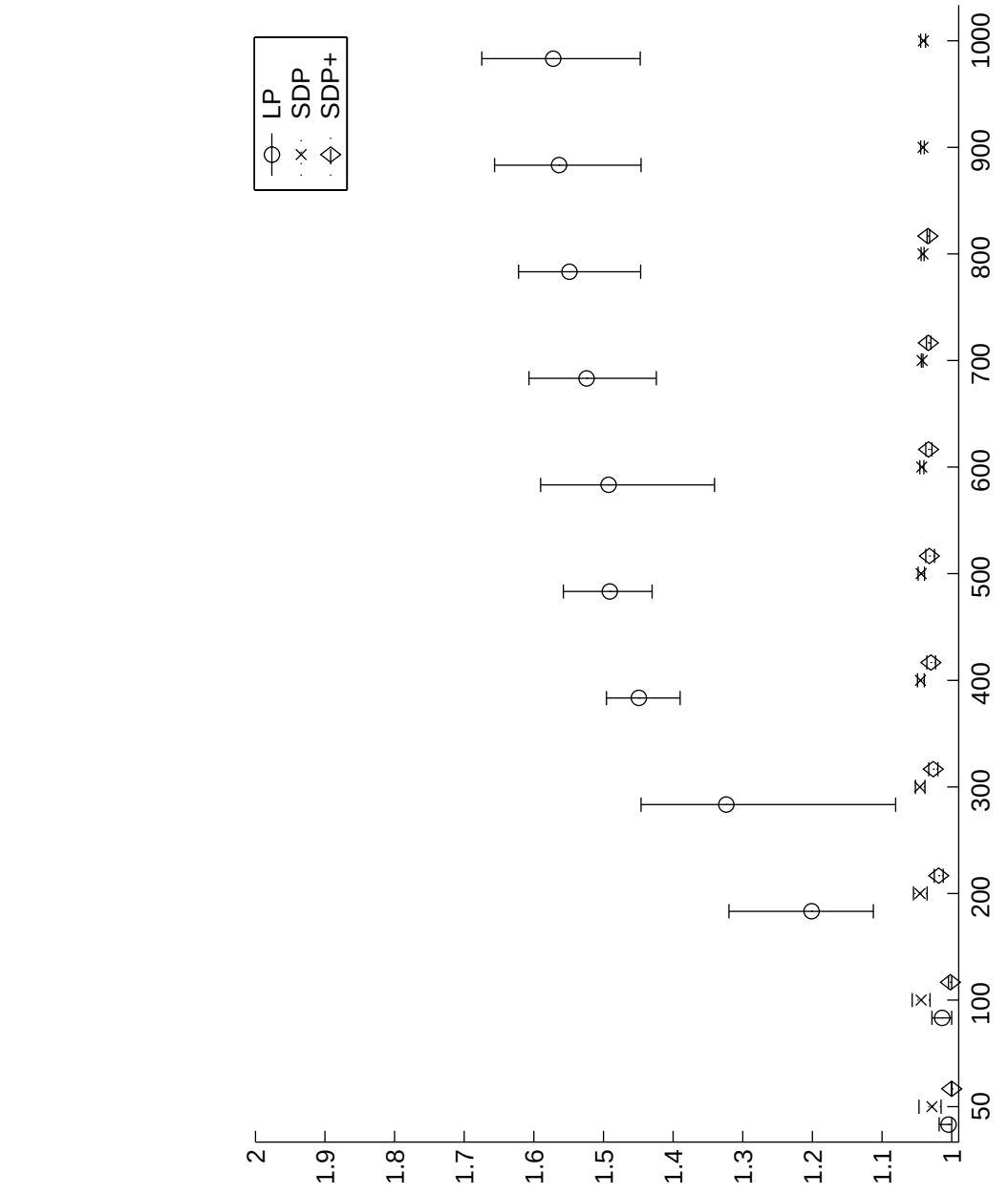


Figure 3.1: Upper bounds for unweighted instances: minimum, maximum, and average ratio to best lower bound

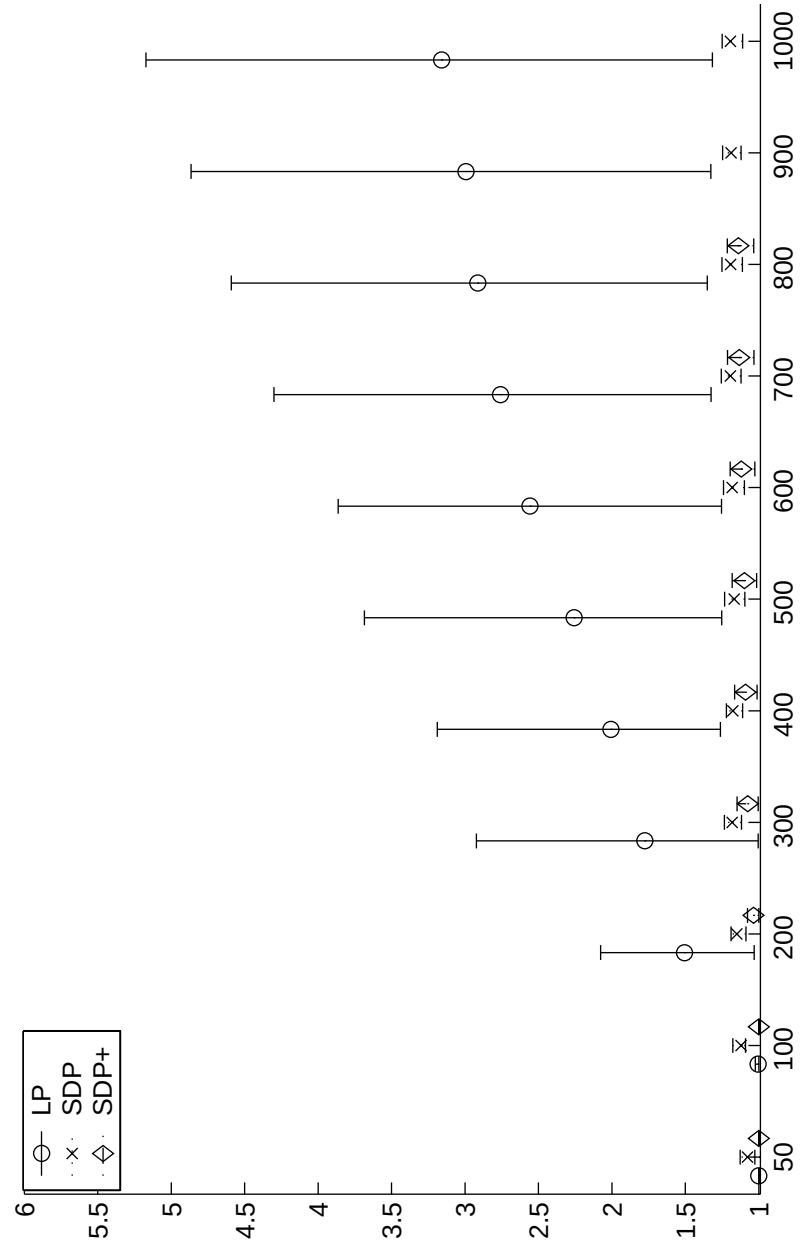


Figure 3.2: Upper bounds for weighted instances: minimum, maximum, and average ratio to best lower bound

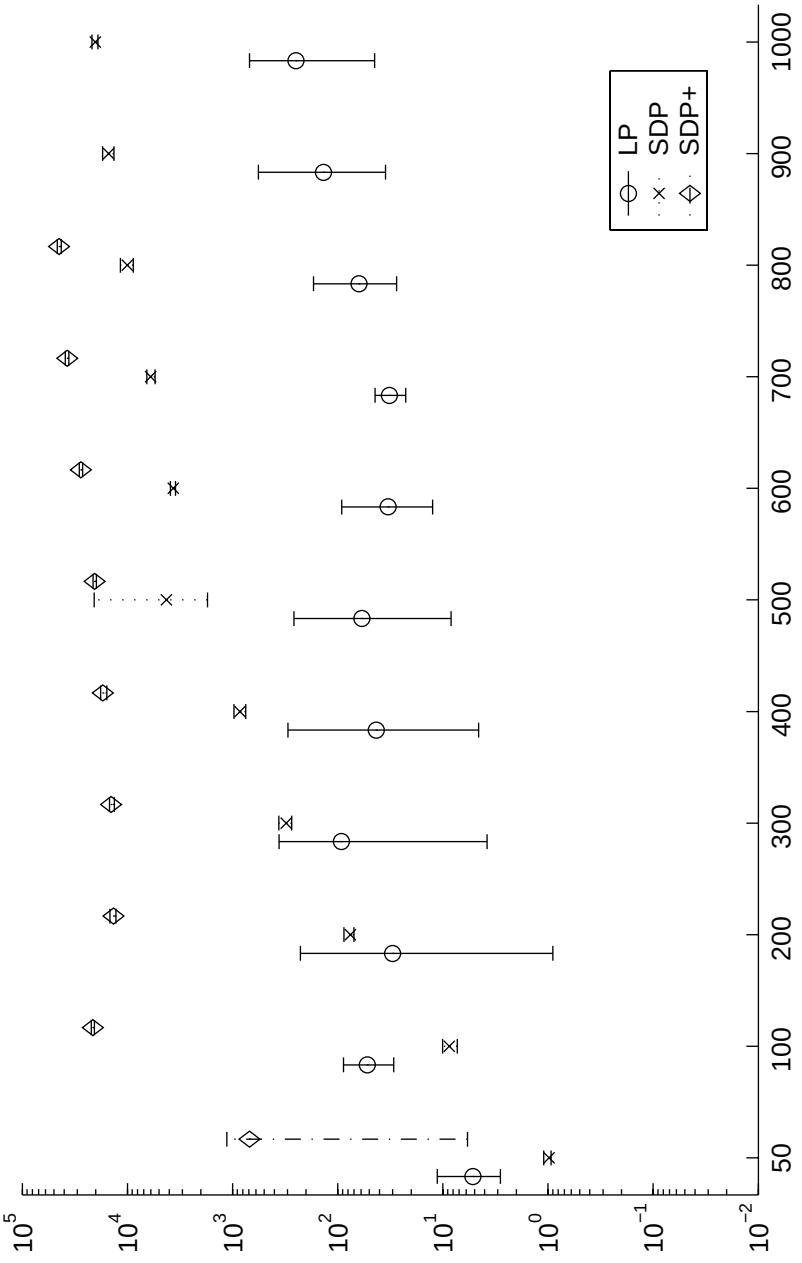


Figure 3.3: Time usage for almost planar graphs: minimum, maximum, and average time usage

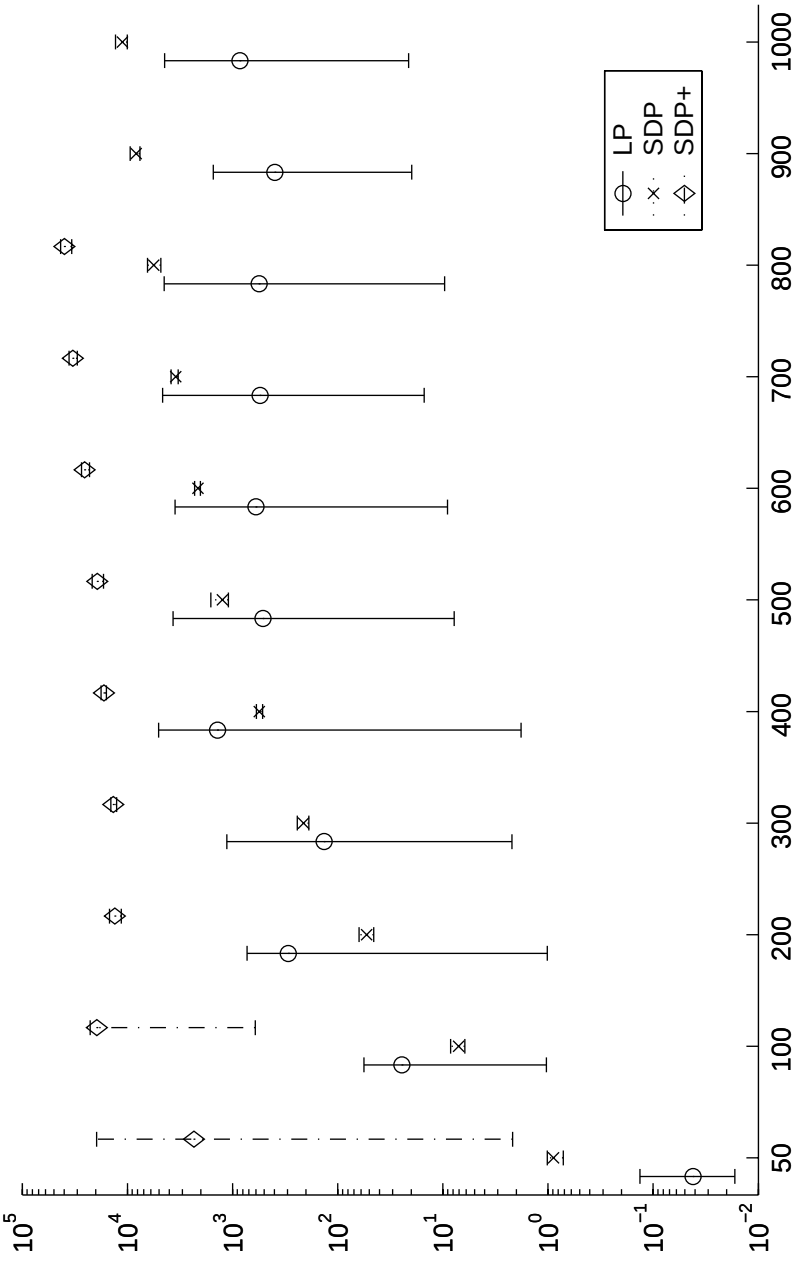


Figure 3.4: Time usage for other instances: minimum, maximum, and average time usage

		LP	SDP	$SDP+$
50	almost planar	7.731 (4.370)	0.979 (0.060)	691.452 (486.883)
	other	0.062 (0.048)	0.886 (0.089)	2327.394 (5835.023)
100	almost planar	78.678 (35.802)	8.694 (0.854)	21374.075 (586.267)
	other	36.698 (22.004)	7.088 (0.671)	19562.456 (6058.644)
200	almost planar	45.109 (118.861)	75.629 (6.266)	13576.806 (849.846)
	other	441.748 (458.763)	53.049 (5.284)	13187.638 (1036.620)
300	almost planar	138.812 (216.808)	309.644 (25.786)	14276.344 (449.588)
	other	201.779 (474.550)	211.297 (14.096)	13605.327 (525.764)
400	almost planar	64.650 (154.100)	851.533 (74.301)	17107.233 (896.006)
	other	2086.738 (3237.235)	557.592 (24.895)	16744.181 (669.912)
500	almost planar	88.653 (129.859)	4265.094 (6657.051)	20482.123 (493.893)
	other	767.648 (1883.494)	1249.038 (145.807)	19355.184 (1782.154)
600	almost planar	49.659 (37.617)	3670.529 (142.764)	27719.696 (759.953)
	other	897.075 (1841.572)	2138.437 (96.474)	25569.642 (1568.883)
700	almost planar	48.372 (12.359)	6021.508 (397.335)	37393.148 (991.841)
	other	822.952 (1855.661)	3478.990 (188.462)	33046.105 (2150.615)
800	almost planar	94.609 (69.548)	10050.567 (1015.959)	44703.392 (1392.681)
	other	839.235 (1780.554)	5632.110 (449.884)	39781.328 (3576.028)
900	almost planar	205.203 (266.315)	15129.900 (1273.643)	
	other	595.458 (607.490)	8252.656 (516.801)	
1000	almost planar	373.488 (399.289)	20176.985 (854.201)	
	other	1277.160 (1887.007)	11168.191 (807.100)	

Table 3.2: Average time usage in seconds (standard deviation)

		Greedy	Coin Flip	GW
50	unweighted	0.986 (0.011)	0.805 (0.062)	0.999 (0.002)
	weighted	0.951 (0.028)	0.503 (0.080)	0.994 (0.011)
100	unweighted	0.960 (0.010)	0.785 (0.047)	0.981 (0.013)
	weighted	0.907 (0.030)	0.378 (0.061)	0.977 (0.017)
200	unweighted	0.944 (0.009)	0.788 (0.037)	0.968 (0.006)
	weighted	0.821 (0.038)	0.229 (0.052)	0.918 (0.041)
300	unweighted	0.933 (0.007)	0.786 (0.022)	0.958 (0.004)
	weighted	0.765 (0.050)	0.137 (0.060)	0.881 (0.057)
400–800	unweighted	0.927 (0.004)	0.796 (0.012)	0.949 (0.004)
	weighted	0.713 (0.054)	0.129 (0.037)	0.839 (0.066)
900–1000	unweighted	0.920 (0.008)	0.801 (0.026)	0.941 (0.007)
	weighted	0.655 (0.035)	0.110 (0.032)	0.775 (0.047)
all	unweighted	0.936 (0.021)	0.795 (0.030)	0.958 (0.018)
	weighted	0.753 (0.101)	0.187 (0.129)	0.863 (0.086)

Table 3.3: Constructive heuristics: average ratio to upper bound (standard deviation)

to 1.7% worse for graphs with 1000 vertices. For the Coin Flip procedure the cuts obtained using this smaller time bound resulted in cuts that are about 10% worse.

The results of applying local search methods are given in Table 3.4, which gives the average ratio to the best upper bound and its standard deviation, and Figures 3.7–3.10, in which the minimum, maximum, and average ratio are given. We applied multi-start local search to the Greedy and Coin Flip starting solutions. The number of starting solutions is chosen such that the time spent on each of these local search procedures is the same as the time spent on the GW heuristic plus that of the local search method. As there is hardly any difference in the performance of a local search method applied to multiple greedy solutions and that of a local search procedure applied to Coin Flip solutions, we only report on the methods applied to Greedy solutions. For iterative improvement, we implemented two neighbor selecting rules: *first* and *best* improvement. The strategies performed equally well and we therefore only present the results of best improvement. For iterative improvement (II) and tabu search (TS), we see that these methods when they are applied to the GW solution perform better than when they are applied to multiple Greedy solutions. Variable-depth flip (VDF) performs better when it is applied to multiple Greedy solution than when it is applied to the GW solution. The best cuts are obtained by applying simulated annealing (SA) applied to either of the starting solutions: on average the ration is 0.973 for the unweighted instances and 0.913 for the weighted ones.

