Efficiency of Local Search

Tobias Brueggemann

The research presented in this thesis was funded by the Netherlands Organization for Scientific Research (NWO) grant 613.000.225 (Local Search with Exponential Neighborhoods) and was carried out at the Discrete Mathematics and Mathematical Programming Group, Department of Applied Mathematics, University of Twente, Enschede, The Netherlands.

Netherlands Organization for Scientific Research
NWO Grant 613.000.225
Local Search with Exponential Neighborhoods

EWI/TW/DWMP
P.O. Box 217, 7500 AE Enschede
The Netherlands.

University of Twente
Enschede - The Netherlands

Keywords: Local Search, Very Large-Scale Neighborhoods, Scheduling, Performance Guarantees.

This thesis was edited with NEdit and typeset with LaTeX 2_ε.

Cover design by Kim McWilliams.

Printed by Wöhrmann Print Service.

Efficiency of Local Search,
by Tobias Brueggemann, University of Twente, Enschede, 2006.
ISBN: 90-365-2404-0

EFFICIENCY OF LOCAL SEARCH

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 15 september 2006 om 15.00 uur

door

Tobias Brueggemann

geboren op 20 mei 1976

te Ibbenbueren

Dit proefschrift is goedgekeurd door

Prof. dr. G.J. Woeginger (promotor)
Dr. J.L. Hurink (assistent-promotor)

Contents

Preface

This thesis is the result of my work carried out in four years, starting in 2002. Doing research is necessary for being able to write a thesis, and research can not be done without working together. Thus, I am thankful to many people I have met in this time. I will for sure forget to mention some of them, so to these: Thank You!

I thank Johann Hurink for being my supervisor. I had a nice time with him and our work was very fruitful. Without the discussions, his help and encouragements, this thesis would look very different. It was not only a pleasure to work with Johann, but also being together on conferences or meetings. Moreover, without Johann Hurink, Walter Kern and Gerhard Woeginger, the NWO project would never have existed and so, I am also thankful to them.

Additionally, I want to mention all my co-authors. I am grateful to all of them. There is Walter Kern, who always was open minded to new problems and sometimes came with astonishing approaches. And there is Tjark Vredeveld, who did his PhD on a similar subject before I began. He visited our university for a week and we had a nice time together in working and recreation. To speak with his words, it is amazing that we needed only one week to complete a subject.

I would like to thank the members of my graduation committee for participating, as there are Prof.dr.ir. K.I. Aardal, Prof.dr.ir. H.J. Broersma, Prof.dr. P. Brucker, Dr. M. van der Heijden, Prof.dr. C. Hoede, Prof.dr. F. Spieksma and Prof.dr. S. van de Velde.

I should not forget the other people of our group of Discrete Mathematics and Mathematical Programming. I enjoyed to be part of this group and will for sure miss it. Tim Nieberg was a fellow sufferer. We both studied in Osnabrueck and started nearly at the same time in Enschede. Therefore, we could share our distress and happiness, mostly during the coffee breaks, with the other group members as there are (in alphabetical order) Hilbrandt Baarsma, Jacob Jan Paulus, Gerhard Post, Georg Still. Of course, I will not forget to thank Paul Bonsma for being my room mate. It is amazing how he maddens people with his hereditary Frisian calmness. And last, but not least, I have to thank Dini, the shepherd, for keeping our group together.

I also want to thank my parents, Wolfgang and Helga, and my sister, Julia, for their support and interest. Most of all, I thank Kim for her love and patience. Without her help and the support of my family, I would not have succeeded.

Tobias Brueggemann
Enschede, July 2006

Chapter 1

Introduction

After being stranded in a mountainous area, Arthur Dent[1] decides to run for shelter because of the extremely foggy weather. Since he still wears his bath robe and is equipped with his towel and the 'Guide', he seems not to be well prepared for hiking through the mountains. His hope is, once the weather becomes clear again, to be picked up by some person friendly to hitchhikers. In order to improve his odds, he seeks for a hut as high as possible. Because of the bad weather, the visible range is restricted and so, he can only see nearby huts. The problem is that there are many huts, scattered in the wide mountain area and so, Arthur Dent has a hard job to find the highest hut.

Arthur Dent's hiking problem can be modeled as a combinatorial optimization problem in the mathematical world. A model in combinatorial optimization describes the problem of determining a feasible solution out of a finite set of feasible solutions. Every feasible solution has a measure for its quality, and the goal is to obtain a feasible solution with the best quality. Such a solution is called an optimal solution. For Arthur Dent's hiking problem, all the huts can be seen as a set of feasible solutions, and the height of a hut is a measure for its quality.

[1] Douglas Adams, The Hitchhiker's Guide To The Galaxy, Pan Macmillan, London.

For many interesting combinatorial optimization problems it is hard to find an optimal solution. The hardness of a problem is reflected by the time needed to solve it, in relation to the size of the instance. For Arthur Dent's hiking problem, an optimal solution (i.e. the highest hut) can be obtained e.g. by drawing a map. In order to draw the map, Arthur Dent may start at a single hut. The hut, Arthur Dent is currently in, is marked by him with a sign. Then he records all nearby huts and their height in his map. Afterwards, he moves to one of the nearby huts that he has not yet visited, and proceeds as before. Arthur Dent may get stuck, if he already visited all nearby huts. Then he has to go back to some already marked hut, and proceed from there to a hut, that he has not yet visited. If there is no unmarked hut left, he can read from his map, where the highest hut can be found, and moves to it.

This process may get very time consuming, and is not the method Arthur Dent would prefer. However, for a lot of hard problems only such time consuming methods for determining an optimal solution are known. In order to tackle this, one possibility is to disengage from retrieving optimal solutions, and resort to using simpler, heuristic approaches. The objective of heuristics is to come up with a feasible solution of good quality regarding the goal, and this should be achieved in a reasonable amount of time. In many cases, no a priori guarantees are available for the quality of the solution obtained by a heuristic.

For the hiking problem, Arthur Dent surely wants to find a hut before it gets dark. Thus, drawing a map is too time consuming and so, Arthur Dent may want to apply a heuristic approach. We distinguish two basic types of heuristic algorithms: constructive heuristics and local search heuristics. A constructive heuristic builds up a feasible solution in a step by step manner by assigning values to the decision variables, which determine a feasible solution. For Arthur Dent's hiking problem, building up a solution may e.g. be done by having knowledge of the direction and the distance to a single hut, which then can be reached by using a compass.

A local search heuristic takes an arbitrary feasible solution and iteratively advances to another feasible solution until some stopping criteria is met. We do not require that a local search heuristic must proceed to better feasible solutions in every iteration, but on a long term sight, this naturally should be the case. The process of advancing to another feasible solution is accomplished by considering the feasible solutions of a neighborhood of the current feasible solution, and by selecting one solution out of this neighborhood as the next current

solution. The neighborhood of a solution reflects the locality of the search. For Arthur Dent's hiking problem, a local search heuristic can be illustrated as follows. The neighborhood of the current solution is given by all nearby huts that are in the visible range from the current one. For Arthur Dent, it is a good choice to proceed to the highest hut in the visible range, which is equivalent to selecting the best solution out of the neighborhood. Arthur Dent stops, if there is no higher hut in the visible range. Then, the current hut must not be the highest hut, but seems to be a good choice.

In general, the efficiency of a local search heuristic is determined by the time the local search heuristic needs, and by the quality of the obtained solutions. Intuitively, we expect that the size of the neighborhood influences both of these aspects.

The goal of this thesis is to develop and examine neighborhoods for some fundamental scheduling problems. We restrict our attention mainly to the performance of certain very large-scale neighborhoods and compare them regarding speed and quality. Before considering these topics, in this chapter, we first give an introduction to the basic notation and definitions. In the following section, we describe basic problems and formalize the intractability of a problem. In Section 1.2 we introduce a measure for the quality of a solution relative to an optimal solution, and we describe different local search methods. Afterwards, in Section 1.3, we introduce the scheduling problems that we are concerned with and show, how feasible solutions for these problems are represented. Finally, in Section 1.4 we formulate the goal of this thesis and discuss its structure.

1.1 Problems and Complexity

In this section, we introduce two different types of problems: decision problems and optimization problems. However, as we will see, there is a connection between both. Moreover, we give a measure for the hardness of solving a problem. We follow standard texts on complexity theory as Garey and Johnson [34] and Papadimitriou and Steiglitz [53].

The general base of both types of problems is the following definition. A *problem* P is given by a set $\mathcal{I}_P$ of *instances* and a set $\mathcal{S}_P(I)$ of *feasible solutions* for each

instance $I \in \mathcal{I}_P$. If there are no ambiguities, we omit the index P.

The first type of problems are decision problems. A *decision problem* P is a problem, for which the set $\mathcal{I}$ of instances is partitioned into a set $\mathcal{I}^+$ of *yes-instances* and a set $\mathcal{I} \setminus \mathcal{I}^+$ of *no-instances*. An instance $I \in \mathcal{I}$ is called a *yes-instance* if and only if the corresponding set of feasible solutions $\mathcal{S}(I)$ is not empty, i.e.

$$I \in \mathcal{I}^+ \text{ if and only if } \mathcal{S}(I) \neq \emptyset.$$

In order to *solve a decision problem*, for a given instance we either have to give a feasible solution or we have to state that no such solution exists.

The 3-*partition problem* is an example of a decision problem. An instance of the 3-partition problem consists of positive integers $a_1, \ldots, a_{3t}$ and b with

$$\begin{aligned} &\frac{b}{4} < a_j < \frac{b}{2},\ j = 1, \ldots, 3t \text{ and} \\ &\sum_{j=1}^{3t} a_j = tb. \end{aligned} \tag{1.1}$$

It is asked, whether there exists a partition of $T = \{1, \ldots, 3t\}$ into pairwise disjoint sets $T_1, \ldots, T_t \subseteq T$, such that

$$\sum_{j \in T_i} a_j = b$$

for all $i = 1, \ldots, t$.

The second type of problems are the optimization problems. An *optimization problem* P is a problem, for which additionally an *objective function* or *measure* $f : \mathcal{S}(I) \to \mathbb{N}$ (or $\mathbb{Q}^+$) is given together with a *goal* for the optimization, which is either to *minimize* or to *maximize* the objective function. The optimization problem is then called a *minimization* or *maximization problem*, respectively. The (positive) value $f(x)$ is called the *objective value* of the feasible solution $x \in \mathcal{S}(I)$. In order to *solve an optimization problem*, for a given instance I, we either have to give a feasible solution $x^* \in \mathcal{S}(I)$ such that there exists no feasible solution with better objective value (regarding the goal), or we have to state that there exists no feasible solution at all. More formally, if an optimal solution x^* exists, for minimization problems the following holds:

$$f(x^*) \leq f(x) \text{ for all } x \in \mathcal{S}(I).$$

For the optimal solution x^* of a maximization problem an analogous condition holds. If such a feasible solution x^* exists, it is called an *optimal solution* and $f(x^*)$ is the corresponding *optimal value*.

The *traveling salesman problem* is an example of an optimization problem. An instance for the traveling salesman problem consists of n cities and a $n \times n$ distance matrix D. Here, d_{ij} denotes the distance between city i and j for $i, j = 1, \ldots, n$ and D is assumed to be symmetric, i.e. $d_{ij} = d_{ji}$. The goal is to find a tour of minimum length along all the cities.

Despite the fact that both, decision problems and optimization problems, are related because they are special cases of a problem, there is a further connection between both. For this, let I be an instance of an optimization problem and t be a given value called *threshold*. The *corresponding decision problem* P_t then asks, whether there exists a feasible solution with objective value not worse than t. For the case of minimization problems, we define the set of feasible solutions $\mathcal{S}_{P_t}(I)$ for the corresponding decision problem P_t and instance I to be:

$$\mathcal{S}_{P_t}(I) := \{x \in \mathcal{S}_P(I) \,:\, f(x) \leq t\}.$$

The set of feasible solutions of the corresponding decision problem of a maximization problem is defined analogously.

A solution approach to solve a problem can be given in the form of an algorithm. An *algorithm* $\mathcal{A}$ is a step by step procedure consisting of instructions. The algorithm takes an input, applies in every step an instruction to this input and if it stops, it produces an output. For the rest of this text, we consider a fixed computational model, which has an instruction set consisting of arithmetic operations, branching operations, comparison operations and memory access operations. This model is related to other computational models (as e.g. the Turing-machine) and is known as a *random access machine*. In order to describe an algorithm we use pseudocode known as "Pidgin Algol" that was presented by Aho et al. [5].

The input and output of an algorithm are both given in the form of a *string*, which is a finite sequence of characters from a finite alphabet. In order to present the input to the algorithm in the form of a string, we have to give an *encoding scheme*. The encoding scheme must encode the input without any superfluous padding, so that the resulting string is as short as possible without any loss of information to represent the input. We consider two different encoding schemes

which mainly differ in the way of encoding integers. The first encoding scheme uses the binary encoding. An integer n encoded in binary results in a string of zeros and ones of length $\lceil \log(n) \rceil$, where $\log(n)$ denotes the logarithm of base 2 and $\lceil x \rceil$ stands for the least integer greater or equal to x. We use as a second encoding scheme the unary encoding. An integer n encoded in unary results in a string of n ones and a trailing zero. Thus, the unary encoding of the integer n has length $n + 1$.

In the following, if I is an input for an algorithm, we denote by $|I|_b$ and $|I|_u$ the length of the string obtained by using a binary encoding scheme or a unary encoding scheme, respectively. For the remainder of the thesis, whenever it is not stated which encoding scheme is used, we are using a binary encoding scheme. A binary encoding scheme is referred to as a reasonable encoding in Garey and Johnson [34]. On the other hand, a unary encoding scheme is not reasonable. This is due to the expensive coding of integers, and hence, the dramatical increase of the length of the input.

The *running time* of an algorithm $\mathcal{A}$ is the number of steps it needs to produce an output for a given input and is measured relative to the length of the input. We say that an algorithm *runs in polynomial time*, if there exists a polynomial p, so that the number of steps the algorithms needs to stop, is at most $p(|I|)$ for any input I, encoded by the chosen encoding scheme. If an algorithm runs in polynomial time, the length of its output is necessarily polynomially bounded in the length of the input. There exist algorithms that run in polynomial time, if a unary encoding scheme is used, but if we switch to a binary encoding scheme, the running time increases beyond polynomial time. Therefore, we say that an algorithm *runs in pseudopolynomial time*, if the running time is polynomially bounded in the length of the input using a unary encoding scheme.

To specify the running time of an algorithm we use asymptotic notation. For given functions $f : \mathbb{N} \to \mathbb{R}_+$ and $g : \mathbb{N} \to \mathbb{R}_+$, we say that $f(n) \in \mathcal{O}\big(g(n)\big)$, if there exist some positive constants c and n_0, such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Thus, the $\mathcal{O}$-notation gives an upper bound on the asymptotic behavior of a function. Using the asymptotic notation, we see that an algorithm $\mathcal{A}$ runs in polynomial time, if there exists some fixed integer k_1, such that $\mathcal{A}$ has running time $\mathcal{O}(|I|_b^{k_1})$. Furthermore, an algorithm $\mathcal{A}$ runs in pseudopolynomial time, if there exists some fixed integer k_2, such that $\mathcal{A}$ has running time $\mathcal{O}(|I|_u^{k_2})$.

An algorithm $\mathcal{A}$ *solves a problem* P, if the algorithm stops for each instance

$I \in \mathcal{I}$ after a finite number of steps and the output either signalizes that I is not a feasible instance, or represents a feasible solution for the considered problem and instance I.

1.1.1 Decision Problems and the Class $\mathcal{NP}$

The algorithms running in polynomial time are deemed to be fast, especially those with low order polynomial asymptotic behavior. A problem is considered to be an *easy problem*, if an algorithm is known, that solves the problem in polynomial time. Hence, we introduce the class $\mathcal{P}$ of easy problems by defining that a decision problem P belongs to the class $\mathcal{P}$, if there exists an algorithm solving the problem in polynomial time.

To define a second class of decision problems, we do not concentrate on solving a problem, but ask whether it is easy to check on the base of some characterization, if a given instance is a yes-instance. For the formal definition of this, we need some notation.

For a problem P, we say that an expression is *polynomially bounded in the size of the instance*, if there exists a polynomial p so that, for all instances $I \in \mathcal{I}$, the expression is at most $p(|I|)$. A *certificate c_I of an instance I* now is a string of length polynomially bounded in the size of the instance.

Using these definitions, we say that a problem P belongs to the class $\mathcal{NP}$, if there exists an algorithm $\mathcal{A}$ and for each instance I a certificate c_I, such that $\mathcal{A}$ applied to I and c_I runs in polynomial time, and $\mathcal{A}$ outputs "yes" if and only if I is a yes-instance. Otherwise, the algorithm responds "no". The algorithm $\mathcal{A}$ is called the *certificate checking algorithm*. The letters $\mathcal{NP}$ stand for Non-deterministic Polynomial (time). The term non-deterministic reflects the circumstance, that the certificate c_I may be guessed and hence, not necessarily is constructed by an algorithm. Notice, that any problem $P \in \mathcal{NP}$ can be solved by complete enumeration. We only have to apply the certificate checking algorithm to every possible certificate. Since a certificate is a string of length polynomially bounded in the size of the instance, there exists a polynomial p, so that there are at most $2^{p(|I|)}$ possible certificates of length at most $p(|I|)$.

The 3-partition problem belongs to the class $\mathcal{NP}$. Consider a certificate checking

algorithm that takes as input an instance and a partition of $T = \{1, \ldots, 3t\}$ into pairwise disjoint sets $T_1, \ldots, T_t \subseteq T$. This algorithm has to check, whether $\sum_{i \in T_k} a_i = b$ for all $k = 1, \ldots, t$.

For a decision problem $P \in \mathcal{P}$ it is obvious that also $P \in \mathcal{NP}$ holds. Since there exists an algorithm solving any instance I of P in polynomial time, we use this algorithm as the certificate checking algorithm (using an empty certificate as input). Therefore, $\mathcal{P} \subseteq \mathcal{NP}$ holds. But the question, whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$ holds, is open. The latter, $\mathcal{P} \neq \mathcal{NP}$, is widely believed to be the case, since there exists the class of $\mathcal{NP}$-complete problems.

In order to give a formal definition for the class of $\mathcal{NP}$-complete problems, we need the notion of a polynomial reduction. For this, consider two decision problems $P, P' \in \mathcal{NP}$. The problem P' *polynomially reduces* to problem P, if there exists a polynomial algorithm that takes as input an instance I' of P' and produces as output an instance I of P, such that I' is a yes-instance of P' if and only if I is a yes-instance of P.

By using the above definition of polynomial reduction, we now say that a problem $P \in \mathcal{NP}$ belongs to the class of $\mathcal{NP}$-complete problems, if for every problem $P' \in \mathcal{NP}$, there exists a polynomial reduction from P' to P.

Polynomial reductions are transitive in the way, that if a problem $P_1 \in \mathcal{NP}$ polynomially reduces to $P_2 \in \mathcal{NP}$ and the problem P_2 polynomially reduces to $P_3 \in \mathcal{NP}$, then P_1 polynomially reduces to P_3. Therefore, in order to show that a problem $P \in \mathcal{NP}$ is $\mathcal{NP}$-complete, we only have to polynomially reduce an $\mathcal{NP}$-complete problem P' to P. What remains is to find a first $\mathcal{NP}$-complete problem. This was done by Cook [27], by proving that any problem $P \in \mathcal{NP}$ polynomially reduces to the satisfiability problem SAT.

Up to this point, we considered a binary encoding scheme. There exist problems which are only $\mathcal{NP}$-complete when using a binary encoding scheme but they become solvable in polynomial time, if a unary encoding scheme is used. Hence, for these problems an algorithm solving every instance of the problem (using a binary encoding scheme) in pseudopolynomial time exists. We call such problems $\mathcal{NP}$-complete *in the weak sense*. In contrast to this, there exist other problems which remain $\mathcal{NP}$-complete even if a unary encoding scheme is used. This type of problems we call $\mathcal{NP}$-complete *in the strong sense*. If for such a problem a pseudopolynomial algorithm would exist, it would immediately follow

that $\mathcal{P} = \mathcal{NP}$.

The 3-partition problem is $\mathcal{NP}$-complete in the strong sense. For a proof, we refer to Garey and Johnson [34]. Since nowadays many problems are known to be $\mathcal{NP}$-complete (see e.g. Garey and Johnson [34] and Karp [45]), and no one discovered an algorithm that polynomially solves an $\mathcal{NP}$-complete problem, it is considered as unlikely that there exists such an algorithm.

1.1.2 Optimization Problems and the Class $\mathcal{NPO}$

We introduce the class $\mathcal{NPO}$ for optimization problems by following Ausiello et al. [8]. An optimization problem P belongs to the class $\mathcal{NPO}$, if for a given instance $I \in \mathcal{I}$ and any string x of length polynomially bounded in the size of the instance, the problem of deciding whether x is a feasible solution can be done in time polynomially bounded in the size of the instance, and if for any feasible solution $x \in \mathcal{S}(I)$, the objective value $f(x)$ can be calculated in time polynomially bounded in the size of the instance.

The traveling salesman problem belongs to the class $\mathcal{NPO}$, since it is easy to check, if a sequence of cities forms a tour, i.e. if all cities are reached exactly once. The corresponding objective value also can be calculated in polynomial time.

Similar to the classes $\mathcal{P}$ and $\mathcal{NP}$, we gather all easy problems of $\mathcal{NPO}$ in a single class. The class $\mathcal{PO}$ is defined to consist of all optimization problems $P \in \mathcal{NPO}$, for which an algorithm exists that finds an optimal solution in polynomial time.

We define the class of $\mathcal{NP}$-hard problems as follows: a problem P is called $\mathcal{NP}$-hard, if a polynomial time algorithm for solving P would imply the existence of a polynomial time algorithm for an $\mathcal{NP}$-complete problem P'. If a problem P remains $\mathcal{NP}$-hard by switching to a unary encoding scheme, we call this problem $\mathcal{NP}$-hard *in the strong sense* and otherwise, it is called $\mathcal{NP}$-hard *in the weak sense*.

The traveling salesman problem is an $\mathcal{NP}$-hard problem, since if there would exist an algorithm solving the traveling salesman problem in polynomial time,

there would also exist a polynomial time algorithm that is capable of solving the $\mathcal{NP}$-complete problem of deciding, if a graph contains a Hamilton cycle (see Garey and Johnson [34]).

1.2 Dealing with $\mathcal{NP}$-hard Problems

The difficulties with $\mathcal{NP}$-hard optimization problems are the lack of algorithms solving such a problem in polynomial time. For smaller instances of a problem it is often easy to obtain optimal solutions. On the other hand, with increasing size of the instances, the time needed to obtain optimal solutions jumps up. It is very unlikely, that there exist algorithms solving $\mathcal{NP}$-hard problems in polynomial time.

In order to cope with $\mathcal{NP}$-hard optimization problems, we disengage from obtaining optimal solutions and consider feasible solutions having objective values close to the optimal value, which possibly can be found faster. We consider one class of methods for obtaining such solutions: local search heuristics.

1.2.1 Approximation

At first, to concretize the notion of "close to optimal" we introduce a measure of the quality of a feasible solution. For an instance $I \in \mathcal{I}$ of an optimization problem, let x^* be an optimal solution and $f(x^*)$ be the corresponding optimal value. The *performance ratio* of a feasible solution $x \in \mathcal{S}(I)$ is the ratio $r(I, x)$ given by

$$r(I, x) := \max\left\{\frac{f(x)}{f(x^*)}, \frac{f(x^*)}{f(x)}\right\}.$$

For the performance ratio, $r(I, x) \geq 1$ holds regardless of the goal of the considered optimization problem. Therefore, the closer the objective value $f(x)$ gets to the optimal value, the closer the performance ratio gets to 1.

An interesting class of algorithms are those leading to feasible solutions for a given optimization problem which do not exceed a certain performance ratio. We say that an algorithm $\mathcal{A}$ has a *performance guarantee* r for an optimization

problem P, if for any instance $I \in \mathcal{I}$ the algorithm delivers a feasible solution $x \in \mathcal{S}(I)$, so that $r(I, x) \leq r$. Such an algorithm is called an r-approximation.

1.2.2 Neighborhoods and Local Search

The idea of a local search heuristic is to define for any feasible solution a set of feasible solutions called the neighborhood. Often, the neighborhood of a feasible solution can be explored in an easy manner compared to solving the overall considered problem. By exploring the neighborhood, we may find a new feasible solution that is better than the current one. By a better solution, we mean a feasible solution that is better regarding the goal of optimization. Then, this method is repeated by exploring the neighborhood of this newly found solution. This is done, until some stopping criterion is met.

In the following, we introduce the concepts of local search and neighborhoods. There are different types of local search heuristics and also different classes of neighborhoods. In order to describe local search we follow the book of Aarts and Lenstra [1].

We concretize the term neighborhood. For this, let P be an optimization problem and let $I \in \mathcal{I}$ be an instance of P. A *neighborhood* or *neighborhood function* is a mapping $\mathcal{N} : \mathcal{S}(I) \to 2^{\mathcal{S}(I)}$, which defines a neighborhood $\mathcal{N}(x)$ for each solution $x \in \mathcal{S}(I)$. Here, with $2^{\mathcal{S}(I)}$ we denote the set of all subsets of $\mathcal{S}(I)$. A solution $x' \in \mathcal{N}(x)$ is called a *neighbor* of x regarding the neighborhood $\mathcal{N}(x)$. The solution x is called *center* of the neighborhood $\mathcal{N}(x)$.

Often, a neighborhood is given by a set of operators. An operator can be applied to a solution $x \in \mathcal{S}(I)$ to slightly change this solution x. Given the operators $op_1, \ldots, op_n$, a neighborhood is defined by $\mathcal{N}(x) := \{op_i(x) : i = 1, \ldots, n\}$.

The neighborhood is a critical issue in a local search heuristic. It directly influences the local behavior of a local search heuristic, since it restricts the choice of a new solution in a single iteration. On the other side, this local behavior also influences the global behavior, i.e. the sequence of feasible solutions obtained by the local search heuristic. This sequence of solutions represents the *navigational behavior* of the local search heuristic. A key element is the size of a neighborhood. The size determines how many choices there are to find a better

solution. This also designates the time the heuristic needs in every iteration. Since with smaller neighborhoods we may not have considered (directly or indirectly) many solutions at the end of a local search heuristic, the size may also have an influence on the solution quality from a global point of view.

Similar to Deĭneko and Woeginger [28] we classify neighborhoods by their size and distinguish two different types. We say that $\mathcal{N}(x)$ is of *polynomial size* if there exists a polynomial p such that $|\mathcal{N}(x)| \leq p(|I|)$ for all instances I and all solutions $x \in \mathcal{S}(I)$. The neighborhoods that are not of polynomial size are called *very large-scale neighborhoods* or *exponential neighborhoods*.

The problem of exploring a neighborhood $\mathcal{N}(x)$ is often an interesting combinatorial problem itself. Regarding the running time needed to explore a neighborhood, we distinguish two different types of neighborhoods. All neighborhoods that can be explored with an algorithm in polynomial time in the worst-case belong to the first type. Especially, all neighborhoods of polynomial size are of this type. The second type contains all other neighborhoods.

Exploring a neighborhood has the goal to return a neighboring solution x' of the neighborhood $\mathcal{N}(x)$, which is better than x regarding the goal of optimization, if such a solution exists. Here, we distinguish different types of exploration. If the algorithm returns the best solution of the neighborhood with respect to the goal, we call the exploration technique *best fit* or *best neighbor* and otherwise, if the algorithm only gives a neighbor x' that is better than the center x of the neighborhood, we call the exploration technique *first fit* or *first (improving) neighbor*. If it takes too long to explore a neighborhood, one can also think of partial exploration techniques, i.e. restrict the search in the neighborhood to the most promising solutions.

An example of a neighborhood, used later on to describe several concepts, is the k-exchange neighborhood for the traveling salesman problem. For a given instance of the traveling salesman problem, the k-exchange neighborhood of a given tour is defined to consist of all the tours that can be obtained by first removing k non-adjacent edges from the tour. In order to get a new feasible solution, we re-insert k edges so that all the edges again form a Hamilton cycle, see Lin [50]. By removing k edges from the tour, we receive k different paths. These k paths have to be reconnected to from a Hamilton cycle. In order to form a Hamilton cycle, we take one of the k paths and fix an endpoint, which is kept unchanged. We now iteratively append the other $k-1$ paths. There, every

Algorithm: Iterative Improvement

input: a feasible solution $x \in \mathcal{S}(I)$ and a neighborhood $\mathcal{N}$
output: a local optimum x
1: **while** x is not a local optimum for $\mathcal{N}(x)$ **do**
2: determine $x' \in \mathcal{N}(x)$ with $f(x') < f(x)$;
3: $x := x'$;
4: **end while**;

Figure 1.1: Algorithm Iterative Improvement for minimization problems.

path can be appended in two different directions. Moreover, the $k-1$ paths can be permuted in $(k-1)!$ ways. Hence, the neighborhood contains $2^{k-1}(k-1)!$ Hamilton cycles. The time needed to examine all neighbors is $\mathcal{O}\left(n^k\right)$ and hence, it is only practical for small k to examine the complete neighborhood.

The basic version of local search is given by *iterative improvement*. Iterative improvement starts with some initial solution (e.g. obtained by a constructive heuristic) and iteratively advances to better solutions in the respective neighborhood of the current solution. Iterative improvement stops if there exists no better solution in the current neighborhood (see Figure 1.1). The solution obtained at the end of an iterative improvement process is a so-called local optimum. Thus, for minimization problems, a solution $x \in \mathcal{S}(I)$ is called a *local optimum* for the neighborhood $\mathcal{N}$, if $f(x) \leq f(x')$ for all $x' \in \mathcal{N}(x)$. For a maximization problem, local optima are analogously defined.

Where iterative improvement stops in a local optimum, there exist other methods that circumvent the problem of getting stuck in a local optimum. The first idea is to use a multistart approach, i.e. restart iterative improvement with many different initial solutions. Another idea is to use a multilevel or iterated local search heuristic. After a local optimum is reached, this method selects a solution from another neighborhood of this local optimum, or perturbs the resulting local optimum slightly. The algorithm is then repeated with this newly obtained solution.

Another approach to escape local optima is simulated annealing. Instead of only accepting improving solutions, it allows to advance to solutions with worse objective value with a certain small probability. This acceptance probability decreases with the number of iterations. Another idea is to lead the search out

of local optima by introducing a tabu list, i.e. solutions to which the algorithm may not advance. This algorithm is called tabu search. A third class of local search heuristics model concepts of nature or biology as genetic algorithms or neural networks. We refer the reader to Aarts and Lenstra [1] for an extensive overview.

1.2.3 Very Large-Scale Neighborhoods (VLSN)

As we have seen in the previous section, the design of neighborhoods is a critical issue for a local search heuristic. In this thesis, we are interested in neighborhoods of large size that can be explored in a fast manner, in order to reach local optima of good quality in less time. In the following, we introduce ideas leading to large neighborhoods.

Very large-scale neighborhoods are often referred to as neighborhoods of exponential size or those, that are cubic (and larger) in the size of the instance. We classify very large-scale neighborhoods according to Ahuja et al. [6]. There, three not necessarily disjoint types of very large-scale neighborhoods are introduced:

- neighborhoods based on variable-depth methods,
- neighborhoods given by network flows,
- neighborhoods resulting from solvable special cases.

Next, we give for every type a general characterization followed by an example.

1.2.3.1 Variable-depth Methods

Variable-depth methods are using a small basic neighborhood in order to determine several small changes to a solution, that are promising and that are in some sense independent from each other, e.g. each small change does not touch portions of the solution altered by the other ones. These small changes are then

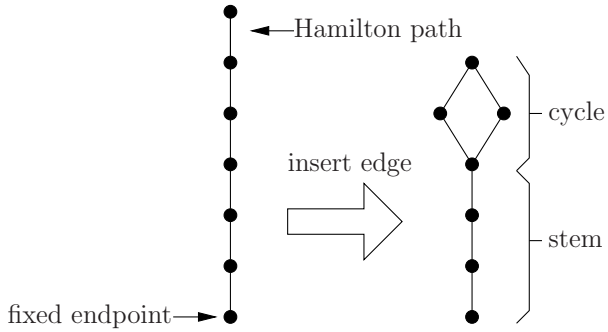

Figure 1.2: A stem and a cycle.

carried out iteratively, and together form a single step in a larger neighborhood. Thus, variable depth-methods heuristically explore this larger neighborhood.

As an example, consider the k-exchange neighborhood for the traveling salesman problem. For larger values of k, it is not practical to use this neighborhood in an iterative improvement algorithm. Variable depth-methods using the k-exchange neighborhood for the traveling salesman problem were introduced by Lin and Kernighan [51]. Applied to the traveling salesman problem, the variable depth-method iteratively breaks a cycle to obtain a Hamilton path. Then, one endpoint is fixed and an edge is inserted at the other endpoint to obtain a stem and cycle structure (see Figure 1.2), i.e. a path leading from the fixed endpoint to one endpoint of the newly inserted edge. This newly inserted edge is part of a unique cycle. In the next iteration, the cycle will be broken at the edge adjacent to one endpoint of the newly inserted edge to obtain a new Hamilton path. Notice, that a Hamilton path is a unique basis for a tour. After k iterations, the best obtained tour is chosen and afterwards, the method is repeated again until some stopping criteria are met, e.g. a local optimum is found.

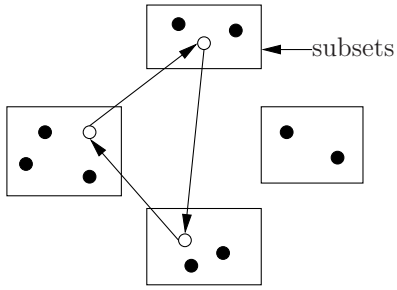

Figure 1.3: Illustrating a cyclic exchange.

1.2.3.2 Network Flows

In network flow based improvement algorithms, an *improvement graph* is given, and the edges of the graph correspond to changes of a solution. The improvement graph heavily depends on the current solution, and structures in this graph comply with the neighbors of the current solution. The best neighbor is determined by the best structure found in the graph. Here, one can think of the following structures:

- Neighborhoods defined by cycles: neighbors arise from subset-disjoint cycles in the improvement graph. Hereby, the vertices of the graph are partitioned into disjoint subsets. A subset-disjoint cycle is a cycle containing at most one vertex of every subset. The subset-disjoint cycle corresponds to a neighbor of the current solution, for which the elements of the disjoint subsets are exchanged along the cycle (see Figure 1.3). A neighborhood of this type is used e.g. for a vehicle routing problem, which is some kind of a multiple traveling salesman problem with additional constraints, see Thompson and Psaraftis [60].
- Neighborhoods defined by paths: the (probably constrained) paths in the improvement graph correspond to the neighbors of the current solution. For example, such a neighborhood is applied by combining independent swaps (see Figure 1.4), as described by Potts and van de Velde [54]. They call their method "dynasearch" and the idea is to use the k-exchange neighborhood on segments that are rearranged by this neighborhood and

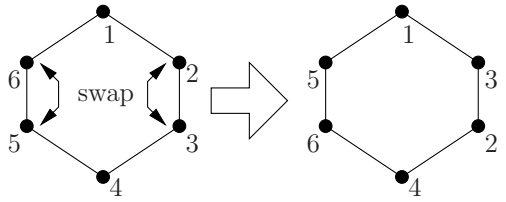

Figure 1.4: Combining independent swaps.

have no edge in common. In this way, they can apply several moves in one iteration of a local search heuristic.

- Neighborhoods defined by assignments and matchings: in neighborhoods of this type, assignments or matchings in the improvement graph correspond to the neighbors of the current solution. An example using assignments in an improvement graph for defining a neighborhood in the traveling salesman problem is given by Gutin et al. [40] and by Punnen [55]. Here, the assignment neighborhood for the traveling salesman problem consists of tours that can be reached by removing a sequence of non-adjacent vertices from the current tour and re-inserting them on the edges of the remaining cycle. Removing a vertex from the tour corresponds to replacing its two adjacent edges by a single edge not adjacent to the considered vertex, so that there remains a cycle in the graph. The resulting improvement graph, needed for determining the best neighbor, is a bipartite graph, and a neighbor corresponds to a matching (see Figure 1.5). The best neighbor can be obtained by determining a minimum weight matching.

1.2.3.3 Solvable Special Cases

Very large-scale neighborhoods can moreover be obtained by restricting the problem under consideration to a special case, for which an algorithm is known solving the special case in polynomial time. An example for the traveling salesman problem are the pyramidal tours used by Carlier and Villon [21] to define a neighborhood. Among the pyramidal tours, the pyramidal tour with minimum length can be obtained in polynomial time, see Gilmore et al. [35]. A tour is

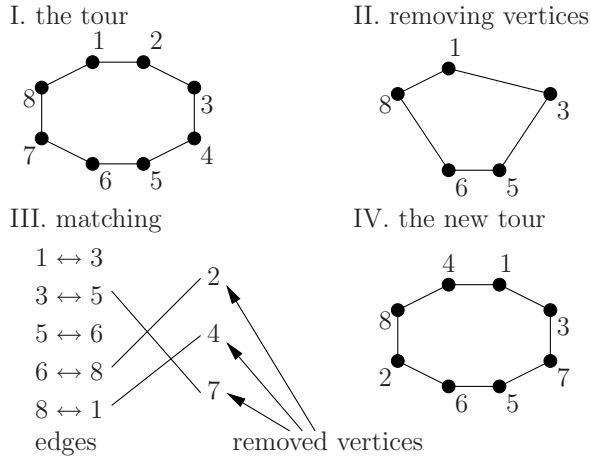

Figure 1.5: Illustrating the matching method.

called *pyramidal*, if it starts in the first city and visits some of the cities in increasing order, until the last city is reached, and finally returns to the first city by visiting all remaining cities in decreasing order. In order to define a neighborhood of a current tour, we rename the cities without loss of generality, so that the current tour is equal to visiting the cities 1 to n in increasing order, and finally traveling from city n to city 1. Then, the neighborhood is defined to consist of all pyramidal tours, and all the pyramidal tours are the so called *pyramidal neighbors*. An improvement graph can be defined in which all pyramidal neighbors of the current tour correspond to paths and a best neighbor can be determined by calculating a shortest path.

1.3 Scheduling Problems and the $\alpha|\beta|\gamma$- Classification

In this section, we briefly introduce the main scheduling problems we consider in this thesis. For a broader overview on scheduling problems we refer, e.g., to the book of Brucker [14].

Generally speaking, a *scheduling problem* is an optimization problem consisting of *machines* which have to process *jobs* while satisfying certain restrictions in order to meet some *quality criteria*. An instance for a scheduling problem is determined by the job and machine data. A solution is given by a *schedule*, that is for each job an allocation of one or more time intervals on one or more machines. If a schedule does not violate any of the restrictions given by the problem or imposed by the instance, it is called a *feasible schedule* and hence, the feasible schedules are the feasible solutions of an instance of the scheduling problem.

To denote certain classes of scheduling problems, the three field classification scheme $\alpha|\beta|\gamma$ introduced by Graham et al. [39] is commonly used. Here, α describes the machine environment, β depicts restrictions on the jobs and γ denotes the objective function. In the following, we introduce the main scheduling problems that will be considered, and we give possible representations of feasible solutions for these problems.

1.3.1 The Problem $1|r_j|\sum C_j$

In Chapter 2 we consider the problem of scheduling n jobs $1,\ldots,n$ with non-negative release dates $r_1,\ldots,r_n$ and processing times $p_1,\ldots,p_n$ on a single machine in order to minimize the sum of job completion times (also called total completion time). A job j is not available before time r_j and needs to be processed for p_j time-units without preemption. This problem is denoted by $1|r_j|\sum C_j$. The considered problem is $\mathcal{NP}$-hard in the strong sense as stated in Lenstra et al. [48]. Without loss of generality, we reorder the jobs such that $r_1 \leq \ldots \leq r_n$ and, if $r_j = r_{j+1}$, that $p_j \leq p_{j+1}$ holds.

A schedule for this problem can be described by a vector $S = (S_1,\ldots,S_n)$ of starting times. It is called a feasible schedule, if and only if

- $S_j \geq r_j$ for $j = 1,\ldots,n$, and
- either $S_j \geq S_i + p_i$ or $S_i \geq S_j + p_j$ for all pairs $i, j = 1,\ldots,n$ with $i \neq j$.

Furthermore, by C we denote the vector of completion times for a feasible schedule S, i.e. $C_j := S_j + p_j$ for $j = 1,\ldots,n$. The goal is to find a feasible

schedule S, such that the objective function

$$f(S) := \sum_{j=1}^{n} C_j \tag{1.2}$$

is minimized.

