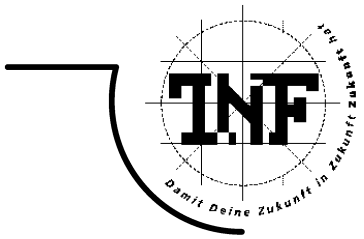
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



GRASP with Path Relinking for a Discrete Lotsizing and Scheduling Problem with Non-trivial Constraints

Master's Thesis

for obtaining the academic title

Master of Science

in

Internationaler Univerisätslehrgang
Informatics: Engineering & Management

composed at ISI-Hagenberg

Handed in by:
Plamen Alexandrov, 0855841

Finished on:
29th June, 2009

Scientific Advisor:
Prof. (FH) Priv.-Doz. DI Dr. Michael Affenzeller

Hagenberg, June, 2009

Abstract

GRASP with Path Relinking for a Discrete Lotsizing and Scheduling Problem with Non-trivial Constraints

Plamen Alexandrov

ISI-Hagenberg

Master's Thesis

This paper addresses a real-world optimization problem from the automotive supply chain industry. After mathematical modeling, it turns out to be a variation of a single-level multi-item Discrete Lot Sizing and Scheduling Problem (DLSP) on parallel machines. DLSP is concerned with allocating production lots and sizes and scheduling them on multiple resources, and is known to be NP-hard. Setups are designed to take only part of the available time in a period. No inventory holding costs are taken into account and backlogging is allowed. Additional non-trivial constraints are introduced for limiting both, parallel production of an item and the total number of setups allowed in a time period.

GRASP with Path Relinking metaheuristic is proposed for the specified problem. Innovatively, a selection pressure concept is introduced for randomized local search and an intelligent path exploration strategy is suggested. An effective parallel implementation is provided to manage the high computational efforts. Additionally, results are compared against those of a Tabu Search metaheuristic and an existing Mixed Integer Programming approach. Parallel GRASP approach performs slightly better in some cases.

Acknowledgments

This Master's project is supported by "RISC Software Gmbh" company in the frame of ISI Hagenberg '08-09 Master's program (Project Nr. 237640).

We would like to acknowledge our industrial advisers DI Dr. Peter Stadelmeyer and DI Roman Stainko from "RISC Software Gmbh" company for their devoted assistance and supervision throughout the project.

Contents

Chapter 1

Introduction

The problem of production planning and scheduling addresses decisions on the effective and cost-efficient use of an in-house production system's resources in order to satisfy customer demands. In practice, such manufacturing environments are organized into several segments, often called parallel machines or plants (flow lines or work centers for instance). This scenario can be further elaborated with heterogeneous configurable segments capable of providing specific operations or producing different sets of products (or batches) at different rates.

There are various levels of decisions, according to the planning period, that formulate a hierarchical process (Drexl [1]). However, this paper considers short-term decisions, where the discrete planning horizon is one or two weeks and due times and sizes of demands have already been determined. Furthermore, we focus on determining production lots sizes and sequences in order to minimize setup costs, backlog costs for unmet demands and maximize production for future demands on parallel machines. This results in a single-level, multi-item Discrete Lotsizing and Scheduling Problem (DLSP) on parallel machines with backlogging. No inventory costs are considered.

1.1 About the Thesis Project

The current paper represents a Master's thesis in the frame of "International School of Informatics - ISI Hagenberg '08-09" and the research associated with the thesis project was already initiated by "RISC Software Gmbh" - a scientific research company. At the time the author was involved into the research, the company had already developed a working Mixed Integer Programming (MIP) solution for a real world

problem of a client from the automotive supply industry. Apparently, all requirements for the project described bellow were readily available (Stainko et al. [2]) and where subsequently simplified for the purpose of this Master's thesis.

1.2 Specific Requirements

The client company produces about 3 million units of various products per year. During the last years the number of product types has strongly increased, whereas the sizes of the batches that have to be delivered to the customers are decreasing. To satisfy the customer needs with this larger number of product types and smaller orders, an intelligent planning of the production is highly necessary. Currently, the client company supports about 300 product types and produces about 3000-3500 product units per one shift of 8 hours.

Roughly speaking, the production line can be split into two big steps, a casting part and a cnc (computerized numerical control) mechanical part. Since the cnc-mechanical part turns out to be the bottleneck of the production line, the aim is to model and optimize the production of this step with an individual production planning software.

A closer look at the cnc-production step reveals the following: There are 21 cnc-machines which work in parallel and can produce different types of products. Each machine has different production capacities which vary with respect to the produced product type. In order to work on a specific product type, the cnc-machines have to be prepared - a setting has to be performed, which can take up to four hours. During this period the machine cannot produce.

The customer demands are known for each of the next 5 days. The planning horizon consists of 5 working days and 3 shifts per day (8 hours each). Additionally, prospective demands are known as future demands and are given in accumulated numbers per week for the next week after the planning horizon. If the amount of the daily jobs is too little to fully load the machines, the free capacity is used to work on future demands, which is called the filling of the machines.

If jobs cannot be processed in time, backlog costs have to be considered for the missing amount. Since the company can store finished and packaged goods even outside, no really limiting storage costs and limitations have to be considered.

The original problem contains also several details, which are not discussed in this paper, but are mentioned in this paragraph. Some product items are very similar

and are grouped together in product families. This yields three different kinds of settings: initial, normal and family settings according to the time they require (i.e. sequence dependent setup times). Additionally, the setting of a machine also includes installation of a clamping device. There are two groups of clamping devices: product specific ones and universal ones (which contain subgroups with different sizes). The number of clamping devices for each group is limited.

1.3 Algorithmic Approaches

Finding an optimal solution to the DLSP is known to be NP-hard (Drexl [1]). Furthermore, if all demands have to be satisfied without any backlogs and, setup times or parallel machines have to be considered, the problem of finding a feasible solution becomes NP-complete. Exact methods for strong single-machine formulations experience difficulties for modest problem sizes and are very sensitive to the number of items (Hoesel [3]). Solving the parallel machine problem is even harder (Monma [4], Cheng [5]). Hence, this is a promising area for applying heuristics.

Well designed heuristic algorithms produce good quality (practically acceptable) solutions in reasonable time. In the case of combinatorial optimization for an NP-hard problem these solutions cannot be verified for optimality in polynomial time. However, some approaches based on mixed-integer programming formulations, which are discussed in Chapter 2, are capable of obtaining tight lower bounds for the optimal solution. Nevertheless, specialized heuristics are often intentionally designed to solve only specific aspects of particular problems or they become too restrictive once their parameters are fixed.

Metaheuristics offer a solution to these problems. A metaheuristic is a combination of diverse strategies for solving a general class of problems by guiding these strategies in order to make them more efficient and more robust. This Master's thesis proposes a Greedy Randomized Adaptive Search Procedure (GRASP) metaheuristic incorporated with a Path Relinking strategy to find good quality production plans for the described problem.

1.4 Solution Method

GRASP is a multi-start iterative strategy which combines a semi-greedy construction heuristic with a local search improvement heuristic. At each iteration first, an initial

solution is composed by a semi-greedy demand-driven construction procedure (one element at a time) and second, randomized local search applies iterative improvement to the initial solution. Adaptive memory is introduced by Path Relinking into the Basic GRASP procedure resulting in a very powerful enhanced metaheuristic. Additionally, a parallel implementation in Open-MP is provided for better runtime performance on modern multi-core systems. Finally, results are compared on the same real-world problem data sets against those obtained by an existing MIP approach and a Tabu Search implemented by Jimborean et al. [6].

The thesis has several contributions. First, concerning the mathematical model, some rather non-trivial parallel production and settings limitation per period constraints were specified by the company in order to reduce the search space. To the best of our knowledge, no similar modeling issues have been studied in literature. Second, the solution method is compared with several other approaches that were independently developed on the same problem configuration. Third, the concept of selection pressure is introduced for randomized local search and is successfully applied to prevent the heuristic from getting "stuck" earlier than expected. Future research may benefit from utilizing this concept at the meta-level. Finally, an intelligent path exploration strategy is used in Path Relinking to examine more thoroughly the generated paths.

In Chapter 2 part of the relevant literature is reviewed and techniques that are typically applied to DLSP are presented. Chapter 3 introduces the exact mathematical model and discusses solution representations. The solution method is described in Chapters 4, 5 and 6 by presenting Basic GRASP, GRASP with Path Relinking and the parallel implementation, respectively. Computational results and the comparison are discussed in Chapter 7. Finally, conclusions and ideas for future research are given.

Chapter 2

State of the Art

Typically, problem configurations with requirements similar with those presented in Section 1.2 are described with simultaneous lot sizing and scheduling models (see Allahverdia [7] for a recent survey) and turn out to be NP-hard (Jansa [8], Gicquel [9]). The problem at hand is described as a single-level multi-item multi-period discrete lot-sizing and scheduling problem on parallel machines and also turns out to be NP-hard (Salomon [10], Bruggeman [11], Webster et al. [12], Monma [4], Cheng [5], Drexl [1]). When confronted with an NP-hard combinatorial optimization problem one usually resorts to specific heuristics and guiding metaheuristics. The purpose of this paper is to solve a real world production planning problem from industry by applying a GRASP with Path Relinking metaheuristic approach.

2.1 Lot Sizing and Scheduling Models

Production planning and scheduling is one of the most challenging areas and represents the beating heart of any manufacturing process. Such a vivid problem has received huge interest during the years and different formulations considering different aspects and variations have been developed (Allahverdia [7], Gicquel [9], Shapiro et al. [13], Volman [14], Graves et al. [15], Trigeiro [16], Dzielinski [17]).

In this section we briefly describe some of the most popular problems in short- to medium-term scope and the elements they modify. For a more detailed practical overview on the integration between short-term and long-term decision models as well as a good distinction between planning models and scheduling models see Kreipl [18].

2.1.1 Infinite Planning Horizon

Research started on continuous time models with infinite planning horizon where demands occur continuously with a constant rate in time (stationary demand). Good representatives are the classical economic order quantity (EOQ) model and the economic lot scheduling problem (ELSP) (Drexl [1]). The EOQ model dose not introduce capacity constraints. Hence, there is no reason to distinguish different product types and it turns out to be a single-item problem. The problem can be solved optimally in polynomial time.

On the other hand, the ELSP introduces capacity restrictions to evolve into a multi-item problem (resources are shared by several items). It still considers stationary demand but solving it optimally appears to be NP-hard (Chatfield et al. [19]). As a general rule, as soon as we introduce capacities we run into hard to solve problems.

Similarly, the problem we described in the previous section also introduces capacity constraints to describe a heterogeneous environment. Parallel machines may have different production capacities for the same product types. However, it has a discrete and finite planning horizon and non-stationary demands. For a more recent research on infinite horizons see Huang [20].

2.1.2 Dynamic Demand

The requirement for a stationary demand turned out to be too restrictive for the industry. Due to practical concerns production planning models where urged to involve dynamic demand. As seen in practice, customer orders quantities may vary over time and due times may be well scattered over the planning horizon.

A typical example for a dynamic demand model is the Wagner-Within (WW) problem. It assumes a finite planning horizon that is divided into discrete periods with equal sizes. Dynamic demand is given per period and zero demands may also appear. However, the WW problem dose not involve capacity limits and turns out to be a single-item problem. Optimal solution algorithms do exist (Sung et al. [21]).

On the other hand, a finite horizon means that demands can be examined only partially in a certain time interval. Future demand addresses this issue by providing information about customer demand beyond the finite planning horizon. As observed by Graves et al. [15], information about future customer requests is usually known only partially and is often a subject to forecasting. Therefore, future demand should only play an inferior role for the result of the optimization.

2.1.3 Capacity Constraints

Introducing the capacity constraints to the WW problem yields one of the most important and difficult problems in tactical production planning (Karimi [22]). The capacitated lot sizing and scheduling problem (CLSP) is a multi-item medium-scale problem with discrete finite planning horizon. It is one of the most intensively studied problems in the area of production planning and scheduling (Trigeiro [16], Graves [23], Shapiro et al. [13], Vollman [14], Allahverdia [7], Nasimento [24]). Similarly, a more advanced scenario is covered by the general lot sizing and scheduling problem (GLSP) (Drexl [1]).

The CLSP may incorporate different kinds of medium-scale problem characteristics such as: multiple stages, production costs and inventory costs, unmet demand and backorder, tardiness costs (Kreipl [18]) and transportation costs (Nasimento [24]). Since multiple items can be produced in a single time period, it is often referred to as a large bucket model. Hence, once a production lot is determined the production runs inside the "large bucket" lot should be further sequenced or scheduled (Graves et al. [15]).

2.1.4 Small Bucket Models

Small bucket models are short-term detailed models with discrete finite planning horizon that allow production of only one item per period (Eppen [25], Salomon [10]). Periods correspond to small time slots as shifts or hours. Typically, the purpose of these models is to make a more detailed schedule for the "large bucket" lots obtained by CLSP. Nevertheless, small bucket approaches can be applied independently.

Several variations exist: discrete lot sizing and scheduling problem (DLSP), continuous setup lot sizing problem (CSLP), proportional lot sizing and scheduling problem (PLSP) (Drexl [1]). All these approaches share one fundamental characteristic in common. Namely, if a machine produces a certain item in a period, production uses the full capacity of the machine. (Production of a single item may last for several periods.)

Finding an optimal solution for the simplest form: the DLSP is NP-hard. Furthermore, if setup times or parallel machines are considered even construction of a feasible solution is shown to be NP-complete (Salomon [10], Bruggeman [11], Webster et al. [12], Drexl [1]).

Apparently, the DLSP is a very suitable model for the original problem. However,

a setting itself does not take all the time of a period in our case and production may also happen in the same period when there was a setting (Stadelmayer et al. [26]). This results into a variation of the DLSP, since two events are modeled in one time period.

2.1.5 Backlogs

As described in Graves et al. [15] for some problems the available resources might not be sufficient to meet all demands. In this case, the optimization process should decide also which demands to meet and to what extent. Also, a backlog cost should be introduced for unmet demands and further optimized by the model. Note, that backlogs are already incorporated to the initial problem specification. Further variations include backorder, totally or partially lost demand and even late shipping.

2.1.6 Sequence Dependent Setups

Modeling family settings (as described in the original client's problem) additionally adds complexity to the model by introducing a sequence dependency of setups (Kreipl [18]). The setup time is supposed to be less if the machine have already been set for another product from the same family. This creates a sequence dependency for both setup costs and setup times (see Zhu [27] and Allahverdia [7] for a more fine-grained classification). This paper dose not consider sequence dependent setups.

2.1.7 Multi-level Models

All approaches discussed so far were single level, where the items to be produced are independent. In more complex manufacturing processes more advanced multi-level models are needed, where production of an item may cause a dependent demand of its component items (Bahl [28]). These problems are not a subject of this thesis (see Drexl [1], Sahling [29] and Gicquel [9] for a more detailed review).

2.2 Available Techniques

A huge variety of approaches for solving production planning and scheduling problems have been published in the literature for the last 50 years. Early techniques can be observed in the following areas: mathematical programming, dispatching rules, expert

systems, neural networks, genetic algorithms, and inductive learning (Jones [30]). More recently neighborhood search methods, specialized heuristics and metaheuristics have shown their capability of obtaining high quality results in reasonable time for significantly larger instances (Karimi [22], Jans [8], Gicquel [9]).

In this section we will outline only some of the available techniques for single-level multi-item lot sizing and scheduling problems with discrete and finite planning horizon. Later on, we will discuss approaches applied to small-bucket models like DLSP and CSLP (for multiple resources), either with sequence dependent setups or without. Finally, we will focus on GRASP applications to problems similar to the original problem of this paper.

2.2.1 Mathematical Programming

Most of the planning and scheduling problems have been formulated using integer programming, mixed-integer programming and dynamic programming. While using a computationally exhaustive solution method might turn out to be inevitable, it is usually worth it spending a lot of effort into modeling the problem. Several researchers have successfully reformulated known production planning problems into more promising models (Eppen [25], Salomon [31], Jans et al. [32]). Also, various model variations are based on the concept of echelon stock (Gicquel [9]).

Similarly, in the past, others have also proposed hierarchical decomposition strategies of mathematical programming problems (Jones [30]). More recently, a novel approach based on decomposition of products into attributes in a particular lot sizing model have been studied by Gicquel [33].

Apart from modeling, popular enumerative solution techniques based on mixed integer programming are branch-and-bound, branch-and-cut, branch-and-price heuristics (Fleischmann et al. [34], Belvaux [35], Jordan [36], Balakrishnan [37], Absi [38], Klosea [39]). These are often applied in conjunction with Lagrangean Relaxation (Diaby [40], Graves [15]) and Column Generation (Cattrysse [41], Jans [42], Hiroaki [43]).

Branching involves a representation of the problem as a decision tree where decision points correspond to partial solutions. Branching possibilities are usually determined by integer constrained variables. The process continues until no further branching is possible. Leaf nodes in the decision tree are solutions to the scheduling problem. During this procedure certain solutions can be evaluated and certain branches of the decision tree can be pruned according to the evaluations.

Lagrangean relaxation is a technique based on removing certain constraints from

the mixed-integer formulation: integer-valued constraints in the most general case (Shapiro et al. [13], Tang [44]) and inventory balance constraints and resource capacity constraints in the case of production scheduling (Gicquel [9]). The method further adds the corresponding costs (for the relaxations) to the objective function. Due to those relaxations, the problem can be decomposed to several simpler problems (usually WW problems). Eventually, this technique allows for obtaining good lower and upper bounds for the cost of the optimal solution (Sural [45]). This process can be further combined with constrained programming aspects.

The main advantage of these approaches is that they provide information about the cost of the optimal solution. This capability of assessing the quality of the found solution is already a great advantage in the area of NP-hard problems. However, these techniques are computationally very expensive for large instances of lot sizing and scheduling problems (Gicquel [9], Jansa [8], Billington [46], Jones [30]).

Several hybrid variations of these techniques with more advanced specific heuristic methods have been proposed in the area of production planning. There is an increasing ongoing research in this direction (Caserta [47], Akartunali [48], Sahling [29], Aghezzaf et al. [49]).

2.2.2 Heuristics

Alternative approaches like specialized heuristics and guiding meta-heuristics are becoming more popular in the area. The main reason for their attractiveness is the flexibility and ability to handle very large instances of NP-hard problems in reasonable time. Appropriate heuristics give good solution quality results with significantly better execution time in comparison to the mixed integer programming approaches (Jansa [8]).

However, a major disadvantage of these approaches is that they do not provide any possibilities of assessing the quality of the obtained solution. Other methods have to be used in order to find information about the optimal solution. Furthermore, a simple heuristic is typically not sufficient in order to obtain nearly optimal results. Typically, a metaheuristic, which operates on one or more simple heuristics, has to be developed in order to guide the search process into more promising regions of the search space.