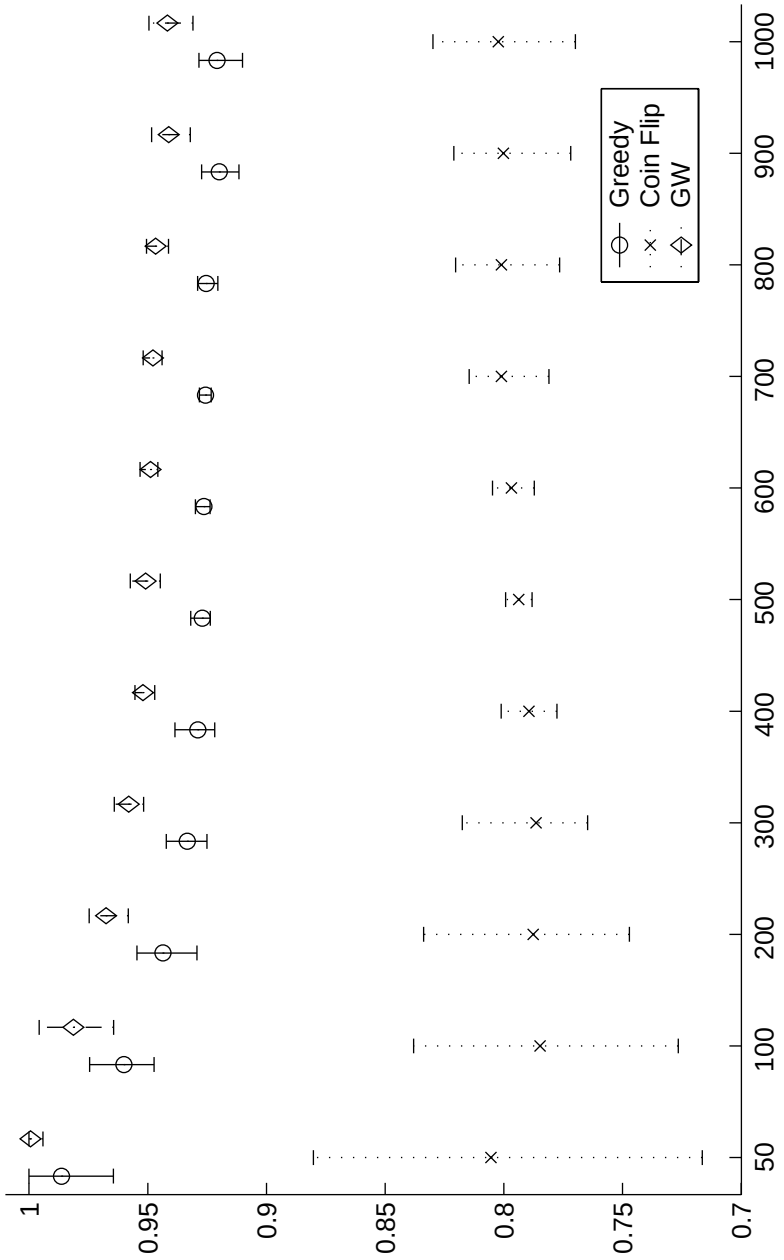


Figure 3.5: Constructive heuristics for unweighted instances: minimum, maximum, and average ratio to upper bound

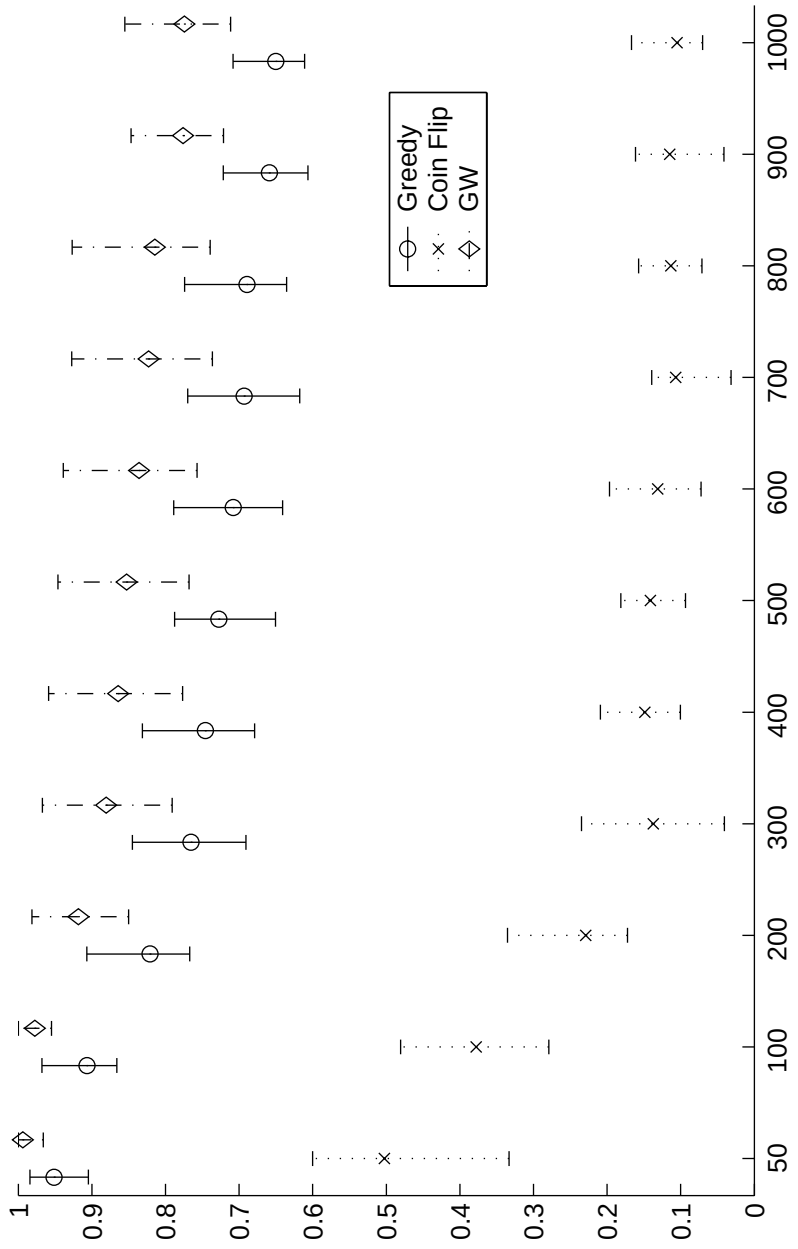


Figure 3.6: Constructive heuristics for weighted instances: minimum, maximum, and average ratio to upper bound

		II		TS		SA		VDF	
		Greedy	GW	Greedy	GW	Greedy	GW	Greedy	GW
50	unweighted	1.000 (0.000)	0.999 (0.002)	0.996 (0.008)	1.000 (0.000)	1.000 (0.000)	1.000 (0.000)	1.000 (0.000)	0.999 (0.002)
	weighted	1.000 (0.000)	0.995 (0.011)	0.997 (0.009)	1.000 (0.000)	1.000 (0.000)	1.000 (0.000)	1.000 (0.000)	0.995 (0.011)
100	unweighted	0.988 (0.012)	0.985 (0.014)	0.985 (0.012)	0.987 (0.013)	0.989 (0.011)	0.989 (0.011)	0.989 (0.011)	0.985 (0.014)
	weighted	0.983 (0.018)	0.990 (0.011)	0.999 (0.002)	0.998 (0.004)	1.000 (0.000)	1.000 (0.000)	1.000 (0.000)	0.992 (0.009)
200	unweighted	0.975 (0.006)	0.976 (0.003)	0.978 (0.004)	0.980 (0.005)	0.981 (0.006)	0.982 (0.004)	0.981 (0.005)	0.977 (0.004)
	weighted	0.920 (0.025)	0.942 (0.032)	0.955 (0.026)	0.957 (0.025)	0.963 (0.028)	0.964 (0.024)	0.965 (0.026)	0.948 (0.026)
300	unweighted	0.966 (0.004)	0.970 (0.005)	0.971 (0.005)	0.973 (0.005)	0.974 (0.005)	0.974 (0.005)	0.973 (0.005)	0.970 (0.005)
	weighted	0.879 (0.032)	0.911 (0.045)	0.923 (0.042)	0.921 (0.043)	0.930 (0.044)	0.930 (0.043)	0.929 (0.045)	0.919 (0.047)
400–800	unweighted	0.958 (0.003)	0.963 (0.004)	0.965 (0.004)	0.966 (0.004)	0.968 (0.004)	0.968 (0.004)	0.965 (0.003)	0.965 (0.004)
	weighted	0.832 (0.030)	0.877 (0.050)	0.885 (0.057)	0.890 (0.051)	0.898 (0.052)	0.898 (0.053)	0.891 (0.055)	0.887 (0.051)
900–1000	unweighted	0.949 (0.007)	0.956 (0.004)	0.956 (0.003)	0.959 (0.003)	0.960 (0.002)	0.960 (0.002)	0.957 (0.004)	0.957 (0.003)
	weighted	0.770 (0.011)	0.817 (0.033)	0.821 (0.038)	0.828 (0.033)	0.840 (0.034)	0.840 (0.034)	0.829 (0.038)	0.826 (0.035)
all	unweighted	0.965 (0.017)	0.969 (0.014)	0.970 (0.013)	0.972 (0.013)	0.973 (0.013)	0.973 (0.013)	0.971 (0.014)	0.970 (0.013)
	weighted	0.860 (0.076)	0.894 (0.069)	0.902 (0.072)	0.906 (0.067)	0.913 (0.066)	0.913 (0.066)	0.908 (0.070)	0.902 (0.067)

Table 3.4: Local search methods: average ratio to best upper bound (standard deviation)

For graphs with 400 or more vertices, we have also ran the local search procedures applied to the Coin Flip and Greedy solutions with a time limit of 450 seconds. In the case of simulated annealing this time bound is obtained by using a faster cooling schedule: the parameter δ is increased. For all local search procedures the average performance is about 0.1% worse than those reported in the table and figures.

3.6 Concluding remarks

We have investigated how well polynomial-time ρ approximation algorithms behave in practice for the max cut problem and we have also considered the quality of some upper bounds for this problem. The best upper bounds are obtained by using a semidefinite programming relaxation to which some triangle constraints are added: about 2.5% above the best lower bound for unweighted instances and 8% for the weighted graphs. The time usage for solving this relaxation is large, and the relaxation could not be solved for graphs with 900 or more vertices. If the triangle constraints are not added, then the difference between the obtained upper bound and the best lower bound is approximately doubled, whereas the time usage for the large instances is still large. The LP relaxation could be solved in reasonable time, but the quality of the bound is poor.

Of the constructive heuristics, the one with the best performance guarantee, i.e., the heuristic of Goemans and Williamson, yields the best cuts: on average 4% below the best upper bound for unweighted graphs and for the weighted graphs

this deviation was 14%. Our results coincide with the results of Goemans and Williamson (1995), who tested their algorithm on a few unweighted instances.

For iterative improvement and tabu search better solutions were obtained when they were applied to the solution obtained by GW heuristic than when they were applied to multiple Greedy or Coin Flip solutions, whereas variable-depth search performed better on multiple Greedy solutions than on the GW solution. The best results are obtained by simulated annealing regardless of the starting solution used: on average the ratio of the obtained cut to the best upper bound is 0.973 for the unweighted instances and 0.913 for the weighted ones. If we impose a time bound of 450 seconds for the local search method applied to multiple Greedy solutions, the performance of each method decreases on average by 0.1%.

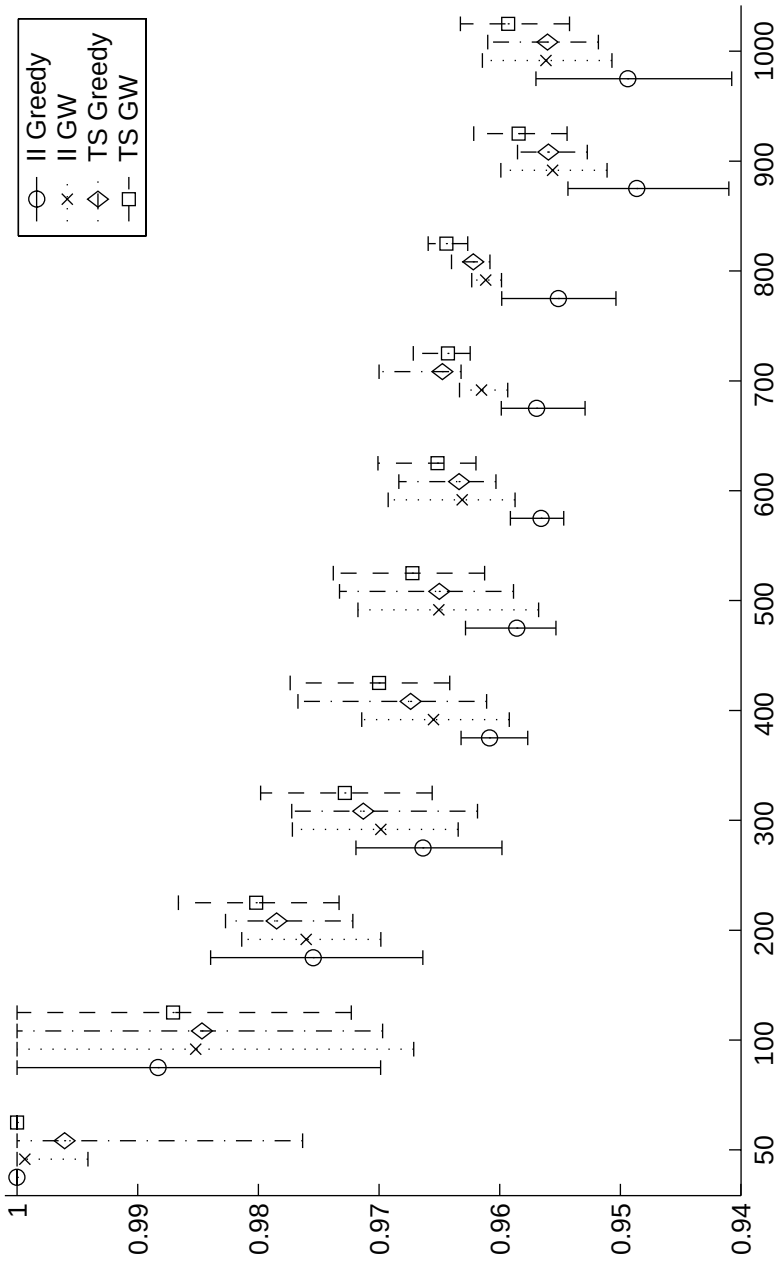


Figure 3.7: Iterative improvement and tabu search for unweighted instances: minimum, maximum, and average ratio to upper bound

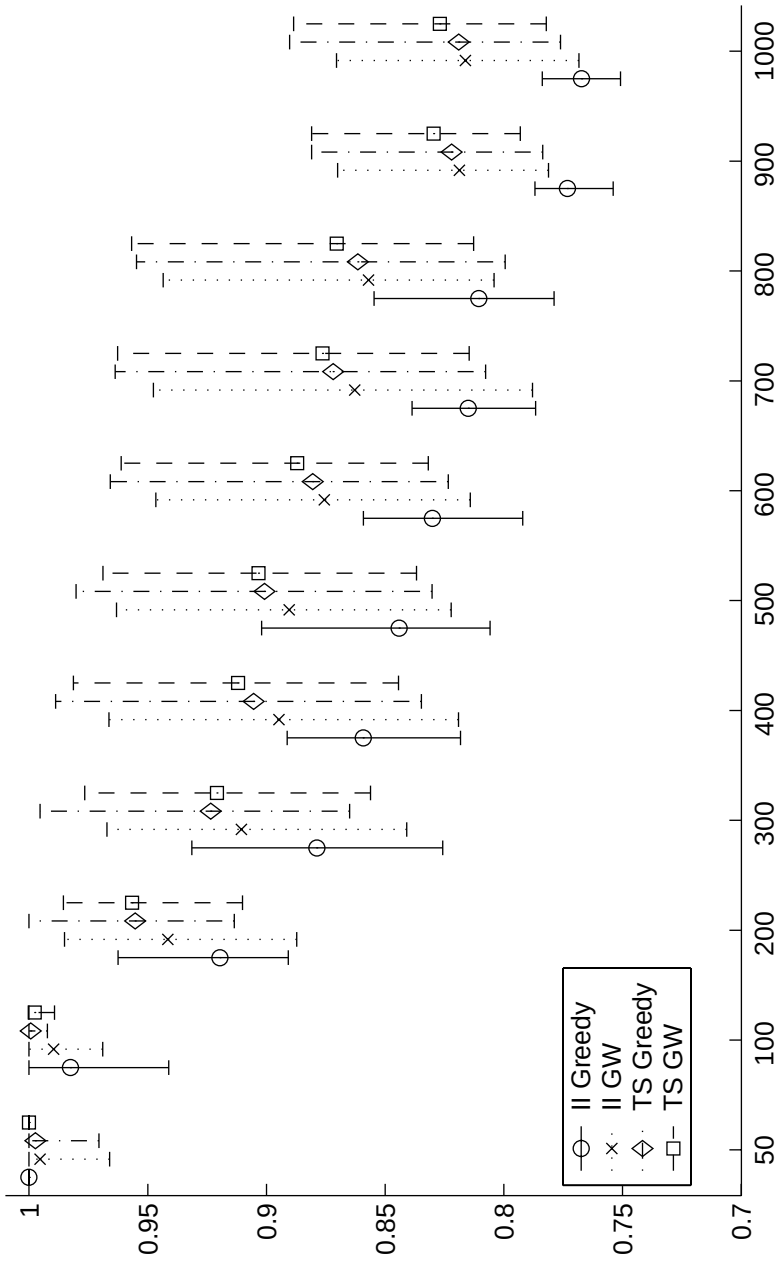


Figure 3.8: Iterative improvement and tabu search for weighted instances: minimum, maximum, and average ratio to upper bound