Without loss of generality, we restrict to feasible schedules that are semi-active, i.e. we restrict to schedules where no job can be processed earlier without changing the processing order or violating the constraints. As a consequence, a feasible schedule for the problem can be characterized by a sequence of the jobs, which represents the processing order of the jobs. For a given sequence $\pi = (\pi(1), \ldots, \pi(n))$, the corresponding feasible schedule S is given by:

$$\begin{array}{lcl} S_{\pi(1)} & := & r_{\pi(1)} \text{ and} \\ S_{\pi(j)} & := & \max\{r_{\pi(j)}, S_{\pi(j-1)} + p_{\pi(j-1)}\} \text{ for } j = 2, \ldots, n. \end{array} \tag{1.3}$$

The calculation of this schedule needs $\mathcal{O}(n)$ time. By using sequences and (1.3) in connection with (1.2), we see that the problem $1|r_j|\sum C_j$ belongs to $\mathcal{NPO}$.

1.3.2 The Problems $P|\,|\sum w_j C_j$, $P|\,|\sum L_i^2$ and $P|\,|C_{\max}$

In Chapters 3 and 4, the problem of scheduling n jobs $1, \ldots, n$ with processing times $p_1, \ldots, p_n$ and weights $w_1, \ldots, w_n$ on m identical parallel machines without preemption is considered. The goal is to find a solution that minimizes the sum of weighted job completion times (also called total weighted completion time). This problem is denoted by $P|\,|\sum w_j C_j$.

The problem $P|\,|\sum w_j C_j$ is $\mathcal{NP}$-hard in the strong sense, as described in Lenstra et al. [48] and problem [SS13] in Garey and Johnson [34]. However, if all weights are equal, the problem becomes easy. In this case, the problem $P|\,|\sum C_j$ can be solved in time $\mathcal{O}(n \log n)$ as a special case of an assignment problem, see e.g. Brucker [14]. The problem remains $\mathcal{NP}$-hard (in the weak sense) for $m \geq 2$, if m is not part of the input (i.e. problem $Pm|\,|\sum w_j C_j$), see Bruno et al. [20] and Garey and Johnson [34]. In the case of having one machine ($m = 1$), the resulting problem $1|\,|\sum w_j C_j$ is solvable in polynomial time by sorting the jobs in order of non-increasing weight to processing time ratios w_j/p_j (Smith's rule, cf. Smith [59]).

A feasible schedule of the problem consists of an *assignment*

$$A : \{1, \ldots, n\} \to \{1, \ldots, m\}$$

of the jobs to the machines and a vector $S = (S_1, \ldots, S_n)$ of starting times of the jobs, such that for all jobs that are processed on the same machine no two jobs overlap, i.e.

$$\text{either } S_j \geq S_i + p_i \text{ or } S_i \geq S_j + p_j$$

for all pairs $i, j = 1, \ldots, n$ with $A(i) = A(j)$ and $i \neq j$. We denote the vector of completion times corresponding to the feasible schedule by C, i.e. $C_j := S_j + p_j$, for $j = 1, \ldots, n$.

We are interested in an assignment A of jobs to machines and a vector S of starting times leading to a feasible schedule, such that the objective function

$$f(S) := \sum_{j=1}^{n} w_j C_j \tag{1.4}$$

is minimized.

Again, without loss of generality, we restrict to feasible schedules that are semi-active, i.e. we restrict to schedules where no job j can be processed earlier without reassigning job j to another machine, changing the processing order on a machine or violating the constraints. Since the jobs scheduled on different machines in a feasible schedule are not influencing each other, the objective function (1.4) splits up into m separate parts for the m machines.

Thus, if an assignment A of jobs to machines is given, the problem decomposes into m independent single machine problems, and an optimal vector S of starting times respecting this assignment can easily be determined: order all jobs processed on the same machine according to Smith's rule. Summarizing, the calculation of the optimal vector S of starting times belonging to a given assignment A and the corresponding objective value $f(S)$ can be done in $\mathcal{O}(n \log n)$, and in $\mathcal{O}(n)$ if an ordering of the jobs according to Smith's rule is already given.

From now on we assume, without loss of generality, that for problem $P|\,|\sum w_j C_j$ the jobs are ordered according to Smith's rule, i.e. such that

$$\frac{w_1}{p_1} \geq \frac{w_2}{p_2} \geq \ldots \geq \frac{w_n}{p_n}$$

holds. Additionally, for two jobs j and k we say that job j has a *higher priority* than job k, if $j < k$. For a given assignment A, we denote by M_i the set of jobs processed by machine i, i.e. $M_i := \{j : A(j) = i\}$. Let $n_i := |M_i|$ be the number of jobs assigned to machine i. Using the machine sets M_i, an optimal vector S of starting times belonging to a given assignment A can be determined by

$$S_j := \sum_{\substack{k \in M_i \\ k<j}} p_k, \text{ for all } j \in M_i \text{ and } i = 1, \ldots, m. \tag{1.5}$$

Based on the above considerations, without loss of generality we restrict to feasible schedules characterized by assignments. We denote by $f(A)$ the objective value of the feasible schedule given by an optimal vector S of starting times belonging to assignment A. By using assignments and (1.5) in connection with (1.4), we see, that problem $P|\,|\sum w_j C_j$ belongs to $\mathcal{NPO}$.

For some argumentation in Chapters 3 and 4, we use partial sums of processing times and weights defined as follows. For a given assignment A, machine i and job j, we denote by L_{ij} the sum of processing times of all jobs assigned to machine i with higher or equal priority compared to job j. Additionally, by W_{ij} we denote the sum of weights of all jobs assigned to machine i with lower priority compared to job j, i.e.

$$\begin{aligned} L_{ij} &:= \sum_{\substack{k \in M_i \\ k \leq j}} p_k, \text{ for all } j = 1, \ldots, n \text{ and } i = 1, \ldots, m, \\ W_{ij} &:= \sum_{\substack{k \in M_i \\ k > j}} w_k, \text{ for all } j = 1, \ldots, n \text{ and } i = 1, \ldots, m. \end{aligned} \tag{1.6}$$

Observe that $L_{A(j),j} = C_j$ holds for all jobs j. For a fixed machine i, the values L_{ij} and W_{ij} can be determined in $\mathcal{O}(n)$ for all jobs $j = 1, \ldots, n$. Additionally, we define the workload of a set M of jobs by

$$L_M := \sum_{j \in M} p_j.$$

For the workload of a machine i we often use L_i with $L_i := L_{M_i}$.

Besides the problem of minimizing the weighted job completion times, we also consider the similar problems of minimizing the $\mathcal{L}_p$-norm of the vector of machine workloads. The main difference to the problem $P|\,|\sum w_j C_j$ is that the

processing order of the jobs on the machines now is of no importance. Thus, without loss of generality we again restrict to feasible schedules characterized by assignments A and seek for an assignment, such that the objective value

$$f_p(A) = \left(\sum_{i=1}^{m} L_i^p \right)^{1/p}$$

is minimized. For any $p > 1$ the problem is $\mathcal{NP}$-hard in the strong sense, see Garey and Johnson [34].

Actually, in Chapter 4, we consider the problem of minimizing the sum of squared machine workloads, i.e.

$$\widetilde{f}(A) = \sum_{i=1}^{m} L_i^2 = f_2(A)^2. \tag{1.7}$$

The problem of minimizing $\widetilde{f}(A)$ arises for example when placing a set of records on a sectored drum in order to minimize the average latency-time (introduced by Cody and Coffman [24, 25]). We denote this problem by $P|\,|\sum L_i^2$. As it will turn out, the two problems using either the objective function $f_2(A)$ or $\widetilde{f}(A)$ are closely related both from an optimization and from an approximation (in the term of performance ratios, see Section 1.2) point of view.

For $p \to \infty$, we get

$$f_p(A) \to \max_{i=1}^{m} L_i.$$

Thus, by considering the $\mathcal{L}_\infty$-norm, we arrive at the problem of minimizing the makespan which was analyzed by Graham [37, 38] and is denoted by $P|\,|C_{\max}$. By *makespan* we denote the completion time of the last job. The makespan $C_{\max}$ corresponding to an assignment A is determined by

$$C_{\max} = \max_{j=1}^{n} C_j = \max_{i=1}^{m} L_i = f_\infty(A).$$

This problem is also $\mathcal{NP}$-hard in the strong sense and is considered in Chapter 5.

1.4 Overview and Contributions

In this thesis, we examine the efficiency of local search. In a local search heuristic there is a local and a global efficiency. The local efficiency is given by the time it takes to explore the neighborhood of the current solution, the quality of solutions in the neighborhood compared to the current solution, and the size of the neighborhood. The global efficiency concerns the number of iterations that are needed to obtain local optima and the quality of these.

The choice of a neighborhood for a local search heuristic is an important matter. The neighborhood influences the local and global efficiency of a local search heuristic. By concentrating on the local efficiency, we develop very large-scale neighborhoods for the introduced scheduling problems and present an efficient way to explore these. We answer the questions which and how many solutions belong to these very large-scale neighborhoods, and how long the exploration takes. From a global point of view, we examine the overall solution quality obtained after a local search by using the introduced neighborhoods. This is done by analyzing the outcome of computational tests, or by giving performance guarantees of local optima, i.e. proving that iterative improvement using the considered neighborhood is an approximation algorithm with some fixed performance guarantee.

In Chapter 2, we present two different types of very large-scale neighborhoods for the single machine problem of minimizing the sum of job completion times. The first arises by combining independent moves, and the best improving neighbor can be obtained by solving a shortest path problem on an appropriate improvement graph. The second very large-scale neighborhood is based on a dominance rule and a special case of the considered problem, which can be solved by sorting. After introducing the neighborhoods, we conduct computational tests to analyze the very large-scale neighborhoods and draw some conclusions on the practical use of them. The ideas and results presented in this chapter are based on joint research with Johann L. Hurink [17].

In Chapter 3, we describe a general approach to obtain very large-scale neighborhoods for the parallel machine problem of minimizing the sum of weighted job completion times. The approach uses an idea of independence similar to that used for the first very large-scale neighborhood in Chapter 2. However, for this neighborhood, we obtain a best improving neighbor by finding a maxi-

mum weight matching in an appropriate improvement graph. A second type of very large-scale neighborhoods is introduced, again arising from a special case. However, this special case is solved with the help of an assignment problem in polynomial time. Moreover, we conduct computational tests to analyze the neighborhoods, and draw some conclusions on the practical use of them. This chapter is based on joint work with Johann L. Hurink [16].

Then, in Chapter 4, we turn to the global efficiency in terms of solution quality obtained at the end of a local search heuristic. We examine a neighborhood introduced in Chapter 3 and we analyze local optima in this neighborhood for the same scheduling problem as considered in Chapter 3, and a special case of this problem. The goal is to obtain a performance guarantee for iterative improvement using this neighborhood. Additionally, we present a performance guarantee for a simple constructive heuristic which uses ideas of local optima. This chapter is based on joint work with Johann L. Hurink and Walter Kern [18].

Finally, in Chapter 5, we consider the problem of minimizing the makespan on parallel machines. For this problem, we introduce a very large-scale neighborhood, again arising from a special case of the problem under consideration. The resulting neighborhood can be explored with the help of an assignment problem. We analyze local optima of this neighborhood, in order to derive a performance guarantee for the iterative improvement using this neighborhood. Afterwards, the same analysis is done for local optima in a neighborhood arising by combining the previous neighborhood with some of the simple neighborhoods analogous to that considered in Chapter 4. This chapter is based on joint work with Johann L. Hurink, Tjark Vredeveld and Gerhard J. Woeginger [19]. We conclude the thesis in Chapter 6 by summarizing the results we obtained.

Chapter 2

VLSN for a Single Machine to Minimize Total Completion Time

In this chapter we present two different approaches for obtaining very large-scale neighborhoods for the problem $1|r_j|\sum C_j$ as introduced in Section 1.3.1.

For this problem, Afrati et al. [2] give a polynomial time approximation scheme (PTAS). They use transformations that simplify an instance without dramatically increasing the objective value in order to show that there exists for every $\varepsilon > 0$ an algorithm that computes a solution with at most a factor of $1+\varepsilon$ away from the optimal solution.

Chu [23] develops a branch and bound algorithm for the problem under consideration. By incorporating a lower bounding scheme received from relaxing the non-preemptive condition and by using already known and new dominance rules, this algorithm is capable of solving instances up to 100 jobs. Yanai and Fujie [63] improve the branch and bound algorithm by introducing additional dominance criteria.

In this chapter, we focus on applying local search to problem $1|r_j|\sum C_j$ using very large-scale neighborhoods. Since for a fixed sequence π there is an efficient method for calculating the best schedule in $\mathcal{O}(n)$, local search may be applied by considering sequences as solutions. The first neighborhood we present is an extension of the adjacent pairwise interchange neighborhood (API). This extension is based on the idea of combining independent operators. Congram et al. [26] and Potts and van de Velde [54] apply the idea of combining independent SWAP-operators to the single-machine total weighted tardiness scheduling problem and the TSP, respectively. They call their approach *iterated dynasearch*. In the paper of Congram et al. [26], the authors show that the size of their neighborhood is $\mathcal{O}(2^{n-1})$ and they give a dynamic programming recursion to find the best neighbor in $\mathcal{O}(n^3)$. Hurink [42] applies combined API-operators in the context of single-machine batching problems and shows that an improving neighbor can be obtained in $\mathcal{O}(n^2)$ by calculating a shortest path in an improvement graph, that is a structure, defined by the possibility of combining API-operators and their change of the objective value.

For problem $1|r_j|\sum C_j$, the independency of changes gets a bit more complicated due to the presence of release dates. Therefore, we first examine in which situations we may combine several API-operators to modify a sequence π describing a solution for $1|r_j|\sum C_j$. We exploit the locality of API-operators, indicating that such an operator causes only small changes to a schedule. It turns out, that the problem of finding a best combined move to a neighboring solution can be solved by calculating a shortest path in an improvement graph, similar to the procedure of Hurink [42]. According to Ahuja et al. [6], this extension of the API-neighborhood belongs to their second category of very large-scale neighborhoods.

The second neighborhood we introduce is based on a dominance rule, that may also be used in a branch-and-bound algorithm for solving the considered problem. This dominance rule states that, for a given solution, the problem is locally not very different from its relaxation $1||\sum C_j$, which can be solved by the *shortest processing time first* (SPT) rule from Smith [59]. This second neighborhood belongs to the third category of Ahuja et al. [6].

The outline of this chapter is as follows. In Section 2.1 we present the $\mathcal{NP}$-hardness proof for the considered problem, which is needed later on in Section 2.2. That section describes the two very large-scale neighborhoods and their main conceptual differences. Afterwards, in Section 2.3 we give some

computational results for these neighborhoods and discuss the possibilities and limitations of the two concepts. Finally, some concluding remarks are given.

2.1 Review on Complexity

In this section, we present the $\mathcal{NP}$-hardness proof for the considered scheduling problem, which is needed in the proof of Theorem 2.3 presented in Section 2.2. In Lenstra et al. [48] it is mentioned, that a reduction of the 3-partition problem to the decision version of the scheduling problem $1|r_j \geq 0|\sum C_j$ can be obtained by adapting the transformation of the knapsack problem to $1|r_n \geq 0|\sum C_j$ presented by Rinnooy Kan [44]. This adaption can be carried out in a straightforward way and leads to a similar construction.

We prove that the 3-partition problem can be reduced pseudopolynomially to $1|r_j \geq 0|\sum C_j$. For the 3-partition problem, we use the notation as introduced in Section 1.1. Additionally, we define $\overline{b} := b + 1$.

An instance of the problem $1|r_j \geq 0|\sum C_j$ corresponding to a given instance of the 3-partition problem is defined as follows. The set of jobs is given by $J = S \cup B$ with

$$\begin{aligned} S &:= \{1, \ldots, 3t\}, \\ B_i &:= \{(i,k) : \ k = 1, \ldots, m\}, \ i = 0, \ldots, t-1, \\ B_t &:= \{(t,k) : \ k = 1, \ldots, M\}, \\ B &:= B_0 \cup \ldots \cup B_t, \end{aligned}$$

where the concrete values of m and M are

$$\begin{aligned} m &:= \frac{3\overline{b}}{2}t(t+1) + 1, \\ M &:= m\frac{3\overline{b}}{2}t(t+1) = \frac{9\overline{b}^2}{4}t^2(t+1)^2 + \frac{3\overline{b}}{2}t(t+1). \end{aligned}$$

The jobs have the following release dates r_j and processing times p_j, $j \in J$:

$$\begin{aligned} r_j &:= 0 \text{ and } p_j := a_j \text{ for } j \in S, \\ r_{(i,k)} &:= i\overline{b} + (k-1)\varepsilon \text{ and } p_{(i,k)} := \varepsilon \text{ for } (i,k) \in B, \end{aligned}$$

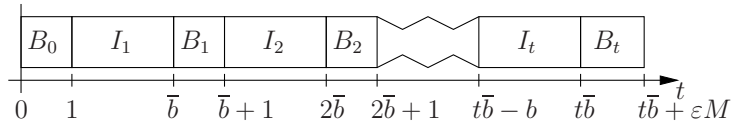

Figure 2.1: Sets B and intervals I

with $\varepsilon := \frac{1}{m}$. It is asked, whether a schedule of the jobs exists with an objective value of at most

$$f^* = \frac{\overline{b}m^2 + 3\overline{b}m}{2m}t^2 + \frac{m^2 - \overline{b}m^2 + 2\overline{b}Mm + 3\overline{b}m + m}{2m}t + \frac{M^2 + M}{2m}.$$

Observe, that if the jobs of all sets B_i are scheduled at their release dates, they leave free time-windows $I_i := [i\overline{b} - b, i\overline{b}[$, $i = 1, \ldots, t$, for the remaining jobs (see Figure 2.1).

Because of the release dates, the contribution f_{B_i} and f_{B_t} of jobs of B_i and B_t to the objective value can be bounded from below as follows:

$$\begin{aligned} f_{B_i} &:= \sum_{k=1}^{m} C_{(i,k)} \geq \sum_{k=1}^{m} \left(r_{(i,k)} + p_{(i,k)}\right) \\ &= \overline{b}mi + \frac{m+1}{2} =: f^*_{B_i}, \text{ for } i = 0, \ldots, t-1, \text{ and} \\ f_{B_t} &:= \sum_{k=1}^{M} C_{(t,k)} \geq \sum_{k=1}^{M} \left(r_{(t,k)} + p_{(t,k)}\right) \\ &= \overline{b}Mt + \frac{M^2 + M}{2m} =: f^*_{B_t}. \end{aligned}$$

With this, we receive for the contribution of jobs of B to the objective value the following lower bound:

$$f_B := f_{B_t} + \sum_{i=0}^{t} f_{B_i} \geq \frac{\overline{b}m}{2}t^2 + \frac{2\overline{b}M - \overline{b}m + m + 1}{2}t + \frac{M^2 + M}{2m} =: f^*_B.$$

Furthermore, if all jobs from $B_0, \ldots, B_t$ start at their release date, we have $f_B = f^*_B$.

Now, assume that we have a solution $T_1, \ldots, T_t$ for the considered instance of the 3-partition problem. In this case we can schedule the jobs of $T_i \subseteq S$ completely in the corresponding window I_i and they contribute at most

$$f_S := \sum_{i=1}^{t} \sum_{j \in T_i} C_j \leq \sum_{i=1}^{t} \sum_{j \in T_i} i\overline{b} = \sum_{i=1}^{t} 3i\overline{b}$$
$$= \frac{3\overline{b}}{2} t(t+1) := f_S^*$$

to the objective value. Furthermore, since the jobs of S fit completely into the time-windows $I_1, \ldots, I_t$, we can schedule the jobs of B at their release dates. Thus, their contribution to the objective value is given by f_B^*. and the objective value of the whole schedule is bounded by $f_S^* + f_B^* = f^*$.

Consider now, that we have an optimal schedule for the jobs of J with objective value of at most $f^* = f_S^* + f_B^*$. We show, that then also the considered instance of the 3-partition problem has a feasible solution. Since all jobs of B have the same processing time, we may assume, that in the optimal schedule the jobs of B are processed according to the order of their release dates.

Now, assume that in the optimal schedule two jobs (i,k) and $(i,k+1)$ from B are not scheduled next to each other. This implies that a subset $\overline{S} \subseteq S$ of jobs is scheduled in between these two jobs. However, since the jobs of $\overline{S}$ have far larger processing times than the jobs of B, and we for the release dates holds $r_{(i,k+1)} = r_{(i,k)} + p_{(i,k)} \leq C_{(i,k)}$, an interchange of job $(i,k+1)$ with $\overline{S}$ reduces the optimal value. As a consequence, all jobs of each set B_i, $i = 0, \ldots, t-1$, are scheduled together in a block of length 1 and the jobs of B_t are scheduled together in a block of length M/m.

In the following, we prove that in each schedule with an objective value of at most $f^* = f_S^* + f_B^*$ all the jobs of B are scheduled at their release dates. Assume that job $(i,1)$ is the first job of B not scheduled at its release date. This implies $S_{(i,1)} = r_{(i,1)} + \delta = i\overline{b} + \delta$, where $\delta \geq 1$, since all jobs from S have integer processing times. This leads to a contribution of the jobs from B_i of at least $f_{B_i}^* + m\delta$. Since

$$m > \frac{3\overline{b}}{2} t(t+1) = f_S^*,$$

this leads to an objective value of more than $f_B^* + f_S^* = f^*$, which is a contradiction.

As a consequence for the optimal schedule, the jobs of S can only be scheduled in the intervals $I_1, \ldots, I_t$ and after $t\overline{b} + \varepsilon M$. If we can prove, that the latter is not possible, we get that the optimal schedule contains a solution for the considered instance of the 3-partition problem.

Assume, there is at least one $j \in S$ with $C_j \geq t\overline{b} + \varepsilon M$. This leads to an objective value of at least $C_j + f_B^* \geq t\overline{b} + \varepsilon M + f_B^*$. Since $\varepsilon = \frac{1}{m}$ and

$$t\overline{b} + \frac{M}{m} > \frac{3\overline{b}}{2}t(t+1) = f_S^*,$$

this contradicts that the objective value of the given schedule is bounded by $f_S^* + f_B^*$.

Thus, the 3-partition problem pseudopolynomially reduces to $1|r_j \geq 0|\sum C_j$. Therefore, the decision problem of $1|r_j \geq 0|\sum C_j$ is $\mathcal{NP}$-complete in the strong sense.

Theorem 2.1 (Lenstra et al. [48]) *The problem $1|r_j \geq 0|\sum C_j$ of finding a schedule with the smallest objective value is $\mathcal{NP}$-hard in the strong sense.*

2.2 Very Large-Scale Neighborhoods

In this section, we present two very large-scale neighborhoods for the problem $1|r_j|\sum C_j$. The neighborhoods rely on two different principles. First, in Section 2.2.1, we build up neighboring solutions by combining several independent pair-interchange operators to one combined neighborhood operator. Next, in Section 2.2.2, we use a reordering of subsequences as the basis of building up a neighborhood structure. Both neighborhoods can have an exponential (in n) number of neighbors and allow efficient exploration. Finally, in Section 2.2.3, we compare the two approaches.

2.2.1 Combined API

In this section, we develop a neighborhood which is based on adjacent-pair interchanges (API). First, we analyze the effects of a single API-operator. We examine which jobs are affected and how the objective function is influenced. Later on, this will be used to combine several API-operators to one combined operator, which results in a neighborhood that may have exponential size and can be searched in polynomial time. Before presenting the mentioned results, we first introduce some notation.

If we consider a schedule S^π, this schedule decomposes uniquely in a set of so-called blocks. Hereby, a block consists of jobs that are scheduled without idle-times, such that the first job of its block starts after an idle period at its release date and all other jobs start at the completion time of their predecessor. More precisely, a *block* is a set of jobs $B := \{\pi(i), \ldots, \pi(i+k)\}$ with $i \in \{1, \ldots, n-1\}$ and $i + k \leq n$, such that the following conditions hold:

- either $i = 1$ or $C_{\pi(i-1)} < S_{\pi(i)}$,
- $C_{\pi(j)} = S_{\pi(j+1)}$ for $j = i, \ldots, i + k - 1$,
- either $i + k = n$ or $C_{\pi(i+k)} < S_{\pi(i+k+1)}$.

A given sequence π of jobs has an unique decomposition into blocks. We denote by $b(\pi)$ the number of blocks and additionally, by $B_1, \ldots, B_{b(\pi)}$ the blocks of the form $B_\beta = \{\pi(i_\beta), \ldots, \pi(i_\beta + k_\beta)\}$, with $i_\beta + k_\beta + 1 = i_{\beta+1}$.

In addition, we denote by g_β the amount of idle-time between the jobs $\pi(i_\beta)$ and $\pi(i_{\beta-1} + k_{\beta-1})$, if $\beta \geq 2$, or the idle-time before the job $\pi(1)$, if $\beta = 1$. This means:

$$g_\beta := \begin{cases} S_{\pi(i_\beta)} - C_{\pi(i_{\beta-1}+k_{\beta-1})} & \text{if } \beta \geq 2, \\ S_{\pi(1)} & \text{if } \beta = 1. \end{cases}$$

In Figure 2.2 an example for blocks and their gaps in a schedule is given.

The neighborhood $\mathcal{N}_{API}$ of a sequence π consists of all sequences received by applying one of the adjacent-pair interchange operators $API_1, \ldots, API_{n-1}$, where

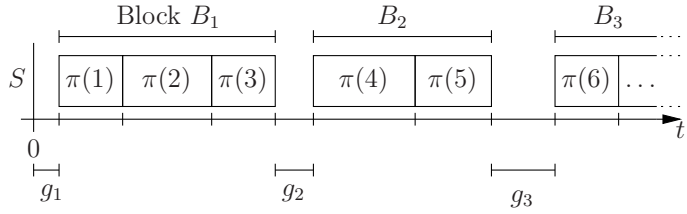

Figure 2.2: Blocks and gaps in a schedule S

the operator API_j interchanges the elements in positions j and $j+1$ of a sequence, i.e.

$$API_j(\pi) := (\pi(1), \ldots, \pi(j-1), \pi(j+1), \pi(j), \pi(j+2), \ldots, \pi(n)).$$

In the following, we examine, how a single API-operator affects a given solution. Considering the two jobs $\pi(j)$ and $\pi(j+1)$ involved in API_j, there are different cases to handle, depending on the position of job $\pi(j)$ in the block and the release date $r_{\pi(j+1)}$ of job $\pi(j+1)$.

Next, consider an operator API_j, where index j belongs to a position of block $B_\beta = \{\pi(i_\beta), \ldots, \pi(i_\beta + k_\beta)\}$. Clearly, if $j = i_\beta + k_\beta$, the application of API_j leads to an increase of the objective value, since the job $\pi(i_\beta + k_\beta + 1)$ is the first job of the next block and, therefore, starts at its release date. Thus, if we are interested in operators API_j, which may lead to better solutions, we only have to consider jobs j which are not the last job of a block. Because of the block-structure, for the schedule S of π holds:

$$S_{\pi(j)} + p_{\pi(j)} = S_{\pi(j+1)}, \text{ for all } j \in \{i_\beta, \ldots, i_\beta + k_\beta - 1\}.$$

In order to calculate the consequences of the exchange of the jobs $\pi(j)$ and $\pi(j+1)$, let S' be the schedule corresponding to $API_j(\pi)$. Furthermore, let

- $d_1 := S'_{\pi(j)} - S_{\pi(j)}$.
- $d_2 := S_{\pi(j+1)} - S'_{\pi(j+1)}$,
- $d_3 := C'_{\pi(j)} - C_{\pi(j+1)}$.

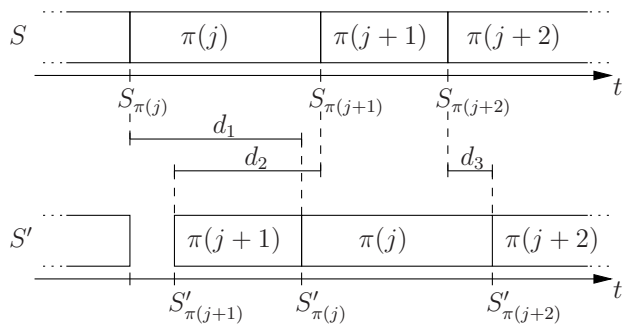

Figure 2.3: Effect of $API_j(\pi)$

Herewith, d_1 describes the absolute value of the change in starting time of job $\pi(j)$, d_2 gives the change for job $\pi(j+1)$ and d_3 presents the effect of the exchange for the succeeding jobs $\pi(j+2), \ldots, \pi(n)$ (see Figure 2.3). The value d_2 is given by

$$d_2 = \begin{cases} \min\{p_{\pi(j)}, S_{\pi(j+1)} - r_{\pi(j+1)}\}, & \text{for } j \neq i_\beta, \text{ and} \\ \min\{p_{\pi(j)} + g_\beta, S_{\pi(j+1)} - r_{\pi(j+1)}\}, & \text{for } j = i_\beta. \end{cases}$$

Furthermore, by scheduling job $\pi(j)$ directly after the finishing of job $\pi(j+1)$, the starting time $S'_{\pi(j)}$ of job $\pi(j)$ becomes $S'_{\pi(j)} = S_{\pi(j)} + p_{\pi(j)} - d_2 + p_{\pi(j+1)}$. Taking into account the release date of job $\pi(j)$, this leads to

$$d_1 = \max\{p_{\pi(j)} + p_{\pi(j+1)} - d_2, r_{\pi(j)} - S_{\pi(j)}\}.$$

Based on these considerations, d_3 becomes:

$$d_3 = d_1 - p_{\pi(j+1)}.$$

If $j > i_\beta$, d_3 is always greater than or equal to 0. However, if $j = i_\beta$, d_3 may also be negative due to the gap g_β before job $\pi(i_\beta)$. The parameter d_3 is useful to calculate the effects of API_j on the jobs $\pi(j+2), \ldots, \pi(n)$.

Applying operator API_j changes the objective value by

$$\begin{aligned}\delta_j := f(S') - f(S) &= \sum_{\mu=1}^{n}(S'_{\pi(\mu)} + p_{\pi(\mu)} - S_{\pi(\mu)} - p_{\pi(\mu)}) \\ &= d_1 - d_2 + \sum_{\mu=j+2}^{n} S'_{\pi(\mu)} - S_{\pi(\mu)}.\end{aligned} \tag{2.1}$$

To calculate δ_j, it remains to calculate

$$penalty := \sum_{\mu=j+2}^{n} S'_{\pi(\mu)} - S_{\pi(\mu)}.$$

As mentioned before, the effects of API_j on the jobs $\pi(j+2), \ldots, \pi(n)$ are characterized by d_3. If $d_3 = 0$, then $penalty := 0$. Otherwise, not only jobs of B_β but also jobs of subsequent blocks may be involved. If $d_3 > 0$, all jobs $\pi(\mu)$ with $j+2 \leq \mu \leq i_\beta + k_\beta$ have to be shifted by d_3 units in order to keep a feasible schedule. This results in $penalty := d_3(i_\beta + k_\beta - j - 1)$. If $d_3 > g_{\beta+1}$, the shift also affects the next block. Thus, we have to shift all jobs $\pi(i_{\beta+1}), \ldots, \pi(i_{\beta+1} + k_{\beta+1})$ of the next block by $d'_3 := d_3 - g_{\beta+1}$, resulting in $penalty := penalty + d_3 k_\beta$. Such a shifting of complete blocks has to be repeated until the gap is sufficiently large.

On the other hand if $d_3 < 0$, we have to shift jobs to the left in the new schedule, beginning with job $\pi(j+2)$. The involved jobs are only the jobs $j+2, \ldots, i_\beta + k_\beta$ and the calculation of *penalty* has to be done by calculating for all these jobs their new starting times by applying (1.3). Note, that in this case the *penalty* is negative and the block B_β may split into several new blocks.

All in all, the effects of such an API-operator are computable in time $\mathcal{O}(n)$. However, on the average, we expect a much lower running time. The neighborhood $\mathcal{N}_{API}$ has size $\mathcal{O}(n)$. Hence, we need in the worst case $\mathcal{O}(n^2)$ to compute the best solution in $\mathcal{N}_{API}$.

In the following, we investigate the possibility of combining API-operators. More precisely, we search for pairs of operators API_i and API_j, where the consecutive application of these two operators to a sequence π leads to a change $\delta_i + \delta_j$ in the objective value. Such an independence of operators allows to evaluate the effects of a combined execution of the operators based on the effects

of the single operators and, therefore, allows a combined execution of several different APIs in one iteration of the local search heuristic leading to a combined neighborhood. To find candidates for combined operators, we have to analyze, which APIs do not have an effect on each other. In the remaining part, we denote by $\circ$ the combination of operators, i.e. $API_i \circ API_j(\pi)$ is equal to $API_i(API_j(\pi))$.

Consider indices i and j with $1 \leq i < j \leq n-1$ and $i+2 \leq j$. We are interested in those cases, where the effect of $API_j \circ API_i(\pi)$ and $API_i \circ API_j(\pi)$ is equal to $\delta_i + \delta_j$, i.e. we look for indices i and j where

$$f(S^\pi) + \delta_i + \delta_j = f(S^{API_j \circ API_i(\pi)}).$$

A sufficient condition for this independency is that after applying API_i to π, the resulting schedule around the jobs in position j and $j+1$ must be the same as in π. To formalize this, we introduce a variable F_i denoting the first position after $i+1$ where the application of API_i to π has no effect.

If, by applying $API_i(\pi)$, the interval which is occupied by the jobs $\pi(i)$ and $\pi(i+1)$ does not change, we have $F_i = i+2$. On the other hand, if the starting time of job $\pi(i+2)$, or the idle period before $\pi(i+2)$ is changed by applying API_i to π, we determine the index of the last affected job

$$l_i := \max\{k : i \leq k \leq n,\ S^\pi_{\pi(k)} \neq S^{API_i(\pi)}_{\pi(k)}\},$$

and we define $F_i := l_i + 2$ (we have to add 2, since there might be a change in the amount of idle-time before $\pi(l_i+1)$ after applying API_i). The value for l_i can easily be determined during the calculations of *penalty*.

Based on the above considerations, we call API_j *π-independent of* API_i if

- neither $\pi(i)$ nor $\pi(j)$ is a last job of its block, and
- $j \geq F_i$.

We call API_i and API_j *π-independent* if either API_j is π-independent of API_i or API_i is π-independent of API_j.

Summarizing, for two π-independent operators API_i and API_j we have

$$f(S^\pi) + \delta_i + \delta_j = f(S^{API_j \circ API_i(\pi)}).$$

An important property of π-independency is its *transitivity*. If, for $i < j < k$, API_j is π-independent of API_i and API_k is π-independent of API_j, then API_k is also π-independent of API_i. Thus, we call a set $M \subseteq \{1, \ldots, n-1\}$ of indices π-*independent* if the API-operators belonging to the elements of M are pairwise π-independent. Hence, for a given π-independent set $M := \{v_1, \ldots, v_k\}$ we can calculate the objective value of the schedule $API_{v_k} \circ API_{v_{k-1}} \circ \ldots \circ API_{v_1}(\pi)$ by

$$f(S^{API_{v_k} \circ API_{v_{k-1}} \circ \ldots \circ API_{v_1}(\pi)}) = f(S^{\pi}) + \sum_{i \in M} \delta_i.$$

We define the neighborhood $\mathcal{N}_{CAPI}$ (Combined Adjacent-Pair Interchange) of a sequence π as all possible sequences resulting from applying all possible combinations of π-independent operators to π. This neighborhood at least contains all API-operators, i.e. $\mathcal{N}_{API}(\pi) \subseteq \mathcal{N}_{CAPI}(\pi)$, but in general there may be an exponential number of sequences in $\mathcal{N}_{CAPI}(\pi)$.

In the following, we develop an efficient method to calculate a set of independent operators that gives the best gain in the objective value over all possible independent operators. For this, we define a structure called improvement graph, which depends strongly on the given sequence π.

For a given sequence π, let $G(\pi) = (V, A(\pi))$ be a directed graph with vertices $V = \{0, 1, \ldots, n-1, *\}$ and a set $A(\pi) \subseteq V \times V$ of directed arcs, where each arc $(i, j) \in A(\pi)$ receives a cost c_{ij}. The vertices 0 and $*$ are called source and sink, respectively. The set $A(\pi)$ contains the following arcs.

- arcs $(0, i)$ with cost $c_{0i} = \delta_i$, for all $1 \leq i \leq n-1$, where $\pi(i)$ is not a last job of a block in S^{π},
- arcs (i, j) with cost $c_{ij} = \delta_j$, for all $1 \leq i < j \leq n-1$, where API_j is π-independent of API_i,
- arcs $(i, *)$ with cost $c_{i*} = 0$, for all $0 \leq i \leq n-1$.

Observe, that there are no directed cycles in the graph.

An arc leading to a vertex $i \leq n-1$ corresponds to an application of the operator API_i. Furthermore, a directed path $P = (0, v_1, \ldots, v_k, *)$ from the

source to the sink, with $1 \leq v_i \leq n-1$, corresponds to a combined operator $API_{v_k} \circ API_{v_{k-1}} \circ \ldots \circ API_{v_1}(\pi)$ of π-independent operators, and the sum of the costs of the arcs on the path describes the gain to the objective value. Hence, if we have a shortest directed path from the source to the sink, this determines the best possible combined operator of π-independent operators.

Because our graph has no directed cycles, we can use Dijkstra's algorithm to obtain a shortest-path in this graph. With this algorithm, we are able to calculate the best possible combined operator of π-independent API_i operators in $\mathcal{O}(n^2)$.

2.2.2 Peaks and Valleys

In this section, we develop a very large-scale neighborhood based on a dominance criterion for the problem $1|r_j|\sum C_j$. Recall, that for the release dates of the jobs holds $r_1 \leq \ldots \leq r_n$. The base of this approach is that each schedule defines in a unique way so-called peaks and valleys. Here, jobs with high indices form the peaks in the sequence and the indices of the jobs between two peaks form a valley.

More precisely, the peaks of a given sequence π are a unique set of indices $1 = i_1 < \ldots < i_k < i_{k+1} = n+1$, with

$$\begin{array}{rll} \pi(j) < & \pi(i_\mu), & \text{for } j = i_\mu + 1, \ldots, i_{\mu+1} - 1, \\ \pi(i_\mu) < & \pi(i_{\mu+1}), & \text{for } \mu = 1, \ldots, k. \end{array} \tag{2.2}$$

Furthermore, we have $\pi(i_k) = n$.

From now on let π be a given sequence of jobs and $i_1, \ldots, i_k, i_{k+1}$ be the peaks belonging to π. We define sets $V_\mu := \{\pi(i_\mu+1), \ldots, \pi(i_{\mu+1}-1)\}$ for $\mu = 1, \ldots, k$. Observe, that

$$\bigcup_{\mu=1}^{k} V_\mu \cup \{\pi(i_1), \ldots, \pi(i_k)\} = \{1, \ldots, n\}.$$

The sets V_μ are called *valleys* and contain the jobs between the peaks. In the following lemma, we show three important properties for the jobs of a valley in the schedule S^π.

Lemma 2.2

(P1) In the schedule S^π, the job $\pi(i_\mu)$ together with the jobs of a valley V_μ are scheduled without idle times, i.e.

$$S^\pi_{\pi(k)} = C^\pi_{\pi(k-1)}, \text{ for } k = i_\mu + 1, \ldots, i_{\mu+1} - 1.$$

(P2) Within the set of all sequences resulting from π by a reordering of the jobs of a valley V_μ, the sequence obtained by reordering these jobs by non-decreasing processing times has minimal objective value.

(P3) Reordering the jobs of $V_\mu \cup \{\pi(i_\mu)\}$ by non-decreasing processing times does not increase the objective value.

PROOF.

(P1) The properties of the peaks (2.2) imply that for all $l \in \{i_\mu+1, \ldots, i_{\mu+1}-1\}$ holds $l < i_\mu$, i.e. $r_{\pi(l)} \leq r_{\pi(i_\mu)}$. This again implies $C_j \geq r_i$, for all $j, i \in V_\mu$. Furthermore, this leads to $S^\pi_{\pi(l)} = \max\{r_{\pi(l)}, C^\pi_{\pi(l-1)}\} = C^\pi_{\pi(l-1)}$, for all $l \in \{i_\mu + 1, \ldots, i_{\mu+1} - 1\}$.

(P2) Due to (P1) of the lemma, the jobs of V_μ are scheduled in S^π in the interval

$$I = [S^\pi_{\pi(i_\mu+1)}, S^\pi_{\pi(i_\mu+1)} + \sum_{j \in V_\mu} p_j].$$

And furthermore, we have $r_j \leq S^\pi_{\pi(i_\mu)}$, for all $j \in V_\mu$. Thus, reordering the jobs of V_μ still allows to schedule the jobs of V_μ in I. Due to Smith's rule, a reordering by non-decreasing processing times is best possible.