Neighborhood search or local search is a simple heuristic that starts from an initial solution and iteratively generates a series of improving solutions. At each iteration a local neighborhood is constructed from the current solution by small modifications

("perturbations"). Then, the generated neighborhood is searched for an improving solution. The procedure ends when no further improvement is possible.

Local search is guaranteed to obtain a local optimum if the small modification neighborhood operators are designed to perform only local changes. The underlying principle of this method is the proximate optimality principle (POP) (Glover [50]). POP says that "good solutions at one level are likely to be found `close to´good solutions at an adjacent level". The principal can be interpreted as a tabu search mechanism where certain level refers to a particular local optima basin of attraction. Neighborhood search heuristics converge to a local optimum which is very unlikely to coincide with the global optimum for hard optimization problems like production lot sizing and scheduling. They heavily depend on the initial solution.

Construction heuristics use specialized and problem dependent techniques to obtain good feasible solutions. They provide a good starting point for other advanced heuristics. Note that for some lot sizing and scheduling problems, already the problem of constructing a feasible schedule is reported to be NP-complete (Salomon [10], Bruggeman [11], Webster et al. [12], Drexl [1]). However, dedicated construction heuristics have been observed in literature (Gicquel [9], Salomon [31], Ozdamar [51], Minner et al. [52]).

2.2.3 Metaheuristics

Several strategies have been developed that try to escape from local optima and guide the search process towards better opportunities. Those techniques usually follow some basic principles and are applied to simpler heuristics. Popular metaheuristics that have been applied to scheduling problems are tabu search (TS), simulated annealing (SA) and genetic algorithms (GA).

Tabu search is a trajectory based approach that starts from only one solution and progressively generates a sequence of solutions (Glover et al. [53], [54]). In contrast to local search, some moves are set tabu (i.e. they are forbidden), because either they lead the search to a local optimum, or they lead to cycles (revisiting old parts of the search trajectory). The result is that the search trajectory will try to explore different regions of the search space from the local search area. Tabu search techniques have been successfully applied in the area of lot sizing and scheduling (Glover et al. [55], Carvalho [56], Pereira [57], Ozdamar [51], Bushera [58]).

Population based approaches like SA and GA have also attracted many researchers. Several applications to production planning problems have been developed (Kirk-

patrick [59], Kuik [60], Kimms [61], Jeffcoat [62], Ferreira [63], Gicquel [9], Jansa [8], Meyr et al. [64], Ozdamar [51], Torabi [65]). However, these metaheuristics are rather computationally expensive and their application to scheduling problems is very unlikely to outperform mixed integer programming approaches.

2.2.4 Techniques for Small-bucket Models

This paper deals with a problem that is described with the Discrete Lotsizing and Scheduling Model. Papers that discuss similar small-bucket models are: Salomon [10, 31], Gicquel [33,66], Drexl [67], Belvaux [35], Jans et al. [32], Paula [68], Tempelmeier [69]. Most of these papers use problem reformulation, heuristics and mixed integer programming approaches. Hence, metaheuristics are a promising alternative in this area and research toward choosing a better metaheuristic approach is well motivated.

Next, some concrete results from previous work on techniques for small-bucket models and parallel machines are shortly listed:

Belvaux and Walsey [35] designed a mixed integer programming based XPRESS-MP branch-and-cut custom solver especially devoted to discrete-time lotsizing problems. They made successful test runs for several small-bucket instances, as well as for big-bucket and multi-level problems. In particular, the small-bucket models that were evaluated allow either only one item per period (group SB-1P), or two items per period (group SB-2P). Group SB-1P contains instances with the following sizes: 1 machine, 3 products and 60 periods; 1 machine, 5 products and 15 periods; 2 machines, 5 products and 12 periods; and, 2 machines, 5 products and 24 periods. Results on the first tree types of problems from group SB-1P were optimal and were obtained for a runtime of 5, 14 and 268 seconds, respectively. The last type from group SB-1P was executed for 2 hours and gave a solution with a 2.3% gap from the obtained lower bound. Group SB-2P contains instances of problem sizes with 3 to 12 machines, 3 to 47 products and 5 to 20 periods. And, for the big instances from group SB-2P the custom solver finds similar solutions with small gaps after 2 hours of execution time.

A Column Generation approach was used by Jans and Degraeve [42] to solve large instances of a variation of a DLSP for an international tire manufacturer. Lagrangean Relaxation is applied to the master problem and the resulting subproblems are solved by a Dynamic Programming (DP) recursion. They consider problem instances with 3 machines (heaters); 10, 20 and 30 items and 10, 20 and 30 periods. All instances are solved almost optimally allowing only gaps under 1%. Results are compared among

five different variations of the heuristic and the authors conclude that incorporating Lagrangean Relaxation results in a very effective algorithm. Additionally, they find highly capacitated problem instances to be harder to solve but also experience a reduction of the search space of DP for these instances. Other papers concerning Column Generation for DLSP include an earlier research by Cattrysse [41] (without backlogging) and an article by Hiroaki [43] (on identical machines).

An early research by Kimms et al. [61] suggests a GA for a similar small-bucket multi-level Proportional Lotsizing and Scheduling Problem (PLSP) without backlogs that competes with a TS approach proposed at this time. The designed algorithm is applied to a huge variety of small instances with 1 or 2 parallel machines, 5 items and 10 time periods. Run-time is rather low (0.10 CPU-seconds) but the average gap to the optimal solution is 19.89%. The GA is compared with an arc based TS and dominates in runtime performance, however average deviation is rather poor. Nevertheless, considered problem sizes are rather small.

A Tabu Search for a multi-level PLSP with higher dimensionality was implemented by Brockmann [70] on a massive parallel system of 192 Pentium II processors (PVM). The algorithm was tested on problem instances of 8 to 12 machines, 80 to 120 items, 20 to 40 periods and 3 to 5 stages. Significant improvements are reported within a reasonable amount of time (10 to 15 minutes). However, in some cases the quality of the found solution was unsatisfactory due to unappropriated choice of an initial solution. Another more recent article by Carvalho [56] discusses a Tabu Search implementation for large instances of DLSP. TS is reported to obtain much better solutions after 1 hour execution time than solutions obtained by a MIP approach after 5 hours of execution. Another article from the same authors (Pereira [57]) compares random start local search and TS with branch-and-bound approaches by extensive experiments.

Typically, techniques that guarantee optimal or nearly optimal solutions are applied in literature when small problem instances are considered. However, the DLSP problem discussed in this paper consists of 21 parallel machines, 272 items and 15 time periods. Hence, rather large problem instances have to be solved. One possibility is to reduce the search space by more constrained production but as observed by Jans [42], this will make the problem even harder to solve. Another possibility is to use more computational power but this may not always provide satisfactory results. Therefore, investigation into more powerful metaheuristic techniques that can be easily parallelized is desirable.

2.2.5 GRASP Metaheuristic

Greedy randomized adaptive search procedure (GRASP) is a trajectory based metaheuristic approach (Feo [71]) for solving combinatorial optimization problems. GRASP is a multistart iterative procedure that operates on a greedy randomized construction heuristic and a local search improvement heuristic. There are many advanced techniques concerning adaptive memory that can be applied to GRASP (Resende et al. [72]). The main concept of GRASP is to explore different regions of the search space by using a randomized construction procedure.

This paper uses GRASP in order to solve a variation of the DLSP without sequence dependent setups. The metaheuristic have been successfully used for other types of scheduling problems: Feo [73], Nasimento [24], Binato [74], Rocha [75], Rojanasoonthon [76]. However, GRASP hasn't been applied before to a small-bucket problem like DLSP. Furthermore, results are compared to those obtained by a Mixed Integer Programming and a Tabu Search approach (Jimborean et al. [6]). No similar comparison exists in literature.

Chapter 3

Problem Statement

This paper solves a real world production planning problem of a customer from the automotive supply chain industry. The customer's requirements were already presented in the previous chapter 1.2. Furthermore, the problem was classified in the realm of short-term decision lot sizing and scheduling problems. Specifically, it was identified as a variation of DLSP on parallel machines.

In this chapter the exact mathematical modeling is based on the requirements from Section 1.2. In the first section an exact mixed integer programming formulation is presented in details. First, a compact visual solution representation is given to explain basic components of the model. Second, the problem's input parameters are specified. Third, examples are provided to show important aspects of the optimization problem. Then, the objective function and the constraints of the problem are presented. Finally, a brief discussion of solution representation alternatives is given as a transition to the next chapter.

3.1 Mathematical Formulation

3.1.1 Visual Solution Representation

The manufacturing workplace is organized in a single plant with a set M of heterogeneous and configurable parallel machines. The planning horizon consists of a set $T = T' / \{t_0\}$ of time periods, where t_0 is an initial period indexed with 0. See Fig. 3.1 for a simpler illustrative example with 6 parallel machines and 15 time periods - 5 days with 3 shifts per day.

Machines can produce different types of products called items. Each machine can

M \ T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2		16					41				16				12
2	37							16								
3	36	34									50					
4	12						41				34					
5	50	34														
6	5		14									18				

Figure 3.1 A visual solution representation for 6 parallel machines and 15 time periods.

produce one item at a time period from a subset of all possible items J. Furthermore, production capacity cap_{mj} specifies a number of units of item $j \in J$ that machine $m \in M$ is able to produce in one time period. It is possible that different machines have different production capacities for producing the same item. For example, if machine 6 needs 9 time periods to produce 981 units of item 14, machine 1 would require at least 16 time periods to do the same job (it will not be possible in the given planning horizon).

In order to produce a specific item $j \in J$ starting from time period $t \in T$, machine $m \in M$ has to be setup. A setting has to be performed (i.e. $x_{mjt} = 1$) in the beginning of time period t. This may take up to 4 hours. During this time the machine cannot produce. After the setup process is finished, the machine will produce item j for a number of following periods $t' \in T, t' > t$ (when no setup is done: $\forall_{i \in J}\ x_{mit'} = 0$), until a new setting is performed on m or until the end of the planning horizon. In Fig. 3.1 the settings are specified with their item in the time periods when they appear.

When performing a setting in a time period t the setting takes only a certain ratio v_{mj} (3/8 is the default ratio) of the available time in the period (see Fig. 3.2). Hence, two events are modeled in a single time period: first, the setting may occur only in the beginning of a period and second, production may happen immediately after the setting in the same time period. This results into a variation of the DLSP, since both setting and production are possible in a period.

Production is specified by the *state variables* $y_{mjt} \in \{0, 1\}$, which describe the state of machine $m \in M$ for item $j \in J$ in time period $t \in T$ as either "producing" or "not producing". Once a machine m is set for producing item j in time period t (i.e. $x_{mjt} = 1$) the corresponding state variable y_{mjt} is automatically set to "producing" (i.e. $y_{mjt} = 1$). Furthermore, production of item j will continue in the following time periods $t' \in T, t' > t$ (i.e. $y_{mjt'} = 1$), until a new setting is performed on machine m or until the end of the planning horizon. Dependency between setup and state

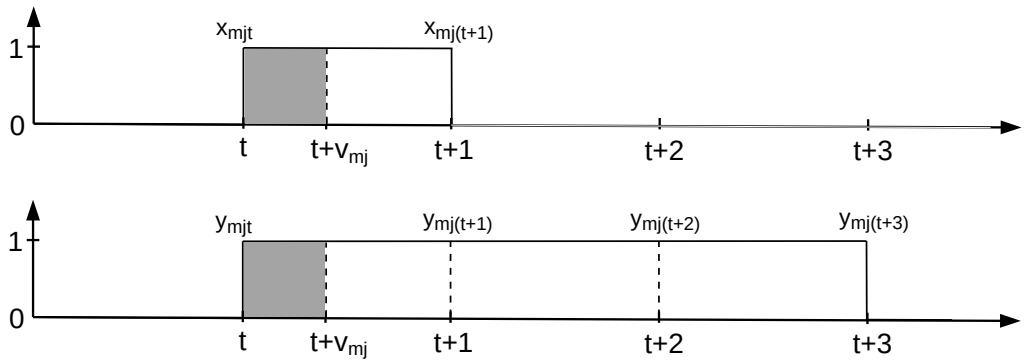


Figure 3.2 A setting for a new item j takes a certain ratio v_{mj} (3/8 is the default ratio) of the available time in a period t. In the remaining time of the period $[t + v_{mj}, t + 1)$ the machine produces the new item. Production may continue in the following periods $t + 1, t + 2, \ldots$.

variables on machine $m \in M$ for an item $j \in J$ is shown in Fig. 3.2.

This assumption comes from the DLSP principle that a machine should work to its full capacity (Drexl [1]). It leads to the natural formation of *production lots*. A *production lot* on machine $m \in M$ for item $j \in J$ consists of the longest sequence of consecutive time periods $\Delta \subseteq T$, for which the state of the machine for item j is set to "producing" (i.e. $\forall_{t \in \Delta}\ y_{mjt} = 1$). All production lots in a schedule cover the whole planning horizon for all machines. The visual representation on Fig. 3.1 identifies only the *production lots*; the *setting variables* x_{mjt} and the *state variables* y_{mjt} are implicitly specified by the lots.

3.1.2 Input Parameters

The input parameters of the problem are given in Table 3.1. The 0-th time period is used to specify initial inventory I_{j0} of the items $j \in J$ available in the storage and "initial setup" state variables y_{mj0} of all machines $m \in M$ in the beginning of the plan.

Input Parameters:

M	set of machines (21 machines of only one type are considered);
J	set of items (product types);
T	set of periods - describes the planning horizon (5 days $\cdot$ 3 shifts = 15 periods);

cap_{mj}	capacity matrix - represents the number of units of item $j \in J$ that machine $m \in M$ can produce in one time period;
$prod_{mj}$	$\in \{0, 1\}$ - indicates whether machine $m \in M$ can produce item $j \in J$ ($prod_{mj} = 1 \Leftrightarrow cap_{mj} > 0$);
I_{j0}	initial inventory (storage) of item $j \in J$ in number of units;
y_{mj0}	initial state of machine $m \in M$ for producing item $j \in J$ at the beginning of the plan ($\in \{0, 1\}$);
d_{jt}	external demand in number of units of item $j \in J$ for time period $t \in T$;
f_j	future demand in number of units of item $j \in J$ for the next week;
s_{mj}	setting cost for setting machine $m \in M$ to produce item $j \in J$ (default value: 1.0);
c_{jt}	backlog cost of not producing one unit of item $j \in J$ from the external demand d_{jt} for time period $t \in T$ (default value: 0.1);
v_{mj}	ratio of setting time per working time in a period (default value: 3/8);
b_j	buffer parameter describing how much overrun is allowed for item $j \in J$ on a single machine;
p_j	parallel factor of item $j \in J$ indicating on how many machines parallel production of j is possible in the same time period (default value: 3);
σ	maximum number of settings per period (default value: 6);

Table 3.1 Input Parameters of the Problem

Production planning aims at meeting all customer requests on time. Customer requests are known as (external) demands: d_{jt}. They are given as numbers of units of items $j \in J$ at the end of each day (every third period $t \in T$). Demands for all other time periods are 0. Additionally, future demands f_j for the following week are given as units of items $j \in J$. On one hand, Late delivery is not allowed. Thus, if an external demand d_{jt} cannot be met on time, a backlog cost c_{jt} has to be considered for each unit of item j which was not produced from the demand. On the other hand, If working for the external demands is not sufficient to fully load the machines, the remaining capacity is used to work for future demands. This is called *filling* of the machines.

The parallel factor p_j for item $j \in J$ specifies maximum parallel production of item j. This parameter has an effect of reducing the search space of the optimization problem significantly. However, it also has the side effect of making the optimization

landscape more narrow and also, making it harder to obtain a feasible solution. A default value of 3 for all items turns out to be practically reasonable for the purpose of the optimization and was agreed with the customer.

Finally, the buffer parameter b_j controls the production overrun of item $j \in J$ on a single machine. If production for item j exceeds the buffer quantity $p_j \cdot b_j$ at the end this results in a redundancy of a certain amount of units. Typically, similar situations are referred to as hyper production. Buffer parameters provide the mathematical formulation with a mechanism for controlling hyper production - an important issue when machines are working at their full capacity.

3.1.3 Decision Variables

Decision variables of the optimization problem are listed in Tables 3.2 and 3.3. They play an important role in the optimization since their values can be varied in order to find better production plans. The primary setting variables x_{mjt} and state variables y_{mjt} were already explained in the previous section 3.1.1. All other secondary decision variables can be calculated according to the constraints from a solution based on x or y. It is sufficient to consider only one of them, because x_{mjt} and y_{mjt} are dependent on each other as shown in Fig. 3.2.

Primary Decision variables:

x_{mjt}	$\in \{0,1\}$ - indicates whether there is a setting performed on machine $m \in M$ in order to produce item $j \in J$ in time period $t \in T$;
y_{mjt}	$\in \{0,1\}$ - indicates the state of machine m for producing item j in time period t (as either "producing" or "not producing" item j);

Table 3.2 Primary Decision Variables of the Problem

Secondary decision variables are the production quantity q_{mjt} in units, the unmet external demand u_{jt} and the available inventory I_{jt} in each time period (see Table 3.3). The new inventory can be easily calculated with Formula (3.1) by accumulating the production in the current time period to the old inventory and meeting as much as it is possible from the external demands. The upper bound for unmet future demand r_j and the slack variable for maximum inventory violation k_j are parts of the mechanism for controlling hyper production, which is explained in section 3.1.6.

Secondary Decision Variables:

q_{mjt}	number of units of item j produced on machine m in period t;
u_{jt}	unmet external demand of item j in time period t, in number of units;
r_j	upper bound for unmet future demand of item j at the end, in number of units;
k_j	slack variable for maximum inventory violation of item j at the end, in number of units;
I_{jt}	inventory[1] of item j after time period t, defined as: $$I_{jt} = I_{j(t-1)} + \sum_{m \in M} q_{mjt} - (d_{jt} - u_{jt}), \quad \forall j \in J, \quad \forall t \in T. \tag{3.1}$$

Table 3.3 Secondary Decision Variables of the Problem

3.1.4 Examples

Let's consider two more examples with respect to the first example in Figure 3.1. The purpose is to evaluate the quality of the presented production plans and to identify a stable criteria for measuring how good a schedule is. Note, that two of the main goals of production planning are:

- to increase effectiveness of the manufacturing process and
- to optimize efficiency of resources in the workplace.

Effectiveness can be interpreted as how well are external demands fulfilled in the schedule. Efficiency refers to good utilization and workload of the machines. If the plan contains too many settings this means that machines are spending too much time in preparation for production, rather than using this time for actual production. This is especially emphasized in the model by introducing an additional parameter σ for maximum number of settings per shift (see Table 3.1). Smaller values of parameter σ reduce the search space, but also influence to what extent large external demands can be satisfied and in general, how hard the problem is.

[1]The inventory variable is not a primary decision variable, since it is actually represented with a formula of parameters and other decision variables.