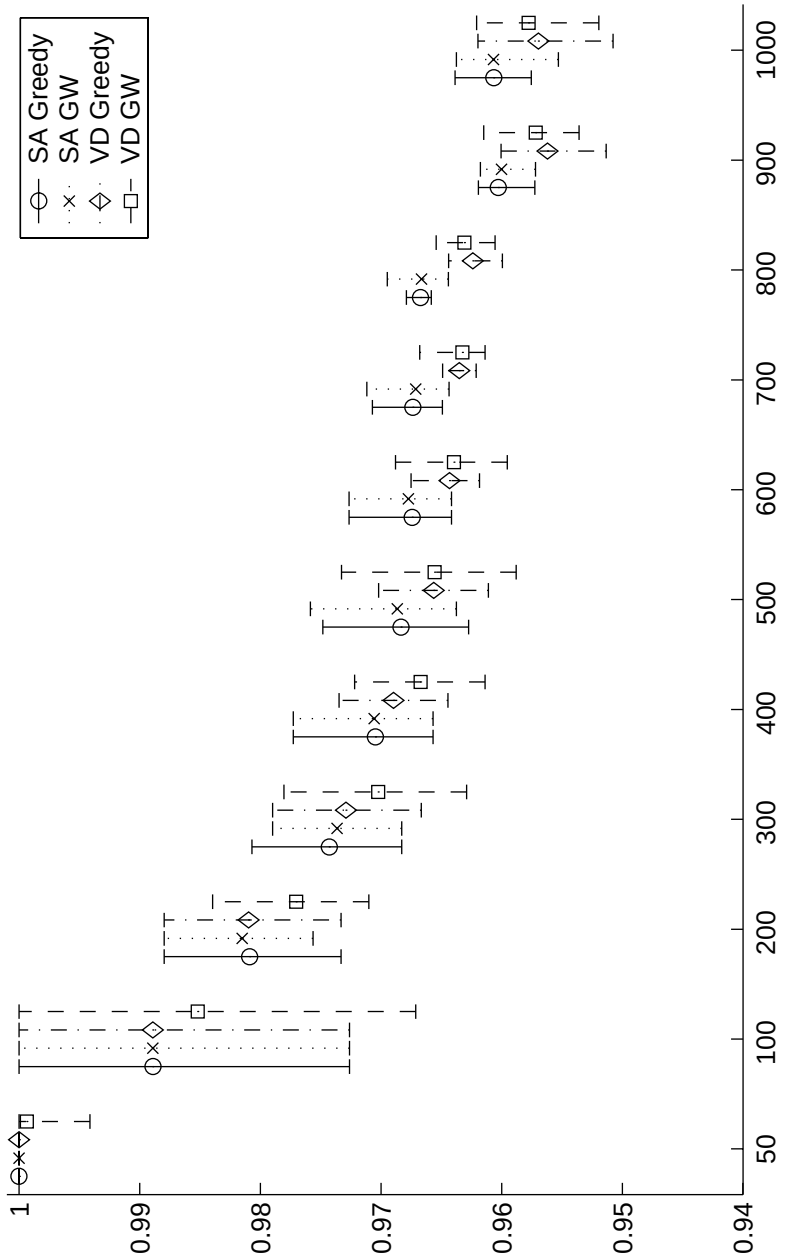


Figure 3.9: Simulated annealing and variable-depth flip for unweighted instances: minimum, maximum, and average ratio to upper bound

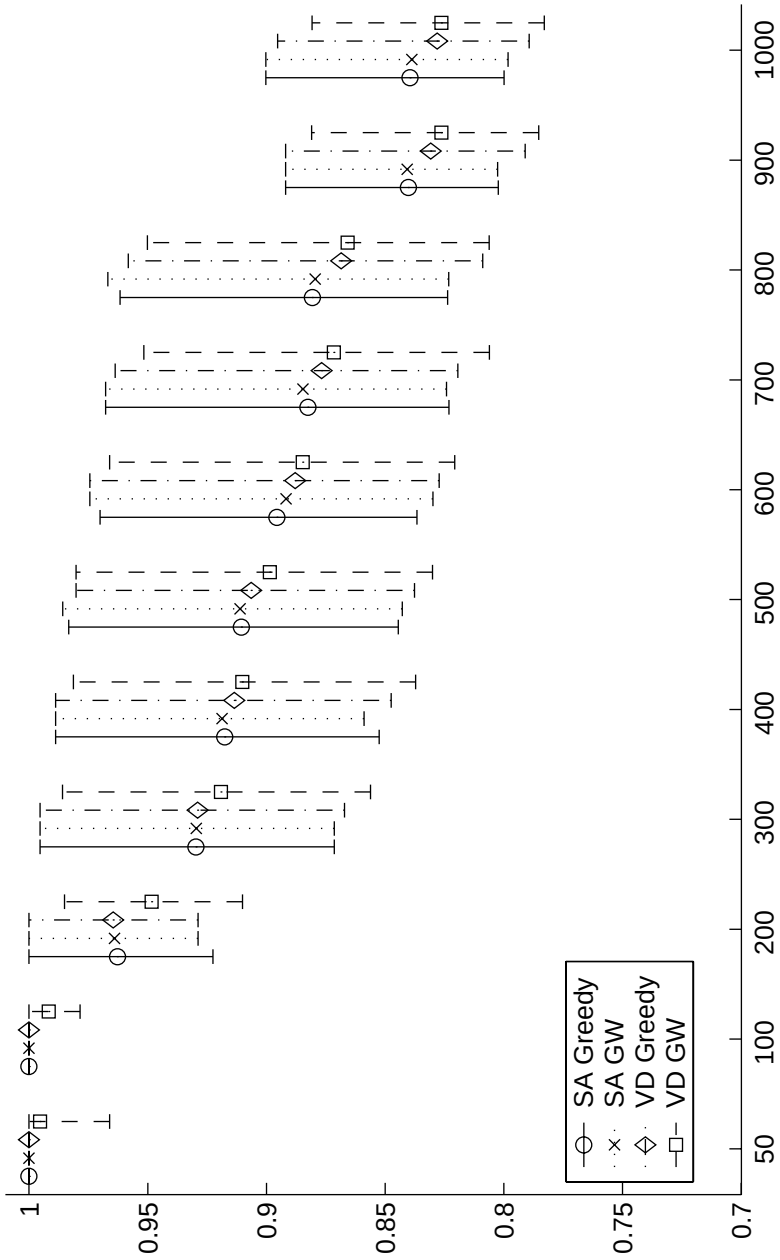


Figure 3.10: Simulated annealing and variable-depth flip for weighted instances: minimum, maximum, and average ratio to upper bound

4

Multiprocessor scheduling: guarantees for local search

4.1 Introduction

We now turn to the worst-case analysis of the quality of local optima. In this chapter, we consider multiprocessor scheduling problems. In these problems, we are given a set of n jobs, $J_1, \ldots, J_n$, each of which has to be processed without preemption on one of m machines, $M_1, \ldots, M_m$. A machine can process at most one job at a time and all jobs and machines are available at time 0. The objective we consider is *makespan minimization*, that is, we want the last job to complete as early as possible. The time, p_{ij}, it takes for a job J_j to be fully processed on a machine M_i depends on the machine environment:

- Identical parallel machines, denoted by P: a job has the same processing time on all machines, i.e., $p_{ij} = p_j$, where p_j is a positive integer.
- Uniform parallel machines, denoted by Q: the machines have positive integral speeds, $s_1, \ldots, s_m$, and each job has a given positive integral processing requirement p_j; the processing time is $p_{ij} = p_j/s_i$.
- Unrelated parallel machines, denoted by R: the time it takes to process job J_j on machine M_i is dependent on the machine as well as the job; p_{ij} is a positive integer.

In the case that the number of machines is part of the input, Graham et al. (1979) denote these problems by $P\|C_{\max}$, $Q\|C_{\max}$, and $R\|C_{\max}$. If the number of

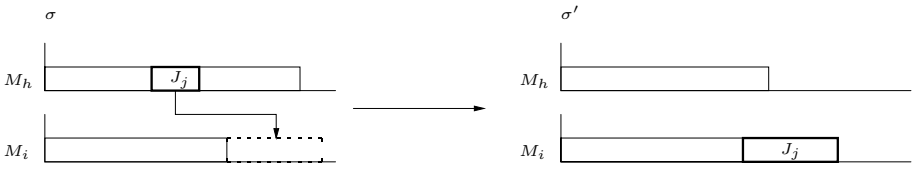


Figure 4.1: jump

machines is a constant m, then they denote the problems by $Pm\|C_{\max}$, etc.

Even the simplest case, $P2\|C_{\max}$, is NP-hard; see Garey and Johnson (1979). Therefore, it is unlikely that there exist polynomial-time algorithms that solve these problems to optimality. Hence, we search for approximate solutions. For $P\|C_{\max}$ and $Q\|C_{\max}$, Hochbaum and Shmoys (1987, 1988) develop polynomial-time approximation schemes. For $R\|C_{\max}$, Lenstra, Shmoys, and Tardos (1990) present a polynomial-time 2-approximation algorithm; they also prove that there does not exists a polynomial-time $(\frac{3}{2} - \epsilon)$-algorithm for any $\epsilon > 0$, unless P = NP. For an overview of previous work on the worst-case analysis of local search, we refer to Chapter 1.

As mentioned in Chapter 1, one way to find approximate solutions is through local search. In this chapter, we analyze the performance of local search for the jump, the swap and the newly defined push neighborhood from a worst-case perspective.

This chapter is organized as follows. In the following section we discuss the neighborhoods and in Section 4.3 we establish performance guarantees for the various local optima and scheduling problems. In Section 4.4, we make some remarks on the running time for iterative improvement to obtain the local optima, and in Section 4.5 we make some concluding remarks.

4.2 Neighborhoods

Before discussing the neighborhoods, we first describe our representation of a schedule. As the sequence in which the jobs are processed does not influence the makespan of a schedule for a given assignment of the jobs to the machines, we represent a schedule by such an assignment. This is equivalent to a partitioning of the set of jobs into m disjoint subsets $\mathcal{J}_1, \ldots, \mathcal{J}_m$, where $\mathcal{J}_i$ is the set of jobs scheduled on M_i. The *load* of a machine is the total processing time of its jobs. A *critical machine* is a machine with maximum load.

The first neighborhood that we consider is the *jump* neighborhood, also known as the *move* neighborhood: we select a job J_j and a machine M_i on which job J_j is not scheduled. The neighbor is obtained by moving J_j to M_i, as shown in Figure 4.1. We say that we are in a *jump optimal solution*, if no jump decreases the makespan or the number of critical machines without increasing the makespan.

In the *swap* neighborhood, we select two jobs, J_j and J_k, scheduled on different machines. The neighbor is formed by interchanging the machine allocations of the jobs (see Figure 4.2). If all jobs are scheduled on the same machine, then the swap neighborhood is empty; therefore, we define the swap neighborhood as one that

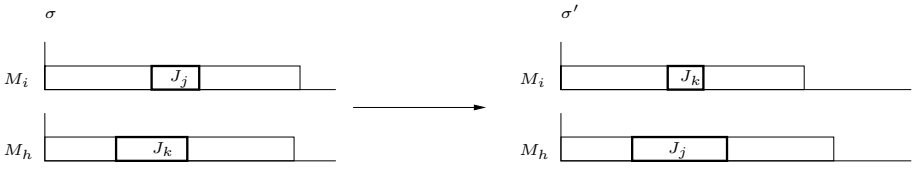


Figure 4.2: swap

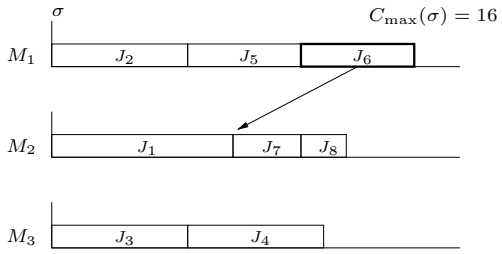
consists of all possible jumps and all possible swaps. A *swap optimal solution* is a solution in which no swap decreases the makespan or the number of critical machines without increasing the makespan.

As we will show in the next section, the jump and swap neighborhoods have no constant performance guarantee for $Q\|C_{\max}$. Therefore, we introduce a *push* neighborhood, for which any local optimum is at most a factor $2 - \frac{2}{m+1}$ of optimal for $Q\|C_{\max}$. The push neighborhood is a form of variable-depth search, introduced by Kernighan and Lin (1970) for graph partitioning; see also Lin and Kernighan (1973). A push is a sequence of jumps.

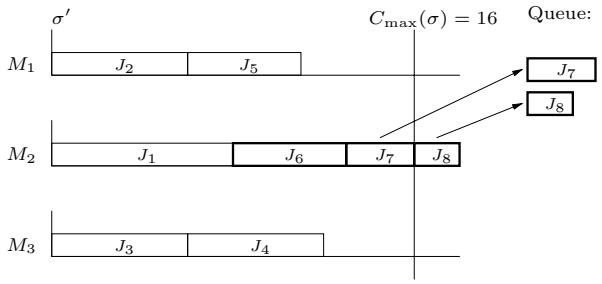
Starting with a schedule $\sigma = (\mathcal{J}_1, \ldots, \mathcal{J}_m)$ having makespan $C_{\max}(\sigma)$, a push is initiated by selecting a job J_k on a critical machine and a machine M_i to move it to. We say that J_k fits on M_i if $\sum_{J_j \in \mathcal{J}_i : p_{ij} \geq p_{ik}} p_{ij} + p_{ik} < C_{\max}(\sigma)$. If J_k does not fit on any machine, then it cannot be pushed. If, after moving J_k to M_i, the load of this machine is at least as large as the original makespan, that is, if $\sum_{J_j \in \mathcal{J}_i} p_{ij} + p_{ik} \geq C_{\max}(\sigma)$, then we iteratively remove the smallest job from M_i until the load of M_i is less than $C_{\max}(\sigma)$. The removed jobs are gathered in a queue. We now have a queue of pending jobs and a partial schedule that has lower makespan or fewer critical machines. If the queue is non-empty, then the largest job in the queue is removed and moved to some machine on which it fits, in the same way as the first job was pushed. Thus, if necessary, we allow some smaller jobs to be removed. If the largest job in the queue does not fit on any machine, then we say that the push is unsuccessful. We repeat the procedure of moving the largest job in the queue to a machine until the queue is empty or we have determined that the push is unsuccessful. When pushing all jobs on the critical machines is unsuccessful, we are in a *push optimal solution*.

We illustrate a push in the following example.

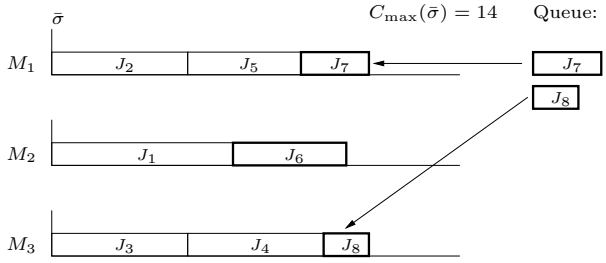
Example 4.1. Consider an instance for $P3\|C_{\max}$, with $n = 8$ jobs. The processing times are $p_1 = 8$, $p_2 = p_3 = p_4 = 6$, $p_5 = p_6 = 5$, $p_7 = 3$ and $p_8 = 2$. The starting schedule is $\sigma = (\mathcal{J}_1, \mathcal{J}_2, \mathcal{J}_3)$, with $\mathcal{J}_1 = \{J_2, J_5, J_6\}$, $\mathcal{J}_2 = \{J_1, J_7, J_8\}$, and $\mathcal{J}_3 = \{J_3, J_4\}$, and has makespan $C_{\max}(\sigma) = 16$. This schedule is depicted in Figure 4.3a. In the first step of the push, we select job J_6, with $p_6 = 5$, to be pushed onto machine M_2. When moving J_6 to M_2, jobs J_8 and J_7 have to be removed from M_2 (Figure 4.3b). At this point, we have a partial schedule $\sigma' = (\mathcal{J}'_1, \mathcal{J}'_2, \mathcal{J}'_3)$, $\mathcal{J}'_1 = \{J_2, J_5\}$, $\mathcal{J}'_2 = \{J_1, J_6\}$, $\mathcal{J}'_3 = \{J_3, J_4\}$ and a queue of pending jobs containing J_7 and J_8. In the next step, we remove J_7 from the queue and move it to M_1 and then



(a)



(b)



(c)

Figure 4.3: push

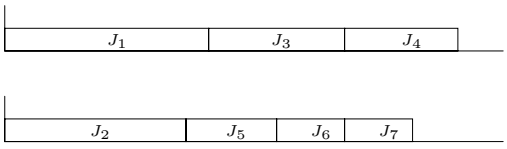


Figure 4.4: push optimal schedule

we move J_8 to M_3. After moving J_8, the queue is empty and we have a new schedule $\bar{\sigma} = (\bar{\mathcal{J}}_1, \bar{\mathcal{J}}_2, \bar{\mathcal{J}}_3)$, with $\bar{\mathcal{J}}_1 = \{J_2, J_5, J_7\}$, $\bar{\mathcal{J}}_2 = \{J_1, J_6\}$, and $\bar{\mathcal{J}}_3 = \{J_3, J_4, J_8\}$, which has makespan $C_{\max}(\bar{\sigma}) = 14$ (Figure 4.3c).

A push optimal solution is given in Example 4.2. As the schedule in this example is not swap optimal, it shows that a push optimal solution is not necessarily swap optimal. Of course, as a push is a sequence of one or more jumps, a push optimal solution is jump optimal.

Example 4.2. In Figure 4.4, we give an example for $P2\|C_{\max}$. There are $n = 7$ jobs, $J_1, \ldots, J_7$, and the processing times are $p_1 = 9$, $p_2 = 8$, $p_3 = 6$, $p_4 = 5$, $p_5 = 4$, and $p_6 = p_7 = 3$. The schedule $\sigma = (\mathcal{J}_1, \mathcal{J}_2)$, with $\mathcal{J}_1 = \{J_1, J_3, J_4\}$ and $\mathcal{J}_2 = \{J_2, J_5, J_6, J_7\}$, is a push optimal schedule. When trying to push J_1, jobs J_7, J_6, and J_5 are moved to the queue of pending jobs. Then jobs J_5 and J_6 are moved to machine M_1, resulting in a partial schedule with a load of 18 for M_1 and of 17 for M_2. The queue of pending jobs consists of job J_7 of length $p_7 = 3$. As $18 + 3 \geq 20 = C_{\max}(\sigma)$ and $17 + 3 \geq 20 = C_{\max}(\sigma)$, J_7 does not fit on M_1 as well as on M_2. Hence, pushing J_1 is unsuccessful. In the same manner, we see that pushing J_3 or J_4 is also unsuccessful. The schedule σ can be improved by swapping e.g. J_1 and J_2, and thus it is not swap optimal.

By our way of defining a push, we know that when moving a job J_k, only smaller jobs than J_k can be removed from the machine. Hence, during one push, at most n jobs need to be moved. As one move can straightforwardly be implemented such that it needs $\mathcal{O}(n)$ time, a push requires $\mathcal{O}(n^2)$ elementary operations. If we use appropriate data structures, like binary heaps for the queue of pending jobs and for the list of machines and doubly linked lists for the jobs, and if we select the machine to move a job to in a greedy manner, a push neighbor can be found in $\mathcal{O}(n \log n)$ time.

Note that, as we always take the largest job from the queue, the push neighborhood is only defined for scheduling problems where the largest job is defined unambiguously. Therefore, in the case of unrelated parallel machines, a push is not well defined.