(P3) Due to (P2) of the lemma, we first may reorder the jobs of V_μ by non-decreasing processing times, without increasing the objective value. Since $r_j \leq r_{\pi(i_\mu)}$, interchanging job $\pi(i_\mu)$ with jobs of V_μ, which have a smaller processing time, leads to a decrease of the objective value. □

Consider a given sequence π, with peaks $i_1, \ldots, i_k, i_{k+1}$ and valleys $V_1, \ldots, V_k$. Based on (P2) of the lemma it is possible to find a best sequence respecting

the peaks and valleys in time $\mathcal{O}(n \log n)$, by simply sorting the jobs of V_μ by increasing processing times. If we allow a change in the peaks and valleys, we even might get a better solution by sorting the whole sets $V_\mu \cup \{\pi(i_\mu)\}$. In the latter case, it may happen that the new sequence does not have the same peaks and valleys as before and, therefore, again may be optimized by re-sorting the valleys.

In summary, if peaks $i_1, \ldots, i_k, i_{k+1}$ and valleys $V_1, \ldots, V_k$ are given, we can easily find an optimal sequence π that respects these peaks and valleys. However, as the next theorem shows, it is not easy to find an optimal sequence π respecting given positions $i_1, \ldots, i_k, i_{k+1}$ of the peaks and corresponding jobs $\pi(i_1), \ldots, \pi(i_n)$, without the knowledge of the valleys.

Theorem 2.3 *Let $1 = i_1 < \ldots < i_k < i_{k+1} = n + 1$ be a given set of integers and $\pi(i_1) < \ldots < \pi(i_k)$ the jobs to be scheduled on the positions $i_1, \ldots, i_n$. The problem of completing the sequence π so that the objective function $\sum C_j$ is minimized and*

$$\pi(j) < \pi(i_\mu), \text{ for } j = i_\mu + 1, \ldots, i_{\mu+1} - 1,$$

is $\mathcal{NP}$-hard in the strong sense.

PROOF. The proof can be given in the same way as the proof that the problem $1|r_j|\sum C_j$ is $\mathcal{NP}$-hard in the strong sense. We identify the positions of the jobs of B as the indices $i_1, \ldots, i_k$ and the corresponding jobs of B as the peaks. Then finding a schedule respecting these peaks with objective value less than f^* solves the 3-partition problem. □

Based on the properties of peaks and valleys given in Lemma 2.2, a second very large-scale neighborhood called *PAV* (Peaks And Valleys) is defined. Given a solution π with corresponding peaks and valleys we define the neighborhood $\mathcal{N}_{PAV}$ of a sequence π to contain all re-orderings of the valleys. Hence, $\mathcal{N}_{PAV}$ may have an exponential size depending on the instance and the sequence. According to (P2) of Lemma 2.2, the best neighbor in $\mathcal{N}_{PAV}$ can be determined by sorting the jobs of the valleys by non-decreasing processing times in $\mathcal{O}(n \log n)$.

The neighborhood $\mathcal{N}_{PAV}$ is not directly suited for an iterative improvement

process. If the best neighbor in $\mathcal{N}_{PAV}$ is determined, we are in a local optimum, i.e. (P2) of Lemma 2.2 does not yield any further improvement. In order to circumvent this, (P3) of Lemma 2.2 can be applied to the best sequence of $\mathcal{N}_{PAV}$ to obtain a different peak and valley structure. The peak in front of a valley has to be inserted in the valley on the base of its processing time (which can be realized in linear time). This whole process can be repeated and stops when all peaks have a processing time smaller or equal to the minimum processing time in their valley.

2.2.3 Comparison of the Neighborhoods

The two approaches, $\mathcal{N}_{CAPI}$ together with the block structure and $\mathcal{N}_{PAV}$ with its peaks and valleys, are related. The jobs that belong to a block are processed without idle-times as well as the jobs of a valley defined by the peaks. The difference results from the fact that valleys for two adjacent peaks may be processed without idle-time in between, whereas two blocks are always separated by an idle-time. Therefore, there are at least as many valleys as there are blocks for a given sequence π, but there may be more.

However, the underlying ideas to develop the very large-scale neighborhoods are quite different. The neighborhood $\mathcal{N}_{CAPI}$ is built upon the simple neighborhood $\mathcal{N}_{API}$ and has in principle the same navigational behavior as that neighborhood, i.e. a local search heuristic using the neighborhood $\mathcal{N}_{API}$ likely advances to the same solutions as an approach using the neighborhood $\mathcal{N}_{CAPI}$ does. The only difference with $\mathcal{N}_{API}$ is that the API-operators are not chosen and executed sequentially, but in parallel. Thus, from a quality point of view, we may expect from $\mathcal{N}_{CAPI}$ only a better behavior than from $\mathcal{N}_{API}$, if the parallel choice fits better to the problem. From a computational point of view, both neighborhoods have a worst case complexity of $\mathcal{O}(n^2)$ to compute a best neighbor. But since the chosen operator of $\mathcal{N}_{CAPI}$ may contain several API-operators, one may suppose that the total time to reach a certain solution quality may be shorter for algorithms using $\mathcal{N}_{CAPI}$. Both of these aspects are investigated via computational tests which are reported in the next section.

The neighborhood $\mathcal{N}_{PAV}$ is based on a local priority criterion, which allows the interchange of two jobs under certain conditions (first job has larger processing time and after the interchange the first scheduled job does not start later).

Thus, in principle this neighborhood also relies on $\mathcal{N}_{API}$. However, in contrast to $\mathcal{N}_{CAPI}$ we do not restrict to independent operators but allow a complete reordering of certain subsets of jobs. This may indicate that an algorithm using $\mathcal{N}_{PAV}$ is able to reach (good) local optima in short time. On the other hand, when a local optima has been reached, $\mathcal{N}_{PAV}$ may not be a good choice to proceed further using this neighborhood, since within this neighborhood a job is interchanged only with successors with smaller processing time; i.e. we have a monotonous behavior. Note that, under $\mathcal{N}_{API}$, also an interchange with a succeeding job with larger processing time may lead to an improving neighbor. Again, computational results have to give insight in these questions.

2.3 Computational Results

In this section we report on computational experiments to indicate how the two approaches perform regarding solution quality and running time for small and large instances. Furthermore, for smaller instances we compare the results with optimal solutions found by a branch-and-bound algorithm. We use the branch-and-bound algorithm of Yanai and Fujie [63] to obtain these exact solutions. As an initial heuristic we use, besides some simple methods, the *APRTF-heuristic* which was presented by Chu [23].

The APRTF-heuristic iteratively extends a partial schedule by

- either scheduling a non-planned job having minimal earliest starting time (ties broken by choosing the job with minimal processing time),
- or scheduling a non-planned job having minimal sum of earliest starting time and earliest completion time.

The second option is chosen, if either the release date of the corresponding job is smaller than or equal to the earliest starting time of the job corresponding to the first option, or if some dominance criterion is fulfilled. For details, we refer to the work of Chu [23]. The two simple initial heuristics are called *RSORT* and *RND*. The first sorts all jobs by non-decreasing release dates and ties are broken by choosing a job with smallest processing time. The second sorts the

jobs randomly. APRTF is superior to RSORT regarding solution quality and RSORT again retrieves better results than RND.

To get some insight in the quality of the obtained solutions, we use a lower bounding scheme LB resulting from a relaxation of the problem by allowing preemption. In this case, the problem becomes solvable in time $\mathcal{O}(n \log n)$ by the *shortest remaining processing time* (SRPT) priority rule, see Baker [10]. The optimal solution of the relaxed problem given by the SRPT-rule has been shown by Ahmadi and Bagchi [4] to be a good lower bound for the considered problem regarding running time and quality.

The problem instances are generated as described by Yanai and Fujie [63] and Chu [23]. The processing times are randomly chosen between 1 and 100. The release dates are generated between 0 and $\frac{101n\lambda}{2}$, where λ is a parameter indicating the density of the problem in terms of how the jobs are distributed w.r.t. their release dates, and n denotes the number of jobs. The tests done by Yanai and Fujie [63] show, that the hardest instances are those for $0.6 \leq \lambda \leq 1.2$. To get an idea about the influence of λ, we show in Figure 2.4 the average optimal values of 100 randomly generated instances with $n = 100$ jobs and different values of λ. With a constant number of jobs, the optimal objective value increases with λ, due to the increasing range of release dates.

In order to get an indication about the effectiveness of the neighborhoods in practice, we implemented them in ANSI-C with an iterative improvement method. We have chosen not to use tabu search or simulated annealing since we are interested in the structural behavior of the neighborhoods and not in the potentials of local search methods. In the following, we denote with *BAPI* (*best* API), *CAPI* and *PAV* the iterative improvement process using the neighborhood $\mathcal{N}_{API}$, $\mathcal{N}_{CAPI}$ and $\mathcal{N}_{PAV}$, respectively. Hereby, each method advances in every iteration to the best solution in the corresponding neighborhood, as introduced. Additionally, because the best solution of the neighborhood $\mathcal{N}_{PAV}$ constitutes a local optimum, we use the earlier described method to receive a new peak and valley structure in each iteration of PAV. As a fourth approach we combine PAV and BAPI. This means that we apply PAV until we receive a local optimum and, afterwards, advance to the best improving neighbor in $\mathcal{N}_{API}$ (i.e. applying one BAPI-move). This process is repeated until no further improvement is possible. This algorithm we call PAVBAPI. The computational tests were done on a PC with an Intel Pentium IV processor running at 2.4 GHz.

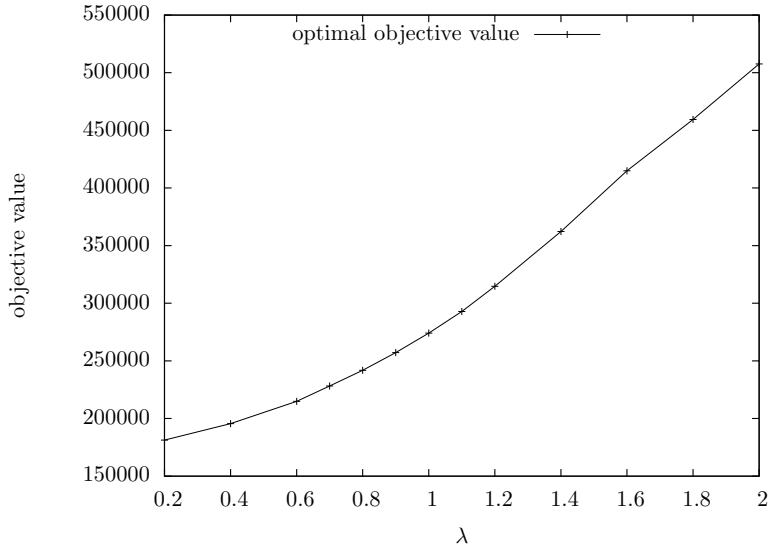

Figure 2.4: Average objective values of optimal solutions.

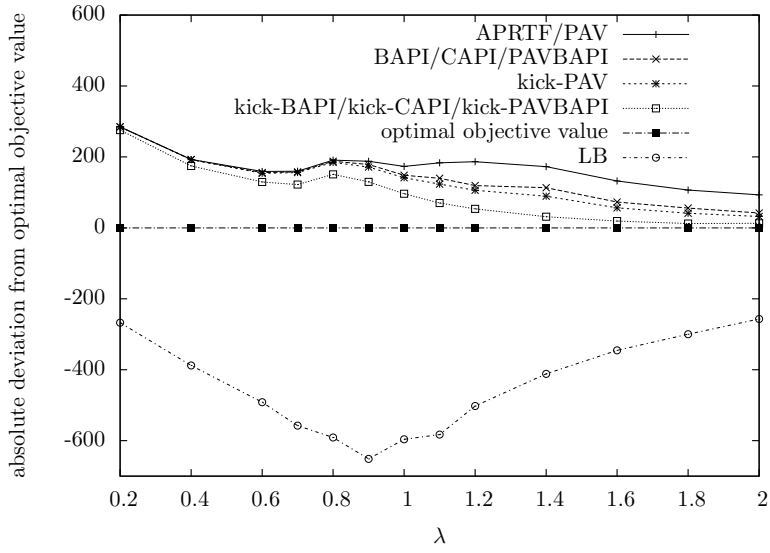

Figure 2.5: Iterative improvement using APRTF to obtain initial solutions.

In a first series of tests, we use the APRTF-heuristic as initial solution for the iterative improvement methods. In Figure 2.5 the performances of the considered approaches are given, besides the results of some other approaches which are described further on. For $n = 100$ and different values of λ, the average absolute deviation of the objective values of the solutions to the optimal value for 100 randomly generated instances is given. Since a solution received by the APRTF-heuristic is mostly locally optimal in the neighborhood $\mathcal{N}_{PAV}$, the values obtained by PAV are almost identical in average to the values of APRTF. The figure also shows, that BAPI, CAPI and PAVBAPI perform nearly the same on average. The corresponding solutions are similar in structure and there is not much difference to the initial solution. This may be caused by the fact that APRTF delivers often a local optima or a solution of good quality.

BAPI and CAPI arrive at solutions of similar quality. This indicates that, for the problem $1|r_j|\sum C_j$, the possibility of the very large-scale neighborhood $\mathcal{N}_{CAPI}$ to combine several moves and to look at their overall performance does not help to get solutions different to solutions obtained by using the neighborhood $\mathcal{N}_{API}$. Because BAPI also delivers solutions of similar quality compared to PAVBAPI, it seems that mainly BAPI-moves are applied in PAVBAPI, if an initial solution of good quality is used.

To get more insight in the navigational behavior of the neighborhoods around solutions of good quality, we added an extra component to the local search heuristic. After iterative improvement stops, we give the resulting local optima a kick and restart the iterative improvement procedure. This kick simply takes one of the jobs not starting at its release date and reinserts it at a position in the sequence so that this job will start at its release date. By doing so, we slightly perturb the solution and arrive at a different but not necessarily better solution compared to the original local optimum. Then we again start iterative improvement possibly arriving at a better solution. We apply the kick to every job where it is possible, and the best received solution from this kick followed by iterative improvement is taken as next solution. We iterated this process and stop if no kick to a job followed by iterative improvement leads to a better solution. In the following, we call the corresponding methods *kick-BAPI*, *kick-CAPI*, *kick-PAV* and *kick-PAVBAPI*.

The average solution quality received by using the kick method is also presented in Figure 2.5. Here one can see that the kick method has in general a large impact on solution quality. Although PAV was hardly able to improve the

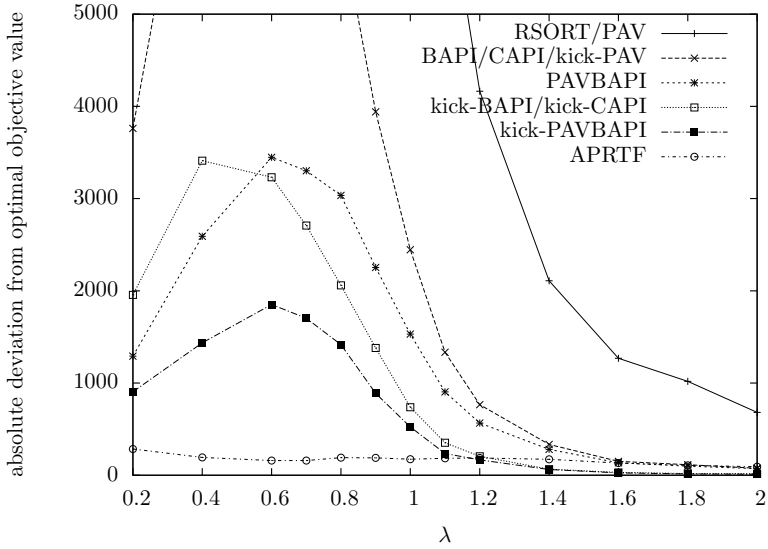

Figure 2.6: Iterative improvement using RSORT to obtain initial solutions.

initial solution given by the APRTF-heuristic, it now improves this solution considerably. The other methods kick-BAPI, kick-CAPI and kick-PAVBAPI perform nearly the same, but much better than kick-PAV.

The above mentioned test gives some indications of the navigational behavior of the neighborhoods in regions of high quality solutions. In a further series of tests, we investigate how they behave if a weak initial solution obtained by RSORT is chosen. In Figure 2.6 we compare the initial solutions and local optima with the optimal solution for the generated instances. Due to the nature of PAV, an initial solution received by RSORT already constitutes a local optimum, and therefore, PAV performs as weak as RSORT. The methods provide solutions of bad quality the smaller λ gets (for $\lambda = 0$, RSORT gives optimal solutions). The shape of the curve for BAPI/CAPI/kick-PAV is similar to the other ones, except that the maximum is reached for $\lambda = 0.6$ at an average objective value of roughly 9000. By looking at $\lambda \geq 0.6$, the methods BAPI and CAPI perform the same and this time PAVBAPI delivers solutions of better quality compared to BAPI. By using the kick methods we get the same ranking between the different neighborhoods, except that the differences to the optimal values become smaller.

Finally, we conducted tests using RND to obtain initial solutions. The initial solutions are of a very bad quality and the differences to the optimal solutions are large. The simple methods BAPI, CAPI and PAV (as well as PAVBAPI) very soon get stuck in a local optima and, thus, deliver solutions of bad quality. BAPI, CAPI and PAVBAPI perform superiorly over PAV alone and give solutions of comparable quality. They reduce the gap of the initial solutions to the optimum by roughly 50%. In Figure 2.7 we only present the results using the kick methods. The solutions of the simple methods are far off limits as well as kick-PAV for smaller values of λ. We again get a situation, where kick-BAPI and kick-CAPI perform the same and slightly better than kick-PAVBAPI. One observation is, that the kick methods are successful in improving on randomly generated initial solutions.

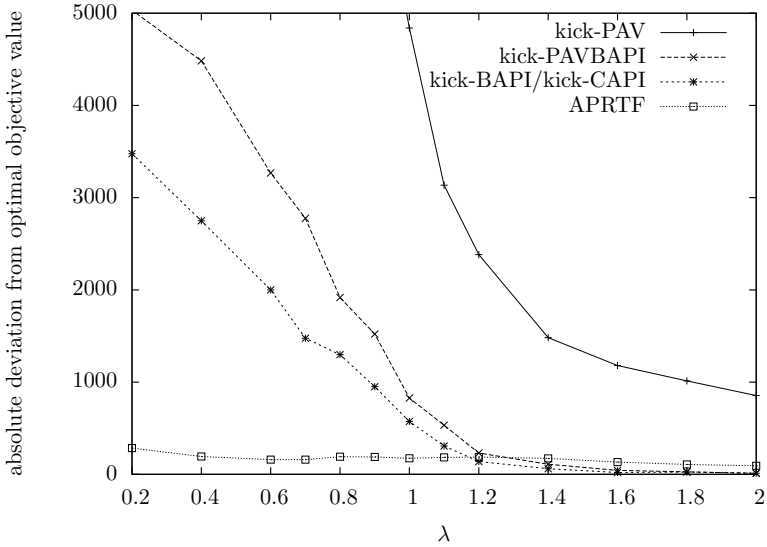

Figure 2.7: Iterative improvement using RND to obtain initial solutions.

The additional tests confirm that the navigational behavior of the very large-scale neighborhood $\mathcal{N}_{CAPI}$ is not better than that of the underlying basic neighborhood $\mathcal{N}_{API}$. Taking always the best solution of the neighborhood $\mathcal{N}_{API}$ in general does not cut off possible moves which are contained in a best solution of $\mathcal{N}_{CAPI}$. Furthermore, a priority based very large-scale neighborhood is not very successful by itself. However, to incorporate these priorities in other

neighborhoods (either via switching between the neighborhoods or by directly incorporating them) seems to be a good idea. The presented results till now only give a qualitative judgment of the neighborhoods. Although the very large-scale neighborhoods were not superior in this direction, compared to $\mathcal{N}_{API}$, they still may be efficient if they reduce the computational effort. Therefore, in the following we investigate this aspect in more detail.

n	BAPI	CAPI	PAV	PAVBAPI
100	0.018	0.022	0.000	0.002
200	0.159	0.242	0.000	0.030
500	2.837	6.244	0.000	0.300
1000	24.715	77.086	0.003	1.594

Figure 2.8: Average running time per instance averaged over λ.

In Figure 2.8 the average running time is presented for iterative improvement using an initial solution received by RND on instances with 100, 200, 500 and 1000 jobs. The outcome is averaged over all considered values for λ. It can be seen that the running times for CAPI are roughly twice as long as BAPI, and PAV and PAVBAPI are fast compared to the other methods.

n	BAPI	CAPI	PAV	PAVBAPI
100	1669	1466	7	249
200	7037	6439	9	657
500	46674	44291	11	2195
1000	195104	189052	12	5272

Figure 2.9: Average number of iterations per instance averaged over λ

Thus, also in this aspect, the very large-scale neighborhood $\mathcal{N}_{CAPI}$ does not outperform its simple counterpart $\mathcal{N}_{API}$. To get an explanation for this, we calculate the average number of iterations needed for iterative improvement to reach a local optimum (see Figure 2.9) and the average number of API-moves which were contained in one CAPI-move for $n = 1000$ and different values of λ (see Figure 2.10). The results show that almost the same number of iterations are needed for BAPI and CAPI. Furthermore, CAPI is not able to significantly combine several API-moves.

Finally, we calculated, for $n = 1000$, in how many cases the improvement of a solution in the combined approach PAVBAPI was achieved by BAPI (see

λ	0.6	0.8	0.9	1.0	1.1	1.2	1.4	1.6	1.8	2.0
API	1.02	1.02	1.03	1.03	1.03	1.04	1.05	1.05	1.06	1.07

Figure 2.10: Average moves per iteration in the *CAPI*-neighborhood

λ	0.4	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.4	1.6
%	67.9	69.5	69.0	69.1	69.2	70.4	70.3	70.9	72.3	72.5

Figure 2.11: Average percentage of cases in which improvement is made by BAPI in PAVBAPI.

Figure 2.11). The average need for BAPI increases for higher values of λ because of the structure of theses instances. In order to rearrange a local optimum with respect to $\mathcal{N}_{PAV}$ by BAPI-moves to achieve a solution which is no longer locally optimal with respect to $\mathcal{N}_{PAV}$, in average more than one step of BAPI is needed. Hence, it takes some iterations using BAPI before PAV is able to improve the given solution again. This is especially the case for instances with higher values of λ. These instances have a wider range of release dates, and hence it is more likely to receive empty valleys. And if a non-empty valley is obtained by BAPI, mostly either the valley and the corresponding peak are already sorted by processing times or it would not improve the objective value, if this sorting is done by PAV.

2.4 Concluding Remarks

The very large-scale neighborhood $\mathcal{N}_{CAPI}$ obtained by combining independent API-moves does not automatically lead to solutions of better quality. Also the hope for a faster running time of iterative improvement because of the execution of several moves at once was not fulfilled. In fact, $\mathcal{N}_{CAPI}$ is not able to combine enough moves in any testing to beat $\mathcal{N}_{API}$ regarding computational time.

Furthermore, the second very large-scale neighborhood $\mathcal{N}_{PAV}$ alone is not able to deliver good results. This neighborhood is only useful for improving "unstructured" feasible solutions to some better solution in a very fast way. It does not succeed on solutions of good quality since these solutions have a near optimal structure regarding $\mathcal{N}_{PAV}$. But, in combining $\mathcal{N}_{API}$ and $\mathcal{N}_{PAV}$, iterative

improvement is able to improve an initial solution of bad quality in a fast way in comparison to the stand-alone neighborhood $\mathcal{N}_{API}$ and reaches solutions of comparable quality. Therefore, $\mathcal{N}_{PAV}$ is only useful in combination with other neighborhoods.

Since iterative improvement is not a clever local search heuristic and depends strongly on the initial solution, we added a simple restart technique by perturbing the received local optima. By doing this we observed a vast increase in the solution quality. Hence, we expect that with other more elaborated methods like simulated annealing or tabu search it will be possible to receive even better local optima with the two introduced neighborhoods.

The results of this chapter confirm the conclusions drawn by others on the practical use of very large-scale neighborhoods (see e.g. Hurink [42]). Very large-scale neighborhoods are not good on beforehand for making local search efficient. Based on our experiences, we may conclude that the size of the neighborhood does not guarantee a better quality. Only if structural properties of the considered problem make it useful to combine different neighborhood operators, very large-scale neighborhoods may be successful. This means that these combined neighborhood operators lead to different and better solutions as a sequence of (greedy) chosen neighborhood steps in the underlying basic neighborhood (for our problem, this was not the case!). On the other hand, large neighborhoods resulting from dominance rules are not useful as stand alone methods. Only in combination with other neighborhoods, they may help to improve the quality.

A second possible advantage of very large-scale neighborhoods consisting of combined operators of a basic neighborhood can be a speed up in computational time. Based on our results, we may conclude that this can only be the case if most of the executed combined neighborhood operators combine several basic neighborhood operators, and if the extra computational effort for searching the combined neighborhood in comparison with the basic neighborhood is not large.

All in all, we suggest to develop and use very large-scale neighborhoods of the considered types only if problem specific properties or computational arguments on beforehand give an indication that the very large-scale neighborhoods have some potential to be a success.

Chapter 3

VLSN for Parallel Machines to Minimize Total Weighted Completion Time

In this chapter different approaches are presented for obtaining efficiently searchable very large-scale neighborhoods for the problem $P||\sum w_jC_j$, as introduced in Section 1.3.2.

A polynomial time approximation scheme (PTAS) is presented by Skutella and Woeginger [58]. They use transformations that simplify an instance without dramatically increasing the objective value in order to show that there exists for every $\varepsilon > 0$ an algorithm that computes a solution with at most a factor of $1 + \varepsilon$ away from the optimal solution. Sahni [56] developed an FPTAS (fully PTAS) for the problem $P_m||\sum w_jC_j$ with fixed m, which runs in time bounded by a polynomial in the input size and $1/\varepsilon$.

Belouadah and Potts [13] describe a branch and bound algorithm for the problem $P||\sum w_jC_j$. For receiving lower bounds, they use Lagrangian relaxation. Since the difficulty of the problem results from the existence of more than one machine, they use Lagrangian relaxation on the constraint, that at most m jobs may

be processed by the system during any unit time interval. Additionally, for the branching steps they use dominance rules introduced by Elmaghraby and Park [31]. Their algorithm is capable of solving instances of up to 30 jobs and 8 machines on a CDC 7600 computer in about 60 seconds. Due to the nature of the algorithm, computational time heavily increases with the number of machines.

A different approach is used by van den Akker et al. [7]. They formulate the problem as a set-covering problem with an exponential number of binary variables. The idea is to assign to every machine exactly one set of jobs. There are 2^n different sets of jobs for the machines and, hence, this gives a huge number of columns for the integer linear program and its relaxation. The relaxed linear program is solved by column generation where the pricing algorithm for determining entering columns runs in pseudo-polynomial time, i.e. in $\mathcal{O}(n\sum p_j)$ time and space. If a solution for the relaxed linear program is integral, this constitutes an optimal solution, which happens quite often especially for instances of size comparable to those the algorithm of Belouadah and Potts [13] is able to solve. If otherwise the solution is fractional, they branch by imposing either a deadline or a release date for some job whose average completion time using the (fractional) job-sets is bigger than the minimal completion time over all used job-sets. This strategy can easily be incorporated in the pricing algorithm without changing the running time and space. Their computational testing indicated that the algorithm is capable of solving instances from Belouadah and Potts [13], in one fourth of the time that algorithm would need. Moreover, the column generated approach is not as sensitive as the approach of branch and bound to increasing values of m. As a consequence, it is able to solve instances of size up to $n = 100$ jobs and $m = n/10$ machines in at most an hour on a HP 9000/710 computer, which is about twice as fast as the CDC 7600.

Barnes and Laguna [12] introduce a tabu search algorithm for this problem. They are using a combined neighborhood of job-insertions (an job-insertion we call *move*) and swaps, i.e. they are working on assignments of jobs to machines and change the assignment of one job or exchange the assignments of two jobs. Their computational experiments point out that this method is rather successful in delivering near-optimal solutions. Agarwal et al. [3] develop a very large-scale neighborhood based on moving jobs from one machine to another. In principle, they allow sequences of moves with decreasing job priorities. If in a first part of such a sequence a job j has been moved to a different machine and is inserted there directly before a job k, in the remaining part of the sequence only moves

of jobs with lower priority than k are allowed. Agarwal et al. [3] introduce an improvement graph for this neighborhood, which is searched heuristically and hence, forms a variable depth search algorithm. They run computational tests using this neighborhood in local search frameworks as iterative improvement, tabu search and iterated local search (i.e. perturbing local optima by some random swap of jobs). They conclude that an iterated local search heuristic with several runs using randomly generated initial solutions delivers the best results, regarding running time and solution quality.

The neighborhoods we present consist mainly of matchings in a certain improvement graph. The first neighborhood describes an approach to receive a very large-scale neighborhood by combining independent operators. We examine which properties such operators must have. It turns out, that the problem of finding a best combination of independent operators can be solved by calculating a maximum weight matching in a general graph. Afterwards, we introduce several operators that are suited as the base for the very large-scale neighborhood. According to the work of Ahuja et al. [6], this neighborhood belongs to the second category of very large-scale neighborhoods.

The second neighborhood we introduce is based on a restricted version of the considered scheduling problem. The restricted version turns out to be a special case of an assignment problem which seeks a minimum weight perfect matching in a complete bipartite graph. The best neighbor of the second neighborhood can be determined by solving this restricted scheduling problem. The second neighborhood belongs to the third category of Ahuja et al. [6].

In Section 3.1 and Section 3.2 the two methods to obtain very large-scale neighborhoods are introduced. Afterwards, in Section 3.3 we give computational results for these neighborhoods and discuss the possibilities and limitations of the concepts. Finally, some concluding remarks are given.

3.1 Neighborhoods by Combining Independent Moves

In the following, we introduce a first approach leading to very large-scale neighborhoods of exponential size (in the number of machines) for the problem $P||\sum w_jC_j$. The basic principles presented are not only restricted to neighborhood search for the problem $P||\sum w_jC_j$ but are also adoptable for similar problems, which have the property, that the machines are independent (parallel); e.g. $Q||\sum w_jC_j$ and $R||\sum w_jC_j$.

The main idea behind the presented approach to build very large-scale neighborhoods is to start with a rather simple basis neighborhood $\mathcal{N}_1$ and to build a new very large-scale neighborhood $\mathcal{N}_2$ by allowing combinations of operators from the neighborhood $\mathcal{N}_1$. To be able to evaluate the neighborhood $\mathcal{N}_2$, only combinations of operators which are independent are allowed. The presented approach to build the very large-scale neighborhood is general in the sense that it can be applied to all basis neighborhoods $\mathcal{N}_1$, which have some stated properties. These properties and the proposed construction of the neighborhood $\mathcal{N}_2$ are given in the following.

The basis neighborhood $\mathcal{N}_1$ has to consist of operators $op(i_1, i_2)$ which operate on pairs (i_1, i_2) of different machines (i.e. $i_1 \neq i_2$). Hence, $\mathcal{N}_1$ contains up to $\frac{1}{2}m(m-1)$ neighbors. The operators have to fulfill the following properties:

- the change resulting from the application of an operator $op(i_1, i_2)$, for a fixed pair $i_1 \neq i_2$, is only dependent on the given schedules of the two machines i_1 and i_2, and has only effects on the machines i_1 and i_2,
- the operator is symmetric, i.e. the assignments $A' := op(i_1, i_2)(A)$ and $A'' := op(i_2, i_1)(A)$ are equal.

In the following, we denote by $\circ$ the combination of two operators $op(i_1, i_2)$ and $op(i_3, i_4)$, i.e. $op(i_1, i_2) \circ op(i_3, i_4)(A)$ is equal to $op(i_1, i_2)(op(i_3, i_4)(A))$. The two operators $op(i_1, i_2)$ and $op(i_3, i_4)$ are defined to be *independent* if $i_j \neq i_k$ for $j \neq k$; i.e. the two machine pairs are disjoint. The first property guarantees that for a set of pairwise independent operators it does not matter in which sequence these operators are applied to a given solution. Furthermore, since the

objective function (1.4) splits up in m separate parts for the m machines, the first property also guarantees that for a set of pairwise independent operators the change in the objective value is additive. More precisely, if we apply a set of pairwise independent operators $op(s_1, t_1), \ldots, op(s_k, t_k)$ to a solution A the resulting change in the objective value is given by

$$f(A) - f(op(s_k, t_k) \circ \ldots \circ op(s_1, t_1)(A)) = \sum_{l=1}^{k} \delta_{s_l,t_l}(A),$$

where $\delta_{i_1,i_2}(A) := f(A) - f(op(i_1, i_2)(A))$ denotes the improvement of the objective value resulting from the application of $op(i_1, i_2)$ to assignment A (if the value is negative it actually is a worsening operator for A).

All basis neighborhoods $\mathcal{N}_1$ having the above properties can be used as the base to design an exponential (in m) neighborhood $\mathcal{N}_2$. The neighborhood $\mathcal{N}_2$ of an assignment A consists of all assignments obtained by applying a set of pairwise independent operators to the assignment A. In the following, we treat the neighborhood $\mathcal{N}_2$ in more detail. We describe how a best neighbor in this neighborhood can be obtained and give some bound on the size of the neighborhood.

The operators of the neighborhood $\mathcal{N}_2$ can be represented by matchings. To achieve this, we consider the following weighted graph $G(A) = (V, E, c(A))$, where

- the vertex set V contains a vertex for each machine (i.e. the graph contains m vertices),
- an edge $e = \{i_1, i_2\} \in E$, with $i_1 \neq i_2$, connecting vertices i_1 and i_2 represents the operator $op(i_1, i_2)$ (and due to the above symmetry property also the operator $op(i_2, i_1)$),
- an edge $e = \{i_1, i_2\}$ gets a weight $c(A)_{i_1,i_2} = \delta_{i_1,i_2}(A)$, representing the change in the objective value resulting from applying the operator $op(i_1, i_2)$ to assignment A.

The weights of the graph are calculated by applying the operator $op(i_1, i_2)$ for every pair of machines i_1, i_2, with $i_1 < i_2$, to assignment A; i.e. the complexity

of building up the graph is $\frac{1}{2}m(m-1)$ times the complexity of evaluating the effects of a single operator $op(i_1, i_2)$.

A matching $\mathcal{M}$ in graph $G(A)$ is given by a subset of edges $\mathcal{M} \subseteq E$, such that no two edges have a vertex in common. Therefore, each matching corresponds to a set of pairwise independent operators of $\mathcal{N}_1$ and, thus, to an operator of $\mathcal{N}_2$ and vice versa. Furthermore, the weight $w(\mathcal{M})$ of a matching $\mathcal{M}$ is given by the sum of the weights of all edges present in the matching, i.e.

$$w(\mathcal{M}) := \sum_{\{i_1,i_2\}\in\mathcal{M}} c(A)_{i_1,i_2} = \sum_{\{i_1,i_2\}\in\mathcal{M}} \delta_{i_1,i_2}(A).$$

Again, this weight $w(\mathcal{M})$ is equal to the change in the objective value resulting from applying the operator of $\mathcal{N}_2$ belonging to the matching $\mathcal{M}$. Hence, we can determine the best neighbor of an assignment A in neighborhood $\mathcal{N}_2$ by calculating a maximum weight matching $\mathcal{M}$ in the graph $G(A)$. Observe, that the structure of the graph $G(A)$ is independent of the considered assignment A but the weights heavily depend on it.

Determining a maximum weight matching in a general graph with $|V|$ vertices and $|E|$ edges can be done in different ways. There exists a $\mathcal{O}\left(|V|^3\right)$-algorithm from Gabow [33] and Lawler [47] extending the work of Edmonds [30]. Using this algorithm, the best neighbor in $\mathcal{N}_2$ can be determined in $\mathcal{O}\left(m^3\right)$.

In order to give bounds on the number of neighbors of an assignment A in the neighborhood $\mathcal{N}_2$, we have to give a bound on the number $\mathcal{M}_m$ of different matchings in a complete graph K_m with m vertices:

- the number $\mathcal{M}_m^*$ of maximal cardinality matchings in K_m is given by
$$\mathcal{M}_m^* = \frac{m!}{l!2^l} > \left(\frac{\sqrt{m}}{2}\right)^m,$$
where $l := \lfloor \frac{m}{2} \rfloor$
- a matching of size $k \leq \lfloor \frac{m}{2} \rfloor$ in K_m exists of $m-2k$ isolated vertices and a maximal cardinality matching in a complete subgraph of size $2k$. Therefore, the number $\mathcal{M}_m^k$ of matchings of size k in K_m is given by
$$\mathcal{M}_m^k = \begin{pmatrix} m \\ m-2k \end{pmatrix} \cdot \mathcal{M}_{2k}^* = \begin{pmatrix} m \\ 2k \end{pmatrix} \cdot \mathcal{M}_{2k}^*.$$

- summing this up over all k yields

$$\mathcal{M}_m = \sum_{k=0}^{l} \left(\begin{array}{c} m \\ 2k \end{array} \right) \cdot \frac{(2k)!}{k!2^k} = \sum_{k=0}^{l} \frac{m!}{k!(m-2k)! \cdot 2^k}.$$

Summarizing, the neighborhood $\mathcal{N}_2$ consists of an exponential (in m) number of neighbors, whereby each neighbor represents the results achieved from applying a set of independent operators of the basis neighborhood $\mathcal{N}_1$, and is efficiently searchable. In the following, we introduce some examples for the basis neighborhood $\mathcal{N}_1$, which have the stated properties to build up the neighborhood $\mathcal{N}_2$.

3.1.1 Move-Neighborhood

A first example of a basis neighborhood $\mathcal{N}_1$ is built up of move operators $op^{move}(i_1, i_2)$. Hereby, $op^{move}(i_1, i_2)$ considers the machines i_1 and i_2 of the given assignment A and moves exactly one job between these two machines in such a way that the change in the objective value is best possible. Obviously, these operators have the stated property for the basis neighborhood $\mathcal{N}_1$. It remains to describe how these operators $op^{move}(i_1, i_2)$ can be realized efficiently.

In principle, an operator $op^{move}(i_1, i_2)$ represents a best possible move in a neighborhood consisting of operators $move(j, i)$, that reassign job j to machine i, where

- for j only jobs currently assigned to machine i_1 or i_2 are allowed,
- for i only the machines i_1 and i_2 are allowed, and
- for i only the machine where j currently is not assigned to is allowed.

In the following, we study the effects of a single operator $move(j, i)$ and show how, on the basis of these results, the effect of an operator $op^{move}(i_1, i_2)$ can be calculated.

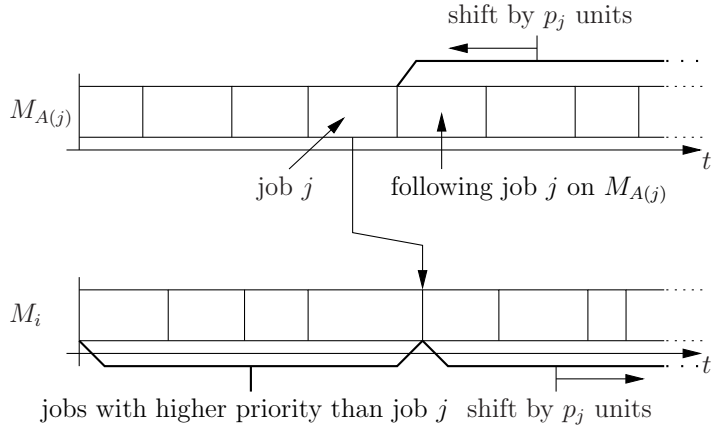

Figure 3.1: Illustration of $move(j,i)(A)$.

If a job j is deleted from its machine $A(j)$, the completion times of all jobs following job j on machine $A(j)$ decrease by p_j units. These jobs have a lower priority than job j. Thus, the deletion of job j lowers the objective value by $\delta_1 = p_j W_{A(j),j}$. Inserting job j on the target machine i increases the completion times of all jobs following job j on machine i by p_j units. Theses jobs have a lower priority than job j. The insertion of job j raises the objective value by $\delta_2 = p_j W_{ij}$, (see Figure 3.1 for an illustration).