M \ T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2		16					41				16				
2	37						16								12	
3	36	34														
4	12							41			50					
5	50	34														
6	5		14								18					

Figure 3.3 A first improvement example production plan with fewer settings.

M \ T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2		16													12
2	37							12					16			
3	36	34														
4	12				41						50					
5	50	34														
6	5		14								18					

Figure 3.4 A second improvement example production plan with less unmet demands.

For example, the plan presented in Fig. 3.3 has one setting less than the initial example in Fig. 3.1. In the initial example production of item 34 on machine 3 stops at the end of time period 9 and continues with a new production lot on machine 4 starting in time period 10. While in the new plan, the production lots on machines 3 and 4 starting in time period 10 are interchanged, allowing production of item 34 on machine 3 to continue until the end. This simple modification allows to not only decrease the total number of settings in the plan but also to produce 35 more units of item 34, which decreases the unmet demand in time period 12.

Overall, the observation is that the new plan introduces a more balanced way of handling demands. Furthermore, this observation is proved by the fact that the first improvement example in Fig.3.3 produces 193 units of item 12 more for unmet external demands in the last two days. Similarly, the plan produces 102 units of item 16 and 84 units of item 18 more for unmet demands. However, it also introduces some new unmet demands for items 37 and 41 (of 69 and 93 units respectively).

The new plan is not optimal and further modifications may be performed in order to improve its quality. The second improvement example in Fig. 3.4 has even less settings and less unmet demands than the first improvement example. This is achieved by appending the lot on machine 1 for item 41 starting at time period 7 to the front of the lot on machine 4 starting at time period 7. Also some small modifications have

been performed on machine 2. Unmet demands are reduced significantly for items 16, 37 and 41.

Unmet demands and total number of settings are good quality measure properties for assessing effectiveness and efficiency of a production plan. These two factors have a strong correlation between them and they should be interpreted rather together than separately. However, we did not discuss in this section some less quality influential properties of a plan like unmet future demands and maximum inventory violations. Nevertheless, these properties are rather important for both the mathematical formulation of the problem and the optimization procedure.

3.1.5 Objective Function

The objective function is a universal way of defining a solution quality measure for an optimization problem. It makes it possible to compare solutions according to their properties. Equation (3.2) defines the objective function of the minimization problem, which is based on the observations from the previous section 3.1.4.

Objective function:

$$\min \quad \omega_s \cdot \sum_{m \in M} \sum_{j \in J} \sum_{t \in T} s_{mj} \cdot x_{mjt} + \omega_u \cdot \sum_{j \in J} \sum_{t \in T} c_{jt} \cdot u_{jt} + \omega_f \cdot \sum_{j \in J} r_j + \omega_k \cdot \sum_{j \in J} k_j, \quad (3.2)$$

where ω_s, ω_u, ω_f and ω_k are weight coefficients (default values are 1, 1, $\frac{1}{\sum_{j \in J} f_j}$ and 1, respectively).

The objective function (3.2) aims at minimal cost of settings, backlog cost and inventory violation cost and maximal production for future demand by minimizing the lower limits of unmet future demand cost, subject to the following constraints.

3.1.6 Constraints

Most of the constraints were already explained in this chapter. For example, equation (3.3) defines the production quantity produced by a single machine in a period either with or without a setting, discussed in Fig. 3.2. In section 3.1.1 it was stated that a machine can produce one item at a time period - this is now formally defined by Equation (3.5). Constraints (3.6) and (3.7) refer to the correlation between setting variables and state variables and were explained at the end of section 3.1.1. The

number of settings per period is bounded by the first non-trivial constraint (3.8) and was mentioned in the beginning of section 3.1.4.

Constraints:

$$q_{mjt} = cap_{mj} \cdot (y_{mjt} - v_{mj} \cdot x_{mjt}), \quad \forall m \in M, \quad \forall j \in J, \quad \forall t \in T \tag{3.3}$$

$$y_{mjt} \leq prod_{mj}, \quad \forall m \in M, \quad \forall j \in J, \quad \forall t \in T \tag{3.4}$$

$$\sum_{j \in J} y_{mjt} = 1, \quad \forall m \in M, \quad \forall t \in T \tag{3.5}$$

$$x_{mjt} \leq y_{mjt}, \quad \forall m \in M, \quad \forall j \in J, \quad \forall t \in T \tag{3.6}$$

$$x_{mjt} \geq y_{mjt} - y_{mj(t-1)}, \quad \forall m \in M, \quad \forall j \in J, \quad \forall t \in T \tag{3.7}$$

$$\sum_{m \in M} \sum_{j \in J} x_{mjt} \leq \sigma, \quad \forall t \in T \tag{3.8}$$

$$\sum_{m \in M} y_{mjt} \leq p_j, \quad \forall j \in J, \quad \forall t \in T \tag{3.9}$$

$$r_j \geq f_j - I_{jT},^{2} \quad \forall j \in J \tag{3.10}$$

$$I_{jT} \leq \max\{f_j + p_j \cdot b_j, I_{j0}\} + k_j, \quad \forall j \in J \tag{3.11}$$

$$q_{mjt} \geq 0, I_{jt} \geq 0, r_j \geq 0, k_j \geq 0, \quad \forall m \in M, \quad \forall j \in J, \quad \forall t \in T \tag{3.12}$$

$$x_{mjt}, y_{mjt} \in \{0, 1\}, \quad \forall m \in M, \quad \forall j \in J, \quad \forall t \in T \tag{3.13}$$

$$0 \leq u_{jt} \leq d_{jt}, \quad \forall j \in J, \quad \forall t \in T \tag{3.14}$$

Production matrix $prod_{mj}$ (see Table 3.1) defines the capability of a machine $m \in M$ to produce item $j \in J$ (see Equation (3.4)) and has a value of 1 when the capacity cap_{mj} is non-zero. The second non-trivial constraint (3.9) limits parallel production of an item $j \in J$ in any time period by the corresponding parallel factor parameter p_j.

According to the inequality (3.10), the decision variables r_j (see Table 3.3) represent a lower bound for unmet future demands of an item $j \in J$. By minimizing these bounds in the objective function (3.2) it is guaranteed that the future demand will be fulfilled as much as possible. Since r_j is non-negative (see Constraints (3.12)) the additional sum in the objective will not be negative. But the right-hand-side of (3.10) still can get smaller than 0. So, we have to provide some reasonable bounds for I_{jT} (like in (3.11)) which are restrictive, but never make the resulting optimization problem infeasible.

[2] I_{jT} denotes the inventory in the last time period.

In constraint (3.11), the buffer parameters b_j describe how much overrun is allowed for item $j \in J$ on a single machine. The parallel factor p_j incorporates the case of parallel production on multiple machines. Additionally, the constraint is softened by the slack variables k_j. These slack variables are bounded by zero in (3.12), and are penalized by an additional term in the objective (3.2). Originally, this limit was referred to as a hyper production prevention mechanism at the end of section 3.1.2.

Finally, non-negativity of production quantity q_{mjt} and inventory I_{jt} are guaranteed in (3.12). Setting variables x_{mjt} and state variables y_{mjt} are declared binary in (3.13). And constraint (3.14) limits the unmet demand u_{jt} between 0 and the actual demand d_{jt}.

3.2 Solution Representation

The mixed integer formulation presented in the previous section is a formal definition of the problem. However, a plan representation of binary state variables is not suitable for a good memory efficient solution method. A more compact representation is needed which still contains all the relevant information of a schedule.

As stated in section 3.1.3, the setting variables or the state variables contain all the relevant information of a plan. Furthermore, they are correlated in a special manner, according to the DLSP principle for full capacity production (see section 3.1.1). Therefore, it is sufficient to use only one of them for the representation.

3.2.1 Using State Variables

State variables implicitly specify what item is produced in each time period and for every machine. Little effort is needed to determine if there is a setting at a particular time period (by looking at the state variable for the previous time period on the same machine). However, instead of using the binary state variables y_{mjt} it is more reasonable to use integral state variables:

$$y'_{mt} = j \Leftrightarrow y_{mjt} = 1, \qquad \forall m \in M, \quad \forall t \in T. \tag{3.15}$$

Correctness of this definition and uniqueness of item j can be verified by constraints (3.5) and (3.12). Further, the integral variables can be stored either in a matrix, or in a collection indexed by the number of the machine or even by the item.

3.2.2 Using Setting Variables

While production state of a machine can be determined directly by the state variables, it requires some more effort using the setting variables. If production has to be determined on machine $m \in M$ in time period $t \in T$, then the last non-zero setting variable $x_{mjt'} = 1, t' \leq t$ has to be found and the item produced is j. Similarly, integral setting variables can be defined:

$$x'_{mt} = \arg_j \max_{t' \leq t} \{t' \in T \mid x_{mjt'} = 1, j \in J\}, \qquad \forall m \in M, \quad \forall t \in T. \tag{3.16}$$

Correctness of this definition and uniqueness of item j can be verified by constraints (3.5) and (3.12).

This is a more compact definition since settings can be stored only for the time periods when they appear and not for all $t \in T$. Settings are directly accessible and they uniquely identify beginning of production lots. Integral setting variables can be stored in collections indexed either by the machine numbers or by the items (for direct access to similar lots). Total production can be computed lot by lot instead of period by period.

3.2.3 Other

As it was mentioned in section 2.2.1 mathematical reformulations may lead to more promising models. Several attempts were made in order to find a more compact (based on production lots) or more familiar (graph based) representation. However, all of them were unsuccessful, because of the high dimensionality of the problem. Nevertheless, such redefinitions of the DLSP are known in literature. Hoesel [3] developed a shortest-path formulation of the multi-item DLSP on a single machine resulting in a linear integer programming formulation. Other examples are a graph representation of the problem by Salomon [31] and a decomposition of products into attributes by Gicquel [33].

Chapter 4

Basic GRASP

The optimization problem presented in the previous Chapter 3 is a variation of the DLSP defined in Drexl [1]. No polynomial time algorithms for solving it exist in the literature. Furthermore, the problem turns out to be NP-hard and even construction of a feasible solution without unmet demands is NP-complete (Salomon [10], Bruggeman [11], Webster et al. [12], Drexl [1]).

Heuristics are not guaranteed to find an optimal solution but they can generate practically acceptable, good quality solutions in polynomial time. However, designing a good heuristic algorithm that would perform well for various different problem instances is not always possible. Specialized heuristics usually guide the optimization process only into certain areas of the search space. Metaheuristics operate on simpler heuristics for solving a general class of problems in order to make them more efficient and more robust.

Greedy randomized adaptive search procedure (GRASP) is a metaheuristic for solving combinatorial optimization problems (Pitsoulis [77]). It was first introduced by Feo [78] in 1989. This master thesis uses GRASP as a solution method because of its simple structure, fewer parameters, randomization capabilities and its complementary combination of specialized heuristics. Furthermore, advanced techniques based on adaptive memory and parallel implementations are easy to incorporate.

4.1 GRASP Overview

GRASP is a multi-start iterative procedure that executes a separate trajectory based optimization process in each iteration. Basically, there are two phases of this process: construction and improvement. A trajectory starts at an initial solution generated

by a construction heuristic. Consequently, this solution is modified by a local search heuristic to generate series of improving solutions. The basic structure of GRASP for a minimization problem is shown in Algorithm 4.1. Two termination criteria are illustrated: one is based on a maximum number of iterations max_iter and the other is based on a maximum number of non-improving iterations max_term_iter (a logical *or* is used between them).

Algorithm 4.1 Basic Structure of GRASP

Input: max_iter - maximum number of iterations
max_term_iter - maximum number of non-improving iterations
Output: the best solution found X^* and its objective value f^*

1: $f^* = \infty$;
2: $term\_iter = 0$; // number of non-improving iterations
3: Initialize pseudo-random number generator $rand$
4: **for** $i = 1 \ldots max\_iter$ **and** $term\_iter < max\_term\_iter$ **do**
5: $X_s = GreedyRandomizedConstruction(rand)$; // Phase 1: Construction
6: $X_l = LocalSearch(X_s, rand)$; // Phase 2: Improvement
7: **if** $f(X_l) < f^*$ **then**
8: $f^* = f(X_l)$; $X^* = X_l$;
9: $term\_iter = 0$;
10: **else**
11: $term\_iter = term\_iter + 1$;
12: **end if**
13: **end for**

Local search is based on searching a local neighborhood for an improving solution. A local neighborhood of a solution is constructed by small modifications, which are not necessarily designed to improve the solution. The neighborhood is then searched for either a *best-improving* or a *first-improving* solution (see Resende [79]). Iteratively, starting from an initial solution a sequence of improving solutions is generated. When no further improvement is possible the procedure ends with a local optimum.

The effectiveness of local search depends on several aspects, such as the solution modification operators, the neighborhood exploration approach, the evaluation of the objective function, and the initial solution. Furthermore, the number of moves needed to reach the local optimum can be large, even exponential to the problem size (Pitsoulis [77]). Therefore, construction of the initial solution plays an important role

for the overall performance of the metaheuristic.

A greedy construction algorithm operates on a *partial solution*. At each step, a set of current candidate elements for insertion into the solution is searched. Candidate elements are evaluated according to a *greedy function* which also may depend on the *partial solution*. The element with the best value is inserted into the solution. The procedure is repeated until the solution becomes complete or until no more candidate elements are available for insertion.

Greedy algorithms usually provide the same solution for a particular problem instance. This determinism leads the search process always in a specific area of the search space which may not contain the global optimum. If this is the case, then local search will always be stuck in the same local optimum. Randomization of the construction causes variation of the initial solution and thus, tries to solve this problem. This allows GRASP to explore different regions of the search space.

Randomization can be achieved at various points of the greedy construction algorithm. For example, selection of current candidates for insertion may not choose all possible candidates, but only a part of them and put them in a *Candidate List* (CL) for further selection. Also, selection of a candidate element may not always choose the best element from CL, but rather may choose randomly from a subset of high value elements called *Restricted Candidate List* (RCL). The few parameters of GRASP are usually concerned with controlling randomization of the semi-greedy construction heuristic.

4.2 Construction Phase

The construction phase implementation of the basic GRASP algorithm consists of three main stages:

1. Working for external demands,
2. Working for future demands and
3. Finalization stage of *filling* gaps .

The first two stages are very similar to each other and they represent the main construction heuristic. In fact, they are implemented with the same semi-greedy construction algorithm 4.2, but executed on different types of demands - external and future demands. The finalization stage is a solution repairing strategy used

in order to insert production lots on machines with empty (undefined) production state variables for certain time periods called "gaps". It tries to achieve feasibility of a production plan that may have been intentionally left unfinished by the main construction procedure (not only in the end of the planning horizon).

4.2.1 Construction Heuristic

The main construction procedure applied in Stage 1 and Stage 2 of the construction phase of GRASP is shown in Algorithm 4.2. It is a demand driven algorithm which assembles a production plan by iteratively inserting production lots for meeting selected demands. Furthermore, a value-based scheme (see Pitsoulis [77]) is used for controlling both the size of the RCL and contents of the CL by parameters α and β, respectively. Note that in this implementation the candidate list CL contains demands, while the restricted candidate list RCL contains insertion costs and positions of production lots for meeting the demands. However, the resulting production schedule might be incomplete because of the greedy constrained insertion at line 15, which is discussed in Section 4.2.4.

In Step 1 at line 3 first, all "unprocessed" demands are evaluated according to their *candidate cost* (see Section 4.2.2). Then, a candidate threshold *c_threshold* is determined using parameter β and the candidate list CL is constructed from all "unprocessed" demands that have a candidate cost bellow the threshold. This mechanism controls which demands are processed first by the semi-greedy algorithm. Furthermore, the determined threshold defines a level of acceptance of a demand that is further applied to partially met demands (because their candidate cost may change).

Remaining inventory from previous production lot insertions may contain units of products of the new unmet demands. In this case demands are (partially) fulfilled from available production in inventory in Step 2 at line 7 of the algorithm. The CL is updated according to the threshold level and completely met (zero valued) demands are marked as "processed".

Step 3 tries to assign production lots to the machines for all demands that remained in the candidate list CL. For this purpose the *best insertion cost* (described in section 4.2.3) is computed for all demands $d \in CL$. Insertion cost specifies production lots and their contribution to the overall quality of the schedule. Formation of the RCL is achieved by the value-based scheme using parameter α.

Algorithm 4.2 A Semi-greedy Construction Heuristic

Input: external demands (j, t, d_{jt}) or future demands (j, T, f_j)
α parameter for controlling the size of RCL
β parameter for controlling which demands are fetched into CL first
rand - a pseudo-random number generator

Output: a production plan with possible "gaps"

1: Store all demands (j, t, d_{jt}) in U ("unprocessed");
2: **while** U is not empty **do**
3: // Step 1: Fetch demands in CL according to their candidate cost.
4: Compute the candidate cost[3] $ccost(d)$ for each demand $d \in U$;
5: $c\_threshold := ccost_{min} + \beta \cdot (ccost_{max} - ccost_{min})$;
6: $CL := \{d \in U \mid ccost(d) \leq c\_threshold\}$;
7: // Step 2: Consume demands in CL from inventory.
8: Try to meet demands in CL (partially) from available inventory;
9: $CL := \{d \in CL \mid ccost(d) \leq c\_threshold\}$ and remove met demands from U;
10: // Step 3: Perform assignment of production lots for unmet demands.
11: **while** CL is not empty **do**
12: Compute the best insertion cost[4] $icost(d)$ for each demand $d \in CL$;
13: $RCL := \{d \in CL \mid icost(d) \leq icost_{min} + \alpha \cdot (icost_{max} - icost_{min})\}$;
14: Choose randomly insert. cost $icost_d$ for a demand $d = (j, t, d_{jt})$ from RCL;
15: Perform the best insertion[5] $icost_d$ for demand d at calculated position;
16: **if** Insertion could not be performed **then**
17: Mark demand d as unmet and remove it from CL and U;
18: **else if** $d_{jt} == 0$ **then**
19: Mark demand d as met and remove it from CL and U;
20: **else if** $ccost(d) > c\_threshold$ **then**
21: // Demand was only partially met and its candidate cost was changed;
22: Remove demand d from CL;
23: **else**
24: Leave demand d in both CL and U for further processing;
25: **end if**
26: **end while**
27: **end while**

Next, the algorithm tries to perform insertion for a randomly selected insertion cost from the RCL. If no production lot can be allocated on the partial schedule the demand is left unmet and marked as "processed". In case the demand was completely met, it is again marked as "processed". In case it was only partially met, the candidate threshold is applied to the demand and either it is removed from the current candidate list CL, or it is left there for subsequent lot insertions.