4.3 Performance guarantees

In this section, we establish performance guarantees for the various local optima and scheduling problems. These are given in Table 4.1. "UB = ρ" denotes that ρ

	jump	swap	push
$P2\|\|C_{\max}$	$\frac{4}{3}$	$\frac{4}{3}$	$\frac{8}{7}$
$P\|\|C_{\max}$	$2-\frac{2}{m+1}\dagger$	$2-\frac{2}{m+1}\dagger$	UB $= 2-\frac{2}{m+1}$ LB $= \frac{4m}{3m+1}$
$Q2\|\|C_{\max}$	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$	$\frac{\sqrt{17}+1}{4}$
$Q\|\|C_{\max}$	$\frac{1+\sqrt{4m-3}}{2}\ddagger$	$\frac{1+\sqrt{4m-3}}{2}\ddagger$	UB $= 2-\frac{2}{m+1}$ LB $= \frac{3}{2}-\epsilon$
$R2\|\|C_{\max}$	LB $= \frac{p_{\max}}{C^*_{\max}}$	LB $= n-1$	undefined
$R\|\|C_{\max}$	LB $= \frac{p_{\max}}{C^*_{\max}}$	LB $= \frac{p_{\max}-1}{C^*_{\max}}$	undefined

Table 4.1: performance guarantees: $C^{LS}_{\max}/C^*_{\max}$; † UB due to Finn and Horowitz (1979); ‡ UB due to Cho and Sahni (1980)

is a performance guarantee and "LB $= \rho$" denotes that the performance guarantee cannot be less than ρ; "ρ" denotes that UB $= \rho$ and LB $= \rho$. For the unrelated parallel machines cases, we use $p_{\max}$, which is defined by $p_{\max} = \max_{i,j} p_{ij}$.

In the following subsection we prove the performance guarantees for the identical parallel machines cases, and in the subsequent two subsections we consider the uniform and unrelated parallel machines cases, respectively. In the following, we denote the value of an optimal schedule by $C^*_{\max}$ and $C^{\mathrm{J}}_{\max}$, $C^{\mathrm{S}}_{\max}$, and $C^{\mathrm{P}}_{\max}$ denote the makespan of respectively a jump optimal, a swap optimal, and a push optimal schedule.

4.3.1 Identical parallel machines

Recall that in the identical parallel machine environment, all machines need the same amount of time to process a job. Hence the processing time of a job J_j on a machine M_i is $p_{ij} = p_j$. Theorem 4.1 is due to Finn and Horowitz (1979); for completeness we include the proof.

Theorem 4.1. *A jump optimal schedule for $P\|C_{\max}$ has makespan at most $2-\frac{2}{m+1}$ times the optimal solution value.*

Proof. Consider a jump optimal schedule with value $C^{\mathrm{J}}_{\max}$ and assume w.l.o.g. that machine M_1 is a critical machine. If M_1 processes only one job, say J_k, then $C^{\mathrm{J}}_{\max} = p_k$, and as the optimal makespan is at least as large as the size of any job, we know that $C^{\mathrm{J}}_{\max} = C^*_{\max}$.

If M_1 processes at least two jobs, then the smallest job on this machine, say job J_k, has processing time $p_k \leq \frac{1}{2}C^{\mathrm{J}}_{\max}$. Jump optimality implies for all machines M_i

that

$$\sum_{J_j \in \mathcal{J}_i} p_j + p_k \geq C^{\mathrm{J}}_{\max}.$$

Summing this inequality over all machines M_i for $i > 1$ and adding $C^{\mathrm{J}}_{\max} = \sum_{J_j \in \mathcal{J}_1} p_j$ yields

$$mC^{\mathrm{J}}_{\max} \leq \sum_i \sum_{J_j \in \mathcal{J}_i} p_j + (m-1)p_k = \sum_j p_j + (m-1)p_k. \tag{4.1}$$

As $C^*_{\max} \geq \frac{1}{m} \sum_j p_j$, inequality (4.1) implies

$$C^{\mathrm{J}}_{\max} \leq \frac{1}{m} \sum_j p_j + \frac{m-1}{m} p_k \leq C^*_{\max} + \frac{m-1}{2m} C^{\mathrm{J}}_{\max},$$

where the last inequality is due to $p_k \leq \frac{1}{2} C^{\mathrm{J}}_{\max}$.

Rearranging terms yields the performance guarantee:

$$C^{\mathrm{J}}_{\max} \leq \frac{2m}{m+1} C^*_{\max} = (2 - \frac{2}{m+1}) C^*_{\max}.$$

□

Corollary 4.2. *A swap optimal schedule for $P\|C_{\max}$ has makespan at most $2 - \frac{2}{m+1}$ times the optimal makespan.*

The bounds given in Theorem 4.1 and Corollary 4.2 are tight, as can be seen in the following example.

Example 4.3. For given $K > 1$, consider the instance in which there are m machines and the number of jobs is $n = 2 + (m-1)(Km+1)$. The processing times of the jobs are $p_1 = p_2 = Km$ and $p_j = 1$ $(j = 3, \ldots, n)$. The optimal makespan is obtained by a schedule in which jobs J_1 and J_2 are processed on different machines and the other jobs are divided over the machines such that the loads of any two machines differ by at most 1. The optimal makespan is $C^*_{\max} = K(m+1) + 1$.

In Figure 4.5, a swap optimal schedule is given: jobs J_1 and J_2 are both processed on machine M_1 and the other jobs are equally divided over the machines $M_2, \ldots, M_m$. It is easy to see that this schedule is jump optimal and swap optimal, as swapping job J_1 or J_2 with any of the unit length jobs does not decrease the makespan and a jump of J_1 or J_2 increases the makespan. The value of this schedule is $C^{\mathrm{S}}_{\max} = 2Km = (\frac{2m}{m+1+1/K}) C^*_{\max}$ and for large values of K, the ratio $C^{\mathrm{S}}_{\max}$ / $C^*_{\max}$ is close to $2 - \frac{2}{m+1}$. For a jump optimal schedule, we can even remove one job from each of the machines $M_2, \ldots, M_m$. The value of the optimum is then $C^*_{\max} = K(m+1)$ and the jump optimal schedule remains $C^{\mathrm{J}}_{\max} = 2Km = (2 - \frac{2}{m+1}) C^*_{\max}$.

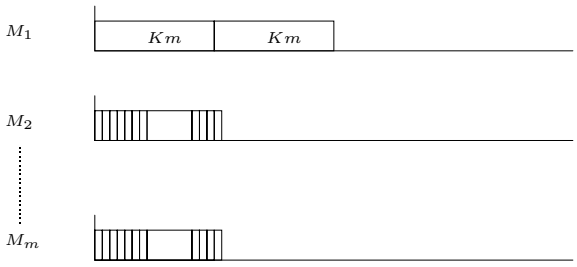


Figure 4.5: swap optimal schedule

This example also holds for $m = 2$, and thus we have the following corollary to Theorem 4.1.

Corollary 4.3. *The performance guarantee for a jump optimal schedule and a swap optimal schedule for* $P2\|C_{\max}$ *is* $\frac{4}{3}$*, and this bound is tight.*

As a push optimal schedule is also jump optimal, the following is also a corollary to Theorem 4.1.

Corollary 4.4. *A push optimal schedule for* $P\|C_{\max}$ *has value at most* $2 - \frac{2}{m+1}$ *times the optimal solution value.*

In contrast to the jump and swap optimal solutions, we have no tightness guarantees. The following example shows that the performance guarantee for push optimal solutions for $P\|C_{\max}$ cannot be less than $\frac{4m}{3m+1}$.

Example 4.4. Consider the following instance, with m machine and $n = 2m+2$ jobs. The processing times of jobs $J_1, \ldots, J_{n-4}$ are $p_j = m + \lceil \frac{j}{2} \rceil$ for $j = 1, \ldots, n-4$, i.e., there are exactly two jobs of size p for $p = m+1, \ldots, 2m-1$. The processing times of the last four jobs are $p_{n-3} = p_{n-2} = p_{n-1} = p_n = m$. The optimal makespan is $C^*_{\max} = 3m + 1$ and is attained by $\sigma^* = (\mathcal{J}^*_1, \ldots, \mathcal{J}^*_m)$, with $\mathcal{J}^*_1 = \{J_1, J_{n-3}, J_{n-2}\}$, $\mathcal{J}^*_2 = \{J_2, J_{n-1}, J_n\}$, and $\mathcal{J}^*_i = \{J_i, J_{n-1-i}\}$ for $i = 3, \ldots, m$.

In the schedule in Figure 4.6 the jobs of size $p_j = m$ are assigned to M_1 and M_i processes J_{i-1} and J_{n-i-2}, for $i = 2, \ldots, m$. This is a push optimal schedule with makespan $C^{\mathrm{P}}_{\max} = 4m = \frac{4m}{3m+1} C^*_{\max}$.

The performance guarantee for push optimal solutions for $P2\|C_{\max}$ is better than the $\frac{4}{3}$ obtained by jump and swap optimal schedules, as stated by the following theorem.

Theorem 4.5. *A push optimal solution for* $P2\|C_{\max}$ *has value at most* $8/7$ *times the optimal solution value.*

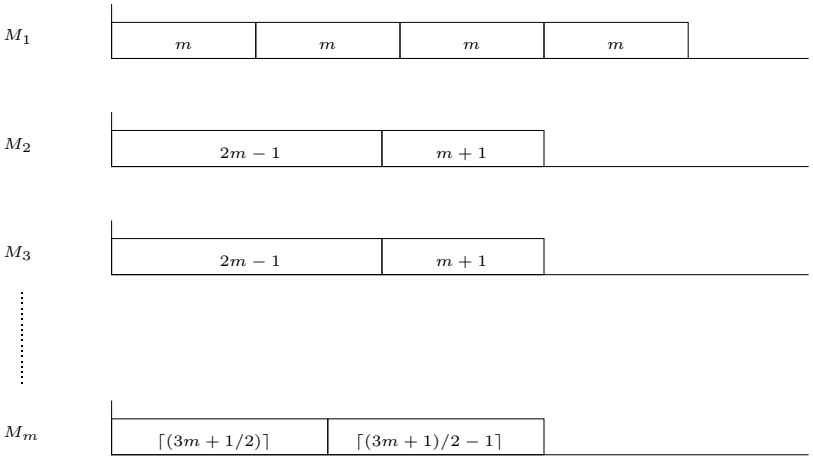


Figure 4.6: push optimal schedule

PROOF. Suppose, to the contrary, that there exists a push optimal schedule with makespan $C^{\mathrm{P}}_{\max} > \frac{8}{7}C^*_{\max}$. Let $L_i = \sum_{J_j \in \mathcal{J}_i} p_j$ be the load of machine M_i $(i = 1, 2)$. W.l.o.g. we assume that $L_1 \geq L_2$, thus $C^{\mathrm{P}}_{\max} = L_1$.

For the difference in loads of the two machines, we know that

$$L_1 - L_2 = L_1 - (\sum_j p_j - L_1) \geq 2L_1 - 2C^*_{\max} > \frac{1}{4}L_1.$$

The first inequality is due to the lower bound $C^*_{\max} \geq \frac{1}{2}\sum_j p_j$ and the second inequality is due to the assumption that $L_1 > \frac{8}{7}C^*_{\max}$. Let J_1 be the smallest job on M_1. Then, by push optimality, we know that $p_1 \geq L_1 - L_2 > \frac{1}{4}L_1$. Hence, there are at most three jobs on M_1. We assume that M_1 processes three jobs, i.e., $\mathcal{J}_1 = \{J_1, J_2, J_3\}$, $\mathcal{J}_2 = \{J_4, \ldots, J_n\}$, and that $p_1 \leq p_2 \leq p_3$ and $p_4 \geq \ldots \geq p_n$. Let J_k be the job such that the smaller of J_k and J_1 is the largest job that has to be removed, when pushing J_1 to M_2, i.e.,

$$\begin{aligned} p_4 + \ldots + p_{k-1} + p_1 &< L_1, \\ p_4 + \ldots + p_k + p_1 &\geq L_1. \end{aligned} \tag{4.2}$$

As $L_1 - L_2 > \frac{1}{4}L_1 \geq \frac{3}{4}p_1$, we know that $L_2 + p_1 < L_1 + \frac{1}{4}p_1$ and

$$\frac{1}{4}p_1 > L_2 + p_1 - L_1 \overset{(4.2)}{\geq} L_2 - (p_4 + \ldots + p_k) = p_{k+1} + \ldots + p_n. \tag{4.3}$$

Because of push optimality, we know that $p_1 \leq p_k + \ldots + p_n$ and thus

$$p_k \geq p_1 - (p_{k+1} + \ldots + p_n) \overset{(4.3)}{>} \frac{3}{4}p_1. \tag{4.4}$$

If $p_k < p_1$, then by pushing J_1 to M_2, J_k is moved to M_1 and jobs $J_{k+1}, \ldots, J_n$ are distributed among M_1 and M_2 yielding a schedule with makespan

$$C'_{\max} \leq \max(L_1 - p_1 + p_k,\ L_2 + p_1 - p_k) \overset{(4.4)}{<} \max(L_1, L_1 - \frac{1}{2}p_1) = L_1.$$

Thus the schedule is not push optimal and it must be the case that $p_1 \leq p_k$.

Suppose M_2 processes at least three jobs at least as large as J_k, then $L_2 \geq 3p_1$. By push optimality, we know that $p_1 \geq L_1 - L_2 > \frac{1}{4}L_1$ and thus $L_2 \geq 3p_1 > \frac{3}{4}L_1$. However, as $L_1 - L_2 > \frac{1}{4}L_1$, it must be that $L_2 < \frac{3}{4}L_1$. Therefore, M_2 can process at most two jobs of size greater than or equal to p_1.

If M_2 processes only one job of size at least p_k, i.e., $k = 4$, then the current schedule is optimal, as the sub-schedule for J_1, J_2, J_3, J_4 is optimal, because by (4.2) $p_1 + p_4 \geq p_1 + p_2 + p_3$. In the case that M_2 processes two jobs of size at least p_k, i.e., $p_4 \geq p_5 = p_k$, we consider two sub-cases: $p_3 \leq p_5$ and $p_3 > p_5$. If $p_3 \leq p_5$, then the sub-schedule for $J_1, \ldots, J_5$ is optimal and $C^*_{\max} \geq C^*_{\max}[1,5] = L_1 = C^{\mathrm{P}}_{\max}$, where $C^*_{\max}[1,5]$ denotes the makespan of an optimal sub-schedule for $J_1, \ldots, J_5$.

If $p_3 > p_5$, then an optimal schedule for $J_1, \ldots, J_5$ has value $C^*_{\max}[1,5] \geq p_1 + p_2 + p_5$. As $L_2 < 3p_1$, we know that $p_4 < 2p_1$ and by push optimality we know that $p_5 + \sum_{j \geq 6} p_j \geq p_3$. Hence,

$$\frac{C^{\mathrm{P}}_{\max}}{C^*_{\max}} \leq \frac{p_1 + p_2 + p_3}{p_1 + p_2 + p_5} \leq \frac{p_1 + p_2 + p_5 + \ldots + p_n}{p_1 + p_2 + p_5} \overset{(4.3)}{\leq} 1 + \frac{1/4p_1}{3p_1} = \frac{13}{12} < \frac{8}{7}.$$

This contradicts the assumption that $C^{\mathrm{P}}_{\max} > \frac{8}{7}C^*_{\max}$.

In the case that M_1 processes only two jobs, i.e., $\mathcal{J}_1 = \{J_1, J_2\}$ and $\mathcal{J}_2 = \{J_3, \ldots, J_n\}$, we can prove in a similar way that $C^*_{\max} = C^{\mathrm{P}}_{\max}$. If M_1 only processes J_1, then $C^{\mathrm{P}}_{\max} = C^*_{\max}$.

Hence, $C^{\mathrm{P}}_{\max} \leq \frac{8}{7}C^*_{\max}$. □

The bound in Theorem 4.5 is tight: if we set $m = 2$ in Example 4.4, we have a push optimal schedule with makespan $C^{\mathrm{P}}_{\max} = \frac{8}{7}\, C^*_{\max}$.

4.3.2 Uniform parallel machines

In this environment, jobs have processing requirements p_j for $j = 1, \ldots, n$ and machines have speeds s_i for $i = 1, \ldots, m$. The processing time of job J_j on machine M_i is $p_{ij} = p_j / s_i$.

The following theorem is a restatement of Lemma 1 of Cho and Sahni (1980), who prove that that a performance guarantee for list schedules in the uniform parallel machine environment is $\frac{1+\sqrt{4m-3}}{2}$. For completeness, we include the proof.

Theorem 4.6. *A jump optimal schedule for $Q\|C_{\max}$ has makespan at most $\frac{1+\sqrt{4m-3}}{2}$ times the optimal makespan.*

PROOF. We assume w.l.o.g. that $p_1 \geq p_2 \geq \ldots \geq p_n$ and $s_1 \geq s_2 \geq \ldots \geq s_m$. To prove the theorem, we use two lower bounds on the optimal makespan:

$$C^*_{\max} \geq \frac{\sum_j p_j}{\sum_i s_i}, \tag{4.5}$$

$$C^*_{\max} \geq \frac{p_1}{s_1}. \tag{4.6}$$

Let job J_k be a job on a critical machine in a jump optimal schedule. Then we know that for each machine M_i,

$$\sum_{J_j \in \mathcal{J}_i} \frac{p_j}{s_i} + \frac{p_k}{s_i} \geq C^{\mathrm{J}}_{\max}. \tag{4.7}$$

Multiplying this inequality with s_i and summing over all machines yields

$$\sum_i s_i C^{\mathrm{J}}_{\max} \leq \sum_j p_j + (m-1)p_k,$$

or equivalently,

$$C^{\mathrm{J}}_{\max} \leq \frac{\sum_j p_j}{\sum_i s_i} + (m-1)\frac{p_k}{\sum_i s_i} \leq C^*_{\max} + (m-1)\frac{s_1}{\sum_i s_i} C^*_{\max}.$$

The last inequality is due to inequalities (4.5) and (4.6). On the other hand, using inequality (4.7) for machine M_1, yields

$$C^{\mathrm{J}}_{\max} \leq \frac{\sum_j p_j}{s_1} \overset{(4.5)}{\leq} \frac{\sum_i s_i}{s_1} C^*_{\max},$$

Combining both bounds on $C^{\mathrm{J}}_{\max}$ and replacing $\frac{\sum_i s_i}{s_1}$ by x, we have

$$C^{\mathrm{J}}_{\max} \leq \min(x, 1 + (m-1)/x) C^*_{\max}. \tag{4.8}$$

The maximum of the right hand side of inequality (4.8) is obtained when $x = 1 + (m-1)/x$, i.e., for $x = \frac{1+\sqrt{4m-3}}{2}$. Hence, $C^{\mathrm{J}}_{\max} \leq \frac{1+\sqrt{4m-3}}{2} C^*_{\max}$. □

Corollary 4.7. *A swap optimal schedule for $Q\|C_{\max}$ has makespan at most $\frac{1+\sqrt{4m-3}}{2}$ times the optimum.*

In Example 4.5, we give a jump and swap optimal schedule for $Q\|C_{\max}$ with $C^{\mathrm{S}}_{\max} = \frac{1+\sqrt{4m-3}}{2} C^*_{\max}$ and thus the bounds in the above theorem and corollary are tight.