Taking into account the change in completion time of job j, the overall change in the objective value $\delta^A_{move(j,i)} := f(A) - f(move(j,i)(A))$, resulting from an application of $move(j,i)$ to assignment A, is given by

$$\begin{aligned}\delta^A_{move(j,i)} &= \delta_1 - \delta_2 + w_j\left(C_j - p_j - L_{ij}\right) \\ &= p_j\left(W_{A(j),j} - W_{ij}\right) + w_j\left(L_{A(j),j} - L_{ij} - p_j\right).\end{aligned} \tag{3.1}$$

Thus, if the corresponding values for L and W from (1.6) are known, $\delta^A_{move(j,i)}$ can be calculated in $\mathcal{O}(1)$.

For the operator $op^{move}(i_1,i_2)$ we now have to find the best move of a job between the two machines i_1 and i_2. This can be achieved by evaluating all moves $move(j,i_2)$ for jobs j with $j \in M_{i_1}$ and all moves $move(j,i_1)$ for jobs j with $j \in M_{i_2}$. Since in a preprocessing, the relevant values for L and W can be cal-

culated in $\mathcal{O}(n)$, the overall complexity to evaluate the operator $op^{move}(i_1, i_2)$ is $\mathcal{O}(n)$. For the neighborhood $\mathcal{N}_2$ we have to evaluate all operators $op^{move}(i_1, i_2)$ with $i_1 < i_2$ to build up the graph $G(A)$. This can be realized in $\mathcal{O}(nm)$.

The neighborhood $\mathcal{N}_2$ contains an exponential (in m) number of neighbors. Each of these neighbors again dominates a certain number of neighbors w.r.t. the neighborhood defined by $move(j, i)$ operators (we call this neighborhood $\mathcal{N}_0$). More precisely, a matching $\mathcal{M}$ containing the edges $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$ in $G(A)$ leads to an assignment A', with

$$A' = op^{move}(s_1, t_1) \circ op^{move}(s_2, t_2) \circ \cdots \circ op^{move}(s_k, t_k)(A).$$

Each edge $\{s_l, t_l\}$ can be understood as an operator $op^{move}(s_l, t_l)$ applied to assignment A. Observe, that $op^{move}(s_l, t_l)$ dominates $n_{s_l} + n_{t_l}$ operators in the neighborhood $\mathcal{N}_0$. Therefore, the considered matching $\mathcal{M}$ using operator $op^{move}(s_l, t_l)$, for each edge $\{s_l, t_l\}$, represents the best solution in a neighborhood of size $\prod_{l=1}^{k}(n_{s_l} + n_{t_l})$. In contrast to this, neighborhood $\mathcal{N}_1$ (i.e. move a job between a pair of machines i_1 and i_2) contains only $\frac{1}{2}m(m-1)$ neighbors, where each neighbor represents the best solution in a neighborhood of size $n_{i_1} + n_{i_2}$.

3.1.2 Combining Several Moves

A second class of examples for a basis neighborhood $\mathcal{N}_1$ can be obtained by not only moving one job from a machine to another, but swapping two jobs between a pair of machines. Of course, this can be generalized by allowing a fixed number of moves between a pair of machines in one step. In the next part we introduce the foundations to evaluate such operators efficiently. A general method is introduced to calculate the effects to the objective value of applying operators exchanging several jobs between machines i_1 and i_2, where i_1 and i_2 are fixed.

In order to combine several move-operators, we first consider assignments A and A_1, where assignment A_1 was obtained from applying several move-operators to assignment A. We want to know in which situations the values $\delta^A_{move(j,i)}$ can still be used to calculate the objective change resulting from applying $move(j, i)$ to assignment A_1. Recall, that the values $\delta^A_{move(j,i)}$ denote the change in the

objective value by moving job j to machine i in assignment A. Clearly, it makes only sense to consider the case that job j has not changed its machine, i.e. we restrict ourselves to assignments A_1 in which job j is still processed by the same machine as in assignment A.

Consider a job j processed by machine $i_1 := A(j)$ in assignment A and A_1. We want to move job j to machine i_2, i.e. we want to apply $move(j, i_2)$ to A_1. By emanating from assignment A to assignment A_1, insertions and deletions of jobs with higher priority than job j may have happened on machine i_1. This influences the completion time of job j. The change in completion time of job j, by emanating from assignment A to A_1, is given by

$$\Delta L_{i_1} := C_j^{A_1} - C_j^A = L_{i_1,j}^{A_1} - L_{i_1,j}^A. \tag{3.2}$$

Furthermore, by emanating from assignment A to A_1, insertions and deletions of jobs with lower priority than job j may have happened on machine i_1. This also influences the move of job j to machine i_2 in assignment A_1. In order to take this into account, we define the value ΔW_{i_1} to denote the change of weight of jobs with low priority on machine i_1. The value ΔW_{i_1} is calculated as

$$\Delta W_{i_1} := W_{i_1,j}^{A_1} - W_{i_1,j}^A. \tag{3.3}$$

Next, we concentrate on the target machine i_2. To calculate the effect of moving job j to machine i_2 we need to know the completion time of job j, if it would be processed by machine i_2. This completion time may have changed in assignment A_1, compared to what would have been its completion time in A. Hence, we define the value ΔL_{i_2} describing this change as

$$\Delta L_{i_2} := L_{i_2,j}^{A_1} - L_{i_2,j}^A. \tag{3.4}$$

Of course, also insertions and deletions of jobs with lower priority than job j on machine i_2 influence the move of job j to machine i_2 in assignment A_1. For this we define the value ΔW_{i_2} for the target machine to be

$$\Delta W_{i_2} := W_{i_2,j}^{A_1} - W_{i_2,j}^A. \tag{3.5}$$

An illustration of the situation is given in Figure 3.2. The assignments A and A_1 are shown as well as job j and the prior described effects influencing the change in the objective value. By "high priority" we denote all jobs with higher priority than job j, and by "low priority" we denote all jobs with lower priority.

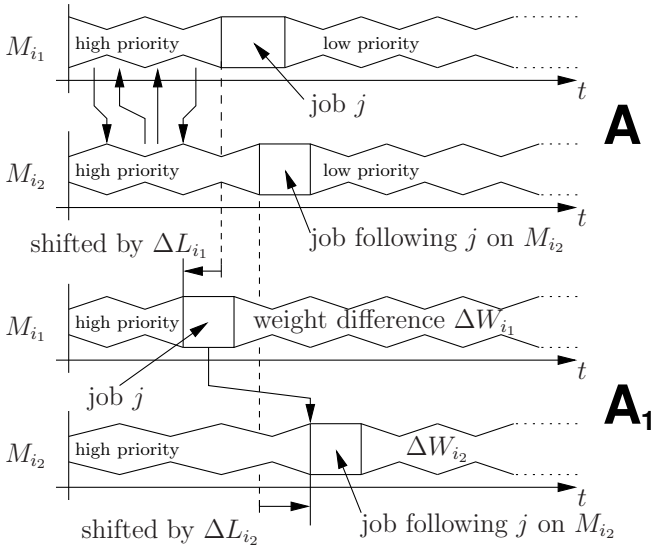

Figure 3.2: Emanating from assignment A to A_1.

Insertions and deletions of jobs with lower priority cause the weight differences on the machines. With the help of the values defined in (3.2), (3.3), (3.4) and (3.5), we can state the following lemma.

Lemma 3.1 *Let A and A_1 be assignments with the above mentioned properties, let j be a job scheduled on machine i_1 in assignment A and A_1, and let i_2 be the designated target machine. Then*

$$\delta^{A_1}_{move(j,i_2)} = \delta^{A}_{move(j,i_2)} + w_j(\Delta L_{i_1} - \Delta L_{i_2}) + p_j(\Delta W_{i_1} - \Delta W_{i_2}).$$

PROOF.

$$\begin{aligned}
\delta^{A_1}_{move(j,i_2)} &= p_j W^{A_1}_{i_1,j} - p_j W^{A_1}_{i_2,j} + w_j L^{A_1}_{i_1,j} - w_j L^{A_1}_{i_2,j} - w_j p_j \\
&= p_j \left(\Delta W_{i_1} + W^{A}_{i_1,j}\right) - p_j \left(\Delta W_{i_2} + W^{A}_{i_2,j}\right) + \\
&\quad w_j \left(\Delta L_{i_1} + L^{A}_{i_1,j}\right) - w_j \left(\Delta L_{i_2} + L^{A}_{i_2,j}\right) - w_j p_j \\
&= \delta^{A}_{move(j,i_2)} + p_j \left(\Delta W_{i_1} - \Delta W_{i_2}\right) + w_j \left(\Delta L_{i_1} - \Delta L_{i_2}\right). \qquad \square
\end{aligned}$$

Lemma 3.1 is the basis of combining several moves. In the next two subsections, we give examples of possible combinations of moves. The combined moves then again can be used to build a very large-scale neighborhood that can be explored by the matching method.

3.1.2.1 Swap-Neighborhood

The first combination of moves to build up a basis neighborhood $\mathcal{N}_1$ is given by swap operators $op^{swap}(i_1, i_2)$. Hereby, $op^{swap}(i_1, i_2)$ is a combination of two move-operators, which considers the machines i_1 and i_2 of the given assignment A and moves exactly one job from machine i_1 to i_2 and one job from machine i_2 to i_1. This is done in such a way that the change in the objective value is best possible. Obviously, these operators again have the stated property for the basis neighborhood $\mathcal{N}_1$. In the following, we describe how these operators $op^{swap}(i_1, i_2)$ can be realized efficiently. Hereby, amongst other things, Lemma 3.1 will be used.

The operator $op^{swap}(i_1, i_2)$ represents a best possible neighbor in a neighborhood, consisting of operators $swap(j, k)$, that reassign job $j \in M_{i_1}$ to machine $i_2 = A(k)$ and job $k \in M_{i_2}$ to machine $i_1 = A(j)$, where $i_1 \neq i_2$. We study the effects of a single swap of jobs $swap(j, k)$ and show how on the basis of these results the effect of an operator $op^{swap}(i_1, i_2)$ can be calculated.

In the following, let i_1 and i_2 be a pair of machines with $i_1 \neq i_2$. Additionally, let job j be a job assigned to machine i_1 and job k be a job assigned to machine i_2. A swap of jobs $swap(j, k)$ consists of two moves $move(j, i_2)$ and $move(k, i_1)$, i.e. $swap(j, k)(A) = move(j, i_2)(move(k, i_1)(A))$ (see Figure 3.3 for an illustration of a swap of jobs). For sake of simplicity we assume that job k has higher priority than job j.

In order to examine a swap we make use of Lemma 3.1. We define assignment A_1 as the assignment that arises from A by moving job k from machine i_2 to i_1, i.e. $A_1 := move(k, i_1)(A)$. Then $swap(j, k)(A) = move(j, i_2)(A_1)$.

It remains to express the value $\delta^{A_1}_{move(j,i_2)}$ in relation to $\delta^{A}_{move(j,i_2)}$. The assignments A and A_1 differ only by job k, which is processed by machine i_1 in assignment A_1. Hence, the starting times of all jobs following job k on machine

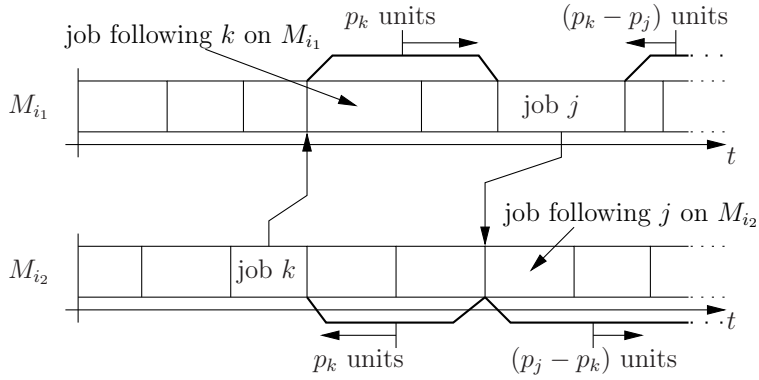

Figure 3.3: Illustration of $swap(j,k)(A)$ for $k < j$.

i_2 have decreased by p_k time units. Because job k has a higher priority than job j and the deletion of job k influences the completion time of scheduling job j on machine i_2, for (3.4) we obtain $\Delta L_{i_2} = -p_k$. Additionally, job k has been inserted on machine i_1. Therefore, all jobs with lower priority on machine i_1 start p_k time units later, resulting in an increase of job j's starting time by p_k time units. This causes that $\Delta L_{i_1} = p_k$, see (3.2).

By emanating from assignment A to A_1 we only move job k from its machine i_2 to machine i_1. Hence, we have not done any insertions or deletions on jobs with lower priority than job k. We obtain that $\Delta W_{i_1} = 0$ and $\Delta W_{i_2} = 0$, respectively, see (3.3) and (3.5). With Lemma 3.1 we can conclude, that

$$\delta^{A_1}_{move(j,i_2)} = \delta^{A}_{move(j,i_2)} + 2w_j p_k.$$

Summarizing, the change in the objective value by applying a swap of jobs $swap(j,k)$ with $k < j$ and $i_1 = A(j) \neq A(k) = i_2$, on assignment A is calculated as

$$\delta^{A}_{swap(j,k)} := \delta^{A}_{move(j,i_2)} + \delta^{A}_{move(k,i_1)} + 2w_j p_k. \tag{3.6}$$

Thus, if the values $\delta^{A}_{move(j,i_2)}$ and $\delta^{A}_{move(k,i_1)}$ are known in advance, $\delta^{A}_{swap(j,k)}$ can be calculated in $\mathcal{O}(1)$.

The overall preprocessing to calculate all values $\delta^{A}_{move(j,i)}$, with $i \neq A(j)$, can be done in $\mathcal{O}(nm)$. To calculate the effect of $op^{swap}(i_1,i_2)$, all swaps of pairs of

jobs on the two machines have to be evaluated, which can be done in $\mathcal{O}(n_{i_1}n_{i_2})$. Finally, for the neighborhood $\mathcal{N}_2$ we have to evaluate all operators $op^{swap}(i_1, i_2)$, with $i_1 < i_2$, to build up the graph $G(A)$. This can be realized in $\mathcal{O}(n^2)$.

The neighborhood $\mathcal{N}_2$ contains an exponential (in m) number of neighbors. Each of these neighbors again dominates a certain number of neighbors w.r.t. the neighborhood defined by the *swap* operators (we again call this neighborhood $\mathcal{N}_0$). More precisely, a matching $\mathcal{M}$ containing the edges $(s_1, t_1), \ldots, (s_k, t_k)$ in $G(A)$ leads to an assignment A', with

$$A' = op^{move}(s_1, t_1) \circ op^{move}(s_2, t_2) \circ \cdots \circ op^{move}(s_k, t_k)(A).$$

Each edge $\{s_l, t_l\}$ can be understood as an operator $op^{swap}(s_l, t_l)$, applied to assignment A. Observe, that $op^{swap}(i_l, j_l)$ dominates $n_{s_l}n_{t_l}$ operators in the neighborhood $\mathcal{N}_0$. Therefore, the considered matching $\mathcal{M}$ using operator $op^{swap}(s_l, t_l)$ for each edge (s_l, t_l) represents the best solution in a neighborhood of size $\prod_{l=1}^{k}(n_{s_l}n_{t_l})$. In contrast to this, neighborhood $\mathcal{N}_1$ (i.e. swap a pair of jobs between a pair of machines i_1 and i_2) contains only $\frac{1}{2}m(m-1)$ neighbors, where each neighbor represents a best solution in a neighborhood of size $n_{i_1}n_{i_2}$.

3.1.2.2 $\overline{k}$-Move-Neighborhood

In the following, we generalize the ideas of *move* and *swap* with the help of Lemma 3.1. This generalization leads to another basis neighborhood $\mathcal{N}_1$ built up using $\overline{k} - move$ operators $op^{\overline{k}-move}(i_1, i_2)$. Here, $op^{\overline{k}-move}(i_1, i_2)$ considers the machines i_1 and i_2 and moves up to k jobs between machines i_1 and i_2. This is done in such a way that the change in the objective value is best possible. These operators have the stated properties for the basis neighborhood $\mathcal{N}_1$. It remains to describe how these operators $op^{\overline{k}-move}(i_1, i_2)$ can be realized efficiently.

An operator $op^{\overline{k}-move}(i_1, i_2)$ represents a best possible move of up to k jobs between machines i_1 and i_2. In the following, we define a single operator that moves exactly k jobs between machines i_1 and i_2 and study the effects. For this we consider the fixed pair of machines i_1 and i_2 and only jobs j that are processed by one of these machines, i.e. $A(j) = i_1$ or $A(j) = i_2$. Moreover, by

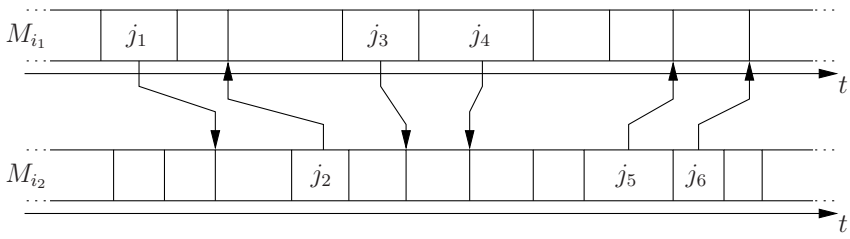

Figure 3.4: Illustration of $k - move(j_1, j_2, \ldots)$

$\overline{A}(j)$ we denote the machine that job j is not assigned to, i.e.

$$\overline{A}(j) = \begin{cases} i_1 & \text{if } A(j) = i_2, \\ i_2 & \text{if } A(j) = i_1. \end{cases}$$

Using this notation we can define $move(j)(A) := move(j, \overline{A}(j))(A)$. For a fixed k we examine a combination of moves

$$k - move(j_1, \ldots, j_k)(A) := move(j_k) \circ move(j_{k-1}) \circ \ldots \circ move(j_1)(A),$$

where $j_1 < j_2 < \ldots < j_k$, i.e. job j_l has a higher priority than job j_{l+1} for $l = 1, \ldots, k-1$.

First we consider the assignment

$$A_2 := move(j_2) \circ move(j_1)(A).$$

Due to Lemma 3.1 and a similar argumentation as used in Section 3.1.2.1 for evaluating a swap, the difference of the objective values of assignments A and A_2 can be calculated by

$$\begin{aligned} f(A) - f(A_2) \quad = \quad & \delta^A_{move(j_1, \overline{A}(j_1))} + \\ & \delta^A_{move(j_2, \overline{A}(j_2))} + 2w_{j_2} p_{j_1} \Delta_{j_2, j_1}. \end{aligned} \tag{3.7}$$

Here, $\Delta_{j_s, j_t} = 1$ if job j_t and j_s are assigned to different machines in A and conversely, $\Delta_{j_s, j_t} = -1$ if job j_t is assigned to the same machine as job j_s. This leads to the following lemma.

Lemma 3.2 *Let $A' := k - move(j_1, \ldots, j_k)(A)$. The change in the objective value $\delta^A_{k-move(j_1,\ldots,j_k)} = f(A) - f(A')$ is calculated as*

$$\delta^A_{k-move(j_1,\ldots,j_k)} = \sum_{s=1}^{k} \delta^A_{move(j_s,\overline{A}(j_s))} + \sum_{s=1}^{k} 2w_{j_s} \sum_{t=1}^{s-1} \Delta_{j_s,j_t} p_{j_t}. \tag{3.8}$$

PROOF. We prove the lemma by induction on k. For $k = 2$ the claim of the lemma coincides with (3.7) and, therefore, is true. Consider the assignments $A_l := move(j_l) \circ \ldots \circ move(j_1)(A)$ for each l, $1 \leq l \leq k$. Suppose that the claim of the lemma is true for operators which move up to $k-1$ jobs, $k \geq 3$. We prove that the claim also holds for operators which move k jobs.

Every job of $j_1, \ldots, j_{k-1}$ that is processed on machine $A(j_k)$ in assignment A is moved to machine $\overline{A}(j_k)$ in assignment A_{k-1}. Every job of $j_1, \ldots, j_{k-1}$ that is processed on machine $\overline{A}(j_k)$ in assignment A is moved to the machine on which job j_k is processed in assignment A_{k-1}. Furthermore, all jobs j_μ, with $\mu = 1, \ldots, k-1$, have a higher priority than job j_k.

Hence, the completion time of job j_k in the schedule belonging to assignment A_{k-1} differs from the completion time in the schedule belonging to assignment A by

$$C^{A_{k-1}}_{j_k} = C^A_{j_k} + \sum_{t=1}^{k-1} \Delta_{j_k,j_t} p_{j_t}.$$

The sum of processing times $L_{\overline{A}_{k-1},j}$ of all jobs having higher priority than job j_k on machine $\overline{A}_{k-1}(j_k) = \overline{A}(j_k)$ changed in a similar way to

$$L^{A_{k-1}}_{\overline{A}_{k-1}(j_k),j_k} = L^A_{\overline{A}(j_k),j_k} - \sum_{t=1}^{k-1} \Delta_{j_k,j_t} p_{j_t}.$$

By using Lemma 3.1, we know that

$$f(A) - f(A_k) = f(A) - f(A_{k-1}) + \delta^A_{move(j_k,\overline{A}(j_k))} + 2w_{j_k} \sum_{t=1}^{k-1} \Delta_{j_k,j_t} p_{j_t}$$

holds, and (3.8) follows for A_k. $\square$

The calculation of the value $\delta^A_{k-move(j_1,\ldots,j_k)}$ with (3.8) needs in the worst case a running time of $\mathcal{O}\left(k^2\right)$, if the values $\delta^A_{move(j_s,\overline{A}(j_s))}$ are known in advance.

For the operator $op^{\overline{k}-move}(i_1,i_2)$ we now have to find the best move of up to k jobs between machines i_1 and i_2. To do so, we denote by $M_{i_1} \cup M_{i_2}$ the set of jobs processed by machine i_1 or i_2. For every l, with $1 \leq l \leq k$, we have to calculate, for every possible subset $\{j_1,\ldots,j_l\} \subseteq M_{i_1} \cup M_{i_2}$ of cardinality l, the value $\delta^A_{l-move(j_1,\ldots,j_l)}$ to determine the best move of up to k jobs between machines i_1 and i_2. Since in a preprocessing the values $\delta^A_{move(j_s,\overline{A}(j_s))}$ can be obtained in $\mathcal{O}(n)$, the overall complexity of $op^{\overline{k}-move}(i_1,i_2)$ is $\mathcal{O}(k^2n^k)$. For the neighborhood $\mathcal{N}_2$ we have to evaluate all operators $op^{\overline{k}-move}(i_1,i_2)$ with $i_1 < i_2$ to build up the graph $G(A)$. This can be realized in $\mathcal{O}(k^2m^2n^k)$. Thus, for constant values of k the neighborhood $\mathcal{N}_2$ can be evaluated in polynomial time.

The best neighbor in a neighborhood consisting of $k-move$ operators working on machines i_1 and i_2 dominates

$$\binom{n_{i_1}+n_{i_2}}{k}$$

neighbors. This means, that the operator $op^{\overline{k}-move}(i_1,i_2)$ retrieves the best solution out of

$$\sum_{l=1}^{k}\binom{n_{i_1}+n_{i_2}}{l}$$

solutions contained in the neighborhood $\mathcal{N}_0$ consisting of operators $l-move$, with $1 \leq l \leq k$.

A matching $\mathcal{M}$ containing the edges $(s_1,t_1),\ldots,(s_\mu,t_\mu)$ in $G(A)$ is contained in the neighborhood $\mathcal{N}_2$ of exponential (in m) size. Each edge (s_ν,t_ν) corresponds to an operator $op^{\overline{k}-move}(s_\nu,t_\nu)$. Therefore, the considered matching $\mathcal{M}$ using operator $op^{\overline{k}-move}(s_\nu,t_\nu)$, for each edge (s_ν,t_ν), represents the best solution in

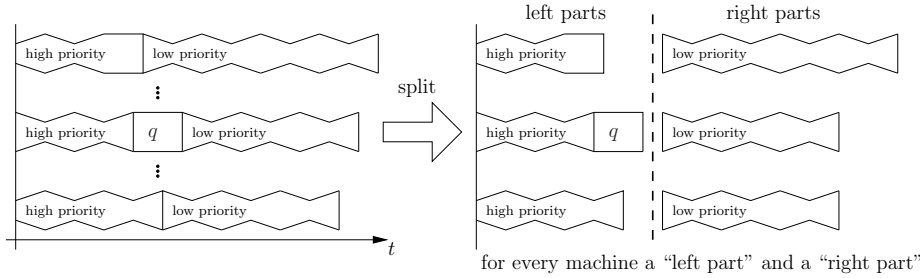

Figure 3.5: Illustration of split-operator for fixed job q.

a neighborhood of size

$$\prod_{\nu=1}^{\mu}\left(\sum_{l=1}^{k}\binom{n_{s_\nu}+n_{t_\nu}}{l}\right).$$

In contrast to this, the neighborhood $\mathcal{N}_1$ consisting of operators $op^{\overline{k}-move}(i_1,i_2)$ for $1 \leq i_1, i_2 \leq m$ and $i_1 \neq i_2$ contains only $\frac{1}{2}m(m-1)$ neighbors, where each neighbor represents the best solution in a neighborhood of size $\sum_{l=1}^{k}\binom{n_{i_1}+n_{i_2}}{l}$.

Observe, that a neighborhood built by the operator $op^{\overline{2}-move}(i_1,i_2)$ represents the union of the neighborhoods built by the two operators $op^{move}(i_1,i_2)$ and $op^{swap}(i_1,i_2)$.

3.2 Split-Neighborhood

In this section we introduce the so-called split neighborhood. The idea of this neighborhood is to first partition the jobs assigned to a machine into a left part and a right part. For a given job q, this partitioning is carried out, so that the left parts only contain jobs with higher or equal priority compared to job q, and the right parts only contain jobs with lower priority. Such a partitioning is illustrated in Figure 3.5

Afterwards, a neighbor is obtained by reassigning the jobs of the right parts to

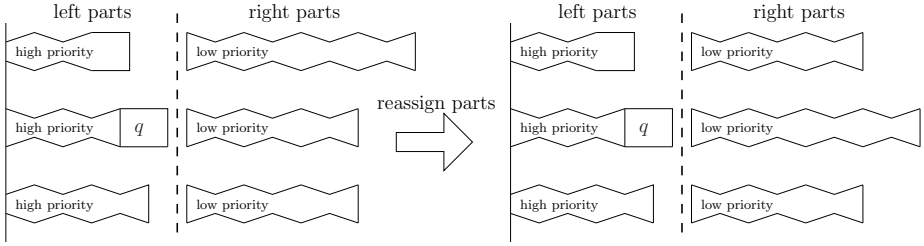

Figure 3.6: Reassigning parts to obtain neighboring assignment.

different machines (see Figure 3.6). The best improving neighbor in this neighborhood can be determined by looking for a minimum weight perfect matching in an improvement graph. In fact, it turns out that the process of searching for the appropriate matching solves a restricted version of the considered scheduling problem.

We start by introducing this restricted version of the problem $P||\sum w_j C_j$. Consider two sets of jobs $J_1 = \{1, \ldots, m\}$ and $J_2 = \{m+1, \ldots, 2m\}$ of equal cardinality. Moreover, every job $j \in J_1$ has a higher priority than any job $k \in J_2$. We enumerate the jobs without loss of generality, so that $p_1 \geq \ldots \geq p_m$ and $w_{m+1} \leq \ldots \leq w_{2m}$. The restricted problem now consists of assigning these $2m$ jobs to the m machines, such that to every machine exactly one job from J_1 and exactly one job from J_2 is assigned and the corresponding schedule minimizes the objective function given by (1.4). In the following, we only consider assignments A that fulfill the introduced restrictions, i.e. $n_i = 2$ for any machine i. Therefore, each assignment can be represented by $M_i = \{i_1, i_2\}$, with $i_1 \in J_1$ and $i_2 \in J_2$, for $i = 1, \ldots, m$.

We are interested in the completion times of the jobs in the schedule corresponding to a given assignment A. For the completion times of the two jobs $i_1 \in J_1$ and $i_2 \in J_2$ assigned to machine i hold $C_{i_1} = L_{i,m} = p_{i_1}$ and $C_{i_2} = L_{i,2m} = L_{i,m} + p_{i_2}$. Observe that for the weight of job i_2 holds $w_{i_2} = W_{i,m}$. For the contribution $f_i(A)$ of machine i to the objective value we obtain

$$f_i(A) = w_{i_1} C_{i_1} + w_{i_2} C_{i_2} = w_{i_1} p_{i_1} + w_{i_2} p_{i_2} + L_{i,m} W_{i,m},$$

and, therefore, we obtain

$$f(A) = \sum_{i=1}^{m} f_i(A) = \sum_{j=1}^{2m} w_j p_j + \sum_{i=1}^{m} L_{i,m} W_{i,m}. \tag{3.9}$$

Thus, an optimal assignment that minimizes the objective function f in fact minimizes

$$\sum_{i=1}^{m} L_{i,m} W_{i,m} = \sum_{i=1}^{m} p_{i_1} w_{i_2}. \tag{3.10}$$

The value of $L_{i,m}W_{i,m}$ depends on the choice, which two jobs $i_1 \in J_1$ and $i_2 \in J_2$ are assigned to machine i. In order to solve this assignment problem, consider the complete bipartite weighted graph $G(A) = (V_1 \cup V_2, V_1 \times V_2, c(A))$, where

- the vertex sets V_1 and V_2 contain a vertex for each job of set J_1 and J_2, respectively,
- an edge $e = (j_1, j_2) \in V_1 \times V_2$ connecting vertices $j_1 \in V_1$ and $j_2 \in V_2$ represents an assignment, in which jobs j_1 and j_2 are assigned to the same machine,
- an edge $e = (j_1, j_2) \in V_1 \times V_2$ gets a weight $c(A)_{j_1,j_2} = p_{j_1} w_{j_2}$ representing the contribution to the objective value (3.10) by assigning jobs j_1 and j_2 to the same machine.

An assignment in the graph (i.e. a perfect matching) with minimum weight gives an optimal solution of the scheduling problem. Since the structure of the weights of an edge (j_1, j_2) is very special (product structure $p_{j_1} w_{j_2}$), the assignment problem can be solved by putting jobs i and $m+i$ to machine i for $i = 1, \ldots, m$. Thus, the restricted problem can be solved in time $\mathcal{O}(m \log m)$ by sorting the jobs of set J_1 by non-increasing processing times and the jobs of set J_2 by non-decreasing weights. A proof that this assignment is optimal can be found, e.g. in Corollary 2.4 in Brucker [14].

In the following, we show how for our original problem $P||\sum w_jC_j$, as described in Section 1.3.2, a very large-scale neighborhood can be obtained using the previous ideas. Let q be a fixed, but arbitrarily chosen job for the rest of this

section. We denote by a *split* of the jobs with *anchor* q a partition of all sets M_i^A into a so-called *left part* M_{i1}^A and a *right part* M_{i2}^A, such that

$$M_{i1}^A := \{j \in M_i^A : j \leq q\},$$
$$M_{i2}^A := \{j \in M_i^A : j > q\}.$$

Using this split, a neighborhood structure can be achieved by defining that an assignment A' is a neighbor of A with respect to the anchor q, if the resulting sets $M_{i1}^{A'}$ and $M_{i2}^{A'}$ are permutations of the sets M_{i1}^A and M_{i2}^A, for $i = 1, \ldots, m$, i.e. if there exist two permutations $\pi_l, \pi_r \in \mathcal{S}_m$, such that $M_{\pi_l(i),1}^A = M_{i1}^{A'}$ and $M_{\pi_r(i),2}^A = M_{i2}^{A'}$. Since the machines are all identical, we assume, without loss of generality, that $\pi_l(i) = i$, for all $i = 1, \ldots, m$.

The neighborhood $\mathcal{N}_{split(q)}(A)$ consists of all neighboring assignments A' with respect to the anchor q. This means, that the neighborhood $\mathcal{N}_{split(q)}(A)$ contains all reassignments of the right parts M_{i2} to the left parts M_{i1}, which are at most $m!$ assignments.

It remains to describe how the neighborhood $\mathcal{N}_{split(q)}(A)$ can be efficiently explored and a best neighboring assignment can be determined. For this, consider M_{i1} and M_{i2}, for $i = 1, \ldots, m$, to be a split of jobs with anchor q for a given assignment A. We define the contribution $f(M_{il})$ of a part M_{il} for $l \in \{1, 2\}$ and $i \in \{1, \ldots, m\}$ to be

$$f(M_{il}) := \sum_{j \in M_{il}} w_j L_{ij}.$$

Using these values, the objective value $f(A)$ is calculated as

$$f(A) = \sum_{i=1}^{m} \big(f(M_{i1}) + f(M_{i2})\big) + \sum_{i=1}^{m} L_{iq} W_{iq},$$

where $L_{iq} = \sum_{j \in M_{i1}} p_j$ and $W_{iq} = \sum_{j \in M_{i2}} w_j$.

For the rest of this section, let $A' \in \mathcal{N}_{split(q)}(A)$ be a neighboring assignment of A. Then there exists a unique permutation $\pi_r \in \mathcal{S}_m$, such that $M_{i1}^A = M_{i,1}^{A'}$

and $M^A_{\pi_r(i),2} = M^{A'}_{i,2}$, for $i = 1, \ldots, m$. For the objective value of A' we get

$$\begin{aligned} f(A') &= \sum_{i=1}^{m} \left(f(M^{A'}_{i1}) + f(M^{A'}_{i2}) + L^{A'}_{iq} W^{A'}_{iq} \right) \\ &= \sum_{i=1}^{m} \left(f(M^{A}_{i1}) + f(M^{A}_{\pi_r(i),2}) + L^{A}_{iq} W^{A}_{\pi_r(i),q} \right) \\ &= \sum_{i=1}^{m} \left(f(M_{i1}) + f(M_{i2}) \right) + \sum_{i=1}^{m} L^{A}_{iq} W^{A}_{\pi_r(i),q}. \end{aligned}$$

Hence, the objective value of a neighboring assignment A' differs from A in the value of $L^A_{iq} W^A_{\pi_r(i),q}$. This value is determined by the choice, which set of jobs $M^A_{\pi_r(i),2}$ is combined with M^A_{i1} on machine i, according to assignment A'.

In the following, we give an instance for the restricted problem, for which there exists a one-to-one correspondence between assignments in the restricted problem and neighbors of A in the neighborhood $\mathcal{N}_{split(q)}(A)$. The instance for the restricted problem has $2m$ jobs with the following processing times $\widetilde{p}_i$ and weights $\widetilde{w}_i$:

$$\begin{array}{llllll} \widetilde{p}_i & := L_{iq} & \text{and } \widetilde{w}_i & := W, & \text{for } i = 1, \ldots, m, \\ \widetilde{p}_{m+i} & := L & \text{and } \widetilde{w}_{m+i} & := W_{iq}, & \text{for } i = 1, \ldots, m, \end{array}$$

where:

$$\begin{aligned} L &:= max\{L_{iq} : i = 1, \ldots, m\} \text{ and} \\ W &:= max\{W_{iq} : i = 1, \ldots, m\}. \end{aligned}$$

We have chosen these values for L and W to ensure for the weight to processing time ratios that

$$\frac{\widetilde{w}_{j_1}}{\widetilde{p}_{j_1}} \geq \frac{\widetilde{w}_{j_2}}{\widetilde{p}_{j_2}},$$

for all pairs $j_1 \in J_1 = \{1, \ldots, m\}$ and $j_2 \in J_2 = \{m+1, \ldots, 2m\}$. It is straightforward to see that, by identifying set M_{i1} with job $i \in J_1$ and set M_{i2} with job $m+i \in J_2$, $i = 1, \ldots, m$, there is a one-to-one correspondence between the feasible assignments of the restricted problem and the neighbors of A with respect to the anchor q. It remains to show that the objective values of the restricted problem and the corresponding assignment for the original problem

are equal except for an additive term determined by the chosen split with anchor q.

Consider again the neighboring assignment A'. This neighboring assignment A' corresponds to an assignment $\widetilde{A}$ for the restricted problem, in which job i and job $m + \pi_r(i)$ are assigned to machine i. By denoting with $\widetilde{f}$ the objective function and with $\widetilde{L}$ and $\widetilde{W}$ the partial sums for the restricted problem, we obtain from (3.9):

$$\begin{aligned}
\widetilde{f}(\widetilde{A}) &= \sum_{j=1}^{2m} \widetilde{w}_j \widetilde{p}_j + \sum_{i=1}^{m} \widetilde{L}_{im}^{\widetilde{A}} \widetilde{W}_{m+\pi_r(i),m}^{\widetilde{A}} \\
&= \sum_{i=1}^{m} L_{iq}^{A'} W + \sum_{i=1}^{m} L W_{iq}^{A'} + \sum_{i=1}^{m} L_{iq}^{A'} W_{iq}^{A'} \\
&= \sum_{i=1}^{m} L_{iq}^{A} W + \sum_{i=1}^{m} L W_{iq}^{A} + \sum_{i=1}^{m} L_{iq}^{A} W_{\pi_r(i),q}^{A} \\
&= \sum_{i=1}^{m} L_{iq} W + \sum_{i=1}^{m} L W_{iq} + \sum_{i=1}^{m} L_{iq}^{A} W_{\pi_r(i),q}^{A}.
\end{aligned}$$

Hence, the objective values of the neighboring assignment A' and the corresponding assignment $\widetilde{A}$ for the restricted problem differ only by an additive term determined by the chosen split. Thus, we can calculate the best assignment in the neighborhood $\mathcal{N}_{split(q)}(A)$ by solving the restricted problem for the above given instance.

3.3 Computational Results

In this section, we report on computational experiments conducted to analyze the introduced approaches regarding solution quality and running time. Since no data-sets or codes of the existing exact solution methods of Belouadah and Potts [13] and van den Akker et al. [7] are available, we calculated for a set of smaller instances optimal solutions via some straightforward enumeration method.

For obtaining initial solutions we use two different constructive heuristics and

randomly generated instances. A first method is introduced by Eastman et al. [29] and is called *LRF-heuristic* (*Largest Ratio First*). The heuristic iteratively assigns a non-planned job with highest priority to a machine with minimum workload. Kawaguchi and Kyan [46] prove that schedules received by the LRF-heuristic do not exceed $(\sqrt{2}+1)/2 < 1.208$ times the optimal value. Computational testing from Baker and Merten [11] and Barnes and Laguna [12] indicate that the LRF-heuristic is rather successful in delivering near-optimal solutions.

A second heuristic goes back to an idea of Hoede [41]. If N denotes the set of non-planned jobs, then this method chooses in every iteration a job $j \in N$ for which

$$w_j \sum_{k \in N \setminus \{j\}, k<j} p_k + p_j \sum_{k \in N \setminus \{j\}, k>j} w_k$$

is maximum. This value reflects the importance of job j, since if all jobs of N (except for job j) would be assigned to a single machine (and there are no more), and afterwards job j is also assigned to this machine, the objective value increases by this amount. Then, this job j is inserted on the machine giving the least increase in the objective value and deleted from the set of non-planned jobs. We denote this algorithm as *IF-heuristic* (Important First). Furthermore, we use randomly generated assignments as initial solutions (called *RND-heuristic*).

The problem instances are generated as described by van den Akker et al. [7]. They introduce 3 different types of instances, where the integer processing times and weights are uniformly drawn out of the following intervals:

- type (1) with $p_j \in [1, 10]$ and $w_j \in [10, 100]$,
- type (2) with $p_j \in [1, 100]$ and $w_j \in [1, 100]$,
- type (3) with $p_j \in [10, 20]$ and $w_j \in [10, 20]$.

Additionally, we consider another type (4). Here, the processing times are taken uniformly from the interval $[5, 15]$ or $[35, 45]$ and the weights are determined by choosing the weight to processing time ratio uniformly from the interval $[0.8, 1.2]$ and rounding the resulting weights to the nearest integer. For each type we generate three sets of instances with a constant job to machine ratio of $\frac{n}{m} = 5$, $\frac{n}{m} = 10$ and $\frac{n}{m} = 25$. In order to receive significant results we create instances

for varying numbers of jobs and average each measurement over 100 randomly generated instances. Additionally, for instances with a job to machine ratio of $\frac{n}{m} = 5$ and $n \in \{15, 20, 25\}$ we know the optimal objective value.

First we compare the chosen constructive heuristics and the lower bounding method described by Eastman et al. [29], which we call *LB*. We see that LB is very close to the optimal objective value for instances with $n \in \{15, 20, 25\}$ and $\frac{n}{m} = 5$, but not as close as the LRF-heuristic and the IF-heuristic. Both, the LRF-heuristic and the IF-heuristic deliver solutions of similar quality and except for type (4) instances, the LRF-heuristic performs better than the IF-heuristic (for type (4) and $\frac{n}{m} = 5$ or $\frac{n}{m} = 10$ the IF-heuristic performs better than LRF, but for $\frac{n}{m} = 25$ the LRF-heuristic again performs better than IF). If we compare the differences to the optimal objective value for instances with $\frac{n}{m} = 5$, we see that LB is clearly separated from the optimal objective value, LRF-heuristic and IF-heuristic, which deliver solutions of similar objective value. Here, RND is far off limits. For instances with $\frac{n}{m} = 10$ and $\frac{n}{m} = 25$, the deviations between LB, LRF and IF become larger. The differences in the objective value of the methods increase nearly linear with n for a constant ratio $\frac{n}{m}$. We note, that the average objective values (received by one of the methods LB, LRF, IF, RND) for instances of type (1) and type (3) are of the same order. Instances of type (4) and type (2) have average objective values of roughly 1/3 and 1/6, respectively, compared to those of instances of type (1).