4.2.2 Candidate Cost

The candidate cost is used in Algorithm 4.2 for determining what subset of demands should be processed first. It provides a perspective of how much better is it to work first for an early demand with smaller value or for a later demand that has bigger value and may require more production time to be met. The candidate cost function of a demand $d = (j, t, d_{jt})$ is formally defined in (4.1).

$$ccost(d) := \beta_t \cdot t + \beta_v \cdot \frac{1000}{d_{jt}}, \text{ for a demand } d = (j, t, d_{jt}); \tag{4.1}$$

A greedy algorithm will choose to meet earliest demands first. However, there may be later demands that require more production time even on parallel machines. Production of such demands should start as early as possible. Coefficients β_t and β_v in the candidate cost function control the interplay between selecting an early demand and selecting a demand with bigger value, respectively. Along with the β parameter, they affect randomization of the construction heuristic by generation of the candidate list CL.

4.2.3 Insertion Cost

The best insertion cost $icost_d$ for a demand d specifies the best insertion point of a production lot working for d and a contribution cost of this lot to the overall quality of the production plan. In Algorithm 4.2 in Step 3 (row 10), the restricted candidate list RCL is formed from the lowest value insertion costs from CL. This allows for the random selection of an insertion cost in the next line to choose only from production lots with high quality contribution costs. Therefore, randomized construction will not

[3]Computation of the candidate cost $ccost(d)$ of a demand d is discussed in next Section 4.2.2.
[4]Computation of the best insertion cost $icost(d)$ is shown in Algorithm 4.3.
[5]The procedure for performing the best insertion for a demand is discussed in Section 4.2.4.

produce absolutely random solutions but rather will guarantee good overall quality of the schedules.

Algorithm 4.3 Calculation of the Best Insertion Cost $icost(d)$ for a Demand d

Input: A demand $d := (j, t, d_{jt})$
A partial production plan
Weight coefficients ω_s, ω_k, ω_q, ω_t, ω_l and ω_m for the insertion cost function
Output: $(icost_{best}, m_{best}, t_{best}, \delta_{best})$; // the best insertion cost and position for d

1: $icost_{best} = \infty$; $m_{best} = 0$; $t_{best} = 0$; $\delta_{best} = 0$;
2: **if** $I_{jt} > 0$ **then**
3: // the demand is satisfied from inventory in Step 2 at line 7 of Algorithm 4.2.
4: **return** $(0, m_{best}, t_{best}, \delta_{best})$;
5: **end if**
6: **for** Each machine $m \in M$ **do**
7: Let T_m be the first free time period of machine m;
8: **if** $T_m \leq t$ and $cap_{mj} > 0$ **then**
9: Find a feasible production lot for item j starting from $t' \geq T_m$ to $t'' \leq t$;
10: **if** Such a feasible lot exists **then**
11: // compute the lot's setting, production and unmet demand
12: Locate the previous production lot $l_0 = (m, i, t_0)$ on machine m until T_m;
13: $x_{mjt'} := 1$, if $j \neq i$ // a temporary setting variable;
14: Find the number of periods $\Delta_d := \lceil (d_{jt} - I_{jT_m})/cap_{mj} + x_{mjt'} \cdot v_{mj} \rceil$ necessary in order to satisfy demand d;
15: $q_{mjt''} = cap_{mj} \cdot (\min\{\Delta_d, t'' - t' + 1\} - x_{mjt'} \cdot v_{mj})$;
16: $\delta_k := \max\{0, I_{jT_m} + q_{mjt''} - d_{jt} - p_j \cdot b_j\}$; // expected buffer overrun
17: // compute the insertion cost on machine m
18: $icost := \omega_s \cdot s_{mj} \cdot x_{mjt'} + \omega_k \cdot \delta_k +$
19: $\omega_q \cdot \frac{100}{q_{mjt''}} + \omega_t \cdot \frac{1}{t - t' + 1} + \omega_l \cdot \frac{x_{mjt'}}{t' - t_0 + 1} + \omega_m \cdot x_{mjt'} \cdot (\sum_{\substack{j' \in J \\ k \in [1, t_0]}} x_{mj'k})$;
20: **if** $icost < icost_{best}$ **then**
21: $icost_{best} = icost$; $m_{best} = m$; $t_{best} = t'$; $\delta_{best} = \min\{\Delta_d, t'' - t' + 1\}$;
22: **end if**
23: **end if**
24: **end if**
25: **end for**
26: **return** $(icost_{best}, m_{best}, t_{best}, \delta_{best})$;

Algorithm 4.3 is used for calculating the best insertion cost of a demand. A partial production plan is needed in order to determine what is the available inventory until now, were are the free time periods T_m on machines $m \in M$, where a lot can be inserted and whether a setting needs to be performed for the new production lot.

If a demand $d := (j, t, d_{jt})$ can be satisfied from inventory I_{jt} (even only partially) the insertion cost is 0 - the lowest possible. If no inventory is available all machines $m \in M$ are checked for their capability to produce for the demand d. Even if a machine m is capable to produce for d, it doesn't mean that a feasible production lot can be allocated on m which satisfies (partially) the demand. If such a feasible lot exists on machine m starting at time period t' and finishing at t'' the corresponding lot's setting, production quantity and expected buffer overrun are computed.

The formula for calculating the insertion cost is given at rows 18 and 19. The weighted sum contains setting and buffer violation elements from the objective function (3.2), but no unmet external or future demands are accounted. This was experimentally found to be more effective. Other terms of the insertion cost function are designed to emphasize insertion for maximum production quantity, as early as possible, after larger previous lots, and for less total number of settings on the machine (see Table 4.1).

Note that, the original demands d_{jt} are stored separately and the construction procedure keeps track of what parts of the original demands are satisfied by working with unmet demands u_{jt}. Initially, unmet demands are equal to the original demands. As soon as best insertions are performed for a particular demand d_{jt}, the corresponding unmet demand u_{jt} gets update with the produced quantity.

4.2.4 Performing Insertion

Insertion is concerned with performing the settings, (partially) satisfying the demand d, updating the inventory and the next free time period of a production lot determined by an insertion cost $icost_d$. All these actions were already predetermined in the calculation of the insertion cost in Algorithm 4.3. However, it is important to mention that inventory should be stored per time periods. It is not necessary to store inventory for all time periods $t \in T$, but it is sufficient to keep track of it only per days.

Construction should start from left to right, lot by lot, on all machines at the same time. So, we keep track of markers T_m for each machine m showing where the first free time period is. The first free time period T_m is updated by the last inserted production lot at machine m for item j starting at time period t for demand

$d = (j, t', d_{jt'})$ in one of the following two ways:

- $T_m = t + \lceil (d_{jt'} - I_{jT_m})/cap_{mj} + x_{mjt} \cdot v_{mj} \rceil \quad (\leq t')$ when the requested demand $d_{jt'}$ can be fulfilled by the production lot, or
- $T_m = t'$, when the demand cannot be completely fulfilled.

Performing a new setting depends on the choice of a solution representation. As it was outlined in Section 3.2.2, a setting-based representation with integral setting variables results in a more compact solution r. In representation implementation discussed here, setting variables s'_{mt} are stored in a collection indexed by machine numbers $m \in M$ and sorted according to their time periods $t \in T$ for each machine m. Finding the production state of a machine $m \in M$ in time period $t \in T$ is implemented with a binary search procedure that locates the corresponding $x'_{mt'}, t' \leq t$ variable. A binary search was preferred to a linear search for finding production state because of improving the construction procedure performance two times (in terms of execution time). Then, adding a new setting is achieved by simple insertion of the new setting variable at a position located by the binary search procedure.

An important aspect of insertion is that it may leave "gaps". When finding a feasible lot in Algorithm 4.3 at row 9 it is possible that the beginning of the lot is $t' > T_m$. The reason may be that a parallel production constraint (3.9) or a maximum number of settings per period limitation constraint (3.8) may not allow beginning of the lot exactly at period T_m. In case this insertion cost gets selected, this will results in a "gap". Hence, a solution repairing strategy should be implemented in Stage 3 of the construction heuristic.

4.2.5 Solution Finalization (Filling of Gaps)

A production plan finalization procedure is required for obtaining a feasible solution with the construction heuristic presented in this chapter. However, it is possible to enforce allocation of a feasible production lot in Algorithm 4.3 starting immediately after the previous production lot. This will result in "gaps" only at the end of the planning horizon, which can be further filled with production for future demands. While this approach dose not require any additional repairing strategies, it produces worse quality solutions. This can be easily explained by the fact that instead of not producing at all because of unconstrained beginning of the lot, with the former approach the production lot is simply shifted until it becomes feasible. Hence, we

implement a more complicated greedy insertion which produces better results than the simpler insertion that dose not allow intermediate "gaps".

The solution finalization is a rather complicated, full-backtrack procedure which starts at the resulting solution of Algorithm 4.2 and tries to fill all of its "gaps" by performing local modifications to the current solution. Every modification produces a new solution which is then made the current. This often leads to a production plan that has some of the original "gaps" filled in, but no constrained production can be allocated for filling the remaining "gaps". In this case, a backtrack occurs which results in another attempt to fill in the original "gaps" in a different sequence.

The local modification operators designed to fill in production plan's undefined areas try to make use of available resources in order to satisfy remaining unmet demands. First, all remaining unmet future and external demands are used to find the best insertion cost (based on objective value) for a feasible production lot in the "gap"; if such a lot is found production is allocated. Second, the previous and the next lots are extended into the undefined time periods of the "gap" without causing infeasibility. Third, the "gap" is exchanged with another candidate production lot on a different machine containing the same time period; the previous and the next lots of both the "gap" and the candidate lot are also subject to the exchange. Finally, solutions that have been visited before or that have even more gaps than the current solution are eliminated. The best objective cost production plan is selected as a next current solution among the remaining schedules.

In most of the cases, the repairing strategy ends up with a feasible solution which is made the result of the construction phase of GRASP. However, in rare cases not all the "gaps" can be filled in and the constructed schedule has to be given up. A finalization strategy, which can guarantee feasibility is a subject to further investigation.

4.2.6 Parameters and Randomization Control

A full list of parameters and coefficients of the construction heuristic is given in Table 4.1. The main parameters which may cause semi-greedy randomization of the heuristic are α and β. They control respectively the size of the RCL and the contents of the CL. Proportion of the coefficients β_t and β_v also has a significant effect to the objective value of the constructed plan. The other coefficients may be used to influence specific characteristics of the resulting schedule.

Construction Parameters:

α	randomization parameter for controlling the size of RCL;
β	randomization parameter for controlling which demands are fetched into CL first;
β_t	candidate cost coefficient for selecting an early demand first;
β_v	candidate cost coefficient for selecting a demand with bigger value first;
ω_s	insertion cost coefficient emphasizing insertion for extending the previous lot without performing a setting;
ω_k	insertion cost coefficient emphasizing production with less buffer violations;
ω_q	insertion cost coefficient emphasizing insertion for maximum production quantity;
ω_t	insertion cost coefficient emphasizing insertion as early as possible;
ω_l	insertion cost coefficient emphasizing insertion after larger previous lots;
ω_m	insertion cost coefficient emphasizing insertion for less total number of settings on the machine;

Table 4.1 Parameters of the Semi-greedy Construction Heuristic

4.3 Improvement Phase

The improvement phase of GRASP Algorithm 4.1 (line 6) is implemented with a Local Search heuristic. Local Search and its relation to GRASP was already explained in Section 4.1. However, there are different variations of neighborhood search algorithms. Most of them are concerned with the efficiency vs. solution quality trade-off. In this sense, it is important to know that most of the execution time of GRASP is consumed by improvement of the constructed solutions. This is the reason for using a semi-greedy construction algorithm, which yields relatively good quality solutions, rather than using random initial solutions for the Local Search. In this section the exact implementation of Local Search heuristic is presented and its solution quality compromise is discussed in details.

4.3.1 Local Search Heuristic

The basic structure of the Local Search heuristic is given in Algorithm 4.4. Initially, a trajectory based optimization starts at an initial solution X_s and iteratively applies a randomly selected local neighborhood operator on a good quality solution from the available list of operators: $nhood\_oprs$. An operator op modifies a selected solution x and generates a local neighborhood $nhood$ of similar feasible solutions. Then, the neighborhood is evaluated and the best solution X_{best} found in neighborhood according to objective value is tested for a current solution in the trajectory (i.e. better than the last solution). The optimization process is repeated until no further improvement is possible; thus, it ends with a local optimum X_l.

Algorithm 4.4 A Randomized Local Search Heuristic

Input: X_s - an initial solution
$nhood\_oprs$ - a list of local neighborhood solution modification operators
$rand$ - a pseudo-random number generator
Output: a feasible (local optimum) solution X_l and its objective value f_l

1: $f_l = \infty$;
2: $nhood \leftarrow X_s$; // put the initial solution in a local neighborhood
3: **while** No further improvement **do**
4: // Randomly select a modification operator and a solution from $nhood$
5: $op = SelectOperator(nhood\_oprs, rand)$;
6: $x = SelectSolution(nhood, rand)$;
7: $ExecuteOperator(op, x, nhood, rand)$;
8: $X_{best} = Evaluate(nhood)$;
9: **if** $f(X_{best}) < f_l$ **then**
10: $f_l = f(X_{best})$;
11: $X_l = X_{best}$;
12: **end if**
13: $Filter(nhood)$; // leave only good quality solutions in $nhood$
14: **end while**

Local neighborhood contains only feasible solutions, generated by local modifications applied to a selected good quality solution. However, in this implementation the neighborhood always contains also, the current best solution found so far. For example, in the beginning the initial solution X_s is inserted into $nhood$. In each iteration, after the neighborhood is evaluated it is filtered at row 13 in order to be

prepared for the next iteration. The current best solution is the only one solution which is guaranteed to remain in the neighborhood. Additionally, other good quality solutions may not get filtered out and thus, will be available for selection at row 6 in the next iteration. As a consequence, the local neighborhood is never empty.

Neighborhood Exploration Strategy Neighborhood search strategies are computationally very expensive techniques because they have to repeatedly generate and evaluate enormous local neighborhoods of solutions. Therefore, the choice of a local neighborhood exploration strategy is crucial for the overall performance of the improvement phase of GRASP. One possibility is to use a *first-improvement* strategy which will basically stop generation and evaluation of the neighborhood as soon as a better solution is found than the current best solution. This approach is not always guaranteed to significantly increase performance, since improvement might tend to occur in the last stages of neighborhood generation. Furthermore, it may require more steps to converge since always the first better solution gets selected (results in smaller steps), rather than the best solution in neighborhood (results in bigger steps).

In Algorithm 4.4, a *best-improvement* strategy is implemented where all solutions in *nhood* are evaluated in order to find the best one. The best solution found is then tested for improvement. However, local neighborhood operators are designed to generate a large number of similar feasible solutions. As a result, the improvement heuristic would have serious performance issues. In order to escape from this effect a *max_nhood_size* parameter is introduced which enforces exploration of only a small portion of the available operators neighborhoods (Stadelmeyer et al. [26]). Furthermore, operators are designed to start generation of this neighborhood subset every time from a random point. This issues an assumption that a sufficient number of non-improving iterations should be executed before termination in order to guarantee that the whole neighborhood of an operator will be explored.

Randomized partial neighborhood exploration is a technique applied in order to boost performance. On one hand, it may lead to a selection of a suboptimal solution for improvement which would result in smaller trajectory optimization steps. As a result the algorithm may have to perform more improvement steps than the full neighborhood exploration strategy. On the other hand, each step requires proportionally less computational effort. Furthermore, the solution that gets selected for improvement depends on the randomized neighborhood construction start. The neighborhood exploration strategy used in Algorithm 4.4 represents a good compro-

mise between size of the improvement steps and performance. However, special care has to be taken when choosing a value for the *max_nhood_size* parameter and when specifying a terminating condition.

Termination Criteria The termination criteria used for Algorithm 4.4 is only based on the concept of no further improvement. An additional parameter *term_iterations* specifies a maximum number of subsequent non-improving iterations of the algorithm after which the heuristic should terminate. When this happens, the resulting solution cannot be verified to be a theoretical local optimum since not all possible solutions in the local area can be generated by the neighborhood operators. This is further emphasized by the fact that only a certain part of the local neighborhood generated by the operators is examined in one iteration. Hence, the termination parameter has to be carefully adjusted so that almost all portions of the operators neighborhoods get explored before the current best solution is proclaimed to be a "local optimum".

A local search is considered to be "stuck" in the current best solution, if:

$$\frac{term\_count}{term\_iterations} > nhood\_pressure, \tag{4.2}$$

where *term_count* indicates the current number of subsequent non-improving iterations and *nhood_pressure* is an additional parameter with a value between 0 and 1. The state of being "stuck" is different from the termination criteria of the algorithm. However, local search will always get "stuck" before it terminates allowing for preliminary termination detection. When this happens the algorithm will try to change its behavior in order to prevent premature termination. Thus, escaping from the current best solution and forwarding to the real local optimum (see Fig. 4.1).

Selection Pressure Selection pressure is a concept used in Evolutionary Strategies (Rechenberg et al. [80] and Schwefel et al. [81]) and also interpreted in Genetic Algorithms (Affenzeller et al. [82], [83]). It is often exploited as the proportion of the number of parent individuals in the population to the number of newly generated individuals (by recombination and mutation) for further selection. Usually, it is varied during the execution of the optimization process in order to intensify the search into more promising areas. However, local search is a trajectory based approach and at each iteration only one solution gets modified. Furthermore, the number of new modified solutions is constant and equal to *max_nhood_size*. Hence, selection pressure cannot be analogously defined.

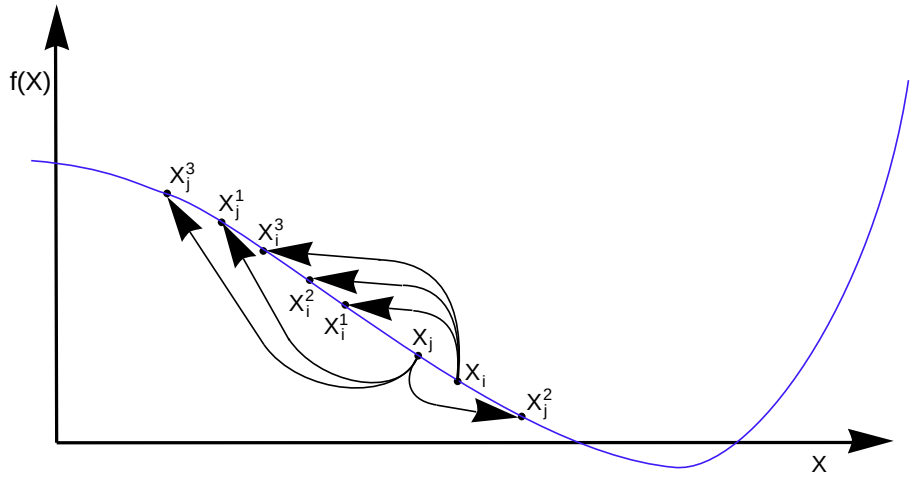


Figure 4.1 Local search might get "stuck" due to neighborhood structure. Suppose that solutions X_i and X_j are both available for selection. X_i is the better solution but its neighborhood (X_i^1, X_i^2, X_i^3) dose not contain any improving solution and the search would get "stuck". However, solution X_j has worse objective value but its modification X_j^2 would lead to an improvement.