Example 4.5. For given $s > 1$, consider the following instance. There are $n = m+1$ jobs and $m = s^2 - s + 1$ machines, that is, $s = \frac{1+\sqrt{4m-3}}{2}$. Job J_1

has processing requirement $p_1 = s$ and all other jobs have processing requirement $p_j = 1$ $(j = 2, \ldots, n)$. Machine M_1 has speed $s_1 = s$ and all other machines have speed $s_i = 1$ $(i = 2, \ldots, m)$. In an optimal schedule, machine M_1 processes job J_1 and one job of unit length and each of machines $M_2, \ldots, M_m$ processes exactly one job of unit length. The optimal makespan is $C^*_{\max} = 1 + \frac{1}{s}$.

In Figure 4.7 a swap optimal schedule is given. The second machine processes job J_1 and machine M_1 processes all other jobs. The makespan of this swap optimal schedule is $C^{\mathrm{S}}_{\max} = s = \frac{s}{1+1/s} C^*_{\max}$ and for large s the ratio $C^{\mathrm{S}}_{\max} / C^*_{\max}$ is close to $s = \frac{1+\sqrt{4m-3}}{2}$. For a jump optimal schedule, we can even remove one of the unit sized jobs, so that the optimum has value $C^*_{\max} = 1$ and the value of the local optimal solution remains $C^{\mathrm{J}}_{\max} = s = \frac{1+\sqrt{4m-3}}{2} C^*_{\max}$.

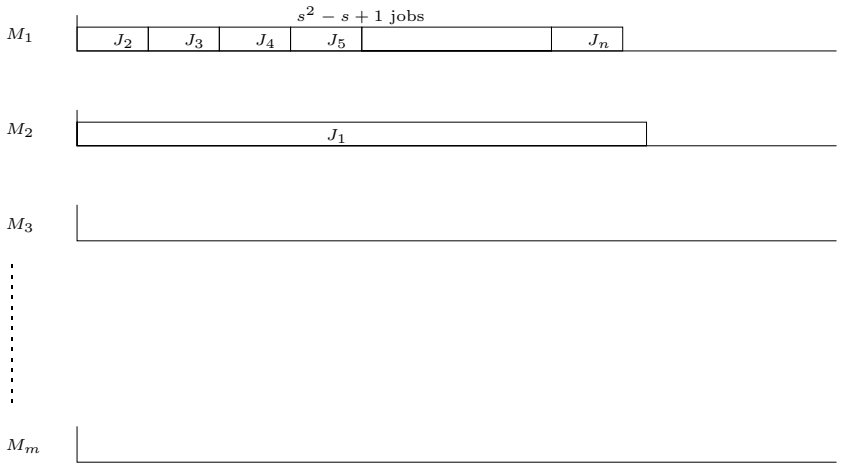


Figure 4.7: Swap optimal schedule for $Q\|C_{\max}$.

Corollary 4.8. *The performance guarantee of jump and swap optimal schedules for $Q2\|C_{\max}$ is $\frac{1+\sqrt{5}}{2}$ and this is tight.*

Proof. The performance guarantee is a direct consequence of Theorem 4.6. To see that the bound is tight, consider the instance with two jobs with processing requirements $p_1 = \frac{1+\sqrt{5}}{2}$ and $p_2 = 1$ and with speeds $s_1 = \frac{1+\sqrt{5}}{2}$ and $s_2 = 1$ for machine M_1 respectively M_2. Obviously, the optimal makespan is $C^*_{\max} = 1$ and the schedule in which J_1 is processed by M_2 and J_2 is scheduled on M_1 is a jump optimal schedule with makespan $C^{\mathrm{J}}_{\max} = \frac{1+\sqrt{5}}{2}$.

This schedule clearly is not swap optimal. To make a swap optimal one, we chop job J_2 of unit length into $\frac{1}{\epsilon}$ jobs with processing requirements $p_j = \epsilon$, for some

$\epsilon > 0$, and we add one job of size ϵ. The optimal makespan is $C^*_{\max} = 1 + \frac{2\epsilon}{1+\sqrt{5}}$ and the schedule in which all jobs of size ϵ are scheduled on M_1 and J_1 is processed by M_2 is a swap optimal schedule, with makespan $C^{\mathrm{S}}_{\max} = \frac{1+\sqrt{5}}{2}$. Hence, for small ϵ the ratio $C^{\mathrm{S}}_{\max}$ / $C^*_{\max}$ is close to $\frac{1+\sqrt{5}}{2}$. □

Theorem 4.9. *A push optimal schedule for $Q\|C_{\max}$ has makespan at most $2 - \frac{2}{m+1}$ times the optimal solution value.*

Proof. Assume w.l.o.g. that $s_1 \geq \ldots \geq s_m$. If we consider an unsuccessful push, then there is a partial schedule, $(\mathcal{J}'_1, \ldots, \mathcal{J}'_m)$, and a queue of pending jobs. The largest job in this queue does not fit on any machine. Let job J_k be this job and let $L'_i = \sum_{j \in \mathcal{J}'_i : p_j \geq p_k} \frac{p_j}{s_i}$ be the total processing time of the large jobs on machine M_i, i.e., at least as large as J_k. By push optimality, we know that for all $i = 1, \ldots, m$:

$$L'_i + \frac{p_k}{s_i} \geq C^{\mathrm{P}}_{\max}. \tag{4.9}$$

Let M_h be the slowest machine on which job J_k has a processing time that is not larger than the optimal makespan, that is, $h = \max\{i : \frac{p_k}{s_i} \leq C^*_{\max}\}$. Thus $C^*_{\max} \geq \frac{p_k}{s_h}$. Another lower bound on the optimal makespan is then

$$C^*_{\max} \geq \frac{\sum_{j:p_j \geq p_k} p_j}{\sum_{i=1}^{h} s_i}.$$

By push optimality (4.9), we know that

$$\sum_{i=1}^{h} s_i C^{\mathrm{P}}_{\max} \leq \sum_{i=1}^{h} s_i L'_i + h p_k \leq \sum_{j:p_j \geq p_k} p_j + (h-1)p_k. \tag{4.10}$$

If $s_1 \geq 2s_h$, then $\sum_{i=1}^{h} s_i \geq (h+1)s_h$ and rearranging the terms in (4.10) yields

$$C^{\mathrm{P}}_{\max} \leq C^*_{\max} + \frac{h-1}{h+1}\frac{p_k}{s_h} \leq C^*_{\max} + \frac{h-1}{h+1} C^*_{\max} \leq \frac{2m}{m+1} C^*_{\max}.$$

The second inequality is due to our choice of h. If $s_1 \leq 2s_h$, then $\sum_{i=1}^{h} s_i \geq \frac{h+1}{2} s_1$. We may assume that $C^*_{\max} \geq \frac{2p_k}{s_1}$, as otherwise there are at most h large jobs and the push optimal schedule is optimal. Rearranging terms in (4.10) yields

$$C^{\mathrm{P}}_{\max} \leq C^*_{\max} + \frac{h-1}{(h+1)/2}\frac{p_k}{s_1} \leq \frac{2m}{m+1} C^*_{\max}.$$

□

Theorem 4.10. *The performance guarantee of push optimal schedules for $Q\|C_{\max}$ is at least $\frac{3}{2} - \epsilon$, for $\epsilon > 0$.*

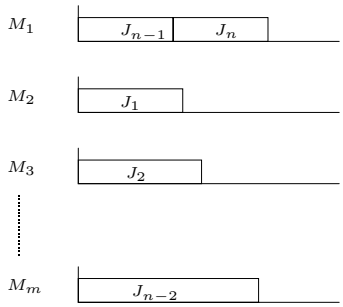


Figure 4.8: Push optimal schedule for $Q\|C_{\max}$

PROOF. Consider the following instance for $Q\|C_{\max}$. For given $r \in (\frac{2}{3}, 1)$, there are $m = \lceil \log(\frac{2-3r}{1-2r}) \rceil$ machines and $n = m + 1$ jobs. The speeds of the machines are given by $s_1 = 2$ and $s_i = rs_{i-1} + 1$, $i = 2, \ldots, m$. The processing requirements are given by $p_j = rs_j$ for $j = 1, \ldots, n-2$ and $p_{n-1} = p_n = 1$. As by our choice of m we have that $\frac{2}{s_m} \leq r$, the optimal makespan is $C^*_{\max} = r$.

The schedule in Figure 4.8 is push optimal: M_1 processes both jobs of size 1 and J_j is scheduled on M_{j+1} for $j = 1, \ldots, m-1$. This schedule has makespan $C^{\mathrm{P}}_{\max} = 1 = \frac{1}{r} C^*_{\max}$. Hence, for any $\epsilon > 0$ there exists a push optimal schedule with $C^{\mathrm{P}}_{\max} \geq (\frac{3}{2} - \epsilon) C^*_{\max}$. □

In the case of two uniform machines, we establish a better performance guarantee. To do so, we need the following two lemmata, in which we consider instances with only three jobs and different speeds for the two machines. We prove in Theorem 4.13 that a smallest worst-case instance for push for the problem of scheduling two uniform parallel machines has exactly three jobs.

Lemma 4.11. *Consider an instance for $Q2\|C_{\max}$ with three jobs in which the machines do not have the same speed. Assume w.l.o.g that $p_1 \geq p_2 \geq p_3$ and $s_1 > s_2$. If in an optimal schedule J_1 is processed on M_1, then in any push optimal schedule a job of size p_1 is scheduled on M_1 and this push optimal schedule is globally optimal.*

PROOF. Suppose to the contrary that there is a push optimal schedule in which J_1 is processed on M_2. Then J_2 is scheduled on M_1, as otherwise a push is possible.

If $p_1 = p_2$, then the first part of the lemma is proven. Consider the case that $p_1 > p_2$. If M_2 is the critical machine, then J_1 can be pushed, and the schedule is not push optimal. Therefore, M_1 is the critical machine and it processes J_2 as well as J_3. In the optimal schedule J_1 is assigned to M_1 and the optimal makespan has value $C^*_{\max} \geq \min\{\frac{p_1+p_3}{s_1}, \frac{p_2+p_3}{s_2}\}$. As $\frac{p_1+p_3}{s_1} > \frac{p_2+p_3}{s_1} = C^{\mathrm{P}}_{\max}$ and $\frac{p_2+p_3}{s_2} > \frac{p_2+p_3}{s_1} = C^{\mathrm{P}}_{\max}$, we have that $C^*_{\max} > C^{\mathrm{P}}_{\max}$, which is a contradiction. Therefore, in a push optimal schedule, J_1 must be processed by M_1, whenever J_1 is scheduled on M_1 in an optimal schedule.

By enumerating over all possible schedules with J_1 scheduled on M_1 for the push optimal schedule as well as the optimal schedule, it is easy to see that whenever such a schedule is push optimal it is globally optimal. □

Lemma 4.12. *Consider an instance for $Q2\|C_{\max}$ with three jobs in which the machines do not have equal speed and assume w.l.o.g. that $p_1 \geq p_2 \geq p_3$ and $s_1 > s_2$. If $C^{\mathrm{P}}_{\max} > C^*_{\max}$, then in the optimal schedule M_1 processes J_2 and J_3, and J_1 is scheduled on M_2. In a push optimal schedule with $C^{\mathrm{P}}_{\max} > C^*_{\max}$, the machine allocation of the jobs is reversed, that is, J_1 is scheduled on M_1, and M_2 processes J_2 and J_3. This push optimal schedule has makespan $C^{\mathrm{P}}_{\max} = \frac{p_2+p_3}{s_2}$.*

Proof. By Lemma 4.11, we know that whenever $C^{\mathrm{P}}_{\max} > C^*_{\max}$, in the optimal schedule J_1 is scheduled on M_2. As $\frac{p_1+p_2}{s_2} \geq \frac{p_1+p_3}{s_2} > \frac{p_2+p_3}{s_1}$, J_2 as well as J_3 is assigned to M_1 in the optimal schedule.

If, in a push optimal schedule, J_1 is processed by M_2, then this schedule must be globally optimal. Hence, for each push optimal schedule with $C^{\mathrm{P}}_{\max} > C^*_{\max}$, J_1 is scheduled on M_1 and the critical machine is M_2 as otherwise J_1 can be pushed. If J_2 or J_3 are also scheduled on M_1, then M_2 cannot be critical and the schedule is not push optimal. Therefore, a push optimal schedule with makespan $C^{\mathrm{P}}_{\max} > C^*_{\max}$ processes J_1 on M_1 and J_2 and J_3 on M_2, and as M_2 is the critical machine $C^{\mathrm{P}}_{\max} = \frac{p_2+p_3}{s_2}$. □

Theorem 4.13. *A push optimal schedule for $Q2\|C_{\max}$ has performance guarantee $\frac{\sqrt{17}+1}{4}$.*

Proof. Consider a push optimal schedule with $C^{\mathrm{P}}_{\max} > \frac{5}{4}C^*_{\max}$. We may assume that such a schedule exists as otherwise $C^{\mathrm{P}}_{\max}/C^*_{\max} \leq \frac{5}{4} < \frac{\sqrt{17}+1}{4}$. Pushing the smallest job on the critical machine leads to an unsuccessful push. Hence, there is a largest job in the queue of pending jobs that does not fit on both machines. Let this job be J_k. Note that this job is at most as large as the smallest job on the critical machine. Because of push optimality, we now have

$$\sum_{J_j \in \mathcal{J}_i : p_j \geq p_k} \frac{p_j}{s_i} + \frac{p_k}{s_i} \geq C^{\mathrm{P}}_{\max}, \qquad i = 1, 2.$$

Thus,

$$(s_1 + s_2)C^{\mathrm{P}}_{\max} \leq \sum_{j : p_j \geq p_k} p_j + p_k \leq (s_1 + s_2)C^*_{\max} + p_k.$$

By the assumption that $C^{\mathrm{P}}_{\max} > \frac{5}{4}C^*_{\max}$, we have that $\sum_j p_j \leq (s_1 + s_2)C^*_{\max} < 4p_k$. Hence, there are at most three large jobs, that is, at least as large as J_k.

If we remove all jobs that are smaller than J_k from the push optimal schedule, then we still have a push optimal schedule and the makespan has not changed, as all the jobs that are smaller than J_k are scheduled on the non-critical machine. As the optimal makespan of the instance with only the large jobs is at most equal to the

optimal makespan of the original instance, the smallest worst-case instance consists of only those, at most three, large jobs.

Any push optimal schedule on an instance with at most two jobs is an optimal schedule, and therefore the worst-case instance for the ratio $C^{\mathrm{P}}_{\max}/C^{*}_{\max}$ consists of three jobs. Consider such a worst-case instance, and assume w.l.o.g. that $p_1 \geq p_2 \geq p_3$ and that $s_1 > s_2$. Note that if $s_1 = s_2$, then we actually have two identical parallel machines and by Theorem 4.5 we know that $C^{\mathrm{P}}_{\max}/C^{*}_{\max} \leq \frac{8}{7}$.

Consider a worst-case instance, and assume w.l.o.g. that $p_1 \geq p_2 \geq p_3$ and that $s_1 > s_2$. By Lemma 4.12, we know that in this worst-case push optimal schedule J_1 is scheduled on M_1 and M_2 processes J_2 and J_3. We also know that $C^{\mathrm{P}}_{\max} = \frac{p_2+p_3}{s_2}$ and $C^{*}_{\max} = \max\{\frac{p_1}{s_2}, \frac{p_2+p_3}{s_1}\}$.

By push optimality, we know that $\frac{p_1+p_3}{s_1} \geq \frac{p_2+p_3}{s_2}$, and thus $C^{\mathrm{P}}_{\max}/C^{*}_{\max}$ is bounded by

$$C^{\mathrm{P}}_{\max}/C^{*}_{\max} = \min\{\frac{s_1}{s_2}, \frac{p_2+p_3}{p_1}\} \leq \min\{\frac{p_1+p_3}{p_2+p_3}, \frac{p_2+p_3}{p_1}\}.$$

This minimum is maximal, when $\frac{p_1+p_3}{p_2+p_3} = \frac{p_2+p_3}{p_1}$. Then $(p_2+p_3)^2 = p_1^2 + p_1p_3$ and thus

$$C^{\mathrm{P}}_{\max}/C^{*}_{\max} \leq \frac{\sqrt{p_1^2+p_1p_3}}{p_1} = \sqrt{1+\frac{p_3}{p_1}}. \tag{4.11}$$

As $p_2 \geq p_3$, we know that $p_1^2 + p_1p_3 = (p_2+p_3)^2 \geq 4p_3^2$ and, thus $p_1 \geq \frac{\sqrt{17}-1}{2}p_3$. Using this bound in inequality (4.11) yields

$$C^{\mathrm{P}}_{\max}/C^{*}_{\max} \leq \sqrt{1+\frac{2}{\sqrt{17}-1}} = \frac{\sqrt{17}+1}{4}.$$

□

In the following example, we have an instance for $Q2\|C_{\max}$ and a push optimal schedule for which $C^{\mathrm{P}}_{\max} = \frac{\sqrt{17}+1}{4}C^{*}_{\max}$.

Example 4.6. Consider the following instance with three jobs: $p_1 = \frac{\sqrt{17}-1}{2}$, $p_2 = p_3 = 1$, and $s_1 = \frac{\sqrt{17}+1}{4}$ and $s_2 = 1$. In the optimal schedule M_1 processes J_2 and J_3 and J_1 is scheduled on M_2. The optimal makespan is $C^{*}_{\max} = \frac{\sqrt{17}-1}{2}$. The schedule in which J_1 is processed by M_1 and J_2 and J_3 are scheduled on M_2 is a push optimal schedule with makespan $C^{\mathrm{P}}_{\max} = 2$. This schedule is depicted in Figure 4.9; $C^{\mathrm{P}}_{\max}/C^{*}_{\max} = 2/(\frac{\sqrt{17}-1}{2}) = \frac{4}{\sqrt{17}-1} = \frac{\sqrt{17}+1}{4}$.

4.3.3 Unrelated parallel machines

In the unrelated parallel machine environment, the processing times are job and machine dependent, i.e., the processing time of job J_j on machine M_i is p_{ij}. The maximum processing time is denoted by $p_{\max} = \max_{i,j} p_{ij}$.

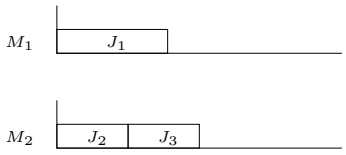


Figure 4.9: Push optimal schedule

Theorem 4.14. *A jump optimal solution for $R\|C_{\max}$ can have makespan $p_{\max}$ times the optimal solution value.*

PROOF. For given $K > 1$ consider the following instance. We are given n jobs and $m = n$ machines. The processing times of the jobs are

$$p_{ij} = \begin{cases} 1 & \text{if} \quad i = j, \\ K & \text{otherwise.} \end{cases}$$

In the optimal schedule, machine M_i processes job J_i, and this schedule has makespan $C^*_{\max} = 1$. The schedule in which machine M_1 processes job J_n and machine M_i processes job J_{i-1} $(i = 2, \ldots, m)$ is jump optimal and has makespan $C^{\mathrm{J}}_{\max} = K$. □

As the above example also is a jump optimal schedule in the case of only two machines, we have the following corollary.