In order to test the effectiveness of the neighborhoods in practice, we implement them in ANSI-C on an PC with Intel Pentium IV processor running at 2.4 GHz. According to the Standard Performance Evaluation Corporation (SPEC), this computer is roughly 50 times as fast as the HP9000/710 used by van den Akker et al. [7] and hence, roughly 100 times as fast as the CDC 7600 used by Belouadah and Potts [13].

First we use neighborhoods consisting of operators $op^{\overline{k-move}}(i, j)$ with values $k \in \{1, 2, 3, 4\}$. Hence, the neighborhood $\mathcal{N}_{k-move}$ of a given assignment A contains all assignments that can be received by moving up to k jobs between a fixed pair of machines i and j. We compare the neighborhood $\mathcal{N}_{k-move}$ to the matching based neighborhood $\mathcal{N}^M_{k-move}$. In order to determine the best improving neighbor in the neighborhood $\mathcal{N}^M_{k-move}$ we need to solve a maximum weight matching problem for the improvement graph corresponding to a given assignment. We implemented this by solving an integer linear program. This

turned out to be fast enough for our purposes. Indeed, the average time to solve the integer linear programs is negligible for small m compared to the average time needed to build up the improvement graph.

We use iterative improvement as local search method for these neighborhoods. We have chosen not to use tabu search or simulated annealing since we are interested in the structural behavior of the neighborhoods and not in the potentials of local search methods. We denote with *k-MOVE* the iterative improvement procedure that advances in every iteration to the best solution in the neighborhood $\mathcal{N}_{k-move}$. If we speak of objective values of a method we always mean the local optimum obtained at the end of the algorithm. Additionally, the iterative improvement algorithm using the matching based neighborhood $\mathcal{N}^M_{k-move}$ is denoted by *MATCHING k-MOVE*.

Secondly, we use the neighborhood $\mathcal{N}_{split(q)}$. For every job $q = 1, \ldots, n$ we calculate the best improving neighbor of the current solution in $\mathcal{N}_{split(q)}$. We then advance to the best solution found, if this solution improves over the current solution. This process is repeated until there can be made no improvement for any job q. We denote this algorithm by *SPLIT*.

Finally, we use iterative improvement on the combined neighborhoods $\mathcal{N}_{split(q)}$ and $\mathcal{N}_{2-move}$. This means, that we run SPLIT and afterwards advance to the best improving neighbor in $\mathcal{N}_{2-move}$. This process is repeated until no further improvement is possible. We denote this algorithm by *SPLIT-2MOVE*.

If it is not mentioned, we always start with an initial solution received by the LRF-heuristic. If we speak of objective values of a method, we always mean the local optimum obtained at the end of the iterative improvement algorithm. In the following, we test the given neighborhoods with different initial solutions for the previously generated instances.

Since we are considering instances with $\frac{n}{m} = 5$, $\frac{n}{m} = 10$ and $\frac{n}{m} = 25$, we expect that the optimal objective values are not far off 4-MOVE. In fact, comparing 4-MOVE with the optimal objective value, we find that both are almost equal. Another motivation to do so is that, for instances with $\frac{n}{m} = 10$, we have in average 10 jobs per machine, iterative improvement by moving at most 4 jobs between a pair of machines is likely to yield balanced machines. Thus, in the following we take 4-MOVE as an indication for the optimal objective value, i.e. we always present deviations from achieved solutions to the objective value

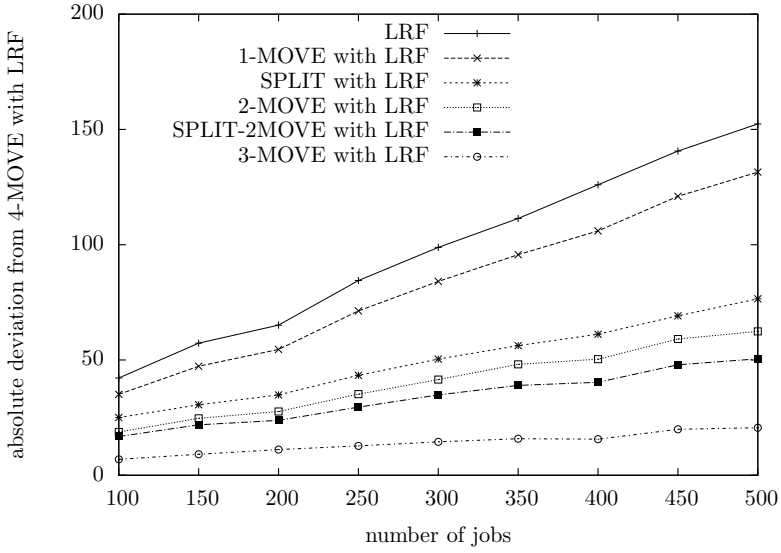

Figure 3.7: Quality of methods for type (1) instances.

obtained by 4-MOVE.

The overall performance of k-MOVE for $k = 1, \ldots, 4$ and SPLIT can be seen in Figure 3.7. This figure shows the outcome for a job to machine ratio $\frac{n}{m} = 25$ for instances of type (1). The situation for all other types looks similar. With increasing k the values of k-MOVE get better for all types of instances except for type (3). Here, the LRF-heuristic performs nearly the same as 1-MOVE. Additionally, 2-MOVE and 3-MOVE yield comparable solutions. Indeed, for instances of type (3) the jobs do not differ much, and thus an improvement mainly can be achieved by exchanging jobs rather than by moving. As a consequence, k-MOVE with an odd k behaves almost the same as $(k-1)$-MOVE.

The quality of the local optima obtained by SPLIT lies between 1-MOVE and 2-MOVE. This can be seen in Figure 3.7 for instances of type (1), which is representative for all types except for type (2). For instances of type (2), the quality of SPLIT is comparable to 3-MOVE (for smaller instances) and 4-MOVE (for larger instances). The combined approach SPLIT-2MOVE performs slightly better than SPLIT alone, for all types of instances.

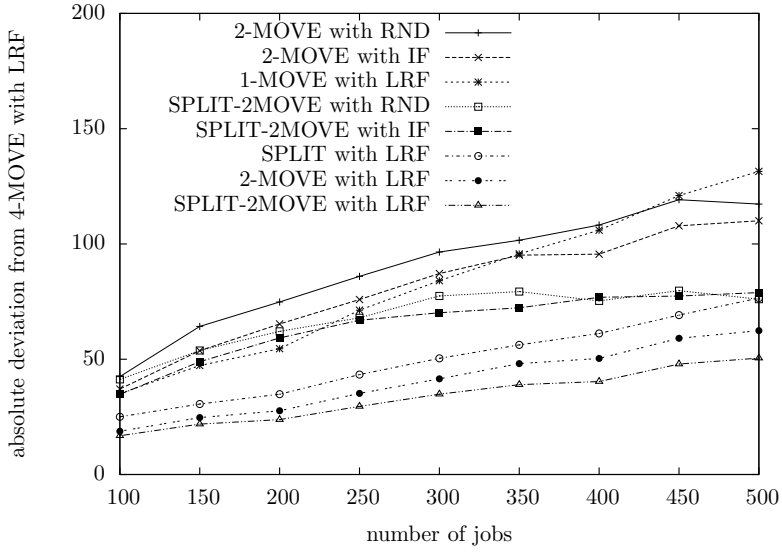

Figure 3.8: Quality of methods for different initial solutions.

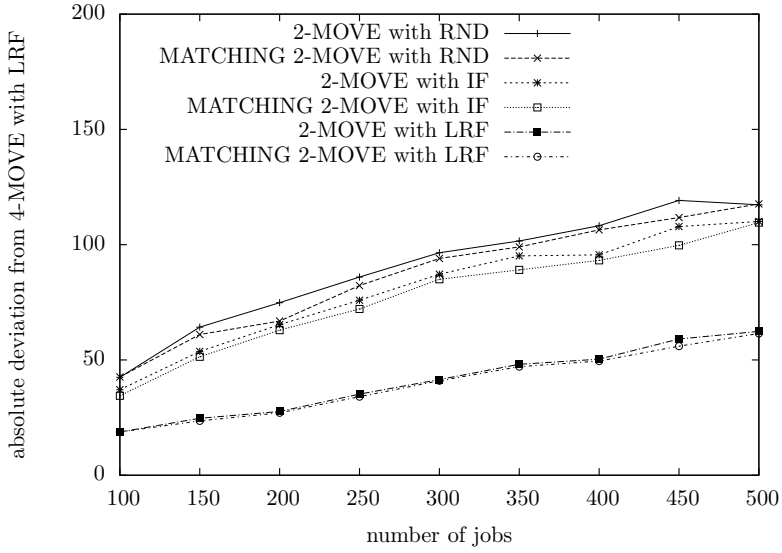

Figure 3.9: Quality of 2-MOVE and MATCHING 2-MOVE.

Next we take a look at how sensitive our neighborhoods are to the choice of initial solutions, see Figure 3.8. First, we see that by starting with a solution of bad quality, the methods only seldom reach the objective values when beginning at a solution of good quality. Nevertheless, the differences are not that big. Indeed, k-MOVE with $k = 2, 3, 4$ is able to bring initial solutions of bad quality in the range of using initial solutions obtained by the LRF-heuristic. This holds for all types of instances. To the contrary, 1-MOVE is very sensitive to the choice of initial solutions in all types of instances. This method delivers local optima of bad quality, if the IF-heuristic or RND-heuristic is chosen to obtain initial solutions. SPLIT is also sensitive to the choice of initial solutions, but performs better than 1-MOVE using bad initial solutions. The behavior of 2-MOVE with initial solutions of bad quality carries over to SPLIT-2MOVE. Moreover, the combined method seems to perform almost equally regardless of the used initial solution for bigger instances.

In a next setting we compare k-MOVE with MATCHING k-MOVE. The experiments indicate, that k-MOVE and MATCHING k-MOVE are performing nearly the same, regarding solution quality, for all types of instances. The results for $k = 2$, $\frac{n}{m} = 25$ and instances of type (1) can be seen in Figure 3.9.

In the results presented till now, we concentrated on the quality of the solutions received. In the next part we investigate the running time. The first observation is that, regardless which method we use, starting with initial solutions of bad quality leads to a higher average running time of the methods, see Figure 3.10. We use instances of type (2) to present the results on the running time, because for this type of instances all our methods need most time to reach a local optima.

Additionally, we observe that the average time needed to reach a local optimum by using MATCHING k-MOVE is smaller than using the basic k-MOVE for all k. Indeed, for instances with $\frac{n}{m} = 25$ the average time needed to solve the matching with the integer linear program needed less than 0.1 seconds in total, which is small in comparison to the average running time of iterative improvement, as shown in Figure 3.11. For $\frac{n}{m} = 25$, the running time of SPLIT is comparable to 2-MOVE, see also Figure 3.11. Finally, the average time needed to reach a local optimum by using the combined approach SPLIT-2MOVE is slightly higher than for SPLIT alone.

The decrease in running time using MATCHING k-MOVE instead of k-move is a result from the fact, that MATCHING k-MOVE needs less iterations to reach

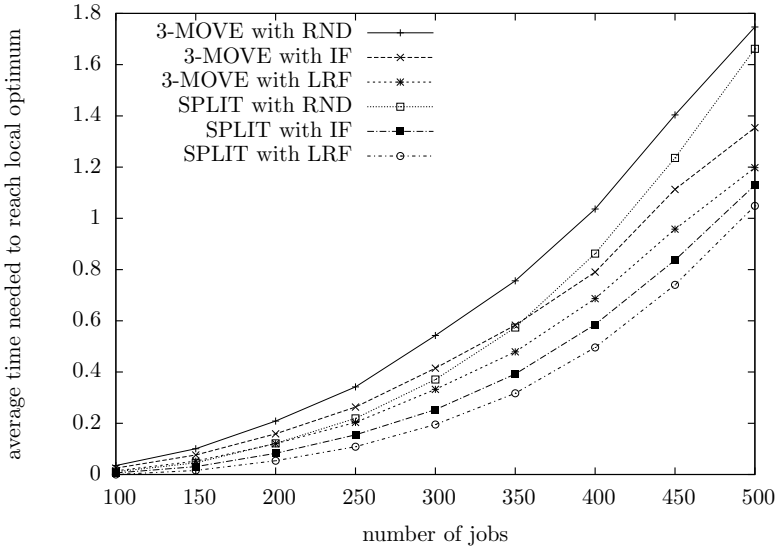

Figure 3.10: Running times for 3-MOVE and SPLIT.

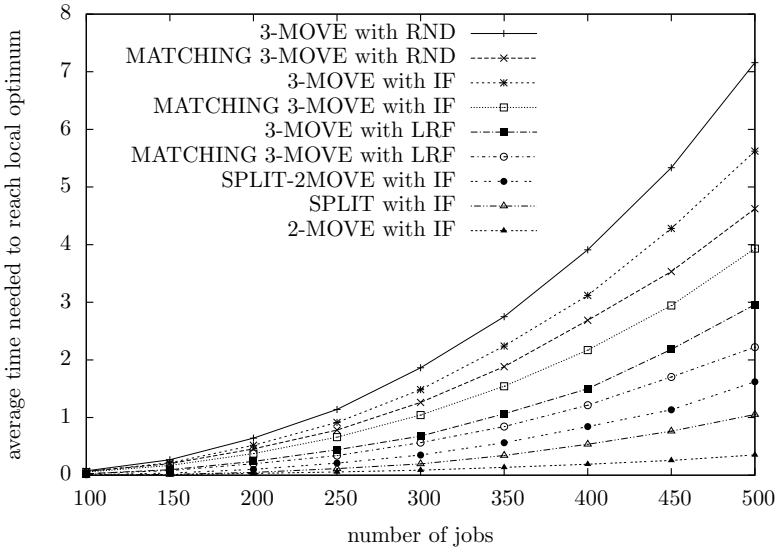

Figure 3.11: Running times for 3-MOVE and MATCHING 3-MOVE.

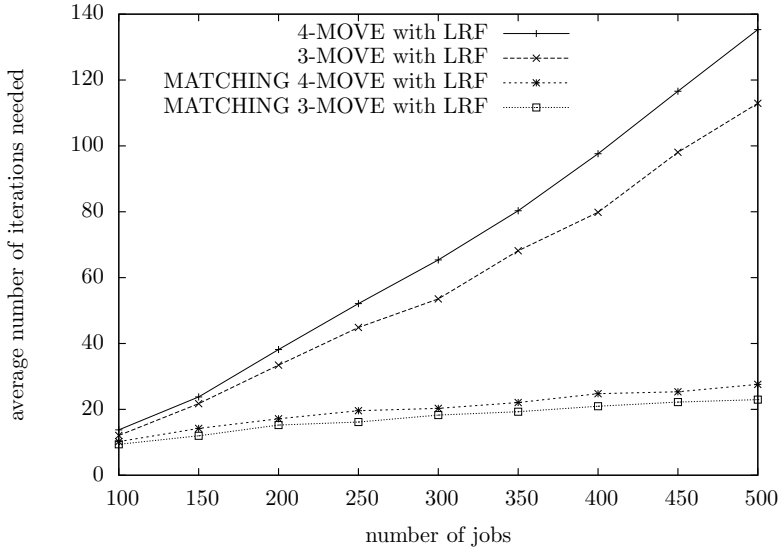

Figure 3.12: Number of iterations for k-MOVE and MATCHING k-MOVE.

the local optimum. We show this result for type (2) instances with $\frac{n}{m} = 25$ and $k = 3, 4$ in Figure 3.12. This result also holds, if the iterative improvement process is started with a different initial solution. Then only the number of iterations increases, but the situation between k-MOVE and MATCHING k-MOVE keeps the same.

The lower running time and less number of iterations for MATCHING k-MOVE compared to k-MOVE is caused by the ability to apply several operators in a single iteration. Of course, it is only possible to combine operators for $m \geq 4$. And with increasing m it is more likely that the average amount of combined operators increases. In Figure 3.13 we present the average number of edges contained in a maximum weight matching for instances of type (2) with $\frac{n}{m} = 25$. Because $m = n/25$, with increasing n also the number of machines increases and, hence, the possibility to combine several moves in one iteration increases.

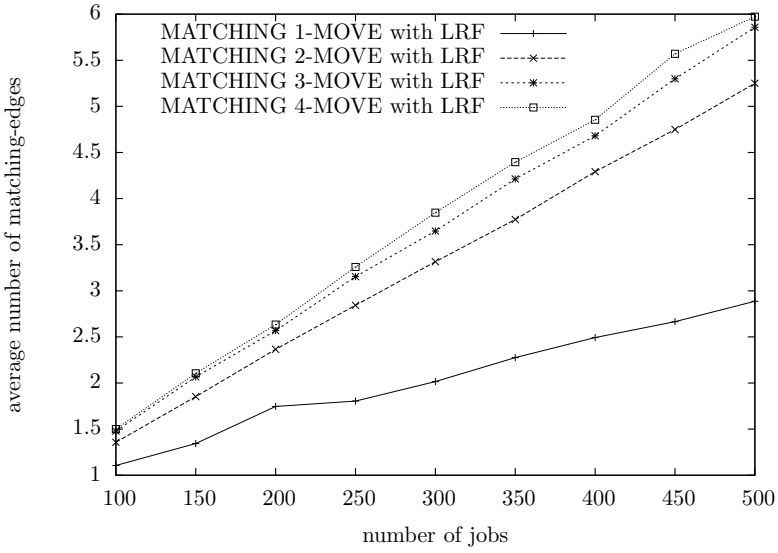

Figure 3.13: Number of edges in maximum weight matching.

3.4 Concluding Remarks

We presented a general approach to build up a neighborhood $\mathcal{N}^M_{k-move}$ of up to exponential size out of a smaller basic neighborhood $\mathcal{N}_{k-move}$. Furthermore, we presented a neighborhood $\mathcal{N}_{split(q)}$ that is based on a restricted version of the considered problem. Computational tests show that using the neighborhood $\mathcal{N}^M_{k-move}$, instead of $\mathcal{N}_{k-move}$, is on average a better choice. Doing so has an impact on the time needed to reach a local optimum. Moreover, the solution quality is not affected by using matching based methods instead of the basic ones. They deliver on average the same quality.

The neighborhood $\mathcal{N}_{split(q)}$ delivers better results compared to $\mathcal{N}^M_{1-move}$ and $\mathcal{N}_{1-move}$, but is not as good as $\mathcal{N}^M_{2-move}$ and $\mathcal{N}_{2-move}$. The average time needed to reach a local optimum is comparable to $\mathcal{N}_{2-move}$. By using the two neighborhoods in a combined approach, the quality of local optima increases, but at the cost of an increase in running time.

The results of this chapter confirm the conclusions drawn in the previous chapter and by others on the practical use of neighborhoods of exponential size (see e.g. Hurink [42]). Very large-scale neighborhoods are not good on beforehand for making local search efficient. Based on our experiences, we may conclude that the size of the neighborhood does not guarantee a better quality. And only if structural properties of the considered problem make it possible to combine several neighborhood operators, very large-scale neighborhoods, derived by combining operators may be successful in speeding up the computational time (which was the case for our problem).

Moreover, large neighborhoods resulting from restricting the considered problem are not useful as stand alone methods. Only in combination with other neighborhoods, they may help to improve the quality.

All in all, we suggest to develop and use very large-scale neighborhoods of the considered types only if problem specific properties or computational arguments on beforehand give an indication that the very large-scale neighborhoods have some potential to be a success.

Chapter 4

Performance Guarantees for Total Weighted Completion Time on Parallel Machines

In Chapter 3 we developed very large-scale neighborhoods for the problem $P||\sum w_jC_j$, introduced in Section 1.3.2. We compared the introduced neighborhoods regarding solution quality and running time by using computational tests. In this chapter we study performance guarantees for some local optima and heuristics for the same problem.

Besides the general problem, we also consider a restricted problem where the jobs have equal weight to processing time ratios. We will show that this problem is similar to the problem $P||\sum L_i^2$, also introduced in Section 1.3.2.

Considering the problem $P||\sum w_jC_j$, Smith's rule gives rise to the so-called *LRF-heuristic* (Largest Ratio First): An *LRF-assignment* is obtained by first ordering the jobs according to Smith's rule and then assigning them successively to the machine with currently minimal workload.

LRF-assignments were analyzed by Eastman et al. [29], and later by Kawaguchi

and Kyan [46]. Relative to the value $f(A^*)$ of an optimal assignment A^*, an LRF-assignment A satisfies

$$\frac{f(A)}{f(A^*)} \leq \frac{1}{2}(\sqrt{2}+1) < 1.208,$$

and this bound is tight. Indeed, examples approaching the upper bound can be found with all jobs having equal weight to processing time ratios, see Kawaguchi and Kyan [46].

The LRF-heuristic is a special case of the *LS-heuristic* (List Scheduling) introduced by Graham [37, 38], since in the LS-heuristic arbitrary sequences of the jobs are allowed before assigning the jobs iteratively to the machine with currently minimal workload. For the special case of equal weight to processing time ratios, the LRF-heuristic allows each sequence of the jobs and, therefore, coincides with the LS-heuristic. The LS-heuristic can also be understood as an online setting. In the online-model, the jobs are available at the same time, arrive one by one and have to be assigned to one of the machines without knowledge of the processing times of the following jobs. Avidor et al. [9] show that the online LS-heuristic has a competitive ratio (i.e. performance guarantee) of $\frac{4}{3}$ for minimizing the sum of squared machine workloads. Hence, the LS-heuristic has a performance guarantee of $\frac{4}{3}$ if the objective is to minimize the sum of squared machine workloads. The difference to the performance guarantee $\frac{1}{2}(\sqrt{2}+1)$ of LRF-assignments, in case of minimizing the sum of weighted job completion times for instances with equal weight to processing time ratios, results from the different objective functions.

In this chapter we study another simple heuristic: a local search which successively modifies a current assignment A by moving a job to another machine. We are interested in the quality of *move-optimal* assignments, i.e. local optima of this local search procedure. An assignment A is move-optimal if

$$f(A) \leq f(A'), \text{ for all } A' \in \mathcal{N}_{1-move},$$

where $\mathcal{N}_{1-move}$ is defined as in Section 3.3.

For the general case with arbitrary weight to processing time ratios, the main result we prove is as follows:

Theorem 4.1 *Let A be a move-optimal assignment of jobs to machines and A^* an optimal assignment. Then*

$$\frac{f(A)}{f(A^*)} \leq \frac{3}{2} - \frac{1}{2m}.$$

The worst example found so far is a small one for $m = 2$ machines and has a ratio

$$\frac{f(A)}{f(A^*)} = \frac{6}{5} = 1.2.$$

Hence, it is unclear (even in case $m = 2$) whether the bound in Theorem 4.1 is tight. If we compare this result with known performance guarantees, we can state that the performance guarantee of $\frac{1}{2}(\sqrt{2}+1)$ given by Kawaguchi and Kyan [46] for LRF-assignments lies between the worst known lower bound example (for $m = 2$) and the upper bound on the performance guarantee, for move-optimal assignments. Furthermore, it is worthwhile to mention that the upper bound of $\frac{3}{2} - \frac{1}{2m}$ is equal to the upper bound for LRF-assignments given by Eastman et al. [29] (this result is an earlier result than the bound of Kawaguchi and Kyan [46], and the proof of this bound is different from the proof of Theorem 4.1 for move-optimal assignments). Therefore, it is an interesting open question, whether it is possible to improve the performance guarantee of move-optimal assignments to a bound as Kawaguchi and Kyan [46] formed for LRF-assignments. Summarizing, for the general case the relation between LRF-assignments and move-optimal assignments stays unclear.

For the special case of equal weight to processing time ratios the situation looks a bit different. We prove the following:

Theorem 4.2 *Let all jobs have the same weight to processing time ratio. Then the objective value $f(A)$ of each move-optimal assignment A satisfies*

$$\frac{f(A)}{f(A^*)} \leq \frac{9 - \sqrt{6}}{6} < 1.092,$$

where $f(A^*)$ denotes the value of an optimal assignment A^*. Moreover, this bound is asymptotically tight.

This gives a better performance guarantee than for general LRF-assignments.

Moreover, for move-optimal assignments in the setting of minimizing the sum of squared machine workloads, we get:

Theorem 4.3 *Let A be a move-optimal assignment. Then the objective value $\widetilde{f}(A)$ of minimizing the sum of squared machine workloads satisfies*

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} \leq \frac{9}{8} = 1.125,$$

where $\widetilde{f}(A^*)$ denotes the value of an optimal assignment A^*. Moreover, this bound is tight.

Chandra and Wong [22] consider the problem of minimizing the sum of squared machine workloads, and investigate the LS-heuristic using the *Longest Processing Time* (LPT) rule of Graham [37, 38] as ordering of the jobs. They prove that the resulting LPT-assignment A satisfies

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} \leq \frac{25}{24} = 1.041\overline{6},$$

and they give a simple example with a ratio

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} = \frac{37}{36} = 1.02\overline{7},$$

where the machines are balanced in the optimal assignment. They also provide an example where in the optimal assignment the machines do not have the same workload. This example has a ratio

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} = \frac{19}{36} + \frac{5\sqrt{13}}{36} > 1.0285,$$

which they prove to be the tight example for instances with $m = 2$ machines. The general upper bound on the performance guarantee was slightly improved by Leung and Wei [49] for the number m of machines, $m \geq 3$, not being divisible by 5.

Goldberg and Shapiro [36] also examine LPT-assignments for the problem of minimizing the sum of squared machine workloads. They restrict themselves to

instances that admit an optimal solution in which all machines have the same workload. For these instances they improve the upper bound on the performance guarantee of LPT-assignments to 37/36. This is done by analyzing a quadratic programming formulation that models the deviation of workloads from the mean and calculates the variance in the objective value.

By using properties of move-optimal assignments, we are able to give performance guarantees for LPT-assignments regarding the objective function $f(A)$ in case of equal weight to processing time ratios.

Theorem 4.4 *Let all jobs have the same weight to processing time ratio and let A be an LPT-assignment. Then the objective value $f(A)$ satisfies*

$$\frac{f(A)}{f(A^*)} \leq \frac{16}{11} - \frac{8}{33}\sqrt{3} < 1.035,$$

where $f(A^*)$ denotes the value of an optimal assignment A^*.

The worst example found so far has a ratio

$$\frac{f(A)}{f(A^*)} = \frac{2}{3} + \frac{1}{4}\sqrt{2} > 1.0202.$$

Whether the upper bound on the performance guarantee of Theorem 4.4 can be improved is still unknown.

In Section 4.1 we introduce some properties of the considered problem and present the lower bound of Eastman et al. [29]. This lower bound is crucial for the proof of Theorem 4.1 given in Section 4.2. Afterwards, in Section 4.3 we examine the complexity of the considered scheduling problems. Especially, we discuss why the problems are computationally intractable. As we will see, for proving performance guarantees we are using properties making the problems intractable. In Section 4.4 we consider move-optimal assignments for instances with equal weight to processing times. There we provide the proof of Theorem 4.2 as well as Theorem 4.3. In Section 4.5 we consider LPT-assignments and use properties of move-optimal assignments to prove Theorem 4.4. Finally, in Section 4.6 we give some concluding remarks and discuss a connection of the considered scheduling problem to non-cooperative game theory.

4.1 Notation and Lower Bound

Throughout this chapter, we use the notation presented in Section 1.3.2. In the following, we introduce some properties of the considered problem and present the lower bound of Eastman et al. [29] on the value $f(A)$.

By using the partial sums of processing times and weights presented in (1.6), the objective value $f(A)$ of an assignment A is calculated as

$$f(A) = \sum_{i=1}^{m} \sum_{j \in M_i} w_j L_{ij} = \sum_{i=1}^{m} \sum_{j \in M_i} p_j \left(w_j + W_{ij}\right).$$

In order to derive a lower bound on the objective value $f(A)$ of an assignment A, we compare $f(A)$ with the optimal objective value f_1 of scheduling the whole job set on a single machine. Due to Smith [59], the value f_1 is given by:

$$f_1 = \sum_{j=1}^{n} w_j \sum_{k=1}^{j} p_k. \tag{4.1}$$

The next theorem is due to Eastman et al. [29] and gives a lower bound for the objective values of arbitrary assignments.

Theorem 4.5 (Eastman et al. [29]) *Let A be an assignment of jobs to machines. Then*

$$f(A) \geq \frac{1}{m} f_1 + \frac{m-1}{2m} \sum_{j=1}^{n} w_j p_j.$$

PROOF. Since several ingredients of the proof given by Eastman et al. [29] are of importance, this proof is given in the following. The proof is based on an induction on the number of distinct weight to processing time ratios.

In a first step we consider instances with equal weight to processing time ratios, i.e. we assume

$$\frac{w_j}{p_j} = r, \text{ for all jobs } j = 1, \ldots, n. \tag{4.2}$$

Observe that the order of the jobs on any machine is of no importance, since the equality of the weight to processing time ratios implies that Smith's rule is automatically obeyed for all job sequences.

By exploiting (4.2), the objective value $f(A)$ of an assignment A is calculated as:

$$\begin{aligned} f(A) &= \sum_{i=1}^{m} \sum_{j \in M_i} w_j L_{ij} = \sum_{i=1}^{m} \sum_{j \in M_i} w_j \sum_{\substack{k \in M_i \\ k \leq j}} p_k = r \sum_{i=1}^{m} \sum_{j \in M_i} \sum_{\substack{k \in M_i \\ k \leq j}} p_j p_k \\ &= \frac{r}{2} \sum_{i=1}^{m} \sum_{j \in M_i} \left(\sum_{k \in M_i} p_j p_k + p_j^2 \right) = \frac{r}{2} \sum_{i=1}^{m} \left(L_i^2 + \sum_{j \in M_i} p_j^2 \right) \\ &= \frac{r}{2} \sum_{i=1}^{m} L_i^2 + \frac{r}{2} \sum_{j=1}^{n} p_j^2 = \frac{r}{2} \sum_{i=1}^{m} L_i^2 + \frac{1}{2} \sum_{j=1}^{n} w_j p_j. \end{aligned} \tag{4.3}$$

On the other hand, also the optimal objective value f_1 of the single machine scheduling problem can be simplified using (4.1):

$$\begin{aligned} f_1 &= \sum_{j=1}^{n} \sum_{k=1}^{j} w_j p_k = r \sum_{j=1}^{n} \sum_{k=1}^{j} p_j p_k = \frac{r}{2} \left(\sum_{j=1}^{n} \sum_{k=1}^{n} p_j p_k + \sum_{j=1}^{n} p_j^2 \right) \\ &= \frac{r}{2} \left(\sum_{j=1}^{n} p_j \right)^2 + \frac{r}{2} \sum_{j=1}^{n} p_j^2 = \frac{r}{2} \left(\sum_{j=1}^{n} p_j \right)^2 + \frac{1}{2} \sum_{j=1}^{n} w_j p_j. \end{aligned} \tag{4.4}$$

In order to give a lower bound on $f(A)$, we first consider the sum of squared workloads. With the help of the quadratic-arithmetic mean inequality (a special case of Cauchy's inequality), we obtain:

$$\sum_{i=1}^{m} L_i^2 \geq \frac{1}{m} \left(\sum_{j=1}^{n} p_j \right)^2. \tag{4.5}$$

Using (4.4) and (4.5) on (4.3) delivers:

$$\begin{aligned} f(A) &= \frac{r}{2}\sum_{i=1}^{m} L_i^2 + \frac{1}{2}\sum_{j=1}^{n} w_j p_j \geq \frac{r}{2m}\left(\sum_{j=1}^{n} p_j\right)^2 + \frac{1}{2}\sum_{j=1}^{n} w_j p_j \\ &= \frac{1}{m}\left(f_1 - \frac{1}{2}\sum_{j=1}^{n} w_j p_j\right) + \frac{1}{2}\sum_{j=1}^{n} w_j p_j \\ &= \frac{1}{m} f_1 + \frac{m-1}{2m}\sum_{j=1}^{n} w_j p_j. \end{aligned}$$

Thus, we have proven Theorem 4.5 in the case that (4.2) holds.

Now we proceed in proving Theorem 4.5 by induction on the number of distinct weight to processing time ratios. For this, we assume that Theorem 4.5 holds for every assignment A and instances with $l-1$ distinct weight to processing time ratios. We prove that the theorem also holds for instances with l distinct weight to processing time ratios. Let q be the job with

$$\frac{w_1}{p_1} = \ldots = \frac{w_q}{p_q} > \frac{w_{q+1}}{p_{q+1}} \geq \ldots \geq \frac{w_n}{p_n}.$$

Consider an instance with the same processing times $p'_j := p_j$ and the following weights:

$$w'_j := \begin{cases} \varepsilon w_j & \text{for } j = 1, \ldots, q, \\ w_j & \text{otherwise,} \end{cases}$$

where $\varepsilon < 1$ is chosen to be

$$\varepsilon := \frac{w_{q+1} p_q}{p_{q+1} w_q}.$$

Observe that in the changed instance we have:

$$\frac{w'_1}{p_1} = \ldots = \frac{w'_q}{p_q} = \frac{w'_{q+1}}{p_{q+1}} \geq \ldots \geq \frac{w'_n}{p_n}.$$

Hence, the number of distinct weight to processing time ratios is reduced by one. We denote by $f'(A)$ the objective value of assignment A and by f'_1 the optimal objective value of the single machine problem in the changed instance.

Because of the induction assumption, we know that

$$f'(A) \geq \frac{1}{m} f'_1 + \frac{m-1}{2m} \sum_{j=1}^{n} w'_j p_j. \tag{4.6}$$

Moreover, the values for the original instance and the changed instance are related:

$$\begin{aligned} f(A) &= f'(A) + (1-\varepsilon) \sum_{j=1}^{q} w_j C_j, \\ f'_1 &= f_1 - (1-\varepsilon) \sum_{j=1}^{q} w_j \sum_{k=1}^{j} p_k, \\ \sum_{j=1}^{n} w'_j p_j &= \sum_{j=1}^{n} w_j p_j - (1-\varepsilon) \sum_{j=1}^{q} w_j p_j. \end{aligned}$$

Hence, by using (4.6) we obtain

$$\begin{aligned} f(A) &= f'(A) + (1-\varepsilon) \sum_{j=1}^{q} w_j C_j \\ &\geq \frac{1}{m} f'_1 + \frac{m-1}{2m} \sum_{j=1}^{n} w'_j p_j + (1-\varepsilon) \sum_{j=1}^{q} w_j C_j \\ &= \frac{1}{m} f_1 + \frac{m-1}{2m} \sum_{j=1}^{n} w_j p_j + \\ &\quad (1-\varepsilon) \cdot \left(\sum_{j=1}^{q} w_j C_j - \frac{1}{m} \sum_{j=1}^{q} w_j \sum_{k=1}^{j} p_k - \frac{m-1}{2m} \sum_{j=1}^{q} w_j p_j \right). \end{aligned} \tag{4.7}$$

Since all jobs $1, \ldots, q$ have higher priority than the remaining jobs, all these jobs are scheduled on their machines before the remaining jobs. Therefore, $\sum_{j=1}^{q} w_j C_j$ can be seen as the objective value of the assignment A restricted to the jobs $\{1, \ldots, q\}$. Since this reduced instance has only one distinct weight to processing time ratio, we can apply the result of Theorem 4.5 to $\sum_{j=1}^{q} w_j C_j$, i.e.

$$\sum_{j=1}^{q} w_j C_j \geq \frac{1}{m} \sum_{j=1}^{q} w_j \sum_{k=1}^{j} p_k + \frac{m-1}{2m} \sum_{j=1}^{q} w_j p_j$$

holds. Incorporating this in (4.7) yields

$$f(A) \geq \frac{1}{m} f_1 + \frac{m-1}{2m} \sum_{j=1}^{n} w_j p_j. \qquad \square$$

Eastman et al. [29] give an instance, for which an optimal assignment attains the lower bound for each m. Their example consists of a set of jobs having two different weight to processing time ratios. We present another simple example with all jobs having the same weight to processing time ratio.

The example has $n = 4$ jobs with weight 1 and processing time 1. These 4 jobs have to be scheduled on $m = 2$ machines. An optimal assignment A^* schedules 2 jobs on the first machine and 2 jobs on the second machine, yielding an optimal objective value $f(A^*) = 6$. The optimal objective value of scheduling the 4 jobs on only a single machine is calculated as $f_1 = 10$. Hence, we obtain for the lower bound on $f(A^*)$, according to Theorem 4.5:

$$f(A^*) \geq \frac{1}{m} f_1 + \frac{m-1}{2m} \sum_{j=1}^{n} w_j p_j = 6.$$

4.2 Different Weight to Processing Time Ratios

In order to derive an upper bound for the performance guarantee of a move-optimal assignment A, we compare the objective value $f(A)$ with the optimal objective value f_1 of the single machine problem with the same set of jobs.

Let A' be an assignment arising from A by reassigning a job j from its machine $A(j)$ to machine i. Observe that the job has to be inserted on machine i at the appropriate position (obeying Smith's rule). As stated in (3.1), the change in the objective value is given by:

$$\begin{aligned} f(A) - f(A') &= \delta_{move(j,i)} \\ &= w_j \left(L_{A(j),j} - L_{ij} - p_j\right) + p_j \left(W_{A(j),j} - W_{ij}\right). \end{aligned} \tag{4.8}$$

Since in a move-optimal assignment we can find no job and target machine that gives an improvement in the objective value, all differences in (4.8) have to be

non-positive. Thus, if we define

$$\Delta_j := w_j L_{A(j),j} + p_j W_{A(j),j},$$

for a move-optimal assignment A the following inequalities hold for all jobs $j = 1, \ldots, n$ and all machines $i \neq A(j)$:

$$\Delta_j \leq w_j L_{ij} + w_j p_j + p_j W_{ij}. \tag{4.9}$$

Furthermore, the values Δ_j and the objective value $f(A)$ are related. If we sum up all Δ_j, for all jobs $j \in M_i$ of a fixed machine i, we get:

$$\sum_{j \in M_i} \Delta_j = \sum_{j \in M_i} w_j L_{ij} + \sum_{j \in M_i} p_j W_{ij} = 2 \sum_{j \in M_i} w_j L_{ij} - \sum_{j \in M_i} w_j p_j.$$

Therefore, for any assignment A and the corresponding objective value $f(A)$ we have

$$2f(A) = 2 \sum_{i=1}^{m} \sum_{j \in M_i} w_j L_{ij} = \sum_{i=1}^{m} \sum_{j \in M_i} \left(\Delta_j + w_j p_j \right). \tag{4.10}$$

Our goal is to bound the objective value $f(A)$ of a move-optimal assignment A in terms of f_1. To calculate the starting time of a job in an optimal schedule for the single machine problem we have to add the processing times of all jobs with higher priority. Therefore, for a given assignment A we can expand the sums in the objective value f_1 to

$$f_1 = \sum_{i=1}^{m} \sum_{j \in M_i} w_j \sum_{t=1}^{m} L_{tj}. \tag{4.11}$$

By a similar argument, we obtain:

$$f_1 = \sum_{i=1}^{m} \sum_{j \in M_i} p_j \left(w_j + \sum_{t=1}^{m} W_{tj} \right). \tag{4.12}$$

Adding (4.11) and (4.12) yields

$$\begin{aligned}2f_1 &= \sum_{i=1}^{m}\sum_{j\in M_i} w_jL_{ij} + \sum_{i=1}^{m}\sum_{j\in M_i} p_j\,(w_j + W_{ij}) + \\ &\quad \sum_{i=1}^{m}\sum_{j\in M_i} w_j \sum_{\substack{t=1\\t\neq i}}^{m} L_{tj} + \sum_{i=1}^{m}\sum_{j\in M_i} p_j \sum_{\substack{t=1\\t\neq i}}^{m} W_{tj} \\ &= 2f(A) + \sum_{i=1}^{m}\sum_{j\in M_i}\sum_{\substack{t=1\\t\neq i}}^{m} (w_jL_{tj} + p_jW_{tj})\,.\end{aligned}$$

If we now incorporate (4.9) and use (4.10) afterwards, the following is obtained:

$$\begin{aligned}2f_1 &\geq 2f(A) + \sum_{i=1}^{m}\sum_{j\in M_i}\sum_{\substack{t=1\\t\neq i}}^{m} (\Delta_j - w_jp_j) \\ &= 2f(A) + (m-1)\sum_{i=1}^{m}\sum_{j\in M_i} (\Delta_j - w_jp_j) \\ &= 2f(A) + 2(m-1)f(A) - 2(m-1)\sum_{i=1}^{m}\sum_{j\in M_i} w_jp_j \\ &= 2mf(A) - 2(m-1)\sum_{j=1}^{n} w_jp_j.\end{aligned}$$

Thus, a move-optimal assignment A satisfies

$$f(A) \leq \frac{1}{m}f_1 + \frac{m-1}{m}\sum_{j=1}^{n} w_jp_j. \tag{4.13}$$

Since $f(A^*) \geq \sum_{j=1}^{n} w_jp_j$ holds, we can conclude from Theorem 4.5 that for the optimal assignment A^* we have

$$f(A^*) \geq \alpha\left(\frac{1}{m}f_1 + \frac{m-1}{2m}\sum_{j=1}^{n} w_jp_j\right) + (1-\alpha)\sum_{j=1}^{n} w_jp_j \tag{4.14}$$

for every $\alpha \in [0,1]$.