In Algorithm 4.4, when the search is "stuck" more than one solution is made available for modification. This is achieved when the neighborhood is filtered at row 13 by an additional parameter *nhood_coef* that specifies which solutions in the neighborhood will be made available for modification in the next iteration:

$$nhood := \{X \in nhood \mid f(X) < (1 + nhood\_coef) \cdot f(X_{best})\}\,. \tag{4.3}$$

The number of solutions which may remain in *nhood* is further limited by a parameter *min_nhood_size*. Nevertheless, only one solution gets randomly selected from *nhood* for modification at row 6. Selection pressure can be defined as the proportion of the number of solutions available for modification to the number of solutions that are generated by the modification operator *op* (i.e. *max_nhood_size*). Initially, its value is $\frac{1}{max\_nhood\_size}$ when only the best found solution gets selected for modification and then, it may vary in the interval $[\frac{1}{max\_nhood\_size}, \frac{min\_nhood\_size}{max\_nhood\_size}]$ according to *nhood_coef* when the local search is "stuck" (4.2).

Premature termination can be interpreted for local search as the condition of getting "stuck" in a solution which is not a "local optimum" in neighborhood space. The main reasons for this might be neighborhood structure and improper selection for modification as visualized in Fig. 4.1. Increasing selection pressure leads to more

randomized selection for modification, since more solutions will be made available in *nhood* after filtering. Therefore, the algorithm will explore more different neighborhoods which would increase the chance of escaping from a suboptimal solution. Selection pressure can be successfully used to prevent premature termination in local search.

Although randomized Local Search with varying selection pressure was successfully applied to this particular problem setup and neighborhood operators, it is not guaranteed that it will work for any problem setup and operators. Generalizations of this improvement scheme are subject to future research. Typically, exploring neighborhoods of different candidates at one level of the search is referred to as a look forward strategy. However, the neighborhood search may benefit from a careful implementation of a multi-level look forward strategy.

Undoubtedly, the Basic GRASP metaheuristic needs intensification into local areas in order to provide close approximations of real local optima. On the other hand, some of the time required to obtain these optima can be used to explore different areas. Therefore, selection pressure may balance the interplay between intensification and diversification when controlled by the metaheuristic (not only by the local heuristic). Furthermore, memory based strategies may be used to store intermediate suboptimal solutions and improve them more intensively, subsequently in the execution of the metaheuristic.

4.3.2 Local Neighborhood Operators

Local neighborhood operators have the role of applying small modifications to a given feasible solution and producing a set of modified feasible solutions. Local changes should not be necessarily designed in order to produce better solutions, although this is what the neighborhood will be searched for afterwords. Modification operators should generate as many feasible solutions as possible in a local area around the specified solution. Thus, Local Search will rather perform small improvement steps and will systematically explore the area instead of jumping all over the search space. Furthermore, the heuristic may benefit significantly from varying neighborhood structure within a search (Paula [68]).

In this section first, some basic modifications that can be applied to a production plan by preserving its feasibility are discussed. Then, it is shown how these changes can improve the solution quality of a schedule. Basically, similar improvements were already presented in Section 3.1.4. Finally, all the specialized neighborhood operators

are presented.

Basic Modifications A basic modification represents a very small change that can be applied to a complete feasible production plan in order to obtain another different feasible plan. Every change, removal or insertion of a new integral setting variable of a schedule is considered to be a basic modification. Besides these changes, other more complicated modifications have been implemented. However, it is not always possible for the resulting plan to remain feasible. In this case the initial solution remains unchanged and the operation fails.

Insert Lots Insertion of a new lot overrides production setting and state variables of a production plan on a specified machine $m \in M$ from $t_{start} \in T$ to $t_{end} \in T$ with production for item $j \in J$. There are two types: head insertion and tail insertion. Head insertion assumes that the new lot will start at t_{start} and will end no later than t_{end}. Tail insertion assumes that the new lot will end at t_{end} and will start not earlier than t_{start}. However, the size of the lot is determined at runtime so that the resulting plan remains feasible.

Extend Lots Extension of a lot in a production plan can happen only if a neighboring lot on the same machine is overridden. Hence, a lot that will be reduced needs to be specified by its machine number $m \in M$ and its starting time period $t \in T$. There are two types of extensions: of the previous lot and of the next lot. Additionally, for the first type an end time period t_{end} that should not be exceeded and for the second type a start time period t_{start} such that production cannot start earlier can be specified. Extension tries to override the specified lot with production either for the same item as the item produced in the previous lot at the head or for the item produced by the next lot at the tail. The size of the extended lot is again determined at runtime so that the resulting solution remains feasible and no additional limits (t_{end} or t_{start}) are exceeded.

Exchange Lots A production lot on machine $m \in M$ starting at $t_m \in T$ and a lot on machine $n \in M$ starting at $t_n \in T$ (or specific parts of them) can be exchanged in a given schedule. There are tree basic strategies: a "head-for-head" exchange, a "tail-for-tail" exchange and a "head-for-tail" exchange (see Fig. 4.2). However, for all types it is essential to determine a maximum exchange lot size in terms of time periods such that the resulting plan will not be infeasible. For example, in Fig.

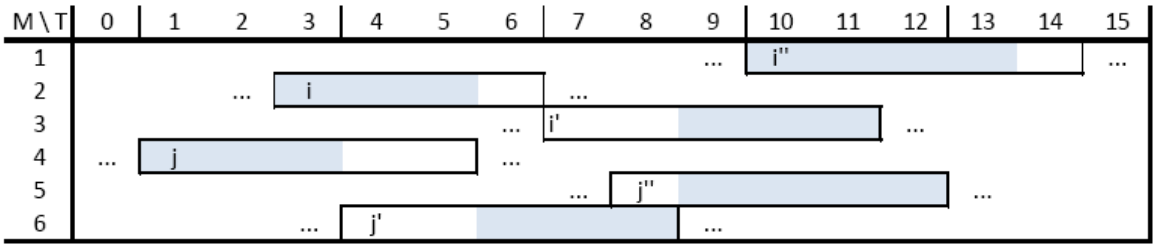


Figure 4.2 Exchange Lots is a basic modification which operates on two lots of a production plan by exchanging specific parts of them. The lots for items i and j are exchanged using a "head-for-head" strategy, i' and j' are exchanged using a "tail-for-tail" strategy, i'' and j'' are exchanged using a "head-for-tail" strategy. A maximum exchange lot size should be determined so that the schedule would remain feasible.

4.2 only 3 time periods from production lots for items i' and j' can be exchanged by a "tail-for-tail" strategy, since the maximum parallel production for item i' may already be full in time periods 4 and 5 ($i' = j$). Nevertheless, an exchange is not always possible.

Neighborhood Operators A neighborhood operator performs a number of specialized modifications at all possible positions in a production plan. For every feasible change a new plan is generated and put in the neighborhood. Furthermore, all operators are designed to start from a random position in the schedule and to stop generating similar plans after *max_nhood_size* number of solutions have been produced.

Algorithm 4.4 selects randomly an operator at line 5 among the following five neighborhood operators:

- Exchange Machine Lots Operator
- Exchange Parallel Lots Operator
- Merge Similar Lots Operator
- Extend Lots for Unmet Demands Operator
- Insert Lots for Unmet Demands Operator

A brief description for each of the operators follows.

Exchange Machine Lots The Exchange Machine Lots operator generates all possible subsets $\{\{m,n\} \mid m \neq n; m,n \in M\}$ of two machines in a certain sequence with a random start. For each such subset of two machines $\{m,n\}$ all the integral setting variables $x'_{mt}, t \in T$ are moved to machine n and vice versa. Feasibility will be preserved if production on the machines is possible (3.4). Note, that parallel production (3.9) and maximum number of settings per period (3.8) constraints cannot be violated by the modification.

Additionally, two more specialized modifications are performed to every two different machines $\{m,n\}$: Exchange Machine Lots Heads and Exchange Machine Lots Tails. In the first type a "head-for-head" basic exchange modification (4.3.2) is applied to the lots at machines m and n starting from the first time period till the end of the planning horizon. In the second type a "tail-for-tail" basic exchange modification is applied to the lots at machines m and n in a reverse order starting from the last time period till the beginning of the planning horizon. If a particular lot for x'_{mt} on machine m cannot be produced on machine n or vice versa it will be simply skipped. The additional specialized modifications are rather designed to produce feasible plans, than the direct exchange which may fail depending on the production capabilities of the machines.

Exchange Parallel Lots Exchange Parallel Lots operator enumerates all possible subsets $\{\{(m,x'_{mt},t),(n,x'_{nt},t)\} \mid x'_{mt} \neq x'_{nt}, m \neq n; m,n \in M; t \in T\}$ of two different lots that start in the same time period t in a specific sequence and with a random start. For each two such lots $\{(m,x'_{mt},t),(n,x'_{nt},t)\}$ a "head-for-head" basic exchange modification (4.3.2) is applied. The operator is designed so that parallel production (3.9) and maximum number of settings per period (3.8) constraints cannot be violated. If machines m and n have the required production capabilities the resulting schedule will be feasible and will be added to the local neighborhood.

Merge Similar Lots In Merge Similar Lots operator, all tuples of similar lots $\left\{((m,x'_{mt_m},t_m),(n,x'_{nt_n},t_n)) \mid x'_{mt} = x'_{nt}; m,n \in M; t_m,t_n \in T\right\}$ are generated in a specific sequence and with a random start. For each such tuple of similar lots $((m,x'_{mt_m},t_m),(n,x'_{nt_n},t_n))$ the first lot is defined to be a source and the second lot is defined to be a target. First, a "tail-for-tail" and a "head-for-tail" basic exchange modifications (4.3.2) are applied to the source and the previous lot of the target. Similarly, a "head-for-head" and a "head-for-tail" basic exchange modifications are applied to the next lot of the target and the source. The goal is to extend the target

lot by transferring production from the source lot to the neighboring lots of the target. Execution of the specialized modification may result in a feasible transfer of the whole source which would result in less settings in the modified schedule.

Extend Lots for Unmet Demands This operator first, fetches all external unmet demands d in a set $U = \{(j_d, t_d, d_{j_d t_d}) \mid d_{j_d t_d} > 0; j_d \in J, t_d \in T\}$. Then, it iterates through all integral setting variables $x'_{mt}, m \in M, t \in T$ of the schedule in a specific sequence and with a random start. If a particular setting variable x'_{mt} indicates production for a demand $d \in U$ (i.e. $x'_{mt} = j_d$) specialized extension operators are applied to the lot on machine m starting at time period t. Specialized extension operators are two types: left and right. A Left Extension operator tries to overwrite the tail of the previous lot by applying a basic next extension modification (4.3.2) with all possible extension lot sizes. Similarly, a Right Extension operator tries to overwrite the head of the next lot by applying a basic previous extension modification with all possible extension lot sizes.

Additionally, after every basic extend modification applied to a production lot (m, x'_{mt}, t) a specialized Left Shift or Right Shift operator is applied with the current extension lot size Δ. A left shift operator iteratively shifts backward previous settings $x'_{mk}, k \leq t, k \in T$ on machine m with Δ time periods resulting in settings $x'_{m(k-\Delta)}, k - \Delta \geq 1$. A right shift operator iteratively shifts forward next settings $x'_{mk}, k > t, k \in T$ on machine m with Δ time periods resulting in settings $x'_{m(k+\Delta)}, k + \Delta \leq T$. If after any successful shift the resulting plan is found out to be feasible it is added to the local neighborhood and the iterative modification continues. A shift basically extends the specified lot with Δ time periods without reducing the size of preceding or following production lots.

Insert Lots for Unmet Demands The insertion operator enumerates all external unmet demands $d \in U$ in a specific sequence and with a random start. For each unmet demand $d = (j_d, t_d, d_{j_d t_d}), d_{j_d t_d} > 0$ and for all machines $m \in M$ (with a random start) basic head and tail insertion modifications (4.3.2) are applied. First, a tail insertion is applied at time period t_d with a maximum possible lot size but without exceeding the required quantity $d_{j_d t_d}$. Then, the latest setting $x'_{mt}, t < t_d$ on machine m is located and a head insertion is applied at time period t with maximum possible lot size but ending not later than t_d and without exceeding the required quantity $d_{j_d t_d}$. Insertion of new lots for unmet demands by overriding old lots may influence the balance of met and unmet demands as discussed in Section 3.1.4.

4.3.3 Parameters and Randomization Control

Parameters of the local search heuristic are shortly listed in Table 4.2. Parameter *max_nhood_size* determines the size of the local neighborhood. Smaller neighborhood sizes will result in more randomization. However, *term_iterations* has to be adjusted accordingly so that operators neighborhoods have the chance to be explored thoroughly.

Local Search Parameters:	
max_nhood_size	a maximum local neighborhood size; default values are 1/5-th, 1/10-th of the largest operators neighborhoods;
term_iterations	a number of non-improving iterations after which the algorithm should terminate; default values should be at least 25 or 50;
nhood_pressure	a limit for a number of non-improving iterations after which local search gets "stuck" (4.2);
nhood_coef	a coefficient which determines solutions from neighborhood that will be made available for modification in the next iteration;
min_nhood_size	a maximum number of solutions that can be made available for modification in the next iteration;

Table 4.2 Parameters of the Randomized Local Search Heuristic

Chapter 5

GRASP with Path Relinking

Reinforcement learning is a problem concerned with acquiring behavior through trial-and-error interactions with a numerical reward outcome within a dynamic environment (Kaelbling [84], Sutton [85]). Similarly, the heuristic approach presented in the previous Chapter 4 has no knowledge about the desired outcome, although no changes can be introduced into the environment (i.e. the objective function cannot be altered). However, a major disadvantage of the Basic GRASP Algorithm 4.1 is that the separate optimization trajectories build in different iterations are independent from each other. Therefore, the algorithm can be further enhanced with the capability of learning from previously obtained knowledge about the search space. Advanced techniques for GRASP are approaches that solve the simplified reinforcement learning problem of heuristic optimization by incorporating adaptive memory into the existing GRASP algorithm (Glover et al. [55], Atkinson et al. [86], Fleurent [87], Taillard [88]).

In this chapter we focus on only one basic enhancement strategies for GRASP. The strategy is called Path Relinking, and it is a metaheuristic on its own which operates upon the trajectory search of GRASP and is based on the idea of combining solutions (Glover et al. [55]). It further utilizes information about already explored regions of the search space in a pool of suitably diverse elite solutions. The strategy tries to enforce the concepts of intensification and diversification in the search process - well known exploitation and exploration mechanisms of Tabu search (Glover et al. [53,54]).

5.1 Path Relinking

Path-relinking was first introduced by Glover et al. [55] as a strategy which combines solutions by generating and exploring linear combinations of selected points in neighborhood space (compare with Euclidean space). A linear combination of two solutions can be obtained by constructing a path between and beyond them. When applied to a set of weighted centers of selected subregions, path relinking may result in non-convex combinations from new subregions. Such a diversified exploration of the search space have been found useful for several NP-hard optimization problems.

The method is seen as a broader generalization of Scatter Search methodology and is often applied in conjunction with Tabu search (Glover [50, 89]). Scatter search retrieves new knowledge by constructing and combining selected solutions from previous knowledge. It maintains adaptive memory of suitably diverse elite solutions in a reference set by using a type of clustering. Only elite solutions within clusters and across clusters are introduced in memory to enable intensification and diversification of the search. These powerful features of scatter search have evolved into a more general Path Relinking framework.

5.1.1 Basic Idea

A path is constructed between an *initiating solution* I and a *guiding solution* G. Special moves have to be designed that would progressively introduce attributes of the *guiding solution* in the *initiating solution* (or reduce their difference). Intermediate solutions I_1, I_2, ... contain a variable subset of old attributes from I and new attributes from G (see Fig. 5.1). Hence, they represent effective combinations with variable "mixes" of the two solutions (Glover [89]).

The process is very similar to the neighborhood search heuristic from Section 4.3 with the difference that the search trajectory is directed to a specific point in neighborhood space with a limited/different set of possible moves, rather than to an unknown better solution. Analogously to the linear combination interpretation, trajectory may be guided by many different solutions or even extended beyond the *guiding solution/s*. These approaches are called Multiple Parents and Extrapolated Relinking, respectively. However, this paper discusses path-relinking in its basic form.

Local moves in path generation may differ from those used in Local Search. Transition neighborhoods in path relinking may allow infeasible solutions resulting in paths that pass through infeasible regions - often called a Tunneling strategy

(Glover [55, 89]). In addition, the purpose of these moves may be implicitly achieved by introducing a dissimilarity measure. Path construction should then progressively decrease the dissimilarity between the current and the guiding solution. In this thesis, no formal definition of an attribute of a production plan is given, but rather a *first-difference* approach is used to choose a local move in a path.

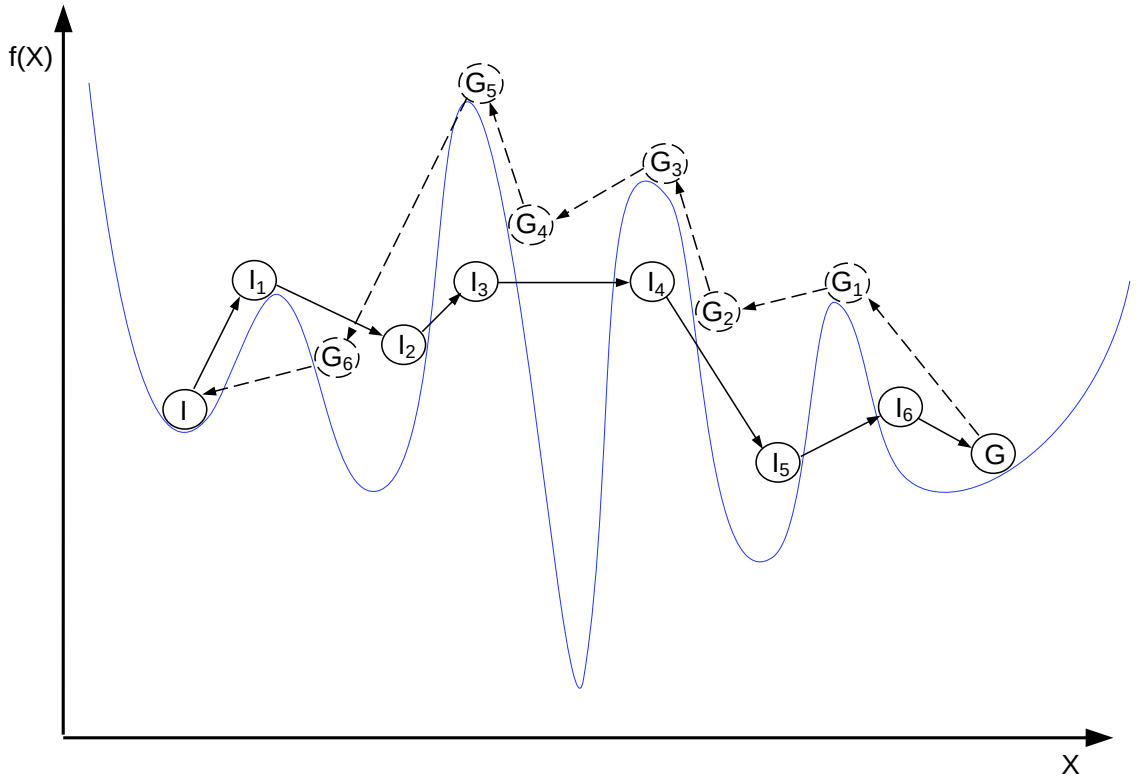


Figure 5.1 Path relinking starts from an *initiating solution* I and performs moves that subsequently introduce attributes from a *guiding solution* G by generating intermediate solutions I_1, I_2, ... The process executes until all attributes of the *guiding solution* are included but may continue beyond. The roles of I and G are interchangeable, allowing for the alternative path to visit new regions of the minimization problem's search space.