Corollary 4.15. *A jump optimal schedule for $R2\|C_{\max}$ has performance guarantee at least $p_{\max}/C^*_{\max}$.*

For the identical and uniform parallel machine environments, a jump optimal schedule with ratio $\rho = C^{\mathrm{J}}_{\max}/C^*_{\max}$ can be converted into a swap optimal schedule with the same ratio ρ, in the same way as was done in the proof of Corollary 4.8. For the unrelated parallel machine environments, this is not possible. In the following two theorems we establish a lower bound on the performance guarantee for swap optimal schedules.

Theorem 4.16. *A performance guarantee for a swap optimal solution for $R\|C_{\max}$ is at least $(p_{\max} - 1)/C^*_{\max}$.*

PROOF. For given $K > 2$ consider the following instance. We are given m machines and $n = m$ jobs. The processing times of the jobs are

$$p_{ij} = \begin{cases} 1 & \text{if} \quad i = j, \\ K + 1 & \text{if} \quad j = i + 2 \pmod{n}, \\ K & \text{otherwise.} \end{cases}$$

The optimal schedule has makespan $C^*_{\max} = 1$.

The schedule in which machine M_i processes job J_{i+1}, $i = 1, \ldots, m-1$, and M_m processes job J_1 has makespan $C^{\mathrm{S}}_{\max} = K$. This schedule is swap optimal, as swapping job J_i with job J_{i+1}, results in a schedule with makespan $C_{\max} = p_{i-1,i+1} = K + 1 > C^{\mathrm{S}}_{\max}$ and any other swap does not change the makespan and the number of critical machines in the schedule. □

The example in the above proof needs at least three machines. In the case of two machines a swap optimal schedule can be as bad as $n-1$ times the optimal makespan.

Theorem 4.17. *The performance guarantee of swap optimal schedules for* $R2\|C_{\max}$ *is at least* $n-1$.

Proof. Consider a instance with n jobs, where J_1 has processing times

$$p_{i1} = \begin{cases} 1 & \text{if} \quad i = 1, \\ n - 1 - \frac{1}{n-1} & \text{if} \quad i = 2, \end{cases}$$

and the other jobs have processing times

$$p_{ij} = \begin{cases} 1 & \text{if} \quad i = 1, j > 1, \\ \frac{1}{n-1} & \text{if} \quad i = 2, j > 1. \end{cases}$$

In the optimal schedule, J_1 is scheduled on M_1 and the other jobs are processed by M_2. The makespan of this schedule is $C^*_{\max} = 1$. The schedule in which J_1 is processed by M_2 and the other jobs are scheduled on M_1 is swap optimal, having makespan $C^{\mathrm{S}}_{\max} = n - 1$. □

4.4 Running time

Until now, we have focused on the quality of local optima with respect to three neighborhoods. In this section, we make some remarks on the time it takes a form of iterative improvement to find jump optimal solutions. For identical machines, Finn and Horowitz (1979) and Brucker et al. (1997) proposed an iterative improvement procedure in which a job jumps from a critical machine to a machine with minimum load. Finn and Horowitz claimed that this procedure would terminate in $\mathcal{O}(n)$ jumps. In the following example we provide an example using $\Omega(n^2)$ jumps and thus contradicting this claim.

Example 4.7. Consider the following instance for two identical parallel machines. Let q be a positive integer. Then the number of jobs is given by $n = 3(q+1)$. The processing times of the jobs are $1, 2, \ldots, 2^q$, each length occurring three times.

The schedules below are described by giving the processing times of the jobs on the two machines. Let the initial schedule be given by

$$S_0 \qquad \begin{matrix} M_1 \\ M_2 \end{matrix} \left\| \begin{matrix} 1,\ldots,2^{q-1} \\ 1,\ldots,2^{q-1} \end{matrix} \right| \begin{matrix} 2^q \\ 2^q \end{matrix} \Big| \; 1,\ldots,2^q$$

In step $i+1$, for $i = 0,\ldots,q-2$ the algorithm moves a job of length 2^i from machine M_1 to machine M_2:

$$S_{i+1} \qquad \begin{matrix} M_1 \\ M_2 \end{matrix} \left\| \begin{matrix} 1,\ldots,2^{q-1} \\ 1,\ldots,2^{q-1} \end{matrix} \right| \begin{matrix} 2^q \\ 2^q \end{matrix} \Big| \begin{matrix} 2^{i+1},\ldots,2^q \\ 1,\ldots,2^i \end{matrix}$$

After this, we end up at schedule S_{q-1}:

$$S_{q-1} \qquad \begin{matrix} M_1 \\ M_2 \end{matrix} \left\| \begin{matrix} 1,\ldots,2^{q-1} \\ 1,\ldots,2^{q-1} \end{matrix} \right| \begin{matrix} 2^q \\ 2^q \end{matrix} \Big| \begin{matrix} 2^{q-1},2^q \\ 1,\ldots,2^{q-2} \end{matrix}$$

Note that the machine loads differ by 2^q+1, and in the next step the algorithm may therefore move a job of length 2^q to arrive at S_q:

$$S_q \qquad \begin{matrix} M_1 \\ M_2 \end{matrix} \left\| \begin{matrix} 1,\ldots,2^{q-1} \\ 1,\ldots,2^{q-1} \end{matrix} \right| \begin{matrix} 2^q \\ 2^q \end{matrix} \Big| \begin{matrix} \\ 1,\ldots,2^{q-2} \end{matrix} \Big| \begin{matrix} 2^{q-1} \\ 2^q \end{matrix}$$

which can be rearranged to

$$S_q \qquad \begin{matrix} M_1 \\ M_2 \end{matrix} \left\| \begin{matrix} 1,\ldots,2^{q-2} \\ 1,\ldots,2^{q-2} \end{matrix} \right| \begin{matrix} 2^{q-1},2^{q-1} \\ 2^q \end{matrix} \Big| \begin{matrix} 2^q \\ 2^q \end{matrix} \Big| \begin{matrix} \\ 1,\ldots,2^{q-2} \end{matrix} \Big| \; 2^{q-1}$$

Thus q jobs have been moved, with lengths $1,\ldots,2^{q-2}$ and 2^q, and we can repeat this procedure, moving $q-1$ jobs from machine M_2 to M_1, and so on. This continues until the machine loads differ by 1. In this scenario, the number of steps is $q+(q-1)+(q-2)+\ldots+1 = \frac{q(q+1)}{2} \approx \frac{n^2}{18}$.

Brucker et al. showed that the proposed procedure terminates in $\mathcal{O}(n^2)$ iterations.

Theorem 4.18. *A jump optimal solution for $P\|C_{\max}$ can be found by the above mentioned iterative improvement procedure using $\mathcal{O}(n^2)$ jumps.*

We next describe an iterative improvement procedure that finds jump optimal solutions in the case of uniform parallel machines. Hereto, we need to define the *slack* of a machine. The slack of a machine is the total amount of processing requirement that can be added to this machine such that its load does not become larger than the makespan. We denote the slack of a machine M_i by $\Delta_i = s_i(C^*_{\max} - L_i)$, where $L_i = \sum_{J_j \in \mathcal{J}_i} p_{ij}$. Note that when no job on a critical machine has a processing requirement that is less than the maximum slack, we have found a jump optimal solution.

We propose an iterative improvement procedure in which in each iteration a job is selected from a critical machine and jumps to a machine with maximum slack.

Theorem 4.19. *The iterative improvement procedure described above finds a jump optimal solution for $Q\|C_{\max}$ after $\mathcal{O}(n^2m)$ jumps.*

PROOF. Note that this algorithm computes a sequence of schedules with non-increasing makespan and maximum slack. We denote the values of L_i, $C_{\max}$, Δ_i, and $\Delta = \max_{1\leq i\leq m} \Delta_i$ in an iteration t by $L_i(t)$, etc.

Consider a machine M_i that was critical in iteration t_0 and had maximum slack in iteration $t_1 > t_0$, where t_0 and t_1 are chosen such that M_i is neither critical nor has maximum slack in iterations t, for $t_0 < t < t_1$. Note that if none of the machines satisfy this condition, the algorithm is finished after $\mathcal{O}(nm)$ iterations: if a job has been moved onto m different machines, it will certainly have been moved to a machine from which it was moved before.

Let job J_j be the job that was moved in iteration t_0. Then $L_i(t_1) = L_i(t_0) - p_j/s_i = C_{\max}(t_0) - p_j/s_i$. By monotonicity of $C_{\max}$, we have $\Delta(t_1) = s_i(C_{\max}(t_1) - L_i(t_1)) \leq s_i(C_{\max}(t_0) - L_i(t_0)) + p_j = p_j$. So, $p_j \geq \Delta(t_1)$ and by monotonicity of Δ job J_j cannot be moved.

Hence, after at most nm iterations, at least one job cannot be moved and thus after $\mathcal{O}(n^2m)$ iterations no job can move and the algorithm terminates. □

This result can be improved by always selecting the largest job on a critical machine that can jump. Using this neighbor selecting rule, the iterative improvement procedure finds a jump optimal solution in $\mathcal{O}(nm)$ iterations. This is stated in the following theorem.

Theorem 4.20. *The iterative improvement procedure that always selects the largest possible job on a critical machine to jump to the machine with maximal slack finds a jump optimal solution for $Q\|C_{\max}$ in $\mathcal{O}(nm)$ iterations.*

PROOF. We show that once a job has jumped away from a machine, it can never return to this machine. Hence, a job can jump at most $m-1$ times and the number of jumps in the described iterative improvement procedure is $\mathcal{O}(nm)$.

We use the same notation as in the previous proof. Let J_j jump from machine M_i in iteration t_0. Let $t_2 > t_0$ be the iteration in which M_i is a machine with maximum slack such that M_i is not a machine with maximum slack in iterations t, for $t_0 < t < t_2$. If this situation does not occur, then J_j can never jump to M_i again. Otherwise, there exists a $t_1 \geq t_0$ such that M_i is a critical machine in iteration t_1 and is not critical in iterations t for $t_1 < t < t_2$. Let J_k be the job that jumps from M_i in iteration t_1. Then, by the same arguments as in Theorem 4.19, we know that $p_k \geq \Delta(t_2)$. As M_i is not a machine with maximum slack in iterations $t_0 \leq t \leq t_1$, the jobs assigned to M_i in iteration t_1 were also scheduled on M_i in iteration t_0. By our selection rule and the fact that Δ is non-increasing, we know that $p_j \geq p_k$ and therefore $p_j \geq \Delta(t_2)$. Hence, job J_j cannot return to machine M_i. □

As a corollary of the above theorem, we can also improve the result of Theorem 4.18.

Corollary 4.21. *The iterative improvement that always selects the largest possible job on a critical machine to jump to a machine with minimum load finds a jump optimal solution for $P\|C_{\max}$ in $\mathcal{O}(nm)$ jumps.*

4.5 Concluding remarks

The main focus of this chapter was the quality of local optima with respect to three neighborhoods. We have seen that, with respect to the three neighborhoods we considered, the local optima have a constant performance guarantee for the problem of minimizing makespan on identical parallel machines. In the case of uniform parallel machines, the two basic neighborhoods do not have a constant performance guarantee. The neighborhood based on variable-depth search provides us with local optima that have a constant performance guarantee. It would be interesting to see whether we can extend the push neighborhood to the unrelated parallel machine environment.

We also saw that only a polynomial number of iterations is needed to find a jump optimal solution for $P\|C_{\max}$ and $Q\|C_{\max}$. It is still an open question how many iterations iterative improvement needs to find a swap or a push optimal solution. We conjecture that a push optimal solution cannot be found in polynomial time through an iterative improvement procedure.

5

Generalized graph coloring: the worst-case of local search

5.1 Introduction

Consider the following problem. Given a graph $G = (V, E)$, a weight function $w : E \to \mathbb{Z}$ on its edges, and an integer $k \geq 2$, find a color assignment $c : V \to \{1, \ldots, k\}$ of the vertices that minimizes the total weight of the *monochromatic edges*, i.e., edges that have end points with the same color. The problem was first stated by Carlson and Nemhauser (1966), who write about 'scheduling to minimize interaction cost'. The problem may occur when one wishes to partition a set of items into a given number of groups so as to minimize the total pairwise interaction cost. The problem is also referred to as the generalized graph coloring problem (GGCP) (Kolen and Lenstra, 1995), graph k-partitioning (Kann, Khanna, Lagergren, and Panconesi 1997), and k-min cluster (Sahni and Gonzalez, 1976). For $k = 2$ the problem is equivalent to the well-known max cut problem, as for two colors minimizing the total weight of the monochromatic edges is equivalent to maximizing the weight of the cut edges. For general k, the problem is equivalent to the max k-cut problem. As max cut is NP-hard, see Karp (1972), the GGCP is also NP-hard, even for fixed k. If k is not part of the input, we denote the problem by k-GGCP.

From an approximation point of view, the GGCP is not equivalent to the max k-cut problem. Under the assumption that P $\neq$ NP, Kann et al. (1997) show that for $k > 2$ and every $\epsilon > 0$ there exists a constant $\alpha > 0$ such that the GGCP cannot be approximated in polynomial time within a factor $\alpha|V|^{2-\epsilon}$ of optimal. For the 2-GGCP, Garg, Vazirani, and Yannakakis (1996) gave a polynomial-time

$\mathcal{O}(\log n)$-approximation algorithm. On the negative side, as a direct consequence of the hardness of approximating max cut by Håstad (1997), there cannot exist a polynomial-time algorithm for 2-GGCP such that the solution is guaranteed to have value within $\frac{18}{17}$ times the optimal solution value, unless P = NP.

In this chapter, we recall a result that is implicit in Carlson and Nemhauser (1966) on the relation between the Karush-Kuhn-Tucker conditions and so-called FLIP-optimal solutions. Moreover, we show that the quality of local optima may be bad, and we mention some results on the time required to find locally optimal solutions.

5.2 Neighborhoods

In this section, we describe the neighborhood FLIP, its extension m-FLIP, and a variable-depth search variant of FLIP, called VD-FLIP.

Given a solution, a FLIP neighbor is obtained by choosing a single vertex and assigning it a different color. A solution is FLIP-*optimal* if flipping any single vertex does not decrease the total weight of monochromatic edges.

To obtain an m-FLIP neighbor of a given solution, we choose at most m vertices and flip them. A solution is m-FLIP-*optimal* if it has no m-FLIP neighbor of smaller objective value.

The third neighborhood, VD-FLIP, is a form of variable-depth search, introduced by Kernighan and Lin (1970) for the graph partitioning problem. To obtain a neighbor of a given solution, we start by labeling all vertices 'unflipped'. We iteratively choose the unflipped vertex that is best to flip, assign it the best new color, and label it 'flipped'. After $|V|$ iterations all vertices have been flipped and we have obtained a series of $|V|$ solutions, of which we choose the best one as our neighbor. Note that if there are only two colors, the last solution in the series is equivalent to the first solution. We say that a solution is VD-FLIP-*optimal* if its VD-FLIP neighbor does not have a smaller objective value.

5.3 KKT conditions and FLIP-optimality

Carlson and Nemhauser (1966) gave a quadratic programming formulation for the GGCP. It uses binary variables x_{hi} for $i \in V$, $h = 1, \ldots, k$: $x_{hi} = 1$ if and only if vertex i is colored with color h. The weight function is extended to the complete graph on V by setting $w_{ij} = 0$ whenever $\{i, j\}$ is not an edge in E. The quadratic formulation is then the following:

$$
\begin{aligned}
& & \min \quad & \frac{1}{2} \sum_h \sum_{i,j} w_{ij} x_{hi} x_{hj} & & & \\
(QP) \qquad & & \text{s.t.} \quad & \sum_h x_{hi} = 1, & & i \in V, & (5.1) \\
& & & x_{hi} \in \{0,1\}, & & i \in V, \quad h = 1, \ldots, k. & (5.2)
\end{aligned}
$$

As there is a one-to-one correspondence between feasible solutions to (QP) and a color assignment of the vertices, we can denote a feasible color assignment c by its corresponding feasible solution $x \in \{0,1\}^{k|V|}$ to (QP).

Let us replace the integrality constraint (5.2) by $x_{hi} \geq 0$. Carlson and Nemhauser showed that there exists an optimal solution to this program that is integral. The Karush-Kuhn-Tucker (KKT) conditions for this quadratic program are

$$\begin{aligned}
\sum_j w_{ij}x_{hj} - \lambda_i &= \mu_{hi}, \quad i \in V, \quad h = 1, \ldots, k, && (5.3)\\
\sum_h x_{hi} &= 1, \quad i \in V, \\
\mu_{hi} &\geq 0, \quad i \in V, \quad h = 1, \ldots, k, && (5.4)\\
x_{hi} &\geq 0, \quad i \in V, \quad h = 1, \ldots, k, && (5.5)\\
\mu_{hi}x_{hi} &= 0, \quad i \in V, \quad h = 1, \ldots, k. && (5.6)
\end{aligned}$$

The following result is implicit in Carlson and Nemhauser (1966), and also mentioned by Lenstra (1976).

Theorem 5.1. *An integral solution satisfies the* KKT *conditions if and only if it is* FLIP*-optimal.*

PROOF. Suppose $x \in \{0,1\}^{k|V|}$ satisfies the KKT conditions, for some $\lambda \in \mathbb{R}^{|V|}$ and $\mu \in \mathbb{R}_+^{k|V|}$, and let x' be the solution obtained by flipping a vertex, say vertex j. Assume w.l.o.g. that this vertex has color g in x and color g' in x'. The change in costs due to this flip is

$$\begin{aligned}
\frac{1}{2}\sum_h \sum_{p,q} w_{pq}x'_{hp}x'_{hq} - \frac{1}{2}\sum_h \sum_{p,q} w_{pq}x_{hp}x_{hq} &= \sum_p w_{pj}x_{g'p} - \sum_p w_{pj}x_{gp} \\
&\overset{(5.3)}{=} (\lambda_j + \mu_{g'j}) - (\lambda_j + \mu_{gj}) \\
&\overset{(5.6)}{=} \mu_{g'j} \geq 0.
\end{aligned}$$

Thus this new solution is not better than x and hence x is FLIP-optimal.