Substituting $\alpha = \frac{2m}{3m-1}$ in (4.14), we obtain

$$f(A^*) \geq \frac{2}{3m-1} f_1 + \frac{2m-2}{3m-1} \sum_{j=1}^{n} w_j p_j.$$

Using (4.13), for $f(A)/f(A^*)$ we get

$$\frac{f(A)}{f(A^*)} \leq \frac{f_1 + (m-1) \sum_{j=1}^{n} w_j p_j}{m} \cdot \frac{3m-1}{2f_1 + (2m-2) \sum_{j=1}^{n} w_j p_j}.$$

This simplifies to

$$\frac{f(A)}{f(A^*)} \leq \frac{3}{2} - \frac{1}{2m}, \tag{4.15}$$

proving Theorem 4.1.

Consider the following example consisting of 4 jobs and 2 machines.

job j	1	2	3	4
p_j	1	1	2	2
w_j	1	1	$\frac{1}{2}$	$\frac{1}{2}$

The assignment A in Figure 4.1 is move-optimal and has $f(A) = 6$, whereas the optimum is $f(A^*) = 5$. So $f(A)/f(A^*) = 6/5 = 1.2$ which is smaller than $5/4 = 1.25$ (obtained from (4.15) with $m = 2$). For any even number of machines we get the ratio 6/5 by taking multiple copies of the instance for two machines. This is summarized by the following lemma.

Lemma 4.6 *The upper bound on the performance guarantee for move-optimal assignments is at least* $\frac{6}{5}$.

4.3 Review on Complexity

The hardness of the considered problems arise from the problem of distributing the workload equally to the machines, as we show in this section. This leveling

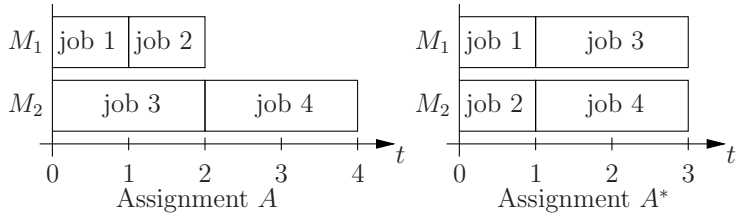

Figure 4.1: Gantt-charts for worst example found so far.

is an important issue in proving certain performance guarantees. The hardness of the considered problems can be shown by derivation from the 3-partition problem, which is $\mathcal{NP}$-complete in the strong sense as shown by Garey and Johnson [34]. For the 3-partition problem, we use the notation as introduced in Section 1.1.

The following lemma illustrates, why the considered problems are hard.

Lemma 4.7 (Garey and Johnson [34], Lenstra et al. [48]) *Below, let I be an instance of the 3-partition problem and I' be an instance obtained by a polynomial transformation for one of the following scheduling problems. The instance I is a yes-instance for the 3-partition problem if and only if for the corresponding scheduling problem the instance I' has an objective value given by the following list:*

(P1) $P||C_{\max}$ *has makespan*

$$C_{\max} \leq b,$$

(P2) $P||\sum L_i^2$ *has objective value*

$$\sum_{i=1}^{m} L_i^2 \leq mb^2,$$

(P3) $P|w_j = p_j|\sum w_jC_j$ *has objective value*

$$\sum_{j=1}^{3m} w_jC_j \leq \frac{m}{2}b^2 + \frac{1}{2}\sum_{j=1}^{3m} a_j^2.$$

Moreover, the instance I is a yes-instance for the 3-partition problem if and only if in the optimal solutions for the corresponding scheduling problems all machines have a workload equal to b.

PROOF. Parts of the proof can be found in Lenstra et al. [48] and problem [SS13] of Garey and Johnson [34].

We first describe how to transform the instance I polynomially to an instance I' for the corresponding scheduling problem. For any instance I, we create an instance I' with $m = t$ machines and $3m$ jobs $1, \ldots, 3m$ with processing times $p_j = a_j$ to be scheduled on the m machines. For the problem $P|w_j = p_j|\sum w_j C_j$ we additionally define the weight of a job j to be $w_j = p_j = a_j$, in order to obtain a valid instance.

Observe that in none of the above given scheduling problems the order of the jobs on the machines is of importance. Hence, a feasible solution for any considered scheduling problem can be given as an assignment of jobs to machines.

First assume that $T_1, \ldots, T_m$ is a feasible solution for the instance I of the 3-partition problem. We define an assignment A by putting the job j, corresponding to an element $j \in T_i$, to machine i, i.e. $A(j) := i$, for all $j \in T_i$ and $i = 1, \ldots, m$. Thus, $M_i = T_i$ for all machines $i = 1, \ldots, m$. Then the workload of a machine i is calculated as

$$L_i = \sum_{j \in M_i} p_j = b.$$

The objective value of the assignment A for the corresponding scheduling problems is given by:

(P1) The makespan is determined by the maximum workload and hence, equals b.

(P2) Since $L_i = b$ for all machines i, we get for the objective value

$$\widetilde{f}(A) = \sum_{i=1}^{m} L_i^2 = mb^2.$$

(P3) By using (4.3), with $r = \frac{w_j}{p_j} = 1$, we obtain

$$f(A) = \frac{m}{2}b^2 + \frac{1}{2}\sum_{j=1}^{3m} a_j^2.$$

We are left to prove the other direction. For this, let A always be an assignment for the corresponding scheduling problem with the desired objective value. We define a partition $T_i := M_i = A^{-1}(i)$, i.e. the i-th set contains all elements corresponding to jobs assigned to machine i. Observe that if for a workload of machine i holds $L_i = b$, then T_i contains exactly 3 elements due to (1.1). Furthermore, we get the following:

(P1) The makespan is less than b, hence there exists no machine having a workload greater than b. Since (1.1), all machines must have a workload equal to b. Therefore, the partition $T_1, \ldots, T_m$ is a feasible solution for the instance I of the 3-partition problem.

(P2) For every machine i we define the deviation of the workload from b as $c_i := L_i - b$. For the workload holds

$$\sum_{i=1}^{m} L_i^2 = \sum_{i=1}^{m} (c_i + b)^2 = \sum_{i=1}^{m} c_i^2 + mb^2 + 2b\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} c_i^2 + mb^2.$$

Since $\sum_{i=1}^{m} L_i^2 \leq mb^2$, we must have $\sum_{i=1}^{m} c_i^2 = 0$ and all machines have a workload equal to b. Therefore, the partition $T_1, \ldots, T_m$ is a feasible solution for the instance I of the 3-partition problem.

(P3) By using (1.1), this can be done in the same way as for (P2) with the help of the deviations c_i.

This completes the proof. □

Lemma 4.7 indicates that the makespan and the balance of the workloads play an important role in the problems $P||\sum L_i^2$ and $P||\sum w_jC_j$. In the following sections, we make use of this idea, i.e. we analyze the makespan by looking for a way to balance the workload of the machines. With the previous lemma we obtain:

Theorem 4.8 *The following problems are $\mathcal{NP}$-hard in the strong sense:*

- $P||C_{\max}$,
- $P||\sum L_i^2$,
- $P|w_j = p_j|\sum w_j C_j$,
- $P|\frac{w_j}{p_j} = r|\sum w_j C_j$,
- $P||\sum w_j C_j$.

4.4 Equal Weight to Processing Time Ratios

In what follows, we assume that all jobs have equal weight to processing time ratios and prove an upper bound $(9-\sqrt{6})/6$ on the performance guarantee of move-optimal assignments for the objective to minimize the sum of weighted job completion times. Because (4.2) holds, according to (4.3), we can write the objective value $f(A)$ of an assignment A as

$$f(A) = \frac{r}{2}\sum_{i=1}^{m} L_i^2 + \frac{r}{2}\sum_{j=1}^{n} p_j^2.$$

In the following, let A denote a move-optimal assignment and A^* an optimal assignment. We are interested in an upper bound for the ratio

$$\frac{f(A)}{f(A^*)} = \frac{\sum\limits_{i=1}^{m}\left(L_i^A\right)^2 + \sum\limits_{j=1}^{n} p_j^2}{\sum\limits_{i=1}^{m}\left(L_i^{A^*}\right)^2 + \sum\limits_{j=1}^{n} p_j^2} = 1 + \frac{\sum\limits_{i=1}^{m}\left(L_i^A\right)^2 - \sum\limits_{i=1}^{m}\left(L_i^{A^*}\right)^2}{\sum\limits_{i=1}^{m}\left(L_i^{A^*}\right)^2 + \sum\limits_{j=1}^{n} p_j^2}. \tag{4.16}$$

To simplify notation, we always scale the processing times and weights such that:

$$\sum_{j=1}^{n} p_j = m.$$

Moreover, we reorder the machines, such that $L_1 \geq L_2 \geq \ldots \geq L_m$ holds. Observe that for the sum of workloads holds

$$\sum_{i=1}^{m} L_i = \sum_{j=1}^{n} p_j = m. \tag{4.17}$$

Different from Section 4.2, let $\Delta_i := L_i - L_m$ denote the deviation of the workload of machine i to the minimal workload L_m. From (4.17) we get:

$$\sum_{i=1}^{m} \Delta_i = m(1 - L_m). \tag{4.18}$$

Using $L_i = \Delta_i + L_m$, we rewrite the sum of squared machine workloads in terms of Δ_i and L_m by exploiting (4.18):

$$\sum_{i=1}^{m} L_i^2 = \sum_{i=1}^{m} \Delta_i^2 + mL_m(2 - L_m). \tag{4.19}$$

Combining (4.17) with the lower bound (4.5), we get:

$$\sum_{i=1}^{m} L_i^2 \geq \frac{1}{m} \left(\sum_{j=1}^{n} p_j \right)^2 = m. \tag{4.20}$$

While the above holds for all assignments A, we now use the move-optimality of A to yield a lower bound for the processing times of jobs.

Lemma 4.9 *Let A be a move-optimal assignment. For all jobs $j \in M_i$ we have $p_j \geq \Delta_i$.*

PROOF. Assume that, for a job $j \in M_i$, we have $p_j < \Delta_i$. Consider the assignment A' arising from A by assigning job j to machine m instead of i. We obtain for the workloads:

$$\left(L_i^{A'}\right)^2 = \left(L_i^A - p_j\right)^2 = \left(L_i^A\right)^2 + p_j^2 - 2L_i^A p_j,$$
$$\left(L_m^{A'}\right)^2 = \left(L_m^A + p_j\right)^2 = \left(L_m^A\right)^2 + p_j^2 + 2L_i^A p_j.$$

By using (4.3), we obtain for the difference of the objective values $f(A)$ and $f(A')$:

$$f(A) - f(A') = p_j \left(L_i^A - L_m^A - p_j\right) = p_j \left(\Delta_i - p_j\right) > 0.$$

This contradicts the move-optimality of assignment A. □

By using (4.18) we obtain

$$\begin{aligned} \sum_{i=1}^{m} \Delta_i L_i &= \sum_{i=1}^{m} \Delta_i (L_m + \Delta_i) = L_m \sum_{i=1}^{m} \Delta_i + \sum_{i=1}^{m} \Delta_i^2 \\ &= \sum_{i=1}^{m} \Delta_i^2 + m L_m (1 - L_m). \end{aligned} \tag{4.21}$$

With the help of Lemma 4.9 we obtain the following for machine i:

$$\sum_{j \in M_i} p_j^2 \geq \Delta_i \sum_{j \in M_i} p_j = \Delta_i L_i.$$

Summing over all machines i leads, together with (4.21), to:

$$\begin{aligned} \sum_{j=1}^{n} p_j^2 &= \sum_{i=1}^{m} \sum_{j \in M_i} p_j^2 \geq \sum_{i=1}^{m} \Delta_i L_i \\ &= \sum_{i=1}^{m} \Delta_i^2 + m L_m (1 - L_m). \end{aligned} \tag{4.22}$$

Using (4.19), (4.20) and (4.22) we obtain for the performance guarantee (4.16) the following:

$$\frac{f(A)}{f(A^*)} \leq 2 - \frac{m(2 - L_m)}{\sum\limits_{i=1}^{m} \Delta_i^2 + m(1 + L_m - L_m^2)}. \tag{4.23}$$

Since we have $L_m \leq 1$, the numerator and the denominator are positive. In order to simplify (4.23) we have to give an upper bound for $\sum_{i=1}^{m} \Delta_i^2$. We do this by looking for an α that satisfies

$$\Delta_i \leq \alpha L_m, \text{ for all machines } i = 1, \ldots, m.$$

Exploiting this upper bound and using (4.18), we get for $\sum_{i=1}^{m} \Delta_i^2$:

$$\sum_{i=1}^{m} \Delta_i^2 \leq \alpha L_m \sum_{i=1}^{m} \Delta_i = m(\alpha L_m - \alpha L_m^2).$$

Incorporating this into (4.23) we obtain:

$$\frac{f(A)}{f(A^*)} \leq 2 - \frac{2 - L_m}{1 + L_m\,(1+\alpha) - L_m^2\,(1+\alpha)}. \tag{4.24}$$

In order to give an upper bound on α, in the following lemma we consider move-optimal assignments that assign many jobs to a single machine i, and show that the corresponding value Δ_i becomes small.

Lemma 4.10 *Let A be a move-optimal assignment and i be a machine with n_i assigned jobs. Then*

$$\Delta_i \leq \frac{1}{n_i - 1} L_m.$$

PROOF. Let j be a job with minimal processing time assigned to machine i. Then, due to Lemma 4.9,

$$L_i - L_m = \Delta_i \leq p_j \leq \frac{1}{n_i} L_i.$$

This simplifies to

$$L_i \leq \frac{n_i}{n_i - 1} L_m.$$

Hence, we obtain for Δ_i:

$$\Delta_i \leq \frac{1}{n_i} L_i \leq \frac{1}{n_i} \cdot \frac{n_i}{n_i - 1} L_m = \frac{1}{n_i - 1} L_m. \qquad \square$$

While the previous lemma is suitable for many jobs assigned to a single machine, the next lemma concentrates on machines containing only one job.

Lemma 4.11 *Let I be an instance, A be a move-optimal assignment and A^* be an optimal assignment with $f(A)/f(A^*) > 1$. If there is only one job j assigned*

to machine i, with $L_i > 1$, then deleting job j and machine i strictly increases the ratio $f(A)/f(A^*)$.

PROOF. We have $p_j > 1$. We claim that job j is scheduled alone in any move-optimal assignment A'. Since A^* is also move-optimal, and by denoting the objective values of the assignments restricted to the changed instance with f', this leads to

$$\frac{f'(A)}{f'(A^*)} = \frac{f(A) - p_j^2}{f(A^*) - p_j^2} > \frac{f(A)}{f(A^*)}.$$

In order to prove our claim, we assume on the contrary, that job j is not scheduled alone on machine i in a move-optimal assignment A'. Let j_0 be a job also scheduled on machine i. By using the fact $p_j > 1 \geq L_m$, we obtain $p_{j_0} \leq L_i - p_j < L_i - L_m = \Delta_i$, contradicting Lemma 4.9. □

Since we are interested in finding instances which maximize the right hand side of (4.24), by Lemma 4.11 we only have to consider instances and move-optimal assignments such that

$$n_i \geq 2, \text{ for all machines } i \text{ with } L_i > 1.$$

The next lemma proves the upper bound on the performance guarantee as given by Theorem 4.2.

Lemma 4.12 *Let A be a move-optimal assignment. Then*

$$\frac{f(A)}{f(A^*)} \leq \frac{9 - \sqrt{6}}{6} < 1.092. \tag{4.25}$$

PROOF. In the case that $L_1 = 1$, it holds $L_i = 1$ for all machines i, and thus $f(A)/f(A^*) = 1$.

On the other hand, consider $L_1 > 1$. Due to Lemma 4.11, we assume, without loss of generality, that at least 2 jobs are assigned to machine 1. With the help of

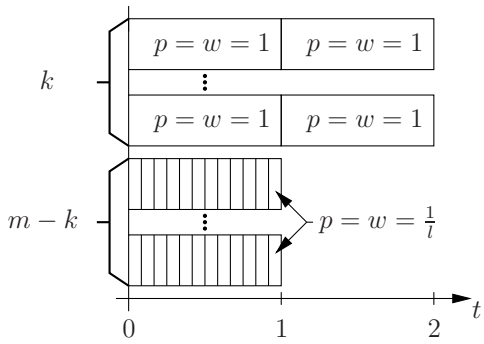

Figure 4.2: Move-optimal assignment A for equal ratios.

Lemma 4.10 we obtain $\Delta_1 \leq L_m$. Moreover, $L_1 \geq L_i$ and thus $L_m \geq \Delta_1 \geq \Delta_i$, for all machines $i = 1, \ldots, m$. The performance guarantee (4.24) simplifies to

$$\frac{f(A)}{f(A^*)} \leq 2 - \frac{2 - L_m}{1 + 2L_m - 2L_m^2}.$$

The maximum is obtained at $L_m = 2 - \frac{\sqrt{6}}{2} \approx 0.775$ yielding (4.25). $\square$

In the following we show that there exists instances and move-optimal assignments approaching the upper bound on the performance guarantee. Consider the following lower bound examples for m machines found by Vredeveld [62]. Let k be an integer with $0 < k \leq \frac{m}{2}$ and l be an integer multiple of m. There are $2k$ jobs with $p = w = 1$ and $(m-k)l$ jobs with $p = w = 1/l$. The move-optimal assignment A shown in Figure 4.2 schedules on each of the first k machines two of the jobs with $p = 1$. Moreover, on each of the last $m - k$ machines l of the jobs with $p = 1/l$ are scheduled. The assignment A has an objective value

$$f(A) = \frac{1}{2}\Big(5k + m + \frac{1}{l}\,(k - m)\Big).$$

In an optimal assignment A^* (see Figure 4.3) kl/m jobs with $p = 1/l$ and 1 job with $p = 1$ are scheduled on each of the first $2k$ machines. Moreover, on each of the last $m - 2k$ machines $l + kl/m$ jobs with $p = 1/l$ are scheduled, yielding

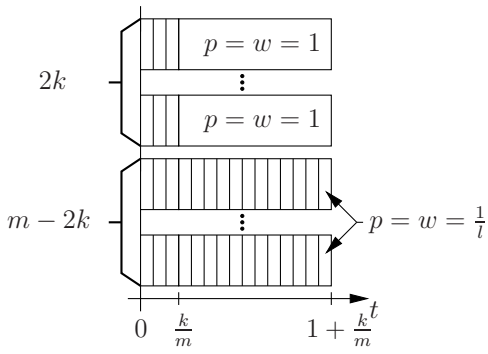

Figure 4.3: Optimal assignment A^* for equal ratios.

an objective value

$$f(A^*) = \frac{1}{2}\Big(\frac{k^2}{m} + 4k + m + \frac{1}{l}\,(k-m)\Big).$$

For $k := \alpha m$, $l \to \infty$, $\alpha \to \frac{\sqrt{6}-1}{5}$ (i.e. ≈ 0.290) and m sufficiently large so that $k \in \mathbb{N}$, the ratio $f(A)/f(A^*)$ approaches the desired value:

$$\frac{f(A)}{f(A^*)} \to \frac{9-\sqrt{6}}{6}.$$

So, the bound (4.25) is asymptotically tight. This proves Theorem 4.2.

In the remaining part we prove Theorem 4.3 in the same way as Theorem 4.2 was proven. We consider the objective function $\widetilde{f}(A)$ as given by (1.7). For a move-optimal assignment A and an optimal assignment A^* we are interested in an upper bound on the ratio

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} = \frac{\sum\limits_{i=1}^{m} \big(L_i^A\big)^2}{\sum\limits_{i=1}^{m} \big(L_i^{A^*}\big)^2}.$$

We again scale the processing times so that $\sum_{j=1}^{n} p_j = m$. By using (4.20) we obtain

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} \leq \frac{1}{m}\sum_{i=1}^{m}\left(L_i^A\right)^2.$$

First we notice that Lemma 4.9, Lemma 4.10 and Lemma 4.11 also hold if the objective function $\widetilde{f}(A)$ is used. In order to give an upper bound on the performance guarantee, similar to the proof of Lemma 4.12, we only have to consider instances and move-optimal assignments A with

$$\Delta_i \leq L_m, \text{ for all machines } i = 1, \ldots, m.$$

By using (4.18) and (4.19) we get

$$\begin{aligned}\sum_{i=1}^{m} L_i^2 &= \sum_{i=1}^{m}\Delta_i^2 + mL_m(2-L_m) \leq L_m\sum_{i=1}^{m}\Delta_i + mL_m(2-L_m)\\ &= mL_m(1-L_m) + mL_m(2-L_m) = mL_m(3-2L_m).\end{aligned}$$

With the help of this, the performance guarantee is bounded by

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} \leq L_m(3-2L_m).$$

The maximum is obtained at $L_m = \frac{3}{4}$ yielding

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} \leq \frac{9}{8}. \tag{4.26}$$

A worst-case example for 3 machines yielding this bound is as follows. There are 3 jobs with processing time $p = 1$ and 3 jobs with processing time $p = 1/3$. The move-optimal assignment A shown in Figure 4.4 has an objective value $\widetilde{f}(A) = 6 = 18/3$. The optimal assignment A^* again has balanced machines, yielding an objective value $\widetilde{f}(A^*) = 16/3$. Therefore

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} = \frac{9}{8}.$$

So, the bound (4.26) is tight. This proves Theorem 4.3.

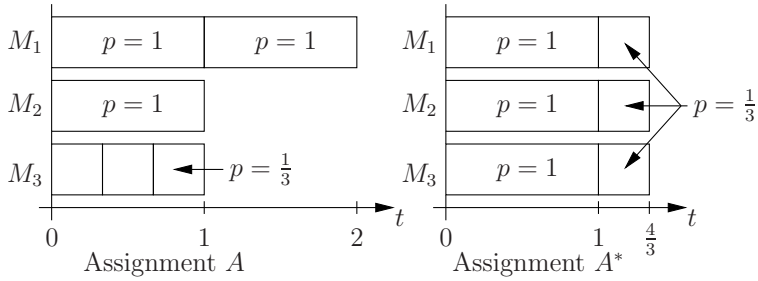

Figure 4.4: Worst case example for $\widetilde{f}(A)$.

4.5 LPT-Assignments

In the following part, we examine LPT-assignments and their performance guarantee for the problem of minimizing $f(A)$ in case of equal weight to processing time ratios, i.e. $w_j/p_j = r$, for all jobs j. We prove Theorem 4.4 by using properties of move-optimal assignments.

We extend the definition of partial workloads (1.6) by defining $L_{i,0} := 0$, for all machines i. For an assignment A, we denote with S_j the starting time of job j given by $S_j := L_{i,j-1}$. We reorder the machines without loss of generality, so that $L_1 \geq \ldots \geq L_m$. For sake of simplicity, we assume without loss of generality that $p_1 \geq \ldots \geq p_n$. An LPT-assignment is achieved by iteratively assigning the remaining job with largest processing time to a machine with minimal workload. The algorithm is given in Figure 4.5 and runs in $\mathcal{O}(n \log n)$ if the jobs are not sorted by their processing times and in $\mathcal{O}(n \log m)$ otherwise.

A direct consequence of this construction is that an assignment A is an *LPT-assignment*, if the following conditions hold:

- if for a pair of jobs j and k the relation $S_j > S_k$ holds, then: $p_j \leq p_k$,
- for every job j we have $S_j \leq L$, with $L := \min_{i=1}^{m} L_i$.

Because of the first property, we know that smaller jobs receive a larger starting

Algorithm: LPT

input: $p_1 \geq \ldots \geq p_n$.
output: an LPT-assignment A.
1: **for** $j := 1$ **to** n **do**
2: Determine i so that $L_{i,j-1} \leq L_{k,j-1}$, for all machines $k = 1, \ldots, m$;
3: $A(j) := i$;
4: **end for**;

Figure 4.5: Algorithm LPT.

time. This is contrary to LRF-assignments, where it may occur that the last job processed by a machine has largest processing time, among the jobs assigned to it. Combining the first with the second property yields, that any LPT-assignment is also move-optimal and we may replace the second property by the move-optimality property.

Furthermore, we make the following observations:

- if we have two jobs k and l, with $S = S_k = S_l$, we may swap the schedules from S onwards on the two machines jobs k and l are assigned to, without changing the objective value or any starting time of a job and without loosing the property that A is an LPT-assignment (see Figure 4.6).
- In the following, we investigate a situation where we decrease the processing time of a certain job j. To have better control and not to lose the property that A is an LPT-assignment, we have to ensure that the machine i (job j is assigned to) has a special schedule. More precisely, if on machine i a job k is scheduled after job j, and if some other job l has $S_l = S_k$, we must have $p_k \geq p_l$.

Due to the above made observation, such a situation can always be ensured. We denote such schedules as *clean with respect to job j*.

In the following, we examine the effects of slightly changing the processing time (and weight) of a job j_1. More precisely, we replace p_{j_1} by $p'_{j_1} = p_{j_1} - \varepsilon$ and all other processing times remain the same (i.e. $p'_j = p_j$ for all jobs $j \neq j_1$).

The next lemma shows that the assignment A remains an LPT-assignment if ε is

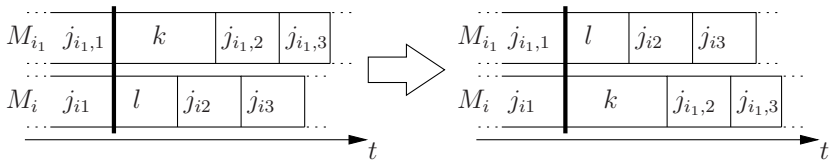

Figure 4.6: Illustration of swapping schedules beginning with job k and l.

chosen sufficiently small. Moreover, the new objective value $f'(A)$ of assignment A in the changed instance is only a slight modification of the value $f(A)$.

Lemma 4.13 *Let j_1 be a job with $p_1 \geq \ldots \geq p_{j_1} > p_{j_1+1} \geq \ldots \geq p_n$ and let A be an LPT-assignment which is clean w.r.t. job j_1. For a sufficiently small $\varepsilon > 0$, the assignment A remains an LPT-assignment for the changed instance.*

The objective value $f'(A)$ in the changed instance is given by

$$f'(A) = f(A) - r\varepsilon \left(L_{A(j_1)} + p_{j_1} - \varepsilon\right).$$

PROOF. Let in the following $i_1 := A(j_1)$ be the machine job j_1 is assigned to. We define the set M of jobs that follow job j_1 on machine i_1 as

$$M := \{j \in M_{i_1} : j \geq j_1 + 1\}.$$

Let $\varepsilon > 0$ be so that the following conditions hold:

(P1) $p_{j_1} - \varepsilon > \max\{p_{j_1+1}, 0\}$,

(P2) for all pairs (j, k) of jobs, with $j \notin M$, $k \in M$ and $S_j < S_k$, we have $S_j \leq S_k - \varepsilon$,

(P3) for all jobs j, with $S_j + p_j = L_{A(j)}$ and $S_j < L_{i_1}$, we have $S_j \leq L_{i_1} - \varepsilon$.

Because the assignment A is clean w.r.t. job j_1, an $\varepsilon > 0$ with the above given properties exists and the assignment A remains an LPT-assignment for the changed instance.

We now turn to the objective value. We consider $f'(A)$ given by (4.3) which is calculated as

$$\begin{aligned} f'(A) &= \frac{r}{2}\sum_{i=1}^{m}\left(L_i'\right)^2 + \frac{r}{2}\sum_{j=1}^{n}\left(p_j'\right)^2 \\ &= f(A) + \frac{r}{2}\left(2\varepsilon^2 - 2\varepsilon L_{A(j_1)} - 2\varepsilon p_{j_1}\right). \qquad \square \end{aligned}$$

The previous lemma shows that we can decrease a processing time by some ε without loosing the LPT-property. As in Section 4.4, we are interested in an upper bound for the ratio $f(A)/f(A^*)$. Again, we scale the processing times and weights so that

$$\sum_{j=1}^{n} p_j = m,$$

without changing the value of $f(A)/f(A^*)$. Since an LPT-assignment is also move-optimal, we may use the results of Section 4.4, i.e. an upper bound on $f(A)/f(A^*)$ is given by (4.24). In the following, we look for an upper bound on α to get a best possible value for the right hand side of (4.24). In order to do so, the next lemma shows that certain instances can be left out of consideration.

Lemma 4.14 *Let I be an instance, A be an LPT-assignment and A^* be an optimal assignment with $f(A)/f(A^*) > 1$. If $L_1^{A^*} \geq L_1^A$, then there exists an instance I' and a corresponding LPT-assignment B with*

$$\frac{f'(B)}{f'(B^*)} > \frac{f(A)}{f(A^*)},$$

where f' denotes the objective function and B^* an optimal assignment for the instance I'.

PROOF. Let $j_1 \in M_1^{A^*}$ be a job processed by machine 1. We assume without loss of generality that A is clean w.r.t. job j_1. Furthermore, by exchanging indices with job j_1, we assume without loss of generality that for the processing times hold $p_1 \geq \ldots \geq p_{j_1} > p_{j_1+1} \geq \ldots \geq p_n$ holds. Due to Lemma 4.13, there exists an $\varepsilon > 0$ such that in the changed instance with $p_j' := p_j$, for all j$\neq j_1$

and $p'_{j_1} := p_{j_1} - \varepsilon$, the assignment A remains an LPT-assignment and for the objective value of A in the changed instance

$$f'(A) = f(A) - r\varepsilon \left(L_{A(j_1)} + p_j - \varepsilon\right)$$

holds. In the changed instance, for assignment A^*, machine 1 has a workload $L'_1 = L_1^{A^*} - \varepsilon$. Thus, the objective value $f'(A^*)$ of assignment A^* in the changed instance is calculated as

$$f'(A^*) = f(A^*) - r\varepsilon \left(L_1^{A^*} + p_j - \varepsilon\right).$$

Furthermore, let B^* be the optimal assignment for the changed instance. This implies $f'(A^*) \geq f'(B^*)$. Therefore, we obtain for the ratio of the objective values of assignments A and A^*:

$$\frac{f'(A)}{f'(B^*)} \geq \frac{f'(A)}{f'(A^*)} = \frac{f(A) - r\varepsilon \left(L_{A(j_1)} + p_j - \varepsilon\right)}{f(A^*) - r\varepsilon \left(L_1^{A^*} + p_j - \varepsilon\right)} > \frac{f(A)}{f(A^*)},$$

where the last inequality is due to $L_1^{A^*} + p_j - \varepsilon \geq L_{A(j_1)} + p_j - \varepsilon > 0$ and $f(A) > f(A^*)$. $\square$

The next lemma is crucial for Theorem 4.4.

Lemma 4.15 *Let A be an LPT-assignment and A^* an optimal assignment with $L_1^A > L_1^{A^*}$. Then*

$$\Delta_i \leq \frac{1}{2} L_m, \text{ for all machines } i = 1, \ldots, m.$$

PROOF. First we show that machine 1 contains at least 2 jobs. For this, assume that $n_1 = 1$ and let j be the only job assigned to machine 1. Then $L_1 = p_j \geq L_i$, for all $i = 1, \ldots, m$. For the machine i with maximal workload according to the optimal assignment A^* then holds $L_i^{A^*} \geq p_j = L_1^A$, which is a contradiction. Therefore, machine 1 contains at least 2 jobs.

Since an LPT-assignment A is also move-optimal, we know by Lemma 4.10 that if many jobs are assigned to a single machine i, the value Δ_i becomes small:

$$\Delta_i \leq \frac{1}{n_i - 1} L_m.$$

Since $\Delta_1 \geq \ldots \geq \Delta_m$, we only have to consider the case $n_1 = 2$. Let j_1 and j_2 be the two jobs assigned to machine 1 according to assignment A. We assume without loss of generality that $p_{j_1} \geq p_{j_2}$. Moreover, let p and x be values, so that $p = p_{j_1}$ and $xp = p_{j_2}$. Hence, the workload of machine 1 equals $L_1 = (1 + x)p$, where x takes values out of the interval $x \in (0, 1]$.

Since A is an LPT-assignment, all other machines $i \neq 1$ must have a workload of at least $L_i \geq p$. Therefore, $\Delta_i \leq xp \leq xL_m$, for all machines i, so for $x \in (0, \frac{1}{2}]$ the statement follows.

Assume now that $x > \frac{1}{2}$. Let I_1 be the set of machines containing a job with processing time larger than p. Such a job is called a *big job*. Observe, that in the optimal assignment A^* no two big jobs may be assigned to the same machine. Since otherwise, we get a makespan of at least $2p \geq L_1^A$, contradicting $L_1^A > L_1^{A^*}$.

Let $I_2 = \{1, \ldots, m\} \setminus I_1$ be the set of machines containing no big job. If $I_2 = \emptyset$, this again leads to a contradiction, since in this case we have at least m big jobs and the job j_2 with $p_{j_2} = xp$, leading to an optimal assignment A^* with makespan at least $(1 + x)p = L_1^A$, contradicting $L_1^A > L_1^{A^*}$. Thus, $I_2 \neq \emptyset$ and $x < 1$.

Since A is an LPT-assignment, all machines $i \in I_2$ contain at least 2 jobs with processing time at least xp. We call a job j with $p > p_j \geq xp$ a *blocking job*. Observe, that there are at least $2|I_2| + 1$ blocking jobs. In the optimal assignment A^* no blocking job can be combined with one or more of the $|I_1|$ big jobs. Because if we do so, we get a makespan of at least L_1^A, contradicting $L_1^A > L_1^{A^*}$.

Hence, in order to obtain an optimal assignment with $L_1^A > L_1^{A^*}$, we have to assign, at least once, 3 blocking jobs to a single machine. Doing so, yields a makespan of at least $3xp$ which is greater than or equal to L_1^A, for $x > \frac{1}{2}$. Hence, this again contradicts $L_1^A > L_1^{A^*}$. Therefore, for $x > \frac{1}{2}$ there exists no optimal assignment with $L_1^A > L_1^{A^*}$. $\square$

In order to give an upper bound on the performance guarantee $f(A)/f(A^*)$, due to Lemma 4.14 we only have to consider instances where for the makespan of A

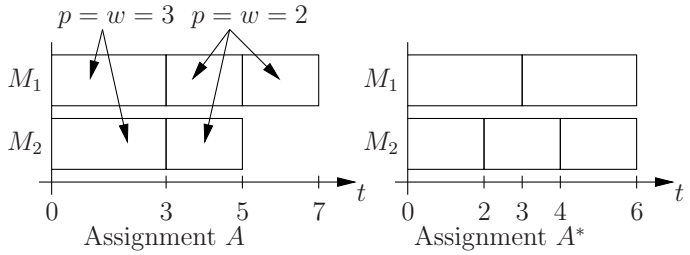

Figure 4.7: Lower bound on performance guarantee of LPT-assignments.

and A^* holds $L_1^A > L_1^{A^*}$. Lemma 4.15 then guarantees, that

$$\Delta_i \leq \frac{1}{2} L_m, \text{ for all machines } i = 1, \ldots, m.$$

Thus in (4.24) we may use $\alpha = \frac{1}{2}$, yielding

$$\frac{f(A)}{f(A^*)} \leq 2 - \frac{2 - L_m}{1 + \frac{3}{2} L_m - \frac{3}{2} L_m^2}.$$

This becomes maximal for $L_m = 2 - \frac{2}{3}\sqrt{3} \approx 0.845$, and thus

$$\frac{f(A)}{f(A^*)} \leq \frac{16}{11} - \frac{8}{33}\sqrt{3} < 1.035.$$

This completes the proof of Theorem 4.4.

The following lower bound example on the performance guarantee of LPT-assignments was given by Chandra and Wong [22] for the problem of minimizing $\widetilde{f}(A)$. This example consists of 2 machines and 5 jobs. There are 2 jobs with processing times and weights equal to 3 and 3 jobs with processing times and weights equal to 2. The LPT-assignment shown in Figure 4.7 has an objective value $f(A) = 52$. The optimal assignment A^* again has balanced machines yielding an objective value $f(A^*) = 51$. This together yields a ratio of

$$\frac{f(A)}{f(A^*)} = \frac{52}{51} > 1.019.$$

If we change the processing times and weights of the three smaller jobs to $\frac{3}{2}\sqrt{2} \approx 2.121$, we receive a better lower bound with a ratio

$$\frac{f(A)}{f(A^*)} = \frac{2}{3} + \frac{1}{4}\sqrt{2} > 1.020.$$

This instance has an optimal assignment, where the machines are not balanced.

If we change the objective $f(A)$ to $\widetilde{f}(A)$, we may use the same proof to get a bound on the objective value of an LPT-assignment A:

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} \leq \frac{25}{24}.$$

As before, we only have to consider instances with $\Delta_i \leq \frac{1}{2}L_m$. By scaling the processing times, using (4.20) and the definition of Δ_i, we obtain

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} \leq \frac{\sum_{i=1}^{m} \Delta_i^2 + m\left(2L_m - L_m^2\right)}{m}.$$

By exploiting the bound on Δ_i and incorporating (4.18) we get

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} \leq \frac{5}{2}L_m - \frac{3}{2}L_m^2.$$

This becomes maximal for $L_m = \frac{5}{6}$, yielding

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} \leq \frac{25}{24}.$$

The previously presented lower bound example gives, for the objective function $\widetilde{f}(A)$, a value

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} = \frac{37}{36}.$$

And again, by changing the processing times of the three smaller jobs (to be $\frac{1}{9} + \frac{5}{9}\sqrt{13} \approx 2.114$), we receive a better lower bound:

$$\frac{\widetilde{f}(A)}{\widetilde{f}(A^*)} = \frac{19}{36} + \frac{5\sqrt{13}}{36} > 1.028.$$

4.6 Concluding Remarks

In this chapter (Section 4.2 and Section 4.4) we discussed the quality of local optima w.r.t. the move-neighborhood. Such a local optimum can be found, e.g. by iterative improvement. However, we do not say anything about how long it takes to get there. It would be interesting to know, whether an iterative improvement process using the neighborhood $\mathcal{N}_{1-move}$ terminates in a polynomially bounded number of steps. But nothing is known in this direction, except for the results of Brucker et al. [15] and Vredeveld [61]. Here, Brucker et al. [15] consider the problem of minimizing makespan on parallel identical machines (i.e. minimizing the $\mathcal{L}_\infty$-norm of the vector of workloads) and prove that the iterative improvement process, by moving a job from one machine to another, terminates in $\mathcal{O}\left(n^2\right)$ steps. There are indeed examples which need n^2 steps as shown by Hurkens and Vredeveld [43]. Moreover, they were able to improve this bound to $\mathcal{O}\left(nm\right)$ iterations by imposing a selection rule for jobs. Vredeveld [61] generalizes this to the problem $Q|\,|C_{\max}$ of unrelated parallel machines, resulting in $\mathcal{O}\left(n^2\right)$ iterations with a machine selection rule and $\mathcal{O}\left(nm\right)$ iterations with a job selection rule.

Another interesting aspect is the relation between move-optimal assignments and the concept of "price of anarchy" proposed by Papadimitriou [52]. The term "price of anarchy" denotes the ratio between the worst-case Nash equilibrium and the optimum payoff of a system. The scheduling problem we consider is similar to the following system. There are n players $1, \ldots, n$ who want to be served by one of m identical and parallel servers. Each player has a given processing time and a weight denoting its importance and has to choose a server to be processed by. After the players have chosen their servers, the system arranges the order in which the players are processed by their machines, i.e. a player can not influence this and, hence, we only have to consider assignments of players to servers.