There are various strategies for path construction. In Forward Relinking, the path is generated starting from one (i.e. the worst) solution among I and G as an *initiating solution* and the other as a *guiding, target solution* (Resende [79]). However, the roles of *initiating* and *guiding* solutions are interchangeable. This may often result in a different path that visits different intermediate solutions (Pitsoulis and Resende [77]). For example, in Fig. 5.1 the backward path starting from G toward I generates a

solution G_4 in a subregion of the search space (for a minimization problem) that was not visited by the forward path. The strategy is called Backward Relinking and is usually applied together with Forward Relinking. Mixed Relinking is an alternative for a well balanced solution combination strategy between computational effort and number of intermediate solutions which is discussed in more details in Section 5.1.3.

Path relinking is usually applied to a special pool of elite solutions, very similar to the reference set in Scatter Search. A memory mechanism which isolates different, good quality solutions tends to focus on *consistent* and *strongly determined* variables (Glover [89]). In addition, most hybridizations of GRASP with Path Relinking (Resende [90]) usually keep only "local optima" in the pool. Hence, path relinking incorporates both intensification within subareas of local optima and diversification across such areas. This is a major enhancement of the basic GRASP metaheuristic, since the Local Search Algorithm 4.4 may yield not exact "local optima" and the Semi-greedy Construction Algorithm 4.2 may not always provide sufficient diversity of solutions.

5.1.2 Dissimilarity Measure

A dissimilarity measure is necessary in order to know what is the distance between two points in neighborhood space. When generating a path the distance between the current and the guiding solutions should be progressively reduced, until it becomes close to 0. Ideally, path relinking should be executed on a set of well scattered in search space diverse elite solutions (Boudia [91]). Therefore, a dissimilarity measure is also necessary in order to maintain a pool of significantly diverse solutions.

However in practice, it is not always straightforward to define a good dissimilarity measure that satisfy all properties of a distance measure (i.e. symmetry, triangle inequality). Furthermore, the neighborhood space of all solutions accessible by construction and local moves is not deterministic in GRASP. Hence, a dissimilarity measure $D(X, Y)$ which estimates the differences between two production plans X and Y can be used instead.

The dissimilarity $D(X, Y)$ consists of the following types of cost:

- *item difference cost* of 3 - added per mismatch of items in lot's item sequences on the same machine,
- *time difference cost* of 1 or 2 - added per mismatch in the starting periods of two lots which were already matched by item sequence comparison, and

- *size difference cost* of 1, 2, or 3 - added per mismatch in produced quantities by two lots which were already matched by item sequence comparison.

The proposed measure dose not introduce an exact comparison of state variables of the two schedules, but rather allows more variation of positioning of production lots. Solutions X and Y are compared independently on all machines $m \in M$ by comparing lot's item sequences of integral setting variables x'_{mt} ordered by their time periods $t \in T$. If there is a mismatch between two settings (of X and Y, respectively) in the lot's item sequences, an *item difference cost* of 3 is added to the dissimilarity measure. Then, the earlier mismatched setting is skipped and comparison of the sequences continues. In case that the two mismatched settings are at the same time period, they are both skipped. However, since positions and sizes of production lots are not exactly matched (i.e. only lot's item sequences are matched) other costs also need to be incurred.

A *time difference cost* is computed for every two production lots (from X and Y) that were matched by item sequence comparison, but do not start at the same time period. If the starting time difference of the two lots in time periods is less than 30% of the whole planning horizon a cost of 1 is incurred, otherwise a cost of 2 has to be considered. Additionally, a *size difference cost* of two item sequence matched production lots is accounted if the production quantities of the lots are not equal. If the difference in production quantity of the two lots is less than 20% of the target lot's quantity a small size cost of 1 is added, if it is between 20% and 50% a medium size cost of 2 is added, if it is more than 50% a large size cost of 3 is added.

5.1.3 Path Construction

Path construction is interpreted as a solution combination method by natural extension of the concept of linear combinations in Euclidean space. However, this paper dose not consider advanced paths generated by Multiple Parents or Extrapolated Relinking (Glover [89]). Scatter search and its principles rely on such paths in order to generate non-convex combinations in neighborhood space from a set of elite solutions. On the other hand, Path Relinking has specialized techniques like Tunneling and Neighborhood Structure, such that when applied to diverse solutions from a multi-modal objective landscape they provide new combinations far beyond a convex objective subarea.

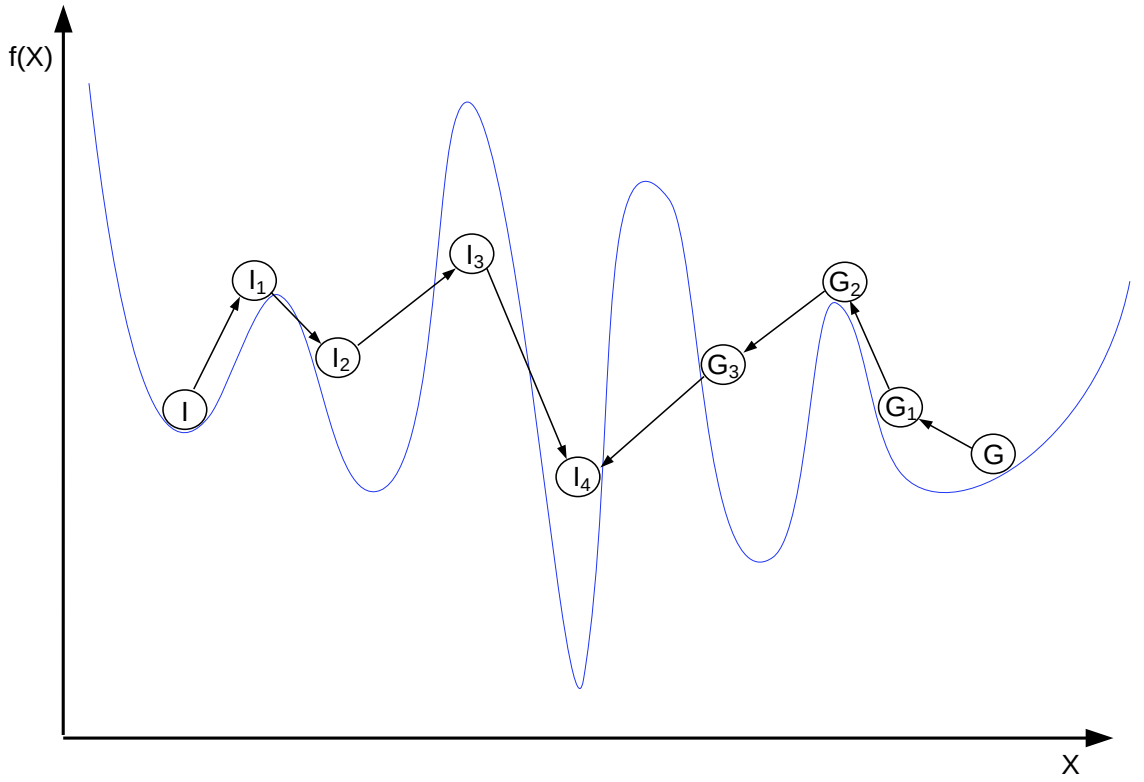


Figure 5.2 Path construction may be accomplished by a Mixed Relinking approach that alternatively changes the roles of *initiating* and *guiding* solutions. Two paths are constructed: one from I and one from G, until they meet in an intermediate solution $I_4 \equiv G_4$. The union of the two paths represents a well balanced single path between I and G.

Mixed Relinking Mixed Relinking is a path construction strategy that alternatively changes the roles of *initiating* and *guiding* solutions after each move. In Fig. 5.2, two paths are simultaneously generated: starting from the *initiating solution* - I, I_1, I_2, I_3 and starting from the *guiding solution* - G, G_1, G_2, G_3. Dissimilarity between the current two solutions on top of the paths is consequently reduced, until it becomes 0 and the two paths meet in a solution $I_4 \equiv G_4$. The resulting single path from I to G is well balanced and more symmetric because its two parts were independently constructed to "climb up" the corresponding search subareas of I and G ("local optima"). Additionally, this approach executes faster than the Fore and Backward Relinking since the number of the intermediate solutions is smaller.

Local Moves Local moves have to extend the current path by producing a new solution that leaves a reduced number of moves remaining to reach the *guiding solution* G. Another more restrictive policy may obligate that a local move that leaves the "fewest" number of remaining moves should always be chosen, resulting in a shorter path. The available local candidates for extending the path from the current solution on top constitute the *transition neighborhood* of path construction. For example, in Fig. 5.3 the *transition neighborhood* consists of solutions I_1^*, I_2^*, I_3^*, I_4^*, I_5^* under the assumption that I is the current solution on top of the path.

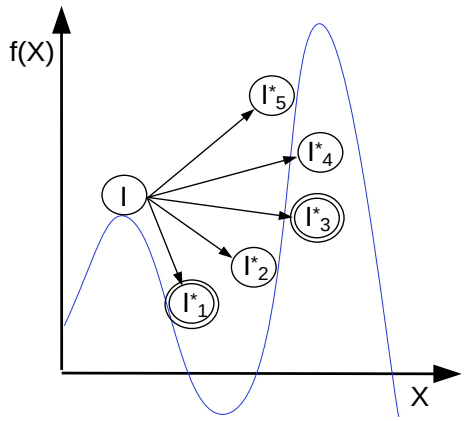


Figure 5.3 Local moves for path construction should always choose the best feasible solution (I_1^* in this case). If no moves that yield a feasible solution exist, unfeasible solutions should be evaluated. Again, the move that gives the best unfeasible solution (I_3^*) should be selected.

The proposed policy selects always a move that yields the best-objective value solution (i.e. I_1^*) among feasible solutions in the neighborhood. Additionally, the neighborhood contains solutions which reduce non-strictly the distance to the *guiding solution* in terms of dissimilarity measure. However, the next solution in the path should be different from the previous solutions in order to escape from cycles. Feasibility of the local neighborhood cannot be always guaranteed. In case that the *transition neighborhood* contains only infeasible solutions (i.e. I_3^* and I_4^*), they should be evaluated with appropriate penalty for their infeasibility and again the best solution (I_3^*) among them should be selected.

Tunneling is a strategy for path construction encouraged by a different neighborhood structure that allows certain parts of the path to go through infeasible regions.

The resulting path "tunnels through" infeasible regions of the objective function landscape and "comes out" by restoring feasibility at new subareas of the search space that have never been explored before. For example, the forward path in Fig. 5.1 contains two infeasible solutions I_3 and I_4 which results in a shorter path that tunnels through a very narrow area of the landscape. After that, a new solution I_5 is generated in a new local optimum subarea with better objective value than I and G. However, allowing infeasibility for intermediate solutions reduces the number of feasible combinations that can be examined in the path. Nevertheless, as soon as the dissimilarity between the current and the guiding solutions is decreasing, feasibility is guaranteed to be restored by the time G is reached (Glover [89]).

Implementation Details Constructing paths between feasible production plans for the DLSP from Chapter 3 is not straightforward because of the non-trivial parallel production and settings limitation feasibility constraints: (3.9) and (3.8). The developed approach is based on the item, time and size mismatches of production lots from the definition of the dissimilarity measure, resulting on corresponding *item*, *time* and *size fixes*. Furthermore, selecting the position of the mismatch that will be fixed in the current move may influence significantly feasibility and quality of the intermediate solutions.

A local move first, finds the earliest mismatch between the current and the guiding solutions. Then, several subsequent *fixes* are applied to this mismatch in order to generate the transition neighborhood. A mismatched lot on machine $m \in M$ is a lot $(m, x'_{mt}, t), t \in T, x'_{mt} \in J$ in the current solution which is different by item, start time period or end time period from its corresponding lot $(m, x'_{mt'}, t'), t' = \max_{k \leq t} \{k \in T \mid \exists x'_{mk}\}$ in the guiding solution. The earliest mismatch is selected from all the earliest mismatched production lots located independently on all machines $m \in M$. However, if the schedule on top of the path is infeasible only lots which cause infeasibility are examined. The resulting *first-diff* procedure tends to locate mismatches in the initial schedule from left to right but also, to choose infeasible lots first in order to restore feasibility.

An *item fix* is applied to a mismatched lot (m, x'_{mt}, t) by item: $x'_{mt} \neq x'_{mt'}$. First, all lots $\{(n, j, k) \mid n \in M, j \in J, k \in T; j = x'_{mt'}\}$ with the target item $x'_{mt'}$ are located in the current solution. For each of the lots, basic "head-for-head", "head-for-tail" and "tail-for-tail" exchange modifications (see Paragraph 4.3.2) are applied to the mismatched lot. Similarly, all lots $\{(n', j', k') \mid n' \in M, j' \in J, k' \in T; j' = x'_{mt}\}$ with

the current item x'_{mt} are located in the guiding solution. Then, the same basic exchange modifications are applied to the mismatched lot and to the corresponding lot $(n', x'_{n'k}, k) \leftarrow (n', j', k'), k = \max_{l \leq k'} \{l \in T \mid \exists x'_{n'l}\}$ in the current solution. Additionally, production for item $x'_{mt'}$ is forced by basic insertion modifications (Paragraph 4.3.2) and replacement of the value of the integral setting variable $x'_{mt} = x'_{mt'}$ which may cause infeasibility. Finally, the resulting schedules are added in the current transition neighborhood and a *time fix* is performed at the same position to all plans generated by the *item fix*.

The proposed strategy for performing an *item fix* is based on feasible basic exchange modifications in order to handle parallel production (3.9) and settings limitation (3.8) constraints. Infeasibility is preferred only if there is no other option. In this case all infeasible plans are evaluated by penalizing total infeasible production quantity with the buffer overrun coefficient ω_k in the objective function (3.2). Furthermore, *time fixes* are applied to all plans into the same transition neighborhood which may result in larger dissimilarity decrements (i.e. shorter paths).

A *time fix* is applied to a mismatched lot (m, x'_{mt}, t) by starting time period: $t \neq t'$. First, basic previous and next extension modifications (Paragraph 4.3.2) are performed to the current and its previous production lots in order to make $t = t'$. Additionally, specialized Left Shift and Right Shift operators (Paragraph 4.3.2) are applied to the mismatched lot with $\Delta = |t - t'|$. Finally, the *time fix* is forced by removing old and inserting new integral setting variables which may cause infeasibility. Generated schedules are added to the current transition neighborhood and a *size fix* is performed to all of them.

A *size fix* is applied to a mismatched lot (m, x'_{mt}, t) by ending time period (i.e. $t = t'$, but $\max_{k \geq t} \{k \in T \mid y_{mjk} = 1, j = x'_{mt}\} \neq \max_{k \geq t'} \{k \in T \mid y_{mj'k} = 1, j' = x'_{mt'}\}$). First, a basic previous or next extension modification (Paragraph 4.3.2) is performed to the next and the current production lots in order to fix the end time period of the mismatched lot. Additionally, a specialized Left Shift or Right Shift operator (Paragraph 4.3.2) is applied to the mismatched lot with appropriate Δ. Finally, a *size fix* is forced by removing old and inserting new integral setting variables which may cause infeasibility. Generated schedules are added to the current transition neighborhood.

5.1.4 Path Exploration

GRASP metaheuristic dose not have deterministic mechanisms for escaping current local optimum as the tabu list in Tabu Search (Glover [50]). This is the reason why initial solutions should be randomized in order to explore more regions of the whole search space. However, due to the greediness of the construction heuristic (in Section 4.2) variability of initial solutions for Local Search (in Section 4.3) may not be sufficient. Path relinking may complement randomization of the construction by solution combination, resulting in a very powerful diversification of the search. If a solution from an unvisited subarea of the search space is generated in a path it may be used in order to explore this area. Therefore, once a path is generated its intermediate solutions should be examined.

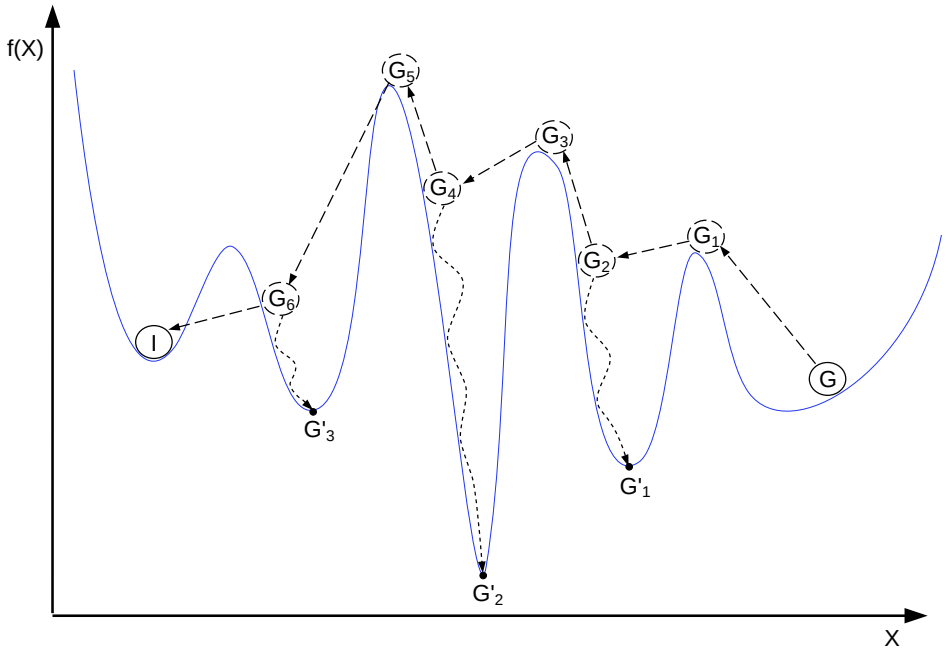


Figure 5.4 A constructed path should be explored. The backward path from G to I contains two *local path minimums*: G_2 and G_4 (i.e. $f(G_1) > f(G_2) < f(G_3)$ and $f(G_3) > f(G_4) < f(G_5)$). Local Search heuristic is applied to all *local path minimums*. Additionally, the best intermediate solution G_6 is also improved.