Now, consider a FLIP-optimal solution $x \in \{0,1\}^{k|V|}$. Let λ_i be the total weight of monochromatic edges incident to vertex $i \in V$, i.e., if $x_{gi} = 1$, then

$$\lambda_i = \sum_j w_{ij}x_{gj}, \quad i \in V.$$

Let μ_{hi} denote the change in the weight of monochromatic edges incident to vertex $i \in V$ when we change its color to h, i.e.,

$$\mu_{hi} = \sum_j w_{ij}x_{hj} - \lambda_i, \quad h = 1, \ldots, k, \quad i \in V.$$

As x is FLIP-optimal, $\mu_{hi} \geq 0$ and $\mu_{gi} = 0$ if $x_{gi} = 1$, and thus (x, λ, μ) satisfies (5.5), (5.4), and (5.6). As x is a feasible solution, it certainly satisfies (5.1) and by definition of λ and μ, (x, λ, μ) satisfies (5.3). Hence, (x, λ, μ) is a KKT-point. □

In their paper, Carlson and Nemhauser propose an iterative improvement procedure that always moves to the best FLIP neighbor, i.e., the one yielding the highest decrease in the objective value; one may escape from local optima by making a zero-cost FLIP. They report that this method is efficient and frequently attains global minima. For an instance with 45 vertices, Kolen and Lenstra (1995) report that iterative improvement over the FLIP neighborhood always finds the same local minimum in the case of two colors, while for three and four colors several local minima are being found. In the subsequent sections, we prove worst-case results on the quality of local optima and the running time of iterative improvement.

5.4 Local optima may be bad

We will now show that a large class of local optima can be arbitrarily bad. The underlying neighborhood functions for this class are so-called *polynomially searchable neighborhoods*, which are neighborhoods for which in polynomial-time either a better neighbor will be found if one exists or it is determined that the current solution is locally optimal.

Theorem 5.2. *Consider the* GGCP *with* $k \geq 3$*. For any constant* $\rho > 1$*, a local optimum w.r.t. a polynomially searchable neighborhood is not guaranteed to have value at most* ρ *times the optimum, unless* $P = NP$*.*

PROOF. Consider a graph $G = (V, E)$ with unit weights on the edges. For $k \geq 3$, the problem of deciding whether G is *k-colorable*, i.e., it can be colored with k colors without monochromatic edges, is NP-complete (Karp, 1972). If G is k-colorable, then the optimal value for GGCP is 0 and otherwise it will be at least 1. If there exists a constant $\rho > 1$ such that a local optimum is guaranteed to have value at most ρ times the optimal value, then any locally optimal solution for a k-colorable graph has value 0 and it would be a global optimum, whereas a local optimum for a graph that cannot be properly colored with k colors has value at least 1. Hence, any procedure that finds a local optimum decides on the k-colorability of G. An arbitrary coloring has value at most $|E|$, because of the unit weights. Thus an iterative improvement procedure needs at most $|E|$ iterations to find a local optimum. As the time spent to find each neighbor is by assumption polynomially bounded in the input size, the total time spent by iterative improvement is polynomially bounded. □

For the three neighborhoods defined in Section 5.2, we show stronger results, as these results hold without the assumption P ≠ NP and are also true for $k = 2$.

Theorem 5.3. *For any constant* $\rho > 1$*, there exists a class of instances and a* FLIP*-optimal solution for each instance, such that the value of this* FLIP*-optimal solution is larger than* ρ *times the optimal solution value.*

PROOF. Consider the graph $G = (V, E)$ with $V = \{1, 2, 3, 4\}$ and $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}\}$ and let the weights on the edges be 1. As this graph is bipar-

tite, the optimum for the 2-GGCP has value 0.

It is easy to see that the solution c in which vertices 1 and 4 are colored blue and the other two vertices are colored red, is FLIP-optimal for the 2-GGCP and that this solution has value 1.

This graph can be extended to the general case of k colors by adding $k-2$ vertices all adjacent to the vertices 2 and 3. This extended graph has unit weights on the edges. As this graph is k-colorable, the optimal value is 0. In the FLIP-optimal solution, the coloring of V remains as in c and the $k-2$ new vertices are matched to the $k-2$ unused colors. □

Theorems 5.4 and 5.5 extend this result to m-FLIP-optimal and VD-FLIP-optimal solutions, respectively.

Theorem 5.4. *For any constant $\rho > 1$, there exists a class of instances and an m-*FLIP*-optimal solution for each instance, such that the value of this m-*FLIP*-optimal solution is larger than ρ times the optimal solution value.*

PROOF. Choose a positive integer $m \geq 2$. Define a graph $G = (V, E)$ by $V = \{1, 2, \ldots, 2m+2\}$, $E = \{\{1,2\}, \{2,3\}, \ldots, \{2m+1, 2m+2\}\}$, let it have unit weights on the edges, and let $k = 2$. This graph is bipartite and therefore the optimum has value 0.

Consider a coloring c in which the odd numbered vertices between 1 and $m+1$ and the even vertices between $m+2$ and $2m+2$ are colored red and the other vertices are colored blue. Because of the edge $\{m+1, m+2\}$, the coloring c has value 1. We claim that c is m-FLIP-optimal.

Suppose we flip at most m vertices to obtain a neighboring coloring c'. If both $m+1$ and $m+2$ are flipped, or if neither of them is, then c' is no better than c. If exactly one of $m+1$ and $m+2$ is flipped, say $m+1$, then consider the connected component of the subgraph induced by the flipped vertices containing vertex $m+1$. Obviously, there is an unflipped vertex m' with $1 \leq m' < m+1$. Hence, in this case c' is no better than c either.

The graph and the locally optimal solution in the above proof can be extended to the general problem with k colors by adding a $(k-2)$-clique to the graph, and making all vertices of this clique adjacent to the $2m+2$ vertices of the bipartite graph. All additional edges have unit weight. The m-FLIP-optimal solution c is extended by matching the $k-2$ new vertices to the $k-2$ unused colors. □

Theorem 5.5. *For any constant $\rho > 1$, there exists a class of instances and a* VD-FLIP*-optimal solution for each instance, such that the value of this locally optimal solution is more than ρ times the optimal solution value.*

PROOF. Consider the graph $G = (V, E)$, with $V = \{1, \ldots, 8\}$ and $E = \{\{1,7\}, \{2,7\}, \{3,4\}, \{3,7\}, \{4,8\}, \{5,8\}, \{6,8\}\}$. The weights on the edges are depicted in Figure 5.1. This graph is bipartite and has optimal value 0. The coloring

c in which we color vertices 7 and 8 red and the other vertices are colored blue has weight 1. We claim that this solution is VD-FLIP-optimal.

In Figure 5.2, we show how the variable-depth search will proceed. All unflipped vertices are denoted by circles and the flipped vertices are denoted by squares. The value next to an unflipped vertex denotes the increase in the objective value if this vertex is flipped. We iteratively choose the best unflipped vertex, which is denoted by an extra circle. In Figure 5.2(a), the coloring c is shown. The best vertex to flip is vertex 3 and we proceed as shown in Figures 5.2(b–i). The intermediate solution in Figure 5.2(f) and the starting and final solution all have objective value 1; the other intermediate solutions have all value at least 2. Thus the coloring c is VD-FLIP-optimal.

We extend this graph to the general case of k colors by adding a $(k-2)$-clique of which all vertices are adjacent to the vertices in V. All added edges have weight at least 7 and in the coloring c, the $k-2$ new vertices are matched to the $k-2$ unused colors. □

5.5 Local optima may be hard to find

For the computational complexity of finding local optima, Johnson, Papadimitriou, and Yannakakis (1988) introduced the class of polynomial-time local search (PLS) problems; see also Yannakakis (1997). This class contains local search problems whose neighborhoods are polynomially searchable. The local search problems of the GGCP with one of the neighborhoods defined in Section 5.2 are all in PLS. Johnson et al. also defined a reduction among problems in this class and showed that there exist PLS-complete problems. If a local optimum for such a complete problem can be found in polynomial time by whatever means, then for all problems in PLS a local optimum can be found in polynomial time. This is generally not believed to be true, as it would require a general approach to finding local optima at least as clever as the ellipsoid algorithm, since linear programming with the simplex neighborhood is in PLS. On the other hand, Johnson et al. showed that if a PLS problem is NP-hard, then NP = co-NP.

Schäffer and Yannakakis (1991) showed that the max cut problem with the FLIP neighborhood is PLS-complete. As a generalization of this the GGCP with the FLIP neighborhood is PLS-complete. As an m-FLIP-optimal and a VD-FLIP-optimal solution are also FLIP-optimal, the GGCP with the m-FLIP or the VD-FLIP neighborhood are PLS-complete too. Schäffer and Yannakakis introduced the notion of *tight* PLS reductions. If there is a tight PLS reduction from a problem Π_1 to a problem Π_2 and Π_1 contains instances and starting solutions for which iterative improvement needs an exponential number of iterations, then there exist instances and starting solutions for Π_2 with the same property. By constructing a tight PLS reduction, they showed that finding a FLIP-optimal solution for max cut by iterative improvement may take an exponential number of iterations, regardless of the neighbor selecting rules. Hence, finding a FLIP-optimal solution for the GGCP by iterative improvement may take an exponential number of iterations as well. As the reductions for

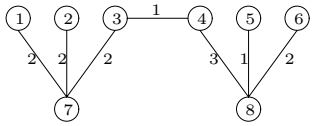


Figure 5.1: Graph

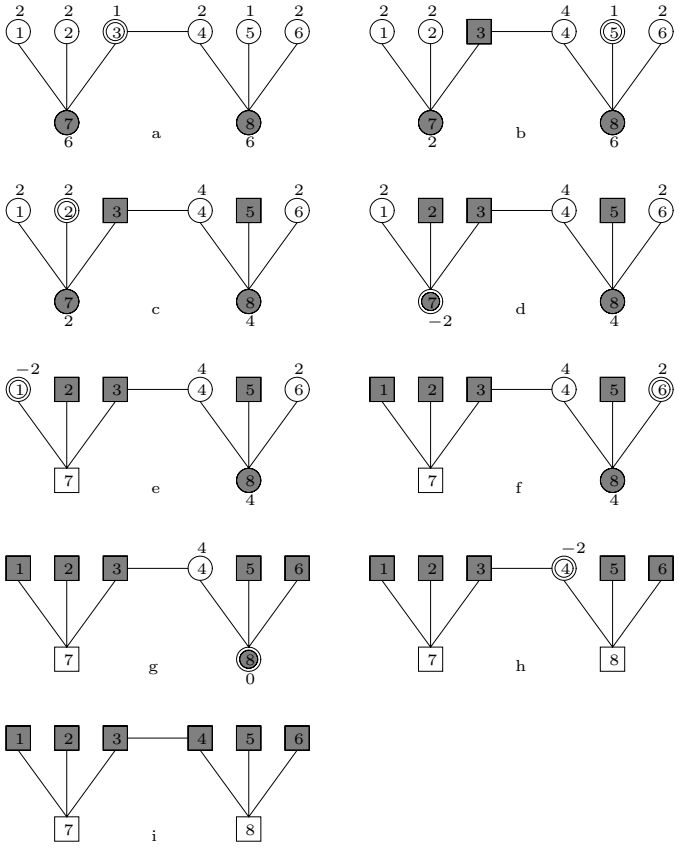


Figure 5.2: Variable-depth flip

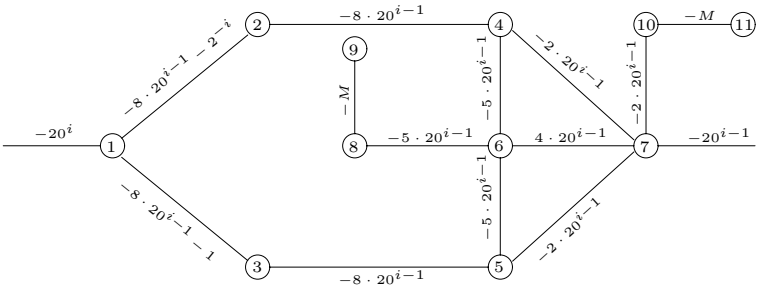


Figure 5.3: Module i

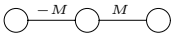


Figure 5.4: Chain

the GGCP with the m-FLIP and with the VD-FLIP neighborhoods are not tight, this result does not extend to iterative improvement procedures for finding m-FLIP- and VD-FLIP-optimal solutions. This does not imply that there does not exist a tight PLS reduction for these problems.

To illustrate the exponential number of iterations needed for finding a FLIP-optimal solution, we give an example of a graph and an initial solution for 2-GGCP for which *best improvement*, i.e., always flipping the best vertex, needs an exponential number of iterations to find a FLIP-optimal solution. This graph consists of K modules with weights on the edges as shown in Figure 5.3 for $i = 1, \ldots, K$ and a chain of three additional vertices as shown in Figure 5.4.

Vertex 1 is called the input node and vertex 7 is called the output node of a module. The input node of module i is adjacent to the output node of module $i+1$, for $i = K-1, \ldots, 1$, and the input node of module K is adjacent to the right most vertex of the chain of Figure 5.4. The output node of module 1 is only adjacent to vertices 4, 5, 6, and 10 of this module. An edge of weight $-M$, where M is some large positive value, makes sure that the two vertices incident to this edge have the same color. We claim that the best improvement procedure starting from the solution in which all vertices are colored red, flips the output node of the first module 2^K times.

In our starting solution, only flipping the right most vertex of the chain yields an improvement. This flip results in a solution in which the input node of module K is *unhappy*, i.e., flipping this vertex improves the solution. We now show by induction on K that the output node of module 1 flips 2^K times. For $K = 0$, the output node is the right most vertex of the chain and it is flipped once.

Assume the claim is true for $K-1$ modules. Consider a graph on K modules of

which the only unhappy vertex is the input node of module K. Flipping this vertex yields a solution in which vertices 2 and 3 of module K are unhappy. Changing vertex 2 yields an improvement of 2^{-K} and by our choice of edge weights, best improvement will only change the color of this vertex when all other vertices are happy. Hence, vertices 3, 5 and 7 are flipped, which results in a solution in which the input node of module $K-1$ is unhappy. By induction we know that the output node of module 1 will now flip 2^{K-1} times and then we have found a solution in which all vertices in the modules $1, \ldots, K-1$ are happy and the only unhappy vertex is vertex 2 of module K. Thus this vertex is flipped and then successively vertices 4 and 6 and the output node of module K are flipped. This yields a solution in which the input node of module $K-1$ is unhappy. By induction, we know that the output node of module 1 flips another 2^{K-1} times and then we have found a FLIP-optimal solution. Hence, the number of times that the output node of module 1 is flipped is 2^K.

Bibliography

E.H.L. Aarts and J.K. Lenstra (editors) (1997). *Local Search in Combinatorial Optimization*, Wiley, Chichester.

E.H.L. Aarts and P.J.M. Van Laarhoven (1985a). A new polynomial time cooling schedule, In *Proceedings IEEE International Conference on Computer-Aided Design*, pages 206–208.

E.H.L. Aarts and P.J.M. Van Laarhoven (1985b). Statistical cooling: A general approach to combinatorial optimization problems, *Philips Journal of Research* 40, 193–226.

F. Alizadeh (1995). Interior point methods in semidefinite programming with applications to combinatorial optimization, *SIAM Journal on Optimization* 5, 13–51.

G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi (1999). *Complexity and Approximation: Combinatorial optimization problems and their approximability properties*, Springer, Berlin.

F. Barahona and A.R. Mahjoub (1986). On the cut polytope, *Mathematical Programming* 36, 157–173.

F. Barahona, M. Grötschel, M. Jünger, and G. Reinelt (1988). An application of combinatorial optimization to statistical physics and circuit layout design, *Operations Research* 36, 493–513.

F. Barahona, M. Jünger, and G. Reinelt (1989). Experiments in quadratic 0–1 programming, *Mathematical Programming* 44, 127–137.

J.W. Berry and M.K. Goldberg (1999). Path optimization for graph partitioning problems, *Discrete Applied Mathematics* 90, 27–50.

F. Bock (1958). An algorithm for solving 'traveling-salesman' and related network optimization problems. Manuscript associated with talk presented at the Fourteenth National Meeting of the Operations Research Society of America. St. Louis, MO.

Y. Boykov, O. Veksler, and R. Zabih (1999). A new algorithm for energy minimization with discontinuities, In *International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition*, pages 205–220, Springer, Berlin.

P. Brucker, J. Hurink, and F. Werner (1996). Improving local search heuristics for some scheduling problems I, *Discrete Applied Mathematics* 65, 97–122.

P. Brucker, J. Hurink, and F. Werner (1997). Improving local search heuristics for some scheduling problems II, *Discrete Applied Mathematics* 72, 47–69.

J.L. Bruno, E.G. Coffman, Jr., and R. Sethi (1974). Scheduling independent tasks to reduce mean finishing time, *Communications of the ACM* 17, 382–387.

R.C. Carlson and G.L. Nemhauser (1966). Scheduling to minimize interaction cost, *Operations Research* 14, 52–58.

M.W. Carter (1984). The indefinite zero-one quadratic problem, *Discrete Applied Mathematics* 7, 23–44.

V. Černy (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm, *Journal of Optimization Theory and Applications* 45, 41–51.

B. Chandra, H. Karloff, and C. Tovey (1999). New results on the old k-opt algorithm for the traveling salesman problem, *SIAM Journal on Computing* 28, 1998–2029.

Y. Cho and S. Sahni (1980). Bounds for list schedules on uniform processors, *SIAM Journal on Computing* 9, 91–103.

G.A. Croes (1958). A method for solving traveling salesman problems, *Operations Research* 6, 791–812.

K.M.J. De Bontridder, B.V. Halldórsson, M.M. Halldórsson, C.A.J. Hurkens, J.K. Lenstra, R. Ravi, and L. Stougie (2001). Approximation algorithms for the minimum test set problem, Manuscript.

C. De Simone, M. Diehl, M. Jünger, P. Mutzel, G. Reinelt, and G. Rinaldi (1995). Exact ground states of Ising spin glasses: new experimental results with a branch-and-cut algorithm, *Journal of Statistical Physics* 80, 487–496.

C. Delorme and S. Poljak (1993). Laplacian eigenvalues and the maximum cut problem, *Mathematical Programming* 62, 557–574.

M.E. Dyer and L.A. Wolsey (1990). Formulating the single machine sequencing problem with release dates as a mixed integer program, *Discrete Applied Mathematics* 26, 255–270.

U. Feige and G. Schechtman (2002). On the optimality of the random hyperplane rounding technique for MAX CUT, *Random Structures and Algorithms*, to appear.

U. Feige, M. Karpinski, and M. Langberg (2000). Improved approximation of MAX-CUT on graphs of bounded degree, Technical Report 85215 CS, Institut für Informatik, Universität Bonn.

C.M. Fiduccia and R.M. Mattheyses (1982). A linear-time heuristic for improving network partitions, In *Proceedings of the 19th IEEE Design Automation Conference*, pages 175–181.

G. Finn and E. Horowitz (1979). A linear time approximation algorithm for multiprocessor scheduling, *BIT* 19, 312–320.

P.M. França, M. Gendrau, G. Laporte, and F.M. Müller (1994). A composite heuristic for the identical parallel machine scheduling problem with minimum makespan objective, *Computers and Operations Research* 21, 205–210.