We first consider the situation that the cost for each player is given by the weighted completion time he would receive. This setting turns out to be equivalent to the scheduling problem we consider. A Nash equilibrium (stable situation) is obtained if no player would receive a lower completion time by switching to another server. Problems arise, when there are players for which Smith's rule does not yield an explicit order, i.e. if two players have the same weight to processing time ratio. In this case, it may happen that there exists no Nash

equilibrium. To circumvent this, we impose the restriction, that the system on beforehand gives every player a priority according to Smith's rule and breaks ties arbitrarily. Then, the priority order of the players is kept fixed.

If we enumerate the players according to their priorities, $S_j \leq S_{j+1}$ holds in a Nash equilibrium, for all players $j = 1, \ldots, n-1$. Moreover, $S_j \leq \min_i L_i$, for all players j. This coincides with an LRF-assignment for the scheduling problem and hence, due to Kawaguchi and Kyan [46], we obtain a worst-case ratio of $\frac{1}{2}(\sqrt{2}+1)$. Observe that a social optimum (i.e. an optimal solution for the scheduling problem) is not always optimal due to the egoistic behavior of the players. Consider the following example consisting of 2 servers and 5 players, with the data given by the following table. For this, let $0 \leq \varepsilon < \frac{1}{3}$.

player j	1	2	3	4	5
p_j	3	3	2	2	2
w_j	$3+\varepsilon$	$3+\varepsilon$	2	2	2

Observe that the players are already ordered according to their priority. An LRF-assignment A assigns the players 1 and 3 to one server and the remaining players $2, 4, 5$ to the second server yielding a *social value* (objective value) $f(A) = 52+6\varepsilon$. In an optimal assignment A^*, the players 1 and 2 are processed by one server and the remaining players $3, 4, 5$ by the second server, yielding a social value $f(A^*) = 51 + 9\varepsilon$. Hence, for $\varepsilon < \frac{1}{3}$ we have $f(A^*) < f(A)$. Moreover, the optimal assignment is not a Nash equilibrium.

In order to derive a problem where an optimal assignment always is a Nash equilibrium, one has to change the cost function for the players (but keep the social value to be the sum of weighted completion times). In the former definition, the cost for a player j is calculated as the weighted completion time $C_j^A = w_j L_{A(j),j}$ obtained by the given assignment A. Now let the cost of player j be given by $\Delta_j := w_j L_{A(j),j} + p_j W_{A(j),j}$. This looks a bit artificial, but the new cost function penalizes unsocial behavior of a player. If player j is processed by a server i, he receives an additional penalty caused by the weights of the low priority players following j on server i. From the proof of Theorem 4.1 we see, that we have a Nash equilibrium if (4.9) is fulfilled. This means, that Nash equilibria and move-optimal assignments coincide. Hence, we have a worst-case ratio between Nash equilibria and the optimum social cost (given by the sum of weighted completion times) of at most $\frac{3}{2} - \frac{1}{2m}$, as stated by Theorem 4.1. Moreover, because every optimal assignment also is move-optimal, any optimal assignment also is a Nash equilibrium.

Thus, if the costs for the players are adjusted to be more social, the optimal assignment also is a stable situation, where no player wants to change its server. It is unclear whether the bound in Theorem 4.1 is tight. Hence, it is unsure if the purely egoistic behavior is preferable to the more social behavior of a player. Nevertheless, the worst-case ratios $\frac{1}{2}(\sqrt{2}+1)$ and $\frac{3}{2}-\frac{1}{2m}$ are close for small values of m.

Chapter 5

VLSN with Performance Guarantees for Scheduling on Parallel Machines to Minimize the Makespan

Amongst other things, in Chapter 4 we studied local optima for the problem of minimizing the sum of squared machine workloads. We now turn to the related problem $P||C_{\max}$ of minimizing the makespan, as introduced in Section 1.3.2. We present a very large-scale neighborhood for this problem similar to that in Section 3.2 and give certain performance guarantees for local optima in this neighborhood.

To obtain reasonable solutions, one may use LPT-assignments as introduced in the previous chapter. Due to Graham [38], a corresponding assignment is no worse than

$$C_{\max}^{LPT} \leq \left(\frac{4}{3} - \frac{1}{3m}\right) C_{\max}^{*},$$

where $C_{\max}^{LPT}$ and $C_{\max}^{*}$ denote the makespan obtained by the LPT-algorithm

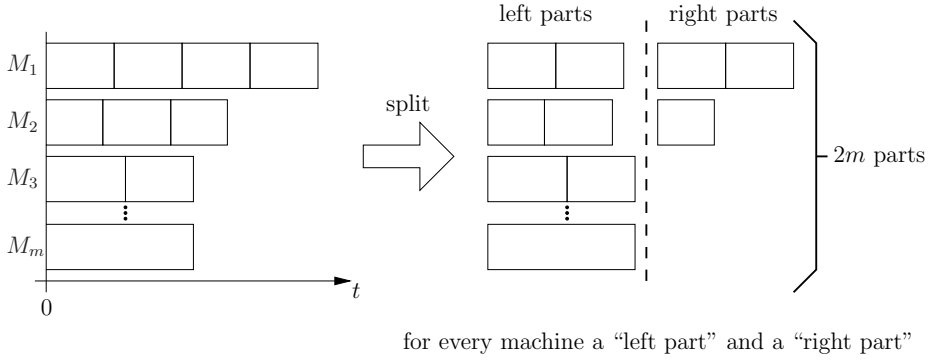

Figure 5.1: Illustration of split-operator.

and the optimal makespan, respectively. This performance guarantee is proven to be tight.

Finn and Horowitz [32] investigate move-optimal assignments obtained by iteratively reassigning jobs to machines as long as either the makespan is decreased or the number of critical machines is reduced, i.e. the number of machines attaining the makespan. They prove for move-optimal assignments a performance guarantee of

$$\frac{C_{\max}^{move}}{C_{\max}^{*}} \leq 2 - \frac{2}{m+1},$$

where $C_{\max}^{move}$ denotes the makespan of a move-optimal assignment. Moreover, this bound is tight as shown by Schuurman and Vredeveld [57]. Schuurman and Vredeveld [57] investigate also the quality of local optima in the swap, the multi-exchange and the push-neighborhood.

In this chapter, we introduce a very large-scale neighborhood of exponential size (in the number of machines) that is based on a matching in a complete graph. This neighborhood is similar to the split-neighborhood of Section 3.2, i.e. the idea is to partition the jobs assigned to the same machine into two sets, a so-called *left part* and *right part*, respectively. This partitioning is done for every machine according to some chosen rule *op* that is called *split-operator*. The idea is illustrated in Figure 5.1.

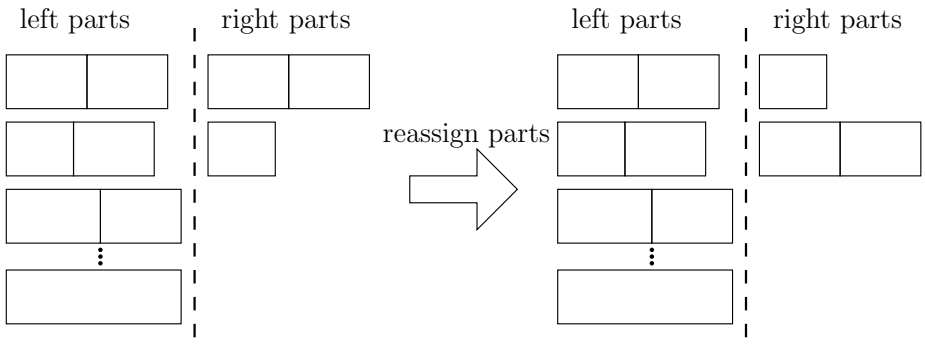

Figure 5.2: Reassigning parts to obtain neighboring assignment.

Afterwards, a new assignment is obtained by putting to every machine i, for $i = 1, \ldots, m$, exactly two parts. The neighborhood $\mathcal{N}^{op}_{split}$ consists of all such possible combinations, and hence, contains exponentially (in the number of machines) many assignments. Figure 5.2 shows, how a split-neighbor is obtained by reassigning the parts. The best assignment of $\mathcal{N}^{op}_{split}$ can be calculated in time $\mathcal{O}(m \log m)$ by determining a perfect matching in a complete graph constituting the best objective value among all assignments in the neighborhood. Here, an edge of the perfect matching corresponds to assigning the jobs of the corresponding parts to the same machine.

First, we only consider move-optimal split-operators. Such an operator delivers for a machine two sets of jobs, where the sum of processing times of the jobs of a set is balanced to the other set. These move-optimal split-operators will be formalized in the next section. Since the partitioning of the jobs of a machine into two sets can be understood as a two machine problem, in a second step we use the LPT-algorithm as a split-operator. Here, we take the jobs of a given machine, sort them by non-increasing processing times and iteratively put them to one of the parts corresponding to the machine. The LPT-algorithm is also a move-optimal split-operator.

A locally optimal solution w.r.t. the neighborhood $\mathcal{N}^{op}_{split}$ is an assignment, for which it is not possible to obtain a better solution by rearranging the left parts and right parts. Such an assignment is called *split-optimal*. For split-optimal assignments we prove the following theorem.

Theorem 5.1 *Let the partition of the sets M_i be done by a move-optimal split-operator op. Then the performance guarantee for a split-optimal solution is given by*

$$\frac{C_{\max}^{split}}{C_{\max}^{*}} \leq \frac{2m}{m+1} = 2 - \frac{2}{m+1},$$

where $C_{\max}^{*}$ denotes the optimal makespan and $C_{\max}^{split}$ the makespan of the split-optimal solution. Moreover, this bound is tight.

From the previous theorem we can conclude that split-optimal schedules have the same worst-case performance guarantee as move-optimal schedules. But, the worst-case examples have a different structure. In a second step, we try to combine these two neighborhoods and get the following theorem.

Theorem 5.2 *Let the partition of the sets M_i be done by a move-optimal split-operator op. Then the performance guarantee for a solution that is both split-optimal and move-optimal is given by*

$$\frac{C_{\max}^{s+m}}{C_{\max}^{*}} \leq \frac{2m+2}{m+3} = 2 - \frac{4}{m+3},$$

where $C_{\max}^{*}$ denotes the optimal makespan and $C_{\max}^{s+m}$ the makespan of the split-optimal and move-optimal solution. Moreover, this bound is tight for $m = 2$ and any odd m.

The performance guarantee for jointly split-optimal and move-optimal schedules is slightly better than for move-optimal or split-optimal schedules alone. But the asymptotic behavior of the guarantees is still the same, i.e. for $m \to \infty$ we have a performance guarantee of 2 for any local optimum of any neighborhood considered so far.

In a last step, we modify the move-neighborhood. A given schedule is called *lexicographic-move-optimal*, if no move of a single job leads to a schedule where the ordered vector of machine workloads is lexicographically smaller than that of the considered schedule. A schedule that is lexicographic-move-optimal has the same worst-case performance guarantee as move-optimal schedules, but if such a schedule is also split-optimal we can prove the following guarantee.

Theorem 5.3 *Let the partition of the sets M_i be done by using the LPT-algorithm as split-operator. If $C_{\max}^{s+lm}$ denotes the makespan of a schedule that is lexicographic-move-optimal* and *split-optimal, then*

$$\frac{C_{\max}^{s+lm}}{C_{\max}^{*}} \leq \frac{3}{2},$$

where $C_{\max}^{*}$ denotes the optimal makespan. Moreover, instances exist that have a ratio

$$\frac{C_{\max}^{s+lm}}{C_{\max}^{*}} = \frac{3}{2} - \frac{1}{2m-2}.$$

Therefore, by combining the neighborhoods in a clever way, we can improve the performance guarantee considerably and even get close to the guarantee for LPT-assignments, which is 4/3 for $m \to \infty$.

In Section 5.1 we introduce the split-neighborhood and show, how to obtain a best neighbor. Moreover, additional definitions are given. Afterwards, in Section 5.2 we prove Theorem 5.1. In Section 5.3 we provide the proof for Theorem 5.2. In Section 5.4 we prove Theorem 5.3. Finally, in Section 5.5 we give some concluding remarks.

5.1 Description and Notation

Throughout this chapter, we use the notation presented in Section 1.3.2. For a given assignment A of jobs to machines, the makespan is equal to the machine with maximal workload:

$$C_{\max}^{A} = \max_{i=1}^{m} L_i^A.$$

We call such a machine with maximal workload a *critical machine*.

For any assignment A, we can give the following lower bound on the makespan:

$$C_{\max}^{A} \geq \frac{1}{m} \sum_{j=1}^{n} p_j. \tag{5.1}$$

In proving the performance guarantees, we make heavy use of this lower bound on the optimal makespan.

Analogous to Section 3.1.1 we define the operator *move* that reassigns a job j_1 from its designated machine $A(j_1)$ to a machine i_1. The neighborhood $\mathcal{N}_{1-move}$ of an assignment A then contains all assignments that can be obtained by moving one job to some machine, i.e.

$$\mathcal{N}_{1-move}(A) = \{\, move(j,i)(A) \,:\, j \in J \text{ and } i = 1,\ldots,m \,\}.$$

We call an assignment A *move-optimal* if $C^A_{\max} \leq C^{A'}_{\max}$ for all $A' \in \mathcal{N}_{1-move}(A)$, and, in case of $C^A_{\max} = C^{A'}_{\max}$, the number of critical machines in A is at most the number of critical machines in A'. Finn and Horowitz [32] gave the following upper bound on the performance guarantee of move-optimal assignments.

Theorem 5.4 (Finn and Horowitz [32]) *Let A be a move-optimal assignment and let $C^*_{\max}$ denote the optimal makespan. If k is a critical machine in A with n_k jobs, then*

$$\frac{C^A_{\max}}{C^*_{\max}} \leq \frac{n_k m}{(n_k-1)m+1}.$$

Moreover, if $n_k = 1$, then $C^A_{\max} = C^*_{\max}$.

The bound in Theorem 5.4 attains its maximum for $n_k = 2$, yielding a performance guarantee of $2 - 2/(m+1)$. This bound has been proven to be tight by Schuurman and Vredeveld [57], see Figure 5.3 for an example with $n_k = 2$ attaining this bound. This example can be extended to arbitrary values for $n_k > 1$ (proving the tightness for arbitrary n_k) by putting $n_k - 2$ additional jobs with processing times equal to 1 to every machine in the assignments A and A^*.

The *split-neighborhood* is of exponential size in the number of machines. It is based on an operator that partitions the set of jobs assigned to a machine i into two disjoint sets. Let $J = \{1,\ldots,n\}$ denote the set of all jobs. A *split-operator* $op : 2^J \to 2^J \times 2^J$ applied to a set M_i of jobs produces $op(M_i) = (M_{i1}, M_{i2})$ and has the following properties:

- the sets M_{i1} and M_{i2} are a partition of M_i,
- for the workloads of the sets holds

$$L_{i1} = \sum_{j \in M_{i1}} p_j \geq \sum_{j \in M_{i2}} p_j = L_{i2}.$$

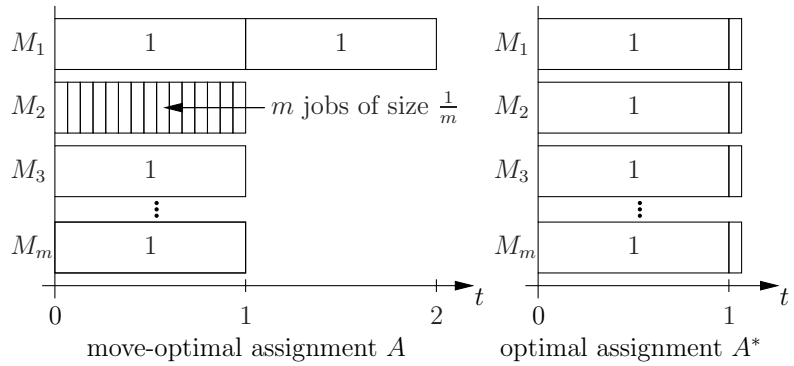

Figure 5.3: Worst-case example for move-optimal assignments.

We refer to the sets M_{i1} and M_{i2} of a machine i (or a set M_i) as the *left part* and *right part*, respectively.

We call a split-operator *op* a *move-optimal split-operator*, if

$$L_{i2} + p_j \geq L_{i1} \text{ for all } j \in M_{i1}. \tag{5.2}$$

Observe that the LPT-algorithm may be used as a move-optimal split-operator.

If we use the split-operator on all sets M_i given by an assignment A, we obtain $2m$ parts M_{i_1} and M_{i_2}, for $i = 1, \ldots, m$. Abusing notation, we denote the set of these $2m$ parts by

$$op(A) = \big\{\, M_{i1}, M_{i2} \,:\, op(M_i) = (M_{i1}, M_{i2}) \text{ for } i = 1, \ldots, m \,\big\}.$$

We call an assignment A' a *neighbor* of A, if A' is achieved by assigning to each machine the jobs of exactly two of the $2m$ parts from $op(A)$. Note, that this definition is different from Section 3.2, where we only allow combinations of left parts with right parts. The neighborhood $\mathcal{N}^{op}_{split}$ of an assignment A using the split-operator *op* is defined as

$$\mathcal{N}^{op}_{split}(A) := \{\, A' \,:\, A' \text{ is neighbor of } A \,\}.$$

We are interested in the *best neighbor* A_1 of the neighborhood $\mathcal{N}^{op}_{split}(A)$, i.e. we are seeking an assignment $A_1 \in \mathcal{N}^{op}_{split}(A)$ with

$$C^{A_1}_{\max} \leq C^{A'}_{\max} \text{ for all } A' \in \mathcal{N}^{op}_{split}(A),$$

and among all assignments $A' \in \mathcal{N}^{op}_{split}(A)$ with $C^{A_1}_{\max} = C^{A'}_{\max}$, A_1 is the assignment with minimal number of machines attaining the makespan.

In order to achieve this, let $P_1, \ldots, P_{2m}$ be the parts obtained by the split-operator, and we renumber the parts without loss of generality, such that $L_{P_1} \geq \ldots \geq L_{P_{2m}}$. We define a so-called *improvement graph* G^A, which heavily depends on the assignment A and the used split-operator *op*. Furthermore, the graph G^A is established, so that any perfect matching in the graph G^A corresponds to a neighbor of A and vice versa.

Let $G^A = (V, E, c(A))$, with $V = \{1, \ldots, 2m\}$, be an undirected, weighted and complete graph with $2m$ vertices, where a vertex i corresponds to part P_i, for $i = 1, \ldots, 2m$. An edge $e = \{i_1, i_2\} \in E$ connecting the vertices i_1 and i_2 gets a weight

$$c(A)_{i_1,i_2} := L_{P_{i_1}} + L_{P_{i_2}}.$$

The cost coincides with the workload obtained by assigning the jobs of the parts P_{i_1} and P_{i_2} to a single machine.

A perfect matching $\mathcal{M}$ in this graph consists of m edges and connects each vertex to exactly one other vertex. Thus, $\mathcal{M}$ also *induces* an assignment in which the jobs of two parts are assigned to the same machine, if the corresponding vertices are connected by an edge contained in the matching. We denote by $w(\mathcal{M})$ the *bottleneck cost* of a perfect matching $\mathcal{M}$, defined as

$$w(\mathcal{M}) := \max_{\{i_1,i_2\} \in \mathcal{M}} c(A)_{i_1,i_2}.$$

By doing so, the bottleneck cost $w(\mathcal{M})$ equals the makespan of the induced assignment and the number of edges with weight $w(\mathcal{M})$ equals the number of machines attaining the makespan in the designated assignment.

We call a perfect matching $\mathcal{M}$ an *optimal perfect matching*, if it has minimum bottleneck cost among all perfect matchings of G^A, and among all perfect matchings with the same bottleneck cost, $\mathcal{M}$ has the minimum number of edges attaining the bottleneck cost.

In order to obtain a best neighbor of $\mathcal{N}^{op}_{split}$ we have to determine an optimal perfect matching. The following lemma shows how to obtain such an optimal perfect matching.

Lemma 5.5 *An optimal perfect matching is given by matching vertex i with $2m+1-i$ for $i = 1, \ldots, m$.*

PROOF. Let $\mathcal{M}$ be the perfect matching consisting of the edges $\{i, 2m+1-i\}$ for $i = 1, \ldots, m$. Let $\mathcal{M}^*$ be an optimal perfect matching. Let k be the minimum k with $1 \leq k \leq m$, such that $\mathcal{M}^*$ does not contain the edge $\{k, 2m+1-k\}$. If there exists no such k, $\mathcal{M}$ and $\mathcal{M}^*$ are equal. Otherwise, $\mathcal{M}^*$ coincides with $\mathcal{M}$ on the first $k-1$ edges.

We claim that there exists a matching M' that is not worse than $\mathcal{M}^*$ and contains the edges $\{i, 2m+1-i\}$ for $i = 1, \ldots, k$, i.e. coincides with $\mathcal{M}$ on the first k edges. Then, by induction on k the statement of the lemma follows.

Since $\mathcal{M}^*$ does not contain the edge $\{k, 2m+1-k\}$ and coincides with $\mathcal{M}$ on the first $k-1$ edges, there exist vertices s, r with $s \neq r$ and $k < s, r < 2m+1-k$, such that $\mathcal{M}^*$ contains the edges $\{k, s\}$ and $\{r, 2m+1-k\}$. Since

$$w_k \geq w_s \geq w_{2m+1-k} \text{ and}$$
$$w_k \geq w_r \geq w_{2m+1-k},$$

we get

$$w_k + w_s \geq w_k + w_{2m+1-k} = c(A)_{k,2m+1-k} \text{ and}$$
$$w_k + w_s \geq w_r + w_s = c(A)_{r,s}.$$

Therefore, the bottleneck cost is not increased, if the two edges $\{k, s\}$ and $\{r, 2m+1-k\}$ are replaced by $\{k, 2m+1-k\}$ and $\{r, s\}$. We now claim, that by advancing from $\mathcal{M}^*$ to $\mathcal{M}'$, also the number of the edges with weight $w(\mathcal{M}^*) = w(\mathcal{M}')$ does not increase.

If the edge $\{k, s\}$ has a weight $c_{k,s} < w(\mathcal{M}^*)$, then $\{k, 2m+1-k\}$ and $\{r, s\}$ are also edges with weight less than $w(\mathcal{M}^*)$. Thus, in this case there is no increase in the number of edges with weight $w(\mathcal{M}^*) = w(\mathcal{M}')$. If the edge $\{k, s\}$ has a weight $c_{k,s} = w(\mathcal{M}^*)$, then the number of edges with weight $w(\mathcal{M}^*) = w(\mathcal{M}')$ may increase, if both new edges of $\mathcal{M}'$ have weight $w(\mathcal{M}^*)$. But in that case $w_k = w_r$ and $w_s = w_{2m+1-k}$, as $c(A)_{k,s} = c(A)_{k,2m+1-k} = c(A)_{r,s}$. Therefore, also the edge $\{r, 2m+1-k\}$ of $\mathcal{M}^*$ has weight $w(\mathcal{M}^*)$. Hence, the number of edges with weight $w(\mathcal{M}^*) = w(\mathcal{M}')$ does not increase in this case. $\square$

By using the previous lemma, the best neighbor of A can be obtained by sorting the workloads of the parts in non-increasing order so that $L_{P_1} \geq \ldots \geq L_{P_{2m}}$. This can be done in $\mathcal{O}(m \log m)$. Afterwards, we assign the jobs of the parts P_i and P_{2m+1-i} to machine i, for $i = 1, \ldots, m$.

We now turn to local optima in the neighborhood $\mathcal{N}^{op}_{split}$ of an assignment A. We call an assignment A *split-optimal* if it is locally optimal with respect to the split-neighborhood, i.e. if for all $A' \in \mathcal{N}^{op}_{split}(A)$ it is $C^{A'}_{\max} \geq C^{A}_{\max}$ and among all assignments $A' \in \mathcal{N}^{op}_{split}(A)$, with $C^{A}_{\max} = C^{A'}_{\max}$, the number of machines attaining the makespan of A' is at least the number of machines attaining the makespan of A.

For a split-optimal assignment A, despite the fact that $L_{i1} \geq L_{i2}$ for all machines $i = 1, \ldots, m$, a critical machine k has the following properties for any machine i:

$$\begin{aligned} &\text{if } L_{i1} \geq L_{k1} \text{ then } L_{i2} \leq L_{k2} \text{ holds,} \\ &\text{if } L_{i1} < L_{k1} \text{ then } L_{i2} \geq L_{k2} \text{ holds.} \end{aligned} \tag{5.3}$$

The first statement follows from the fact that k is a critical machine, and the second statement holds since A is split-optimal.

As we will see in Section 5.2, a split-optimal assignment needs not to be move-optimal. In Section 5.3, we also consider assignments that are both, move-optimal and split-optimal and prove Theorem 5.2. But, as already stated with Theorem 5.3, we get better by considering lexicographic-move-optimal assignments. In the following, we define lexicographic-move-optimal assignments.

For given assignments A and $A' \in \mathcal{N}_{1-move}(A)$, we reorder the machines in A and A' so that

$$\begin{aligned} &L^{A}_{1} \geq \ldots \geq L^{A}_{m} \text{ and} \\ &L^{A'}_{1} \geq \ldots \geq L^{A'}_{m}. \end{aligned}$$

The assignment A' is called *lexicographically better* than A, if there exists a machine k such that

$$\begin{aligned} &L^{A'}_{i} = L^{A}_{i}, \text{ for } i = 1, \ldots, k-1, \\ &L^{A'}_{k} < L^{A}_{k}. \end{aligned} \tag{5.4}$$

We say that A is *lexicographic-move-optimal*, or *lexmove-optimal*, if there exists no move-neighbor $A' \in \mathcal{N}_{1-move}(A)$ that is lexicographically better than A. The following lemma gives a more intuitive characterization of lexmove-optimal assignments A.

Lemma 5.6 *Let A be an assignment and let machine l be a machine with minimal workload (minimal machine). The assignment A is lexmove-optimal if and only if*

$$L_l + p_j \geq L_i, \text{ for all } j \in M_i \text{ and } i = 1, \ldots, m. \tag{5.5}$$

PROOF. We reorder the machines without loss of generality, so that for assignment A holds $L_1 \geq L_2 \geq \ldots \geq L_m$, and we assume $l = m$. Let A be an assignment for which (5.5) is not fulfilled. Thus, there exists a machine k and a job $j \in M_k$, so that $L_m + p_j < L_k$. Consider the assignment A' that arises from A by moving job j from its machine k to machine m. This decreases the workload of machine k and hence, A' is lexicographically better than A. Thus, if A does not satisfy (5.5), then A is not lexmove-optimal.

Consider an assignment A that is not lexmove-optimal. Hence, there exists an assignment $A' \in \mathcal{N}_{1-move}$, such that A' is lexicographically better than A and $A' = move(j, i)(A)$. Let $k \neq i$ be the machine that job j is assigned to in A. In A' only the workloads of the machine i and k differ from the workloads for A: $L'_i = L_i + p_j$, $L'_k = L_k - p_j$ and $L'_l = L_l$ for $l \neq i, k$. Since A' is lexicographically better than A, this yields $L_i + p_j < L_k$. Since $L_m \leq L_i$, also $L_m + p_j < L_k$ holds, and therefore (5.5) is not fulfilled for assignment A. □

Note, that the move-optimal assignment A in Figure 5.3 is also lexmove-optimal. Therefore, the move-optimal and the lexmove-optimal assignments have the same performance guarantee. However, in the next sections we show, that the performance guarantee of a lexmove-optimal and split-optimal assignment is better than that of a move-optimal and split-optimal assignment.

5.2 Split-Optimal Assignments

The performance guarantee of a split-optimal assignment, using a move-optimal split-operator, does not improve on the bound obtained by move-optimal schedules as stated in Theorem 5.1. We now prove this theorem.

We assume without loss of generality that $C^A_{\max} = 1$. Let k be a critical machine, i.e. $L_k = L_{k1} + L_{k2} = 1$. We first consider the case, that

$$\sum_{j=1}^{n} p_j \geq mL_{k1} + L_{k2}.$$

Then, using (5.1), the optimal makespan can be bounded from below by

$$C^*_{\max} \geq \frac{1}{m}\sum_{j=1}^{n} p_j \geq L_{k1} + \frac{1}{m}L_{k2}.$$

Incorporating the fact that $L_{k2} = 1 - L_{k1}$, we obtain

$$C^*_{\max} \geq \frac{(m-1)L_{k1} + 1}{m}. \tag{5.6}$$

Since $L_{k1} \geq L_{k2}$, and therefore $L_{k1} \geq 1/2$, we obtain:

$$\frac{C^A_{\max}}{C^*_{\max}} \leq \frac{m}{(m-1)L_{k1} + 1} \leq \frac{2m}{m+1}.$$

On the other hand, if

$$\sum_{j=1}^{n} p_j < mL_{k1} + L_{k2},$$

it follows that

$$\sum_{i=1}^{m} L_i = \sum_{j=1}^{n} p_j < mL_{k1} + L_{k2},$$

and a machine l with minimal workload satisfies

$$L_l \leq \sum_{i \neq k} L_i/(m-1) < L_{k1}. \tag{5.7}$$

Moreover, by (5.3), we know that (5.7) implies $L_{l2} \geq L_{k2}$. Hence,

$$L_{k1} > L_l = L_{l1} + L_{l2} \geq 2L_{l2} \geq 2L_{k2}. \tag{5.8}$$

Since a move-optimal split-operator is used to obtain the sets M_{i1} and M_{i2}, by using (5.2), from (5.8) follows $p_j \geq L_{k1} - L_{k2} > \frac{1}{2}L_{k1}$, for all $j \in M_{k1}$. Hence, M_{k1} contains only one job j_1 and

$$C^A_{\max} = L_k = L_{k1} + L_{k2} < \frac{3}{2}L_{k1} = \frac{3}{2}p_{j_1} \leq \frac{3}{2}C^*_{\max},$$

as $C^*_{\max} \geq p_j$, for all $j \in J$.

For $m \geq 3$, the theorem is proven, as $\frac{3}{2} \leq \frac{2m}{m+1}$. By using (5.1) with $m = 2$, and since (5.8) guarantees $L_l \geq 2L_{k2}$, we get

$$\begin{aligned} C^*_{\max} &\geq \frac{1}{2}\sum_{j=1}^{n} p_j = \frac{1}{2}(L_k + L_l) \geq \frac{1}{2}(L_k + 2L_{k2}) = \frac{1}{2}(L_{k1} + 3L_{k2}) \\ &= \frac{3}{2} - L_{k1}, \end{aligned}$$

where the last equality holds, since $L_{k2} = 1 - L_{k1}$. Moreover, as $C^*_{\max} \geq L_{k1}$ and M_{k1} contains only one job, we have

$$C^*_{\max} \geq \max\{L_{k1}, \frac{3}{2} - L_{k1}\} \geq \frac{3}{4}.$$

Therefore, for $m = 2$, we have

$$C^A_{\max} \leq \frac{4}{3}C^*_{\max}.$$

This completes the proof on the upper bound on the performance guarantee of Theorem 5.1.

To show that the analysis is tight, consider the following instance consisting of m jobs with processing time 1 and m jobs with processing time $1/m$. In the split-optimal assignment A, we schedule on every machine one job with processing time 1 and, additionally, on the first machine all jobs with processing time $1/m$ are scheduled. It is easy to check that this assignment is split-optimal for a move-optimal split-operator and that it has makespan $C^A_{\max} = 2$. In an optimal assignment A^*, we schedule on every machine one job with processing time 1 and one job with processing time $1/m$. The optimal makespan is $C^*_{\max} = 1 + 1/m$, and thus $C^A_{\max} = 2m/(m+1)C^*_{\max}$ (see Figure 5.4 for an illustration). This completes the proof of Theorem 5.1.

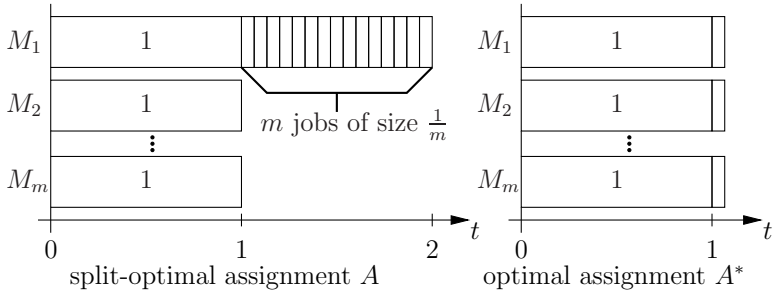

Figure 5.4: Worst-case example for split-optimal assignments.

5.3 ... and Move-Optimal

The worst-case instance for split-optimal assignments in Figure 5.4, showing the tightness of our analysis in the previous section, is obviously not move-optimal. This raises the question whether a combination of the two neighborhoods gives a better performance guarantee, which is answered in Theorem 5.2, for move-optimal split-operators. Moreover, if the sum of processing times of all jobs is small compared to the workload of the left part of a critical machine, we can even prove that this move-optimal and split-optimal assignment is globally optimal.

Like in the previous section, we assume without loss of generality that for a local optimal assignment A property (5.3) holds, and that $C^A_{\max} = 1$ (which can be derived by scaling the processing times).

Lemma 5.7 *Let A be a move-optimal and split-optimal assignment obtained by using a move-optimal split-operator. If there exists a critical machine k with*

$$\sum_{j=1}^{n} p_j < mL_{k1} + L_{k2},$$

then $C^A_{\max}$ is the optimal makespan.

PROOF. Let l be a machine with minimal workload. Then we know, by (5.7), that $L_l < L_{k1}$. By the move-optimality of the assignment A, for any job $j \in M_k$ holds

$$L_l + p_j \geq L_k = L_{k1} + L_{k2} > L_l + L_{k2}.$$

Hence, $p_j > L_{k2}$, for all jobs $j \in M_k$, and thus M_{k2} contains no job at all, i.e. $L_{k2} = 0$. It follows from the move-optimal split-operator that, whenever M_{k2} is empty, M_{k1} contains only one job, j_1. Hence, $C^A_{\max} = L_k = p_{j_1} \leq C^*_{\max}$. □

This lemma implies that we only have to consider cases in which the sum of the workloads on all machines is large enough compared to L_{k1}. Moreover, if L_{k1} is large enough, we can actually prove a bound on the makespan of this local optimal schedule, which is better than the guarantee in Theorem 5.2.

Lemma 5.8 *Let A be a move-optimal and split-optimal assignment obtained by using a move-optimal split-operator. If there exists a critical machine k with $L_{k1} \geq \frac{2}{3}$, then the makespan of A can be bounded by*

$$C^A_{\max} \leq \frac{3m}{2m+1} C^*_{\max},$$

where $C^*_{\max}$ denotes the optimal makespan.

PROOF. Due to Lemma 5.7, we only have to consider situations in which $\sum_j p_j \geq mL_{k1} + L_{k2}$. Therefore, (5.6) holds. Since the right hand side of (5.6) grows monotone with L_{k1}, for $L_{k1} \geq \frac{2}{3}$ this is minimal at $L_{k1} = \frac{2}{3}$. Then

$$\frac{C^A_{\max}}{C^*_{\max}} \leq \frac{m}{(m-1)L_{k1} + 1} \leq \frac{3m}{2m+1}. \qquad \square$$

Let k be a critical machine. Before we prove the performance guarantee on a move-optimal and split-optimal assignment A, we first partition the set of machines into several subsets, based on the properties of the machines in such

a move-optimal and split-optimal assignment:

$$\begin{aligned} S_< &= \{\, i \,:\, L_{i1} < L_{k1} \,\}, \\ S_\geq &= \{\, i \,:\, L_{i1} \geq L_{k1} \,\}, \\ S_{multi} &= \{\, i \in S_\geq \,:\, |M_{i1}| \geq 2 \,\}, \\ S_{single} &= S_\geq \setminus (S_{multi} \cup \{\, k \,\}). \end{aligned} \tag{5.9}$$

That is, $S_<$ is the set of machines that have a left part with workload smaller than L_{k1}, and $S_\geq$ is the set of the remaining machines. This set $S_\geq$ is again partitioned in one set containing all machines that have at least two jobs in the left part, and in one set containing the remaining machines in $S_\geq \setminus \{k\}$ that have exactly one job in the left part. Note that, $S_\geq \setminus \{k\} = S_{multi} \cup S_{single}$.

The workload of a machine in each of the above classes, can be bounded as follows.

Lemma 5.9 *Let A be a move-optimal and split-optimal assignment obtained by using a move-optimal split-operator. Let k be a critical machine, and moreover, let $S_<$ and S_{multi} be as defined in* (5.9)*. Then*

$$\begin{aligned} L_i &\geq 2(1 - L_{k1}) && \text{for } i \in S_< \\ L_i &\geq \tfrac{3}{2} L_{k1} && \text{for } i \in S_{multi}. \end{aligned}$$

PROOF. Consider a machine $i \in S_<$. Then, by property (5.3), we know that $L_{i1} < L_{k1}$ implies $L_{i2} \geq L_{k2}$. Moreover, as $L_{i1} \geq L_{i2}$ and $L_{k2} = 1 - L_{k1}$, we have $L_i = L_{i1} + L_{i2} \geq 2L_{i2} \geq 2L_{k2} = 2(1 - L_{k1})$.

Now, let $i \in S_{multi}$, and let $j_s \in M_{i1}$ be the smallest job in the left part of machine i. As M_{i1} contains at least two jobs, we know that $p_{j_s} \leq \frac{1}{2} L_{i1}$. Due to the move-optimality of the split-operator, $L_{i2} \geq L_{i1} - p_{j_s} \geq \frac{1}{2} L_{i1}$ holds. Hence, $L_i \geq \frac{3}{2} L_{i1} \geq \frac{3}{2} L_{k1}$. $\square$

The following lemma considers machines from the set $S_<$.

Lemma 5.10 *Let A be a move-optimal and split-optimal assignment obtained*

by using a move-optimal split-operator. Let k be a critical machine and moreover, let $S_<$ be as defined in (5.9)*. If $1/2 \leq L_{k1} \leq 2/3$ and $|S_<| \geq 1$, then*

$$\frac{C^A_{\max}}{C^*_{\max}} \leq \begin{cases} \frac{2m}{m+2} & \text{for } m \geq 4, \\ \frac{3m}{2m+1} & \text{for } m \leq 3. \end{cases}$$

PROOF. Using Lemma 5.9 and $|S_\geq| + |S_<| = m$, we obtain:

$$\begin{aligned} \sum_{j=1}^{n} p_j &= \sum_{i \in S_\geq} L_i + \sum_{i \in S_<} L_i \geq 1 + (|S_\geq| - 1)\, L_{k1} + |S_<| 2(1 - L_{k1}) \\ &= 1 + (m - 1 - |S_<|) L_{k1} + |S_<| 2(1 - L_{k1}) \\ &= 1 + (2 - 3L_{k1})|S_<| + (m-1)L_{k1}. \end{aligned}$$

Because we have $|S_<| \geq 1$ and $L_{k1} \leq 2/3$, we obtain:

$$\sum_{j=1}^{n} p_j \geq 3 + (m-4)L_{k1}.$$

For $m \geq 4$, the expression grows monotone with L_{k1}, whereas for $m \leq 3$, the expression declines monotone with increasing L_{k1}. By using the fact that $1/2 \leq L_{k1} \leq 2/3$ holds, we obtain

$$C^*_{\max} \geq \begin{cases} \frac{m+2}{2m} & \text{for } m \geq 4, \\ \frac{2m+1}{3m} & \text{for } m \leq 3. \end{cases} \qquad \square$$

The next lemma deals with the case that the set S_{multi} contains many machines.

Lemma 5.11 *Let A be a move-optimal and split-optimal assignment obtained by using a move-optimal split-operator. Let k be a critical machine and, moreover, let S_{multi} be as defined in* (5.9)*. If $1/2 \leq L_{k1} \leq 2/3$ and $|S_{multi}| \geq 2$, then*

$$\frac{C^A_{\max}}{C^*_{\max}} \leq \frac{2m}{m+2},$$

where $C^*_{\max}$ denotes the optimal makespan.

PROOF. Since $L_{k1} \leq 2/3$ and there is at least one job in the right part of machine k, due to move-optimality we have $L_i \geq L_{k1}$. Hence, using Lemma 5.9, we get:

$$\begin{aligned}\sum_{j=1}^{n} p_j &= L_k + \sum_{i \in S_{multi}} L_i + \sum_{i \in S_{single}} L_i + \sum_{i \in S_<} L_i \\ &\geq 1 + \frac{3}{2} L_{k1} |S_{multi}| + L_{k1} |S_{single}| + L_{k1} |S_<| \\ &= 1 + L_{k1} \left(\frac{3}{2} |S_{multi}| + |S_{single}| + |S_<| \right).\end{aligned}$$

By using $|S_{single}| = m - 1 - |S_<| - |S_{multi}|$, we obtain:

$$\sum_{j=1}^{n} p_j \geq 1 + L_{k1} \left(\frac{1}{2} |S_{multi}| + m - 1 \right) \geq 1 + m L_{k1} \geq \frac{2+m}{2},$$

where the second inequality is due to $|S_{multi}| \geq 2$ and the last holds due to $L_{k1} \geq 1/2$. Therefore, the optimal makespan is bounded by

$$C^*_{\max} \geq \frac{2+m}{2m}. \qquad \square$$

In the following, we prove the upper bound on the performance guarantee given in Theorem 5.2.