A path may contain a large number of intermediate solutions. If all of them are to be explored performance will be influenced significantly (Boudia [91]) (recall that "most of the execution time of GRASP is consumed by improvement", Paragraph

4.3). In contrast, most hybridizations of GRASP and Path Relinking (Pitsoulis [77], Aiex [92, 93], Resende [72, 90], Nasimento [24]) examine only the best intermediate solution. However, a path may visit several subareas with worse quality intermediate solutions. Hence, a new path exploration approach is introduced where all feasible intermediate *local path minimums* G_i, such that $f(G_{i-1}) > f(G_i) < f(G_{i+1})$ (for a minimization problem) are examined (see Fig. 5.4). Additionally, also the best intermediate solution is a subject for improvement.

Local path minimums are good candidates for improvement because they indicate that the path first "went down" in a subregion, stopped at such an intermediate solution, and then started to "climb up" the subregion. This indicates that improvement starts from a relatively good quality solution which results in fewer computational effort. Furthermore, the average number of such *local path minimums* in a path is rather small. The reason for this is that, paths are generated between "local minimums" and thus first, they have to "climb up" the corresponding local areas at the path's edges. Therefore, the new strategy introduces a relatively small additional computational effort.

To see the global effect of the exploration strategy consider Fig. 5.1. Forward Relinking produces a path that has some infeasible solutions but most of the feasible solutions have rather better objective values than the intermediate results of Backward Relinking. However, Backward Relinking generates solutions in some subregions of the search space that are not accessible by the forward path. In Fig. 5.4 the same backward path is explored, resulting in two local optima G'_1, G'_3 and the global optimum G'_2. Notice that, the global optimum G'_2 is not accessible by exploration of the forward path in Fig. 5.1.

Path Relinking is an effective approach for generating diverse solution combinations across the search space. Similarly to Scatter Search (Glover [89]), an improvement heuristic should be applied to intermediate trial solutions in the path in order to transform them into enhanced trial solutions. However, the time required for a more intensified improvement to be applied to all path's intermediate solutions can be used to execute relinking on a larger set of elite solutions. Hence, the proposed exploration strategy tries to find the balance between intensification (that may lead into the same search subareas) and diversification (that would generate more solutions across the search space). Nevertheless, more advanced metaheuristic techniques may benefit from controlling dynamically this balance.

5.1.5 Pool of Elite Solutions

One of the founding principles of Scatter Search says that: "Useful information about the form (or location) of optimal solutions is typically contained in a suitably diverse collection of elite solutions." (Glover [89]). The principle also applies to Path Relinking and, since different "local optima" are good candidates for such elite solutions GRASP is relevant. Indeed, local optimum subareas are natural clusters of neighborhood points with relatively good objective values that are usually located far from each other. Furthermore, the global optimum should be located in such an area. Therefore, maintaining adaptive memory of this information in a pool of elite solutions throughout the search process may result in finding the optimum solution at the end.

A pool P consists of $pool\_size$ $(= |P|)$ number of elite, feasible solutions found in the search process. Initially, P is empty. Next, it is filled by improved solutions returned by iterations of the Basic GRASP. After this initial phase, candidate elite solutions X_c found by the algorithm are tested for acceptance in the pool. Denote the best elite solution with $X_{best} = \arg\max \{f(X) \mid X \in P\}$ and the worst elite solution with $X_{worst} = \arg\min \{f(X) \mid X \in P\}$ in P. Several strategies for updating the pool are described in the literature.

Fluerent and Glover [94] use a *worst replacement* strategy. A candidate elite solution X_c is accepted in the pool, if:

1. $f(X_c) < f(X_{best})$, i.e. X_c is the new best solution found so far, or
2. $f(X_{best}) < f(X_c) < f(X_{worst})$ and for all elite solutions $X \in P, D(X_c, X) > \delta$, where $D(X_c, X)$ is the dissimilarity measure from Section 5.1.2 and δ is an additional parameter, i.e. e_c is better than the worst solution and is significantly dissimilar from all solutions in the pool (Pitsoulis [77]).

If a candidate solution is accepted in the pool, it will replace X_{worst} in order to ensure diversity of elite solutions, dynamically.

Another *most-similar replacement* strategy was reported by Resende [72, 90] to increase the diversity in the pool and was used by Boudia [91]. A candidate elite solution X_c is accepted in the pool, if $f(X_c) < f(X_{worst})$; i.e. $\exists\ X_{worse} \in P$, such that $f(X_c) < f(X_{worse}) \leq f(X_{worst})$, but X_{worse} is not necessarily the same as X_{worst}. Once a candidate X_c is accepted in the pool, it is replaced by the most similar elite solution $X_{like} = \arg\min_{X \in P} \{D(X_c, X) \mid f(X_c) < f(X)\}$ among all worse solutions in the pool.

Other strategies involve evolutionary approaches (Andrade [95]), where lower-quality solutions are forced out of the elite pool when they have stayed there for a long time. However, this paper uses a *most-similar replacement* strategy only to "local optima" (returned by Local Search) as elite candidates for insertion into the pool.

5.2 Structure of GRASP with Path Relinking

Path relinking is applied within GRASP in the following ways:

- periodically, to a "local optimum" X_l obtained during GRASP iterations;
- finally, to all pairs of elite solutions as a post optimization step;

(Resende [72, 79, 90], Boudia [91]). Some of the implementations consider executing path relinking to every local optimum. Alternatively, other implementations suggest periodic optimization of the pool during GRASP iterations (Pitsoulis [77]). Post-optimization is sometimes seen as an evolutionary process (Resende [96], Andrade [95]).

The proposed hybridization in Algorithm 5.1 applies path relinking periodically to a local optimum X_l and a selected elite solution X_e from the pool at line 10. Additionally, post-optimization is performed by relinking all subsets of two solutions $\{\{X, Y\} \mid X \neq Y, X, Y \in P\}$ in the pool at line 15. The process is restarted as soon as an improvement is detected, otherwise it terminates. Hence, it operates on a finite sequence of different pools $P, P_1, P_2, P_3, \ldots$ with strictly improving best elements.

Two solutions are relinked at line 10 either by using Fore and Backward path construction strategy or by using Mixed Relinking (see Sections 5.1.1 and 5.1.3) (i.e. only symmetric approaches). Every time a path is generated its *local path minimums* are improved by Local Search (Section 5.1.4) and the resulting "local optima" are tested for acceptance in the pool P (Section 5.1.5). Additionally, every "local optimum" generated by the Basic GRASP is used as a candidate for insertion into P at line 7. Thus, only "local optima" produced by Local Search heuristic Algorithm 4.4 may become elite solutions in the pool.

Algorithm 5.1 Structure of GRASP with Path Relinking

Input: *pool_size* - maximum number of elite solutions in the pool
relink_interval - parameter for periodic relinking during GRASP
max_iter - maximum number of iterations
max_term_iter - maximum number of non-improving iterations
Output: X^* - the best solution found

1: $term\_iter = 0$; // number of non-improving iterations
2: Initialize pseudo-random number generator *rand*
3: $P = \emptyset$; // an empty pool with *pool_size* maximum capacity
4: **for** $i = 1 \ldots max\_iter$ **and** $term\_iter < max\_term\_iter$ **do**
5: $X_s = GreedyRandomizedConstruction(rand)$;
6: $X_l = LocalSearch(X_s, rand)$; // with randomized *nhood* generation
7: $P \rightarrow Accept(X_l)$;
8: **if** $i \,\%\, relink\_interval = 0$ **then**
9: $X_e = P \rightarrow SelectEdge(X_l, rand)$; // select an appropriate solution from the pool
10: $X_l = P \rightarrow Relink(X_l, X_e, rand)$;
11: **end if**
12: Update *term_iter* to 0, or increase it with 1 if $X_l = P \rightarrow Best()$;
13: **end for**
14: **repeat**
15: $P = P \rightarrow RelinkAll()$; // post-optimization phase
16: **until** No Further Improvement;
17: $X^* = P \rightarrow Best()$;

When Path Relinking is applied periodically to a "local optimum" X_l, an appropriate second edge $X_e \in P$ for the path needs to be selected from the pool at line 9. The second edge X_e is selected randomly among the most similar elite solution $X_{like} = \arg\min \{D(X_e, X_l) \mid X_e \in P\}$ and the most dissimilar elite solution $X_{diff} = \arg\max \{D(X_e, X_l) \mid X_e \in P\}$ to X_l. This leads to simultaneous intensification into the local area of e_{like} in the first case and to diversification in the second case. Post-optimization phase causes rather diversification than intensification of the search process. However, path exploration complements this by intensifying into new subareas that were made accessible by the path.

GRASP with Path Relinking can be influenced by its parameters in order to balance the time reserved for different aspects of the search. On one hand, more often periodic relinking definitely takes longer but at the same time assures that only enhanced and diverse solutions will be put in the pool. On the other hand, a bigger pool would accept more "local optima" such that when relinked they will increase diversity but post-optimization would require more execution time. The metaheuristic may benefit from careful adjustment of *relink_interval* and *pool_size* parameters.

Another aspect can be influenced by dynamically controlling the selection pressure of Local Search introduced in Section 4.3.1. Improvement heuristic may be adjusted to run faster and produce sub-optimal solutions in the beginning, but it may be enforced to return closer approximations of local optima by consuming more runtime at the end of the execution. Hence, path exploration strategy will give more enhanced results in the post-optimization phase. Similar techniques represent the driving force of Evolutionary Strategies (Rechenberg et al. [80], Schwefel et al. [81]) and have their successful interpretation into Genetic Algorithms (Affenzeller et al. [82]).

Chapter 6

Parallel Implementation

Several parallelizations of GRASP have been reported in literature. Most papers discuss distributed parallel implementations of GRASP with Path Relinking. Two main strategies are reported: collaborative and independent (Aiex [92, 93], Resende [79], Pitsoulis [77], Festa [97]). Furthermore, theoretical results show that the time required by GRASP to reach a given target objective value fits a two parameters exponential distribution (Aiex [98]). Thus, it is possible to approximately achieve linear speedups by multiple independent processors (Aiex [93], Resende [79, 90], Pitsoulis [77]). Appropriate implementations that make efficient use of computer resources may result in linear speedups and very robust parallel algorithms (Resende [79]).

However, since parallelization is not the main purpose of this master thesis only a shared-memory collaborative parallel implementation in Open-MP is presented. This is sufficient in order to show that good speedups can be easily obtained and to discuss various synchronization issues of Parallel GRASP with Path Relinking.

This chapter first, presents aspects of the parallel implementation related to Basic GRASP 4.1. Then, more synchronization details are introduced for the parallelization of GRASP with Path Relinking 5.1. Finally, techniques for handling synchronization overhead with larger number of threads are discussed.

6.1 Parallelization of GRASP

Parallel implementations of the Basic GRASP algorithm are trivial (Festa and Resende [97]). However, a termination criteria for a maximum number of non-improving iterations requires a careful implementation. Furthermore, special care has to be taken into account when initializing the random sequences at each of the processors.

6.1.1 Basic Structure

Algorithm 6.1 Parallel Basic GRASP

Input: max_iter - maximum number of iterations
max_term_iter - maximum number of non-improving iterations
Output: the best solution found X^* and its objective value f^*

```
1:  f* = ∞;
2:  term_iter = 0; // number of non-improving iterations
3:  abort = false;
4:  #pragma omp parallel shared(X*, f*, term_iter, abort)
5:  Initialize pseudo-random number generator rand_p for each processor p
6:  #pragma omp for schedule(dynamic, job_size_grasp)
7:  for i = 1 ... max_iter do
8:    #pragma omp flush (abort)
9:    if !abort then
10:     X_s = GreedyRandomizedConstruction(rand_p);
11:     X_l = LocalSearch(X_s, rand_p);
12:     #pragma omp critical
13:     if f(X_l) < f* then
14:       f* = f(X_l); X* = X_l;
15:       term_iter = 0;
16:     else
17:       term_iter = term_iter + 1;
18:       if term_iter ≥ max_term_iter then
19:         abort = true;
20:         #pragma omp flush (abort)
21:       end if
22:     end if
23:   end if
24: end for
```

The iterations of the Basic GRASP are independent from each other. However, input parameters for the construction phase may need to be different or even varied dynamically for separate iterations. Hence, Reactive GRASP may require additional synchronization for accessing and updating the memory for values of parameters α, β and coefficients β_t, β_v. Nevertheless, Basic GRASP requires only one global variable in

order to store the best solution found among all processors. Finally, a good scheduling strategy that distribute sufficient, equal portions of work should be introduced.

Algorithm 6.1 illustrates a simple shared memory parallelization of GRASP in Open-MP. Appropriate values for scheduling parameter *job_size_grasp* are discussed in Section 6.3.

6.1.2 Termination Criteria

If GRASP is designed to terminate only after a fixed number of iterations no additional synchronization should be introduced. However, this is not the case when a termination criteria of a maximum number of non-improving iterations is installed. The Basic GRASP Algorithm 4.1 uses both criteria with a logical *or* between them (rather than a logical *and*). Note that, if the maximum number of non-improving iterations have been reached by one processor, all other processors will have to terminate.

Breaking out of a loop in OpenMP is not possible. Canceling work and throwing an exception in a parallel region are not allowed according to the language specification. Hence, other mechanisms need to be developed. The implementation in Algorithm 6.1 is based on looping through empty cycles once the termination condition becomes *true* on any of the threads. It requires an additional shared variable *abort*, such that all threads should have the same view of its value at the beginning of each iteration. However, this is a workaround designed by Süs et al. [99] that conforms to OpenMP specification. To see a real solution for this problem refer to Süs et al. [100].

6.1.3 Random Number Generator

Quality and efficiency of Basic GRASP metaheuristic depends on randomization capabilities of the algorithm. Diversification of the search can be enforced only by allowing semi-greedy construction to produce effectively different initial solutions. Local search can locate an improving solution much faster by examining different parts of the operator's neighborhood than by evaluating the same parts several times. Both these issues can be positively influenced by a good generator of sequences of different numbers. Although a guided search is based on other principles than a pure Monte Carlo Simulation, efficient randomization may improve its capabilities.

Computers are not capable of producing a true random sequence of numbers due to their usage of finite precision number arithmetics (Knuth et al. [101]). Pseudo-

Random Number Generators (PRNG) are deterministic routines which aim at producing a reasonable approximation of a pure random number sequence (Quinn et al. [102]). Typically, such an algorithm is parametrized by a relatively small set of initial values called *seed*. If the same *seed* is given to two instances of a PRNG they will deterministically produce unique sequences. However, since a PRNG can have only a finite number of states it will eventually fall into a cycle. There are various requirements for the resulting sequence (Coddington [103]) but one of the most important is the length of this cycle (called the period).

Popular PRNGs are Linear Congruential Generators (LCG), Multiple Recursive Generators (MRG), Lagged Fibonacci Generators (LFG) and YARN Generators (YARN) (see TRNG documentation by Bauke et al. [104]). However, linear recurrences have certain deficiencies when generating points in a high dimensional space (Marsaglia [105], Quinn et al. [102], Bauke [106]). Mersenne Twister (MT) is another PRNG with a period of $2^{19937} - 1$ introduced by Matsumoto and Nishimura [107] in 1998 which is guaranteed to produce equidistributed sequences in (up to) 623 dimensions. An improved SIMD-oriented Fast Mersenne Twister (SFMT) (Saito [108]) was found appropriate for GRASP metaheuristic on the DLSP discussed in Chapter 3.

Parallel pseudo-random number generators should not introduce any correlation between the different streams, and should have good scalability and locality (see Quinn et al. [102]). There are various parallelization techniques. Some are based on cyclic/block distribution of a single pseudo-random sequence to all processors (i.e. Leapfrog Method/Sequence Splitting). While others generate separate, different pseudo-random sequences on each processor (i.e. Parametrization). Sequence Splitting requires additional initialization in the sequential part of the algorithm and Leapfrog Method may introduce shorter-range correlations than in the original sequence.

Parametrization was chosen in Algortihm 6.1 despite the fact that it may not be completely independent of the underlying hardware (Bauke [106]). However, using the same type of PRNG on all processors, but initialized with "random" seed will not yield non-overlapping and uncorrelated subsequences of the original sequence (Knuth et. al [101], Bauke [106]). Hence, the PRNG itself must be modified to yield a different sequence. Furthermore, a scalable modifying parametrization technique called Dynamic Creation have been successfully applied to Mersenne Twisters (Matsumoto [109]).

The Parallel GRASP implementation uses a C++ software library of portable

multi-platform code for MT and SFMT designed by Fog et al. [110]. Other libraries that deserve attention are Tina's Random Number Generator (TRNG) (Bauke [104]) and The Scalable Parallel Random Number Generators (SPRNG) (Mascagni [111]). It is worth mentioning that TRNG is to be proposed in the forthcoming revision of the C++ standard.

6.2 Parallelization of Path Relinking

Parallelization of GRASP with Path Relinking (PR) requires more synchronization than the parallel version of Basic GRASP. Weather PR is applied periodically during the GRASP iterations, or finally as a post-optimization phase, access to the pool by more than one threads introduces a race condition. Furthermore, this effect is emphasized by the fact that adaptive memory gets updated very often; elite solutions get replaced by better elite solutions and the old once need to be deleted in order to release memory. In addition, the post-optimization process of relinking all solutions in the pool must be restarted every time a better solution is found.

6.2.1 Basic Structure

The parallel implementation of the main cycle of GRASP with Path Relinking has a very similar structure to the parallel version of Basic GRASP. However, all operations related to the pool require careful synchronization. Algorithm 5.1 is modified so that a critical section is enforced every time a solution is tested for acceptance in the pool. This happens not only at line 7 but also in exploration phase of the resulting path from relinking at line 10. Access to the pool requires another critical section when an appropriate solution is selected at line 9 in order to be relinked with the found "local optimum". Finally, the check whether the local best solution found in the current iteration is a new global best solution for GRASP also needs to be synchronized.

Even more synchronization by mutual exclusion needs to be performed in the parallelization of the post-optimization phase. Algorithm 6.2 illustrates the basic structure of Parallel Path Relinking that is applied finally, after all iterations of GRASP have been consumed. Post-optimization phase is implemented as a recursive procedure that tries to relink all subsets of two elements $\{\{X, Y\} \mid X \neq Y, X, Y \in P\}$ contained in the pool. It either restarts (by a recursive call) as soon as an improving solution have been found, or it terminates when paths have been constructed between all elements in the pool and no improving solution was found.