M.R. Garey and D.S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco.

M.R. Garey, D.S. Johnson, and L. Stockmeyer (1976). Some simplified NP-complete graph problems, *Theoretical Computer Science* 1, 237–267.

N. Garg, V.V. Vazirani, and M. Yannakakis (1996). Approximate max-flow min-(multi)cut theorems and their applications, *SIAM Journal on Computing* 25, 235–251.

C.A. Glass, C.N. Potts, and P. Shade (1994). Unrelated parallel machine scheduling using local search, *Mathematical and Computer Modelling* 20, 41–52.

F. Glover (1989). Tabu search: part 1, *ORSA Journal on Computing* 1, 190–206.

F. Glover (1990). Tabu search: part 2, *ORSA Journal on Computing* 2, 4–32.

M.X. Goemans and D.P. Williamson (1995). Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming, *Journal of the ACM* 42, 1115–1145.

R.L. Graham (1966). Bounds for certain multiprocessing anomalies, *Bell System Technical Journal* 45, 1563–1581.

R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey, *Annals of Discrete Mathematics* 5, 287–326.

M. Grötschel, L. Lovász, and A. Schrijver (1981). The ellipsoid method and its consequences in combinatorial optimization, *Combinatorica* 1, 169–197.

M. Grötschel, L. Lovász, and A. Schrijver (1984). Corrigendum to our paper "the ellipsoid method and its consequences in combinatorial optimization", *Combinatorica* 4, 291–295.

A. Gupta and É. Tardos (2000). A constant factor approximation algorithm for a class of classification problems, In *Proceedings of 32nd ACM Symposium on Theory of Computing*, pages 652–658.

D.J. Haglin and S.M. Venkatesan (1991). Approximation and intractability results for the maximum cut problem and its variants, *IEEE Transactions on Computers* 40, 110–113.

L.A. Hall, A.S. Schulz, D.B. Shmoys, and J. Wein (1997). Scheduling to minimize average completion time: off-line and on-line algorithms, *Mathematics of Operations Research* 22, 513–544.

A.M.A. Hariri and C.N. Potts (1991). Heuristics for scheduling unrelated parallel machines, *Computers and Operations Research* 18, 323–331.

J. Håstad (1997). Some optimal inapproximability results, In *Proceedings of 29th ACM Symposium on Theory of Computing*, pages 1–10.

R. Haupt (1989). A survey of priority rule-based scheduling, *OR Spektrum* 11, 3–16.

C. Helmberg and F. Rendl (2000). A spectral bundle method for semidefinite programming, *SIAM Journal on Optimization* 10, 673–696.

D.S. Hochbaum (editor) (1997). *Approximation Algorithms for NP-hard problems*, PWS Publishing Company, Boston.

D.S. Hochbaum and D.B. Shmoys (1987). Using dual approximation algorithms for scheduling problems: theoretical and practical results, *Journal of the ACM* 34, 144–162.

D.S. Hochbaum and D.B. Shmoys (1988). A polynomial approximation scheme for machine scheduling on uniform processors: using the dual approximation approach, *SIAM Journal on Computing* 17, 539–551.

T. Hofmeister and H. Lefmann (1996). A combinatorial design approach to MAX-CUT, In *Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 1046, pages 441–452, Springer, Berlin.

S. Homer and M. Peinado (1997). Design and performance of parallel and distributed approximation algorithms for maxcut, *Journal of Parallel and Distributed Computing* 46, 48–61.

J.A. Hoogeveen, P. Schuurman, and G.J. Woeginger (1998). Non-approximability results for scheduling problems with minsum criteria, In *Proceedings of the 6th Conference on Integer Programming and Combinatorial Optimization*, LNCS 1412, pages 353–366, Springer, Berlin.

W.A. Horn (1973). Minimizing average flow time with parallel machines, *Operations Research* 21, 846–847.

C.A.J. Hurkens and A. Schrijver (1989). On the size of systems of sets every t of which have an sdr, with an application to the worst-case ratio of heuristics for packing problems, *SIAM Journal on Discrete Mathematics* 2, 68–72.

C.A.J. Hurkens and T. Vredeveld (2002). Bounding the number of local search moves for multiprocessor scheduling problems, Manuscript.

D.S. Johnson and L.A. McGeoch (1997). The traveling salesman problem: a case study, In E.H.L. Aarts and J.K. Lenstra (editors), *Local Search in Combinatorial Optimization*, chapter 8, pages 215–310, Wiley, Chichester.

D.S. Johnson, C.H. Papadimitriou, and M. Yannakakis (1988). How easy is local search?, *Journal of Computer and System Sciences* 37, 79–100.

D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon (1989). Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning, *Operations Research* 37, 865–892.

V. Kann, S. Khanna, J. Lagergren, and A. Panconesi (1997). On the hardness of approximating max k-cut and its dual, *Chicago Journal of Theoretical Computer Science*, `http://cjtcs.cs.uchicago.edu/`.

S.E. Karish (1998). CUTSDP – a toolbox for a cutting-plane approach based on semidefinite programming, Technical Report IMM-REP-1998-10, Department of Mathematical Modelling, Technical University of Denmark.

H. Karloff (1999). How good is the Goemans-Williamson MAX CUT algorithm, *SIAM Journal on Computing* 29, 336–350.

R.M. Karp (1972). Reducibility among combinatorial problems, In R.E. Miller and J.W. Thatcher (editors), *Complexity of Computer Computations*, pages 85–103, Plenum Press, New York.

B.W. Kernighan and S. Lin (1970). An efficient heuristic procedure for partitioning graphs, *Bell System Technical Journal* 49, 291–307.

S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi (1983). Optimization by simulated annealing, *Science* 220, 671–680.

A.W.J. Kolen and J.K. Lenstra (1995). Combinatorics in operations research, In R.L. Graham, M. Grötschel, and L. Lovász (editors), *Handbook of Combinatorics*, pages 1875–1910, Elsevier Science, Amsterdam.

M.R. Korupolu, C.G. Plaxton, and R. Rajaraman (1998). Analysis of a local search heuristic for facility location problems, In *Proceedings of 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10.

K. Lang and S. Rao (1993). Finding near-optimal cuts: An empirical evaluation, In *Proceedings of 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 212–221.

M. Laurent (1997). Max-cut problem, In M. Dell'Amico, F. Maffioli, and S. Martello (editors), *Annotated Bibliographies in Combinatorial Optimization*, chapter 15, Wiley, Chichester.

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys (editors) (1985). *The traveling salesman problem: a guided tour of combinatorial optimization*, Wiley, Chichester.

J.K. Lenstra (1976). *Sequencing by Enumerative Methods*, Ph.D. thesis, Universiteit van Amsterdam.

J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker (1977). Complexity of machine scheduling problems, *Annals of Discrete Mathematics* 1, 343–362.

J.K. Lenstra, D.B. Shmoys, and É. Tardos (1990). Approximation algorithms for scheduling unrelated parallel machines, *Mathematical Programming* 46, 259–271.

S. Lin and B.W. Kernighan (1973). An effective heuristic for the traveling salesman problem, *Operations Research* 21, 498–516.

S. Mahajan and H. Ramesh (1999). Derandomizing approximation algorithms based on semidefinite programming, *SIAM Journal on Computing* 28, 1641–1663.

B. Mohar and S. Poljak (1990). Eigenvalues and the max-cut problem, *Czechoslovak Mathematical Journal* 40, 343–352.

E. Nowicki and C. Smutnicki (1996). A fast taboo search algorithm for the job shop problem, *Management Science* 42, 797–913.

P.M. Pardalos and G.P. Rodgers (1990). Computational aspects of a branch and bound algorithm for quadratic zero-one programming, *Computing* 45, 131–144.

G. Pataki and S.H. Schmieta (1999). The DIMACS library of mixed semidefinite-quadratic-linear programs.
`http://dimacs.rutgers.edu/Challenges/Seventh/Instances/`

C. Phillips, C. Stein, and J. Wein (1997). Task scheduling in networks, *SIAM Journal on Discrete Mathematics* 10, 573–598.

S. Poljak (1995). Integer linear programs and local search for max-cut, *SIAM Journal on Computing* 24, 822–839.

S. Poljak and F. Rendl (1994). Node and edge relaxations of the max-cut problem, *Computing* 52, 123–137.

S. Poljak and F. Rendl (1995). Solving the max-cut problem using eigenvalues, *Discrete Applied Mathematics* 62, 249–278.

S. Poljak and D. Turzik (1986). A polynomial time heuristic for certain subgraph optimization problems with guaranteed worst case bound, *Discrete Mathematics* 58, 99–104.

S. Poljak and Z. Tuza (1995). Maximum cuts and large bipartite subgraphs, In W. Cook, L. Lovász, and P. Seymour (editors), *Special Year on Combinatorial Optimization*, volume 20 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 181–2443, American Mathematical Society.

S. Sahni and T. Gonzalez (1976). P-Complete approximation problems, *Journal of the ACM* 23, 555–565.

M.W.P. Savelsbergh, R.N. Uma, and J. Wein (1998). An experimental study of linear programming-based scheduling heuristics, In *Proceedings of 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 453–461.

A.A. Schäffer and M. Yannakakis (1991). Simple local search problems that are hard to solve, *SIAM Journal on Computing* 20, 56–87.

A.S. Schulz and M. Skutella (1997a). Random-based scheduling: new approximations and LP lower bounds, In J. Rolim (editor), *Randomization and Approximation Techniques in Computer Science*, LNCS 1296, pages 119–133, Springer, Berlin.

A.S. Schulz and M. Skutella (1997b). Scheduling-LPs bear probabilities: randomized approximations for min-sum criteria, In R.E. Burkard and G.J. Woeginger (editors), *Algorithms – ESA'97*, LNCS 1284, pages 416–429, Springer, Berlin.

P. Schuurman and T. Vredeveld (2001). Performance guarantees of local search for multiprocessor scheduling, In *Proceedings of 8th Integer Programming and Combinatorial Optimization Conference*, LNCS 2081, pages 370–382, Springer, Berlin.

D.B. Shmoys and E. Tardos (1993). An approximation algorithm for the generalized assignment problem, *Mathematical Programming* 62, 461–474.

M. Skutella (1998). *Approximation and Randomization in Scheduling*, Ph.D. thesis, Fachbereich Mathematik, Technische Universität Berlin.

M. Skutella (2001). Convex quadratic and semidefinite programming relaxations in scheduling, *Journal of the ACM* 48, 206–242.

W.E. Smith (1956). Various optimizers for single-stage production, *Naval Research Logistics Quarterly* 3, 59–66.

J.F. Sturm (1999). Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones, *Optimization Methods and Software* 11–12, 625–653.

R.N. Uma and J. Wein (1998). On the relationship between combinatorial and LP-based approaches to NP-hard scheduling problems, In *Proceedings of 6th Integer Programming and Combinatorial Optimization Conference*, LNCS 1412, Springer, Berlin.

J.M. Van den Akker, C.A.J. Hurkens, and M.W.P. Savelsbergh (2000). Time-indexed formulations for machine scheduling problems: column generation, *INFORMS Journal on Computing* 12, 111–124.

S.J. Van der Linden (2000). *Convex Quadratic Relaxations and Approximation Algorithms: A Computational Study*, Master's thesis, Department of Operational Research and Management, University of Amsterdam.

V.V. Vazirani (2001). *Approximation Algorithms*, Springer, Berlin.

P.M.B. Vitányi (1981). How well can a graph be n-colored?, *Discrete Mathematics* 34, 69–80.

T. Vredeveld (2001). Test instances max cut, `www.win.tue.nl/~tjark/`.

T. Vredeveld and C.A.J. Hurkens (2001). Experimental comparison of approximation algorithms for scheduling unrelated parallel machines, *INFORMS Journal on Computing*, to appear.

T. Vredeveld and J.K. Lenstra (2002). On local search for the generalized graph coloring problem, Manuscript.

M. Yannakakis (1997). Computational complexity, In E.H.L. Aarts and J.K. Lenstra (editors), *Local Search in Combinatorial Optimization*, chapter 2, pages 19–55, Wiley, Chichester.

Samenvatting

In de combinatorische optimalisering wordt veel onderzoek gedaan naar benaderingsalgoritmen. Dit zijn methoden die voor een optimaliseringsprobleem niet zozeer de beste oplossing als wel een goede oplossing zoeken. Sommige van deze algoritmen zijn ontwikkeld vanuit een theoretisch oogpunt en geven een garantie ten aanzien van de kwaliteit van de oplossing en de benodigde tijd om die te vinden. Andere zijn ontworpen om goed te werken in de praktijk: ze geven goede oplossingen binnen redelijke tijd, maar we kunnen nauwelijks garanties geven ten aanzien van de kwaliteit of de looptijd van zo'n algoritme. Deze laatste categorie bevat *lokale-zoekmethoden*. In deze methoden gaan we steeds van de huidige oplossing naar een buuroplossing totdat aan bepaalde stopcriteria voldaan is. Zo'n lokale-zoekmethode begint dus vanuit een toegelaten oplossing en voor iedere oplossing is een verzameling van buren gedefinieerd, de buurruimte.

In dit proefschrift beschouwen voor een aantal moeilijke problemen methoden uit een van de twee beschreven categorieën en analyseren deze met de maatstaf van de andere categorie. We hebben een aantal approximatie-algoritmen, waarvoor garanties bekend zijn, geïmplementeerd en hun empirische prestatie vergeleken met die van een aantal lokale-zoekalgoritmen. Vervolgens analyseren we, vanuit theoretisch oogpunt, de kwaliteit van lokale optima. We beschouwen ook de tijd die een iteratieve verbeteringsmethode nodig heeft om deze oplossingen te vinden.

In hoofdstuk 2 bekijken we het probleem van het plaatsen van taken op ongerelateerde parallelle machines. In dit probleem heeft iedere taak machine-afhankelijk behandelingsduren en een niet-negatief gewicht. Het doel is om de som van de gewogen completeringstijden te minimaliseren. De benaderingsalgoritmen met garanties ten aanzien van kwaliteit en looptijd zijn alle gebaseerd op het afronden van een oplossing voor een relaxatie van dit probleem. Deze relaxaties geven ondergrenzen voor de optimale waarde. Naast het maken van een empirische vergelijking is in dit hoofdstuk ook aandacht besteed aan de dominantierelatie tussen de verschillende ondergrenzen. Het bleek dat het algoritme dat gebaseerd is op de relaxatie die de beste ondergrens geeft, ook de beste bovengrens levert onder de algoritmen met garanties. Dit algoritme gaf ook betere resultaten dan de lokale-zoekstrategieën *iteratieve verbetering* en *tabu search*, wanneer die vanaf willekeurige startoplossingen uitgaan. De beste resultaten werden verkregen door *tabu search* toe te passen op de oplossing verkregen met het beste benaderingsalgoritme met garantie.

In hoofdstuk 3 beschouwen we het *max cut*-probleem, waarin de knopenverzameling van een graaf in tweeën gedeeld moet worden. De doelstelling is om de som van de gewichten van de kanten tussen knopen in verschillende delen te maximali-

seren. Het algoritme dat de beste garantie geeft voor de kwaliteit van de oplossing, levert ook de beste oplossingen, als we ons beperken tot de algoritmen met garanties. *Simulated annealing* levert gelijkwaardige oplossingen als de tijd die simulated annealing mag gebruiken gelijk is aan de tijd die het beste benaderingsalgoritme gebruikt. Het nadeel van deze twee methoden is dat voor de wat grotere grafen de tijd behoorlijk groot wordt. Als we de tijd die *simulated annealing* mag gebruiken verkleinen, dan wordt de kwaliteit van de gevonden oplossing niet veel slechter: gemiddeld ongeveer 0.1%.

In hoofdstuk 4 kijken we naar een aantal buurruimtes voor machinevolgordeproblemen. De doelstelling is de laatste taak zo vroeg mogelijk te laten eindigen. In deze problemen beschouwen we drie verschillende soorten parallelle machines. Bij identieke machines heeft een taak op iedere machine dezelfde tijd nodig. In het geval van uniforme machines heeft een taak een gegeven behandelingsbehoefte en een machine een gegeven snelheid; de behandelingsduur van een taak is haar behandelingsbehoefte gedeeld door de snelheid van de machine. Tenslotte zijn er ongerelateerde machines, waarbij de behandelingsduur van een taak bepaald wordt door de machine waarop zij geplaatst wordt. Voor deze machinevolgordeproblemen analyseren we de kwaliteit van lokale optima ten aanzien van de *jump*-, de *swap*- en de nieuw gedefinieerde *push*-buurruimte. Voor identieke machines is de waarde van een lokaal optimum ten aanzien van alle drie de buurruimtes altijd kleiner dan twee keer het optimum. Voor uniforme machines levert de *push*-buurruimte nog steeds een garantie van twee, terwijl lokale optima ten aanzien van de *jump* en *swap*-buurruimte slechts een garantie hebben die groeit met de wortel van het aantal machines. De *push*-buurruimte is niet gedefinieerd voor ongerelateerde machines; we laten zien dat *jump*- en *swap*- optimale oplossingen erg slecht kunnen zijn.

In hoofdstuk 5 beschouwen we het gegeneraliseerde graafkleuringsprobleem. Het doel is hier om de knopen van de graaf zodanig te kleuren dat de totale interactie tussen knopen van dezelfde kleur minimaal is. We leggen een verband tussen de Karush-Kuhn-Tucker punten van een kwadratische formulering van dit probleem en FLIP-optimale oplossingen, d.w.z., oplossingen waarbij het verplaatsen van precies één punt naar een andere kleurgroep niet leidt tot een verbetering. We laten zien dat voor drie of meer kleuren lokale optima ten aanzien van een grote verzameling buurruimtes niet een waarde kunnen hebben die gegarandeerd begrensd is door een constante maal de optimale waarde, tenzij P = NP. Voor de FLIP-, m-FLIP- en VD-FLIP-buurruimtes bewijzen we dit ook voor het geval van twee kleuren, zelfs als P = NP. Ook geven we een voorbeeld waaruit blijkt dat het vinden van een FLIP-optimale oplossing exponentieel veel tijd kan kosten.

Curriculum vitae

Tjark Vredeveld was born on July 14th 1973 in Leiden, The Netherlands. In 1991, he received his Atheneum diploma from the St.-Janscollege, Heerlen. In September of the same year he started studying Econometrics with specialty operations research at Erasmus University in Rotterdam. He received his Master's degree in August 1996. From 1995 until 1997, Tjark was employed as a consultant by PLS Point Logic Systems, nowadays known as PointLogic. In February 1997, he started as a PhD student at Erasmus University in Rotterdam. A year later, he switched to Eindhoven University of Technology, which also implied a switch of research subject to approximation algorithms for combinatorial optimization problems. The results of his research are presented in this thesis.