Since $\frac{3m}{2m+1} \leq \frac{2m+2}{m+3}$, for all $m \geq 2$, by Theorem 5.4 we only need to consider assignments A in which a critical machine k contains exactly two jobs. Moreover, by Lemma 5.8, we may restrict ourselves to the case $L_{k1} \in [\frac{1}{2}, \frac{2}{3}]$, and, by Lemma 5.7, we may assume $\sum_j p_j \geq m L_{k1} + L_{k2}$.

Let k be a critical machine. As $\max\{\frac{2m}{m+2}, \frac{3m}{2m+1}\} \leq \frac{2m+2}{m+3}$, due to Lemma 5.10, we can restrict ourselves to the case that there is no machine i with $L_{i1} < L_{k1}$ and, due to Lemma 5.11, we may assume that there is at most one machine i with $|M_{i1}| \geq 2$. Note that if no such machine exists, there are m jobs of processing time at least L_{k1} and one job of processing time $L_{k2} = 1 - L_{k1}$. Then, by the pigeonhole principle $C^*_{\max} = C^A_{\max}$. Hence, we assume that there is exactly one machine s with $|M_{s1}| \geq 2$.

Let j_1 be the smallest job in M_{s1}. If $p_{j_1} \geq \frac{m+3}{2m+2} - L_{k2}$, then there are $m-1$ jobs of processing time at least L_{k1} (contributed by machines $i \in S_{\geq}$), one job of processing time $L_{k2} = 1 - L_{k1}$ (from machine k) and at least two jobs of processing time $\frac{m+3}{2m+2} - L_{k2}$ (from machine s). By the pigeonhole principle then follows $C^*_{\max} \geq \frac{m+3}{2m+2}$.

On the other hand, if $p_{j_1} \leq \frac{m+3}{2m+2} - (1 - L_{k1})$, we can bound the workload of the right part of machine s by $L_{s2} \geq L_{s1} - p_{j_1}$, because we are using a move-optimal split-operator.

Since $L_{s1} \geq L_{k1}$, we can bound the total workload by:

$$\begin{aligned}\sum_{j=1}^{n} p_j = \sum_{i \neq s} L_i + L_s &\geq (m-1)L_{k1} + 1 - L_{k1} + L_{s1} + L_{s2} \\ &\geq (m-2)L_{k1} + 1 + 2L_{s1} - p_{j_1} \geq mL_{k1} + 1 - p_{j_1} \\ &\geq (m-1)L_{k1} + 2 - \frac{m+3}{2m+2}.\end{aligned}$$

Furthermore, by using $L_{k1} \geq 1/2$, we obtain:

$$\sum_{j=1}^{n} p_j \geq \frac{m-1}{2} + 2 - \frac{m+3}{2m+2} = \frac{m^2+3m}{2m+2}.$$

This implies that the optimal makespan can be bounded by

$$C^*_{\max} \geq \frac{1}{m} \sum_{j=1}^{n} p_j \geq \frac{m+3}{2m+2},$$

which proves the upper bound on the performance guarantee given in Theorem 5.2.

For instances with an odd number of machines, the previous analysis is tight. If we schedule m jobs of processing time 1 and m jobs of processing time $2/(m+1)$, as illustrated by the assignment A in Figure 5.5, we obtain a move-optimal and split-optimal assignment for a move-optimal split-operator, with makespan $C^A_{\max} = 2$. In the optimal assignment, all machines have the same workload and $C^*_{\max} = 1 + \frac{2}{m+1}$. For the split-optimality of this example, we need that the left part of machine M_2 has workload equal to 1. Therefore, this example only works

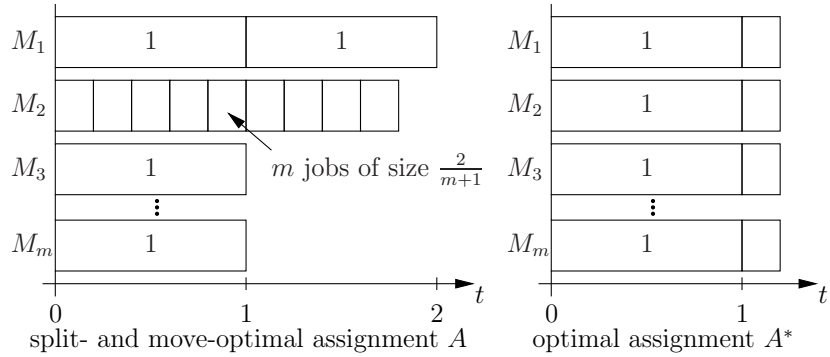

Figure 5.5: Worst-case example for odd number of machines.

for an odd number of machines. For an even number of machines, a lower bound on the performance guarantee is $\frac{2m}{m+2}$. This bound is obtained by an instance with m jobs of processing time 1 and m jobs of processing time $2/m$. In the move-optimal and split-optimal assignment, these jobs are scheduled similar to Figure 5.5.

In the case of $m = 2$ machines, the bound cannot be improved either, as is shown by the example in Figure 5.6, which has makespan $C^A_{\max} = 6$ for the local optimum and $C^*_{\max} = 5$, in an optimal schedule. This proves Theorem 5.2.

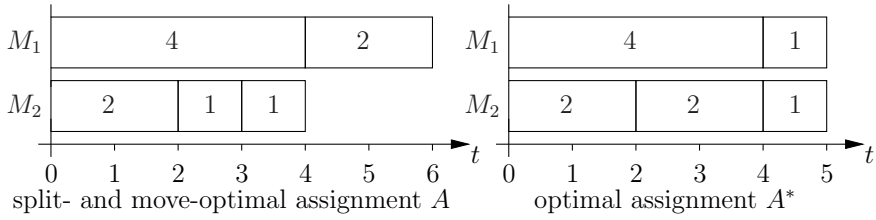

Figure 5.6: Worst-case example for $m = 2$ machines.

5.4 ... and Lexicographic-Move-Optimal

In the previous section, we have seen that the performance guarantee of a move-optimal and split-optimal assignment marginally improves on the performance guarantee of only a move-optimal or only a split-optimal assignment. Moreover, the example, showing the tightness of the guarantee for an odd number of machines, is not lexicographic-move-optimal. Therefore, in this section we consider the lexmove-optimal and split-optimal assignments.

In the following, we only consider a split-operator that retrieves a partition of the jobs of one machine into two parts according to the LPT-algorithm, i.e. we sort the jobs of a given machine i to non-increasing processing times and then iteratively assign a job to the set with minimum workload. In this way, we obtain a move-optimal partition $LPT(M_i) = (M_{i1}, M_{i2})$. Therefore, we can apply Lemma 5.7–5.11. In the remainder of this section, we again assume without loss of generality that for a lexmove-optimal and split-optimal assignment A, that $C^A_{\max} = 1$ and that (5.3) holds. Moreover, we also classify the machines into the sets $S_<$, S_{multi}, and S_{single}, as in (5.9).

Lemma 5.12 *Let A be a move-optimal and split-optimal assignment for a move-optimal split-operator. Let k be a critical machine and let l be a machine with minimal workload. Moreover, let $C^*_{\max}$ denote the optimal makespan. Then, if $l \in S_< \cup S_{multi}$,*

$$\frac{C^A_{\max}}{C^*_{\max}} \leq \frac{3m}{2m+1}.$$

PROOF. By Lemma 5.7 and Lemma 5.8, we can restrict ourselves to the case $\sum_{j=1}^{n} p_j \geq mL_{k1} + L_{k2}$ and $1/2 \leq L_{k1} \leq 2/3$.

If $l \in S_<$ we know, from Lemma 5.9, that $L_l \geq 2(1-L_{k1}) \geq 2/3$ and if $l \in S_{\text{multi}}$, from Lemma 5.9 follows $L_l \geq \frac{3}{2}L_{k1} \geq 3/4 \geq 2/3$. Hence, with $L_i \geq L_l \geq 2/3$, we have

$$C^*_{\max} \geq \frac{1}{m}\sum_{j=1}^{m} p_j = \frac{1}{m}\sum_{i=1}^{m} L_i \geq \frac{1}{m}\left(1 + \frac{2(m-1)}{3}\right) = \frac{2m+1}{3m}. \qquad \square$$

By this lemma, we know that in order to prove the performance guarantee of 3/2 in Theorem 5.3, we can restrict ourselves to local optimal assignments with $l \in S_{single}$. Moreover, as $L_l \geq 2/3$ implies $C^*_{\max} \geq 2/3$, we assume from here on that $L_l < 2/3$.

In the following, we use the concept of blocking jobs. We call a job j a *blocking job*, if $p_j + L_{k1} \geq 2/3$, where L_{k1} is the load of the left part of a critical machine k. Note that if, in some schedule, a blocking job is assigned to the same machine as a job of size at least L_{k1}, then the makespan of this schedule will be at least 2/3.

In the following, we prove the upper bound on the performance guarantee given in Theorem 5.3.

Let l be a machine with minimal workload. By Theorem 5.4, we may assume without loss of generality that $n_k = 2$. Moreover, by Lemma 5.7 and Lemma 5.8, we restrict ourselves to the case $\sum_j p_j \geq mL_{k1} + L_{k2}$ and $L_{k1} \leq 2/3$. Finally, we define the sets $S_<$, S_{multi}, and S_{single} as in (5.9). Then, by Lemma 5.12, we assume $l \in S_{\text{single}}$ and $L_l < 2/3$.

Under these assumptions, we claim that for a machine $i \in S_< \cup S_{multi}$ the sum of processing times of blocking jobs, scheduled on this machine i, is at least 2/3. None of the blocking jobs, which A assigns to a machine in $S_< \cup S_{multi}$, can be scheduled together with a job of size at least L_{k1} in a schedule with makespan smaller than 2/3. Thus, in such a schedule all these jobs need to be distributed over $|S_< \cup S_{multi}|$ machines, as each machine in $S_{single} \cup \{k\}$ processes at least one job with processing time at least L_{k1}. From our claim, it now follows that the machine with maximal workload from these blocking jobs has a workload of at least 2/3. Hence, we always have $C^*_{\max} \geq 2/3$, and the upper bound on the performance guarantee of Theorem 5.3 is proven.

To prove our claim, first we consider a machine $i \in S_<$. From Lemma 5.9 and Lemma 5.6, it follows that for the processing time of each job $j \in M_i$ holds $p_j \geq L_i - L_l \geq 2(1 - L_{k1}) - L_l > 4/3 - 2L_{k1}$. Since $L_{k1} \leq 2/3$, we obtain $p_j + L_{k1} > 4/3 - L_{k1} \geq 2/3$. Thus, each job $j \in M_i$ is a blocking job and the sum of processing times of blocking jobs assigned to machine $i \in S_<$ is equal to $L_i \geq 2(1 - L_{k1}) \geq 2/3$.

Now, consider a machine $i \in S_{multi}$, with $|M_{i1}| \geq 3$. The smallest job in the

left part, say $j_0 \in M_{i1}$, has processing time at most $p_{j_0} \leq L_{i1}/|M_{i1}|$. By move-optimality of the split-operator, we know that the workload of the right part can be bounded by

$$L_{i2} \geq L_{i1} - p_{j_0} \geq \frac{|M_{i1}| - 1}{|M_{i1}|} L_{i1} \geq \frac{2}{3} L_{i1}.$$

Hence, by Lemma 5.6 we know that any job $j \in M_i$ has a processing time $p_j \geq L_i - L_l > \frac{5}{3} L_{i1} - 2/3$. Thus, $p_j + L_{k1} > \frac{8}{3} L_{k1} - 2/3 \geq 2/3$. Hence, for all $i \in S_{multi}$ with $|M_{i1}| \geq 3$, every job $j \in M_i$ is a blocking job, and the sum of processing times of the blocking jobs assigned to such a machine i is equal to $L_i \geq 2/3$.

Finally, consider a machine $i \in S_{multi}$, with $|M_{i1}| = 2$, say $M_{i1} = \{j_1, j_2\}$ with $p_{j_1} \geq p_{j_2}$. Due to the move-optimality of the split-operator, $L_{i2} \geq L_{i1} - p_{j_2} = p_{j_1}$ holds, and since the assignment A is lexmove-optimal, by Lemma 5.6 we also know that $L_l \geq L_i - p_{j_2} = L_{i2} + p_{j_1} \geq 2p_{j_1}$. Hence, $p_{j_1} \leq L_l/2 < 1/3$. This implies $p_{j_2} = L_{i1} - p_{j_1} > L_{k1} - 1/3 \geq 1/6$, and thus, job j_2 is a blocking job, as $L_{k1} + 1/6 \geq 2/3$. Since $p_{j_1} \geq p_{j_2}$, job j_1 is also a blocking job.

Moreover, since the LPT-algorithm is used as a split-operator, there exists at least one job $j_3 \in M_{i2}$, with $p_{j_3} \geq p_{j_2}$. Thus, M_i contains at least three blocking jobs, j_1, j_2, and j_3, and the sum of processing times of these three jobs is at least

$$p_{j_1} + p_{j_2} + p_{j_3} \geq L_{k1} + 1/6 \geq 2/3,$$

which proves the upper bound on the performance guarantee of Theorem 5.3.

To show a lower bound on the performance guarantee, let $\delta = 1/(3m - 4)$ and consider the instance which consists of $2m - 2$ jobs of processing time 3δ, one job of processing time $1 + \delta$ and $m - 1$ jobs of processing time $2 - \delta$. The assignment A depicted in Figure 5.7 is lexmove-optimal and split-optimal and has makespan $C^A_{\max} = 3$, whereas the optimal makespan is $C^*_{\max} = 2 + 2\delta$. This yields a ratio

$$\frac{C^A_{\max}}{C^*_{\max}} = \frac{3m - 4}{2m - 2} = \frac{3}{2} - \frac{1}{2m - 2},$$

and, thus, completes the proof of Theorem 5.3.

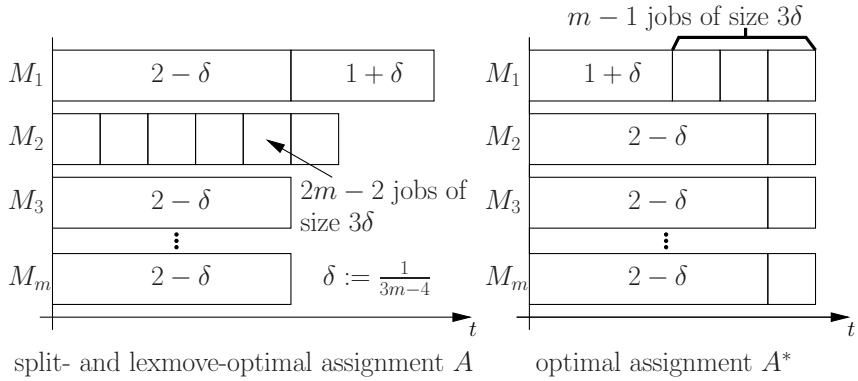

Figure 5.7: Lower bound example for a lexmove-optimal and split-optimal assignment.

5.5 Concluding Remarks

We introduce the split-neighborhood based on perfect matchings in a complete graph. We discuss the quality of this neighborhood alone and in combination with the move-neighborhood.

We observe that the upper bounds on the performance guarantees for local-optima in the split-neighborhood and the move-neighborhood are equal. However, the worst-case examples, attaining the upper bounds, have a different structure. This indicates that a combination of the split-neighborhood and the move-neighborhood improves on the upper bound on the performance guarantee for local optima.

The upper bound on the performance guarantee for local optima in the union of the split-neighborhood and the move-neighborhood is slightly better than for local optima in only one of the neighborhoods. But comparing the asymptotic behavior of the upper bounds on the performance guarantees for a large number of machines, there is no difference, i.e. all the upper bounds up to this point approach 2, for the number of machines becoming large.

By examining the worst-case examples on the performance guarantee for local

optima in the union of the split-neighborhood and the move-neighborhood, we observe that the performance guarantee can be improved by using a modified version of local optima in the move-neighborhood. Here, the local search heuristic is allowed to advance to non-improving (in the objective value) neighbors in the move-neighborhood. This gives a vast improvement in the upper bound on the performance guarantee of local optima in the union of the split-neighborhood and modified move-neighborhood.

Chapter 6

Conclusions

In this thesis, several very large-scale neighborhoods for some basic scheduling problems were investigated. We considered a general approach of combining independent neighborhood operators to obtain a larger neighborhood (the CAPI-neighborhood introduced in Chapter 2) and a matching based neighborhood that arises from a basic neighborhood (see Chapter 3).

The computational tests, using these types of neighborhoods, confirm the conclusions drawn by others on the practical use of very large-scale neighborhoods (see e.g. Hurink [42]): the structure of the neighborhood operators is more important than the size of the neighborhood. In more detail, we have seen, that very large-scale neighborhoods obtained by combining independent neighborhood operators are not good on beforehand, for making local search efficient. Based on our experiences, we may conclude that the size of the neighborhood does not guarantee a better quality. Only if structural properties of the considered problem make it useful to combine different neighborhood operators, very large-scale neighborhoods may be successful. Success in this context means that the combined neighborhood operators lead to different and better solutions than a sequence of (greedy) chosen neighborhood steps in the underlying basic neighborhood (for our problem, this was not the case!).

If the introduced neighborhoods are used in a local search method like tabu

search or simulated annealing, we expect that the order of the neighborhoods regarding quality and computational time stays the same although the absolute differences may get smaller. The conducted computational tests give a hint on this fact since we are testing iterative improvement with random initial solutions on many instances and, additionally in Chapter 2, perturb the obtained local optima.

A further possible advantage of very large-scale neighborhoods consisting of combined operators of a basic neighborhood can be a speed up in computational time resulting from the possibility of the parallel execution of many simple neighborhood operators. Based on our results, we may conclude that this can only be the case if most of the executed combined neighborhood operators combine several basic neighborhood operators, and if the extra computational effort for searching the combined neighborhood, in comparison with the basic neighborhood, is not large.

Secondly, we introduced very large-scale neighborhoods by restricting the considered problem to an easy solvable case, i.e. the PAV-neighborhood described in Chapter 2, the split-neighborhood of Chapter 3 and the split-neighborhood presented in Chapter 5.

We conclude from the computational tests and theoretic arguments, that large neighborhoods resulting from restricting the considered problem also are not better, on beforehand, compared to standard neighborhoods. On the other hand, combining these neighborhoods with standard neighborhoods may give an improvement in solution quality or running time of local search methods in practice.

All in all, we suggest to develop and use very large-scale neighborhoods of the considered types only if problem specific properties or computational arguments on beforehand give an indication that the very large-scale neighborhoods have some potential to be successful.

In the second part of the thesis we derived qualitative results for local optima of the move-neighborhood (Chapter 4) and the split-neighborhood (Chapter 5). There are not many results known on performance guarantees for local optima, especially for scheduling problems. This is an interesting area where more research should be carried out.

A performance guarantee for local optima makes local search to be an approximation algorithm. Although the performance guarantee is worse compared to constructive heuristics (however, for our cases the difference is quite small), the advantage of giving performance guarantees for local optima is that these guarantees hold for all solutions which are locally optimal and not only for a single solution as for most constructive heuristics. It is interesting to see that already simple structured solutions like the local optima allow a reasonable performance guarantee. Since local search heuristics like simulated annealing or tabu search tend to visit a lot of local optima during the search process, we may expect that the best found solution has a much better quality than that given by the performance guarantee. Summarizing, local search is not only a useful approach in practice to receive solutions of good quality, but also allows to have a guarantee for the quality of the achieved solution.

Bibliography

[1] E. Aarts and J. K. Lenstra, editors. *Local search in combinatorial optimization*. John Wiley & Sons Ltd., Chichester, 1997. A Wiley-Interscience Publication.

[2] F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Sviridenko. Approximation schemes for minimizing average weighted completion time with release dates. In *40th Annual Symposium on Foundations of Computer Science (New York, 1999)*, pages 32–43. IEEE Computer Soc., Los Alamitos, CA, 1999.

[3] R. Agarwal, Ö. Ergun, J. B. Orlin, and C. N. Potts. Solving parallel machine scheduling problems with very-large scale neighborhood search. Working paper.

[4] R. H. Ahmadi and U. Bagchi. Lower bounds for single-machine scheduling problems. *Naval Research Logistics. A Journal Dedicated to Advances in Operations and Logistics Research*, 37(6):967–979, 1990.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, Reading, 1974.

[6] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102, 2002.

[7] J. M. Van Den Akker, J. A. Hoogeveen, and S. L. Van De Velde. Parallel machine scheduling by column generation. *Operations Research*, 47(6):862–872, 1999.

[8] G. Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[9] A. Avidor, Y. Azar, and J. Sgall. Ancient and new algorithms for load balancing in the l_p norm. *Algorithmica. An International Journal in Computer Science*, 29(3):422–441, 2001.

[10] K. R. Baker, editor. *Introduction to sequencing and scheduling*. John Wiley & Sons Ltd., 1974.

[11] K. R. Baker and A. G. Merten. Scheduling with parallel processors and linear delay costs. *Naval Research Logistics Quarterly*, 20:793–804, 1973.

[12] J. W. Barnes and M. Laguna. Solving the multiple-machine weighted flow time problem using tabu search. *IIE Transactions*, 25(2):121–128, 1993.

[13] H. Belouadah and C. N. Potts. Scheduling identical parallel machines to minimize total weighted completion time. *Discrete Applied Mathematics*, 48(3):201–218, 1994.

[14] P. Brucker. *Scheduling algorithms*. Springer-Verlag, Berlin, fourth edition, 2004.

[15] P. Brucker, J. Hurink, and F. Werner. Improving local search heuristics for some scheduling problems. part ii. *Discrete Applied Mathematics*, 72(1-2):47–69, 1997.

[16] T. Brueggemann and J. L. Hurink. Matching based exponential neighborhoods for parallel machine scheduling. Memorandum 1773, Departement of Applied Mathematics, University of Twente, Enschede, 2005.

[17] T. Brueggemann and J. L. Hurink. Two very large-scale neighborhoods for single machine scheduling. *OR Spectrum*, 2006. To appear.

[18] T. Brueggemann, J. L. Hurink, and W. Kern. Quality of move-optimal schedules for minimizing total weighted completion time. *Operations Research Letters*, 34(5):583–590, 2006.

[19] T. Brueggemann, J. L. Hurink, T. Vredeveld, and G. J. Woeginger. Very large-scale neighborhoods with performance guarantees for minimizing makespan on parallel machines. Memorandum 1801, Departement of Applied Mathematics, University of Twente, Enschede, 2006.

[20] J. Bruno, E. G. Coffman Jr., and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the Association for Computing Machinery*, 17:382–387, 1974.

[21] J. Carlier and P. Villon. A new heuristic for the traveling salesman problem. *Revue Francaise d'automatique, d'informatique et de Recherche Operationelle (RAIRO)*, 24:245–253, 1990.

[22] A. K. Chandra and C. K. Wong. Worst-case analysis of a placement algorithm related to storage allocation. *SIAM Journal on Computing*, 4(3):249–263, 1975.

[23] C. Chu. A branch-and-bound algorithm to minimize total flow time with unequal release dates. *Naval Research Logistics*, 39:859–875, 1992.

[24] R. A. Cody and E. G. Coffman Jr. Errata: "Record allocation for minimizing expected retrieval costs on drum-like storage devices" (J. Assoc. Comput. Mach. **23** (1976), no. 1, 103–115). *Journal of the Association for Computing Machinery*, 23(3):572, 1976.

[25] R. A. Cody and E. G. Coffman Jr. Record allocation for minimizing expected retrieval costs on drum-like storage devices. *Journal of the Association for Computing Machinery*, 23(1):103–115, 1976.

[26] R. K. Congram, C. N. Potts, and S. L. van de Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS Journal on Computing*, 14(1):52–67, 2002.

[27] S. A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[28] V. G. Deĭneko and G. J. Woeginger. A study of exponential neighborhoods for the travelling salesman problem and for the quadratic assignment problem. *Mathematical Programming*, 87(3, Ser. A):519–542, 2000.

[29] W. L. Eastman, S. Even, and I. M. Isaacs. Bounds for the optimal scheduling of n jobs on m processors. *Management Science*, 11(2):268–279, 1964.

[30] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[31] S. E. Elmaghraby and S. H. Park. Scheduling jobs on a number of identical machines. *American Institute of Industrial Engineers Transactions*, 6:1–12, 1974.

[32] G. Finn and E. Horowitz. A linear time approximation algorithm for multiprocessor scheduling. *BIT*, 19:312–320, 1979.

[33] H. N. Gabow. An efficient implementation of edmonds' algorithm for maximum matching on graphs. *Journal of the Association for Computing Machinery*, 23(2):221–234, 1976.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[35] P. C. Gilmore, E. L. Lawler, and D. B. Shmoys. Well-solved special cases. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors, *The traveling salesman problem*, Wiley-Interscience Series in Discrete Mathematics, pages 87–143. John Wiley & Sons Ltd., Chichester, 1985.

[36] R. R. Goldberg and J. Shapiro. A tight upper bound for the k-partition problem on ideal sets. *Operations Research Letters*, 24(4):165–173, 1999.

[37] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[38] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

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

[40] G. Gutin, A. Yeo, and A. Zverovitch. Exponential neighborhoods and domination analysis for the TSP. In G. Gutin and A. P. Punnen, editors, *The traveling salesman problem and its variations*, volume 12 of *Combinatorial Optimization*, pages 223–256. Kluwer Academic Publishers, Dordrecht, 2002.

[41] C. Hoede. A heuristic for scheduling on parallel machines to minimize total weighted completion time. Personal communication, 2005.

[42] J. Hurink. An exponential neighborhood for a one-machine batching problem. *OR Spectrum*, 21(4):461–476, 1999.

[43] C. A. J. Hurkens and T. Vredeveld. Local search for multiprocessor scheduling: How many moves does it take to a local optimum? *Operations Research Letters*, 31:137–141, 2003.

[44] A. H. G. Rinnooy Kan. *Machine Scheduling Problems: Classification, complexity and computations*. Martinus Nijhoff, The Hague, The Netherlands, 1976.

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

[46] T. Kawaguchi and S. Kyan. Worst case bound of an lrf schedule for the mean weighted flow-time problem. *SIAM Journal on Computing*, 15(4):1119–1129, 1986.

[47] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, NY, USA, 1976.

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

[49] J. Y.-T. Leung and W. D. Wei. Tighter bounds on a heuristic for a partition problem. *Information Processing Letters*, 56(1):51–57, 1995.

[50] S. Lin. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44:2245–2269, 1965.

[51] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21:498–516, 1973.

[52] C. H. Papadimitriou. Algorithms, games, and the internet. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, pages 749–753 (electronic), New York, 2001. ACM.

[53] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Dover Publications Inc., Mineola, NY, 1998. Corrected reprint of the 1982 original.

[54] C. N. Potts and S. L. van de Velde. Dynasearch-iterative local improvement by dynamic programming. part 1. the traveling salesman problem. Technical report, University of Twente, Enschede, The Netherlands, 1995.

[55] A. P. Punnen. The traveling salesman problem: new polynomial approximation algorithms and domination analysis. *Journal of Information & Optimization Sciences. A Journal Devoted to Advances in Information Sciences, Optimization Sciences and Related Aspects*, 22(1):191–206, 2001.

[56] S. K. Sahni. Algorithms for scheduling independent tasks. *Journal of the Association for Computing Machinery*, 23(1):116–127, 1976.

[57] P. Schuurman and T. Vredeveld. Performance guarantees of local search for multiprocessor scheduling. *Informs Journal on Computing*, 2006. To appear.

[58] M. Skutella and G. J. Woeginger. A PTAS for minimizing the total weighted completion time on identical parallel machines. *Mathematics of Operations Research*, 25(1):63–75, 2000.

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

[60] P. M. Thompson and H. N. Psaraftis. Cyclic transfer algorithms for multivehicle routing and scheduling problems. *Operations Research*, 41(5):935–946, 1993.

[61] T. Vredeveld. *Combinatorial approximation algorithms: Guaranteed versus experimental performance*. PhD Thesis. Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2002.

[62] T. Vredeveld. Lower bound example for move-optimal assignments on parallel machines and equal weight to processing time ratios. Personal communication, 2004.

[63] S. Yanai and T. Fujie. On a dominance test for the single machine scheduling problem with release dates to minimize total flow time. *Journal of the Operations Research Society of Japan*, 47(2):96–111, 2004.

Summary

Local search heuristics are an important class of algorithms for obtaining good solutions for hard combinatorial optimization problems. Important issues for these heuristics are solution quality and the time needed to obtain a good solution.

Roughly speaking, a local search heuristic starts with an initial solution, e.g. obtained with the help of a constructive heuristic. The local search heuristic then iteratively explores a neighborhood of the current solution and chooses a new solution out of this neighborhood to become the next current solution. Often, only new solutions are considered, which improve on the current one. The local search heuristic halts if some stopping criterion is met, e.g. there exists no better solution in the neighborhood.

The neighborhood is a critical issue in a local search heuristic. It directly influences the local behavior of a local search heuristic, since it restricts the choice of a new solution in a single iteration. This determines the local efficiency of a local search heuristic. On the other side, this local behavior also influences the global behavior, i.e. the sequence of feasible solutions obtained by the local search heuristic. This sequence of solutions represents the navigational behavior of the local search heuristic and determines the global efficiency of a local search heuristic.

A key element is the size of a neighborhood. The size determines how many choices there are to find a better solution. This also designates the time the heuristic needs in every iteration. Since with smaller neighborhoods we may not have (directly or indirectly) considered many solutions at the end of a local

search heuristic, the size may also have an influence on the solution quality from a global point of view.

In the first part of the thesis, we consider neighborhoods of large size that can be explored in a fast manner, in order to reach local optima of good quality in less time. We consider the scheduling problems of minimizing the sum of job completion times on a single machine with the presence of release dates, and of minimizing the sum of weighted job completion times on parallel and identical machines. By concentrating on the local efficiency, we develop very large-scale neighborhoods for these two scheduling problems. For all the introduced neighborhoods, we explain what solutions they contain and how the neighborhood is efficiently explored to obtain a better neighbor. We also compare their performance to basic neighborhoods regarding solution quality and running time by conducting computational tests.

For the single machine problem, we first introduce a neighborhood that bases on combining independent operators. These operators form a smaller neighborhood, and by combining several operators we obtain a very large-scale neighborhood. A best neighbor in this very large-scale neighborhood can be obtained by determining a shortest path in an improvement graph. In the computational test, this neighborhood does not perform well. We present a second very large-scale neighborhood that bases on a dominance rule for the considered problem. Here, a best neighbor can be obtained by sorting. In the computational test, this neighborhood is fast but regarding solution quality, it is only successful in combination with another neighborhood.

For the parallel machine problem, we also present a very large-scale neighborhood that bases on combining independent operators of a smaller neighborhood. But here, the best neighbor is obtained by determining a maximum weight matching in an improvement graph. In the computational test, this neighborhood is slightly faster than the underlying smaller neighborhood. A second very large-scale neighborhood is introduced that bases on a restricted version of the considered problem, which can be solved in polynomial time. The best neighbor of this neighborhood is obtained by determining a minimum weight perfect matching. In the computational test, this neighborhood performs comparable to the smaller neighborhood used for obtaining the first very-large scale neighborhood.

In the second part of the thesis, we examine the global efficiency in terms of

solution quality obtained at the end of a local search heuristic. We consider the problems of scheduling jobs on parallel identical machines in order to minimize the sum of weighted job completion times and of minimizing the makespan (completion time of last planned job). For these problems, we consider certain neighborhoods and analyze the quality of their local optima. Although the found performance guarantees are worse compared to constructive heuristics (however, for our cases the differences are quite small), the advantage of giving performance guarantees for local optima is that these guarantees hold for all solutions which are locally optimal and not only for a single solution as for most constructive heuristics. It is interesting to see that already simple structured solutions like the local optima allow a reasonable performance guarantee. Since local search heuristics like simulated annealing or tabu search tend to visit a lot of local optima during the search process, we may expect that the best found solution has a much better quality than that given by the performance guarantee.

For the problem of minimizing the sum of weighted job completion times, we consider the neighborhood of moving a job from one machine to another. Our main result is that a local optima in this neighborhood has an objective value of at most $\frac{3}{2} - \frac{1}{2m}$ times the optimal objective value, where m denotes the number of machines. We also consider the behavior of local optima by restricting this problem and moreover, see a relation of local optima to a constructive heuristic.

For the problem of minimizing the makespan, we introduce a very large-scale neighborhood, where the best neighbor is obtained by determining a perfect matching with minimal bottleneck cost. We get the result that a local optima in this neighborhood has an objective value of at most $2 - \frac{2}{m+1}$ times the optimal objective value. We also analyze solutions that are additionally local optimal in a second neighborhood. If local optima of this second neighborhood fulfill certain properties, we are able to improve the performance guarantee to 3/2.

Samenvatting

Lokale zoekmethoden zijn een belangrijke klasse van algoritmen voor het vinden van goede oplossingen voor moeilijke combinatorische optimalisatie problemen. Belangrijke aandachtspunten bij deze methoden zijn de kwaliteit van oplossingen en de tijd die nodig is om deze te verkrijgen.

Grofweg begint een lokale zoekmethode met een initiële oplossing, welke bijvoorbeeld verkregen is met een constructieve heuristiek. De lokale zoekmethode doorzoekt dan in iedere iteratie de omgeving van de actuele oplossing en kiest een oplossing uit deze omgeving als nieuwe actuele oplossing. Vaak worden alleen oplossingen gekozen, die beter zijn dan de huidige actuele. De lokale zoekmethode stopt wanneer een bepaalde toestand bereikt wordt, bijvoorbeeld als er in de omgeving geen betere oplossingen bestaan.

De omgeving is een essentieel onderdeel van een lokale zoekmethode. Het beïnvloedt direct het lokale gedrag van de lokale zoekmethode, omdat de omgeving de keuze van nieuwe oplossingen in ieder iteratie beperkt; hiermee heeft de omgeving directe invloed op de lokale efficiency van de methode. Anderzijds wordt ook het globale gedrag beïnvloed. Met het globale gedrag wordt de rij van toegelaten oplossingen bedoeld, die men krijgt door toepassing van de lokale zoekmethode. Deze rij van oplossingen representeert het navigatie gedrag van de lokale zoekmethode, en daarmee de globale efficiency van de methode.

Een belangrijk aspect is de omvang van een omgeving. De omvang bepaalt hoeveel keuze er bestaat om een betere oplossing te vinden. De omvang bepaalt ook de tijd die de heuristiek in iedere iteratie nodig heeft. Met kleinere omgevingen heeft men aan het eind van een lokale zoekmethode niet zo veel oplossingen (di-

rect of indirect) bekeken. Daarmee heeft de omvang ook invloed op de kwaliteit van de oplossing, vanuit een globaal perspectief.

In het eerste deel van dit proefschrift zijn we geïnteresseerd in grote omgevingen die op een snelle manier doorzocht kunnen worden, om lokale optima van goede kwaliteit in korte tijd te bereiken. We beschouwen twee planningsproblemen. Het eerste probleem is het minimaliseren van de som van de beëindigingstijden van de jobs op één machine als de jobs beperkingen op de begintijden hebben. Het tweede probleem is het minimaliseren van de gewogen som van de beëindigingstijden van jobs op een gegeven aantal parallelle en identieke machines. Van alle voorgestelde omgevingen beschrijven we de structuur van de oplossingen in de omgeving, en hoe de omgeving efficiënt doorzocht kan worden. We vergelijken de resultaten voor deze omgevingen met standaard omgevingen met betrekking tot de oplossingskwaliteit en de rekentijd van de algoritmen.

Voor het één machine probleem ontwikkelen we een zeer grote omgeving, die gebaseerd is op het combineren van onafhankelijke operatoren. Deze operatoren op zich vormen een kleinere omgeving. De beste oplossing in de omgeving wordt gevonden door het berekenen van een kortste pad in een verbeteringsgraaf. Uit rekentests blijkt dat deze omgeving niet goed is. We geven dan ook een tweede zeer grote omgeving, welke gebaseerd is op een dominantie regel voor het beschouwde probleem. Hier wordt de beste oplossing in de omgeving bereikt door sorteren. Uit rekentests blijkt dat deze omgeving snel is, maar goede oplossingen worden alleen verkregen in combinatie met een andere omgeving.

Voor het tweede probleem met meerdere machines ontwikkelen we een zeer grote omgeving, die ook gebaseerd is op het combineren van onafhankelijke operatoren uit een kleinere omgeving. Hier wordt de beste oplossing in de omgeving bereikt door het berekenen van een matching van maximaal gewicht in een verbeteringsgraaf. Uit rekentests blijkt dat deze omgeving iets sneller is dan de ten grondslag liggende kleine omgeving. Een tweede zeer grote omgeving wordt onderzocht, welke gebaseerd is op een beperkte, polynomiaal oplosbare versie van het beschouwde probleem. De beste oplossing in de omgeving wordt bereikt door het berekenen van een perfecte matching van minimaal gewicht. Uit rekentests blijkt dat deze omgeving zich vergelijkbaar gedraagt met de kleine omgeving, die gebruikt werd, om de eerste zeer grote omgeving te ontwikkelen.

In het tweede deel van het proefschrift onderzoeken we de globale efficiency, dat wil zeggen de kwaliteit van de oplossing aan het eind van een lokale zoek-

methode. We beschouwen twee problemen op het gebied van het plannen van jobs op identieke parallelle machines. Het eerste probleem is om de gewogen som van job beëindigingstijden te minimaliseren, en het tweede probleem is om de makespan (beëindigingstijd van de laatst geplande job) te minimaliseren. Voor deze problemen beschouwen we een aantal omgevingen en analyseren de kwaliteit van lokale optima met betrekking tot deze omgeving. Hoewel de gevonden prestatie garanties (iets) slechter zijn dan van constructieve heuristieken, is een voordeel van een prestatie garantie voor lokale optima, dat deze garantie geldig is voor alle oplossingen die lokaal optimaal zijn. Dit in tegenstelling tot de situatie bij constructieve heuristieken, waar de garantie vaak geldt voor een enkele oplossing. Het is interessant om te zien, dat een eenvoudige structuur in oplossingen, zoals lokale optima die geven, al een redelijke prestatie garantie mogelijk maakt. Omdat lokale zoekmethoden zoals simulated annealing en tabu search tijdens het zoekproces meestal veel lokale optima bezoeken, kunnen we verwachten, dat de best gevonden oplossing een veel betere kwaliteit heeft dan de gegeven prestatie garantie.

Voor het probleem van het minimaliseren van gewogen som van job beëindigingstijden beschouwen we de omgeving bestaand uit het verplaatsen van een job van een machine naar een andere. Ons hoofdresultaat is, dat lokale optima in deze omgeving een waarde hebben van hoogstens $\frac{3}{2} - \frac{1}{2m}$ keer de optimale waarde. Hierin is m het aantal parallelle machines. Verder beschouwen we lokale optima in een beperkte versie van het probleem, en bekijken een relatie van lokale optima met een constructieve heuristiek.

Voor het probleem van het minimaliseren van de makespan introduceren we een zeer grote omgeving, waarin de beste oplossing berekend kan worden door het bepalen van een perfecte matching met minimale bottleneck kosten. We bewijzen het resultaat, dat lokale optima in deze omgeving een waarde hebben van hoogstens $2 - \frac{2}{m+1}$ keer de optimale waarde. Verder beschouwen we oplossingen, die tevens lokaal optimaal zijn in een tweede omgeving. Onder bepaalde omstandigheden, kunnen we de prestatie garantie verbeteren naar 3/2.

About the Author

Tobias Brueggemann was born on May 20, 1976 in Ibbenbueren, Germany. He finished secondary school at the Johannes Kepler Gymnasium Ibbenbueren in 1996.

Afterwards, he began studying Mathematics with a minor in Physics at the University of Osnabrueck, Germany. In 2002, he graduated with a thesis on a local search approach for exam timetabling problems.

Since 2002, he was doing his Ph.D. research with the Discrete Mathematics and Mathematical Programming Group, University of Twente, Enschede, The Netherlands, under supervision of Gerhard J. Woeginger and Johann L. Hurink. His interest was in designing and implementing very large-scale neighborhoods for scheduling problems, as well as theoretically examining the efficiency of local search using these neighborhoods. The results of his research are presented in this thesis.