Algorithm 6.2 Parallel Path Relinking

Input: P - the pool of elite solutions

Output: X^* - the best feasible solution that was found

```
1: f* = Evaluate(P → Best()); // the current best value
2: size = P → Size();
3: restart = false; // a flag for recursively restarting the procedure
4: #pragma omp parallel shared(P, f*, size, restart)
5: Initialize pseudo-random number generator rand_p for each processor p
6: #pragma omp for schedule(dynamic, job_size_pr)
7: for k = 0 ... size · (size − 1)/2 do
8:    #pragma omp flush (restart)
9:    if !restart then
10:      {i, j} = GeneratePair(k);
11:      #pragma omp critical
12:      {
13:      X_left = Duplicate(P → Extract(i));
14:      X_right = Duplicate(P → Extract(j));
15:      }
16:      f = P → Relink(X_left, X_right, rand_p);
17:      #pragma omp critical
18:      if f < f* then
19:         f = f*;
20:         restart = true;
21:         #pragma omp flush (restart)
22:      end if
23:   end if
24: end for
25: if restart then
26:   return Restart procedure with updated pool P;
27: else
28:   return X* = P → Best();
29: end if
```

Post-optimization phase operates mainly on the pool of elite solutions, thus it introduces several critical sections. Analogously with the parallel GRASP with Path Relinking maintaining the global best found solution requires a critical section. Ad-

ditionally, mutual exclusion is also introduced when solutions are extracted from the pool in order to be relinked. Relinking itself at line 16 dose not need access to the pool. However, exploration of the constructed path may improve several intermediate solutions and the resulting "local optima" should be tested for acceptance in the pool leading to even more synchronization. Nevertheless, executions in parallel of Path Construction itself and especially of Local Searches in the Path Exploration phase may lead to significant improvement of the runtime.

6.2.2 Linearizing the Loops

Nested parallelism is not supported by all OpenMP implementations and it may be turned off by those that support it. Additionally, it is much harder to tune the work load per thread for nested parallel regions than for a single level parallel region. Algorithm 6.2 operates on all unordered pairs of elite solutions in the pool. Usually, this can be implemented very easily with two nested loops as shown in Code Fragment 6.3 but this would require nested parallelism. In order to prevent unbalanced workload and parallelization overhead for larger number of threads a linearization of the nested loops into a single level loop is proposed as shown in Algorithm 6.2.

Algorithm 6.3 Parallel Nested Loops for Path Relinking

Input: P - the pool of elite solutions

```
1:  size = P → Size();
2:  #pragma omp parallel shared(P, size, ...)
3:  #pragma omp for schedule(dynamic, job_size_pr1)
4:  for i = 0 ... size do
5:     #pragma omp parallel shared(P, size, ...)
6:     #pragma omp for schedule(dynamic, job_size_pr2)
7:     for j = i + 1 ... size do
8:        Generate path/paths for pair {i, j};
9:        ...
10:    end for
11: end for
```

If the number of elite solutions in the pool is $size$ (this is usually the maximum pool size after the main GRASP cycle) then the number of all subsets of two different elite solutions is $N = size \cdot (size - 1)/2$. Furthermore, if the indexes $\{i, j\}$ are to be

generated in a reverse sequence according to Code Fragment 6.3 then 1 pair needs to be generated for $i = size - 2$, 2 pairs need to be generated for $i = size - 3$, ..., $size - 1$ pairs need to be generated for $i = 0$.

Let's denote the reverse index of the current iteration k in Algorithm 6.2 with $k' = N - k$. Hence, for iteration k we can compute the index $i = size - s - 1$, where s is given by the formula:

$$s = \min_{s \in (0, size]} \{s \mid 1 + 2 + \cdots + s \geq k'\}, \tag{6.1}$$

i. e.

$$s = \left\lceil \frac{\sqrt{8 \cdot k' + 1} - 1}{2} \right\rceil. \tag{6.2}$$

Then, the corresponding index j can be obtained by $j = k + 1 + \frac{s \cdot (s+1)}{2} - N$. Finally, a unique pair is generated for all different indexes k and therefore, all pairs of indexes from $\{\{i, j\} \mid i \neq j; i, j \in [0, N)\}$ are generated.

The proposed linearization of the nested cycles from Code Fragment 6.3 generates all pairs of solutions in the pool. It dose not require any additional synchronization. Propagation of the value of *abort* shared variable requires less effort. Additionally, it eases tunning of scheduling parameters (only parameter *job_size_pr*, instead of both *job_size_pr*1 and *job_size_pr*2 needs to be set). This results into a more robust parallel implementation with balanced workload and less synchronization overhead.

6.2.3 Duplication of Data

Typically, adaptive memory is changed very frequently, since it maintains elite and diverse solutions. This introduces a challenge for the parallel implementation of Path Relinking. While one thread has selected a pair of elite solutions and is working to generate a path between them another thread may replace the same solutions by better candidates. At this point the second thread will release the memory for the initial pair of solutions and the first thread will not be able to access them anymore.

The implementation in Algorithm 6.2 solves this problem by creating local copies of the selected solutions for relinking. Duplication of the elite solutions is necessary for the parallel program to run correctly. It dose not introduce any memory overhead since the pool contains only a small number of elite solutions. Hence, the same technique is applied in the main cycle of GRASP when relinking is performed periodically.

6.2.4 Termination Criteria

Post-optimization should terminate after all pairs of elite solutions have been relinked and no improving solution was found. However, as soon as a better solution than the current best found is accepted in the pool the recursive procedure should be restarted with the updated pool. This is accomplished by a shared variable *restart* in a very similar way as described in Section 6.1.2. As soon as an improvement has been detected, first a restart is triggered, then all threads iterate through empty cycles, and finally a recursive call re-initiates the process.

6.3 Synchronization Overhead

When the sizes of the jobs that are executed in parallel are too small the threads would have to wait for each other more often at critical sections. This introduces an overhead of synchronization that can degrade performance of the parallel version significantly. On the other hand, when the number of threads is increased the number of the jobs might not be sufficient to fully load all the processors. In the case of a large number of threads, it is also possible that jobs are too big and when a termination criteria is satisfied on one of the processors it takes additional time until other threads become aware of this. Hence, it is essential that job sizes for parallelization are determined both according to the number of threads and the size of the whole task.

Concerning the main iterations of GRASP, the computation in critical sections is relatively small. In Basic GRASP only the current best solution and some information for the termination criteria needs to be protected by mutual exclusion. However, when Path Relinking is performed regularly during GRASP iterations access to the pool requires significant synchronization. Thus, the following scheme for parallel distribution of work for GRASP iterations is proposed by specification of parameter *job_size_grasp*:

$$num\_threads \in [2^{2\cdot n-1}, 2^{2\cdot n+1}), n \in \mathbb{N} \Rightarrow$$
$$job\_size\_grasp = \left\lceil \frac{\min\left\{2^{2\cdot n-1}, \frac{max\_iter}{relink\_interval}\right\} \cdot relink\_interval}{num\_threads} \right\rceil, \quad (6.3)$$

Since the most significant synchronization is introduced by periodic relinking, different threads should fetch consequent relinking steps. Hence, relink interval plays an important role in determining the size of the jobs to be distributed. Those sizes

must be relatively the same for all threads. However, the fetch size of all threads in total should not exceed the maximum number of iterations.

Concerning post-optimization step of Path Relinking, maintenance of the pool of elite solutions imposes more synchronization overhead threats. Furthermore, path exploration strategy may examine very different number of intermediate path optima which leads to unbalanced jobs sizes. Termination criteria can be triggered very often which causes several restarts of parallel execution. Regular synchronization is required. For larger number of threads the total number of iterations ($\frac{|P|\cdot(|P|-1)}{2}$) should not be exceeded. The parallel distribution of work for Path Relinking is specified by parameter *job_size_pr* in the following way:

$$num\_threads \in (2^n, 2^{n+1}], n \in \mathbb{N}_0 \Rightarrow$$
$$job\_size\_pr = \left\lceil \frac{\min\{2^n, |P|-1\} \cdot \frac{|P|}{2}}{num\_threads} \right\rceil, \tag{6.4}$$

Work load and synchronization overhead are very important issues for an efficient parallel implementation. Thus, appropriate schemes for determining jobs sizes and distributing them among the available computing resources are beneficial for the efficiency of the parallel version of the algorithm. Speedups are discussed in details in the next chapter.

Chapter 7

Computational Evaluation

In this chapter we present results from several experiments with the implemented metaheuristic. First, the testing data set of problem instances is introduced and problem sizes are discussed. Then, a computational comparison of different variations of the algorithm is described. This includes presentation of best and average results, quality charts with trajectories of the optimization process and time-to-target value plots for theoretical assessment of quality of the metaheuristic. Finally, the algorithm is compared to two other approaches: Mixed Integer Programing and Tabu Search, which were independently tuned and implemented for the specific problem configuration.

7.1 Problem Instances (Sizes and Complexity)

The experiments were performed on two different classes of problems: smaller "toy" class with 6 machines, 50 items and 15 time periods and, a "big" class with 21 machines, 272 items and 15 time periods. Three different instances were tested for each of the two classes. Test instances of the original "big" problem were provided by the customer, such that they describe precisely capacities of the available machines and also, their demands were chosen to be hard to satisfy.

Detailed parameter settings for the two problem configurations are shown in Table 7.1. Setting costs are rather expensive although, an additional setup would reduce production quantities itself often leading to unmet demands (as discussed in Section 3.1.4). Backlog costs are accounted for all demanded units that were not produced on time, but the weight for unmet future demands is rather low. Additionally, a setting requires 37.5% ($\approx$ 3 hours) of the available time in a shift. Furthermore, the parallel

factor and the maximum settings per period parameters are adjusted according to the problem sizes.

Parameter	Name	Toy	Big
M	number of machines	6	21
J	number of items	50	272
T	number of periods (shifts)	15	15
s_{mj}	setting costs (for all indexes)	1.0	1.0
c_{jt}	backlog costs (for all indexes)	0.1	0.1
v_{mj}	setting ratio (for all indexes)	0.375	0.375
p_j	parallel factor (for all indexes)	2	3
ω_s	setting weight	1	1
ω_u	backlog weight	1	1
ω_f	future demand weight	$\frac{1}{\sum_{j \in J} f_j}$	$\frac{1}{\sum_{j \in J} f_j}$
ω_k	inventory violation weight	1	1
σ	maximum settings per period	2	6

Table 7.1 Parameter Values for "toy" and "big" problem configurations.

In Section 3.2 it was deduced that a production plan can be uniquely represented by considering only integral setting or state variables. Hence, if non-trivial constraints and production matrices are not regarded and if an integral solution representation is used, the number of all possible setting configurations is limited by $|J|^{|M| \cdot |T|}$. In terms of our "big" problem, integral representation reduces the size of the search space from $2^{21 \cdot 15 \cdot 272}$ to $272^{21 \cdot 15}$ ($\approx 2^{21 \cdot 15 \cdot 8}$). Furthermore, if the maximum number of settings per period constraint is introduced, the number of feasible schedules is reduced to $\binom{|M|}{\sigma} \cdot |T| \cdot |J|^{\sigma \cdot |T|}$ ($\approx 2^{6 \cdot 15 \cdot 8}$). Additionally, parallel production factor as well as production matrix may significantly reduce this number. Moreover, considering only items from external demands limits the search into relevant subareas of the space. Nevertheless, the problem size remains very sensitive to the number of items.

7.2 Comparison of GRASP Variations

In this section different variations of GRASP algorithm with techniques presented throughout the thesis are compared to each other. Results are presented and discussed.

7.2.1 Compared Algorithms

The basic version of GRASP and three other enhanced versions of the algorithm are compared on the datasets. The first enhanced version is a sequential version of GRASP with Fore and Backward Relinking strategy (*GRASP-FBR*) with a regular- and post-optimization phases and path exploration as described in Algorithm 5.1. The second version (also sequential) is very similar to the first, but Mixed Relinking is used instead of FBR (i.e. *GRASP-MR*). The third version is a parallel version of GRASP with Fore and Backward Relinking (*GRASP-PAR*) as described in Algorithm 6.2. Additionally, an improved parallel version (*GRASP-PARi*) with Fore and Backward Relinking that has smaller synchronization overhead is discussed in Section 7.2.3.

A lot of parameters have been introduced throughout the thesis for all implemented heuristics and metaheuristics. For most of them it was sufficient to determine fixed values such that the required behavior is observed. While, others seemed to have more significant impact on the quality of the solution. Nevertheless, more significant parameters can be strategically tuned by alternative metaheuristics during execution. Thus, high parametrization of the basic heuristics might allow for more fine-tuned control of future implementations of GRASP.

Local search parameters did not require too much fine-tuning and optimization. The concept of selection pressure introduced previously was easily applied in practice and thus, randomized local search successfully performs fast improvements until a "local optimum" is reached. However, an evolutionary version of GRASP with Path Relinking may benefit from applying the concept of variable selection pressure during execution. Nevertheless, only fixed values were used for all parameters of Local Search in order to solve the problem of getting "stuck" for both problem configurations and for all variations of the algorithm. They are listed in Table 7.2.

Parameter	Description	Value
max_nhood_size	a maximum local neighborhood size	30
$term\_iterations$	a number of non-improving iterations for termination	30
$nhood\_pressure$	determines when local search gets "stuck"	0.5
$nhood\_coef$	allows candidate solutions for modification in the next iteration	0.015
min_nhood_size	a maximum number of candidate solutions for modification in the next iteration	10

Table 7.2 Parameter Values of the Randomized Local Search Heuristic

Construction heuristic introduces two main parameters that affect the quality of the produced solution by semi-greedy randomization (as discussed in Section 4.2). These parameters are: randomization parameter α for controlling the size of the restricted candidate list (RCL) and parameter β, along with coefficients β_t and β_v for controlling contents of the candidate list (CL). Insertion cost parameters ω_s and ω_k are the same coefficients used in the objective function, coefficient ω_q is fixed to 0.1 and all other coefficients are chosen to be either 1 or 0. However, randomization parameters cannot be simply fixed, since they determine what are the starting solutions for local search and, therefore what are the search subareas that will be explored.

Predetermined fixed values are used for initializing parameters α, β, β_t and β_v. Those values were obtained by preliminary testing on the problem instances. Then during execution, they are randomly varied in a small epsilon region around their initial values.

Execution time of GRASP cannot be predetermined or precisely predicted. However, for the purpose of internal comparison, fixed numbers of maximum iterations and maximum non-improving iterations can be specified. Concerning Path Relinking a suitable relink interval and pool size is also specified. Exact values of these parameters for each problem class are summarized in Table 7.3.

Parameter	Description	Toy	Big
max_iter	maximum number of iterations	1000	200
max_term_iter	maximum number of non-improving iterations	500	100
pool_size	maximum size of the pool	20	20
relink_interval	parameter for periodic relinking	10	5

Table 7.3 Parameter values for determining how long the execution of the metaheuristic should take. These values were used for the purpose of comparison between variations of GRASP.

7.2.2 Results

All GRASP variations were implemented in C++ and SFMT - Mersenne Twister (Fog et al. [110], Saito [108]) was used as a pseudo-random number generator. The parallel implementation was carried out in Open-MP. Computational experiments were performed on a personal computer with Intel Core 2 Duo microprocessor, 2GB of RAM and under the Windows Vista operating system.

In our first experiment we compare basic GRASP algorithm with *GRASP-FBR*, *GRASP-MR* and *GRASP-PAR* variations. A total of 10 runs of every algorithm were executed for each problem instance and detailed results are summarized in Table 7.4.

Prob.	Result	*GRASP*		*GRASP-FBR*		*GRASP-MR*		*GRASP-PAR*	
		Value	Time	Value	Time	Value	Time	Value	Time
	best	59.590	176	37.666	723	39.621	927	36.625	588
*big*1	average	61.496	172	41.208	941	41.295	848	40.357	504
	std. dev	1.165	411	2.184	186	1.371	83	1.937	66
	best	118.69	188	93.576	1605	96.981	1114	92.470	850
*big*2	average	125.12	173	98.334	1138	100.607	1052	97.453	690
	std. dev	4.178	29	3.264	229	2.189	141	2.867	125
	best	263.71	204	245.365	1388	242.324	1599	241.124	558
*big*3	average	275.40	164	247.715	1424	248.812	1068	246.544	777
	std. dev	6.152	34	2.575	365	4.408	236	3.141	156
	best	438.75	48	438.75	85	438.75	53	438.75	43
*toy*1	average	440.09	52	438.75	107	438.76	73	438.75	58
	std. dev	1.081	9	0	16	0.008	12	0	9
	best	26.767	96	21.922	253	22.710	110	21.14	158
*toy*2	average	29.598	71	22.682	264	24.075	164	22.754	139
	std. dev	1.825	16	0.989	44	0.965	26	1.292	14
	best	45.232	116	28.668	211	28.010	185	28.010	114
*toy*3	average	53.028	119	31.715	264	33.071	189	32.054	149
	std. dev	4.155	21	2.559	34	3.405	14	2.580	24

Table 7.4 Results for comparison of GRASP variations. For each problem instance, first the best run and then, the average and the standard deviation of objective value and runtime (in sec.) are shown.

The first observation is that basic GRASP is significantly worse in quality of the obtained solutions than all other variations that use adaptive memory. In most cases, even the best objective values of results obtained by basic GRASP are far from average values obtained by the other approaches that incorporate Path Relinking strategy. Therefore, the combination of GRASP and Path Relinking metaheuristics results in an enhanced algorithm that gives better production plans. Furthermore, one of the reasons for the significant differences in solution quality is adaptive memory and its capability of controlling the search process.

Results for instance *toy1* are different from results obtained for the other instances. The standard deviation is close to 0 on almost all enhanced versions of the algorithm, which brings the assumption that this is the global optimum. The reason for the high quality results for this particular instance is that it was used to inspire the design of local neighborhood operators for Local Search. Hence, neighborhood operators have significant impact on quality of trajectory based approaches.

Another observation is that *GRASP-FBR* is slightly better in terms of solutions quality than *GRASP-MR* in average. This can be explained by the fact that the forward path and the backward path can visit different subareas of the search space and thus, path exploration strategy has the chance to explore more areas. The difference in quality is larger for instances: *big2*, *toy2* and *toy3*, but in total it is not very significant, since these are already very good solutions. However, the average execution time of *GRASP-MR* is better with around 100 seconds than *GRASP-FBR* for almost all of the instances, except *big3* and *toy1*. The reason is that more intermediate path solutions are examined since two times more paths are generated. Notice, that the difference is extremely large for the "toy" instances.

The parallel version *GRASP-PAR* has relatively the same improvement effect over its sequential version *GRASP-FBR*, as *GRASP-FBR* has over *GRASP-MR*. However, the best values obtained by *GRASP-PAR* are among the best values found for these instances by any of the approaches discussed in this Chapter. While this is an abnormal outcome, one possible reason might be that the parallel version uses *num_threads* different Mersenne Twister PRNGs. Apparently, the parametrization scheme used for parallelizing the pseudo-random number sequences effectively generates uncorrelated sequences resulting in better stochastic behavior of GRASP. Executions on more powerful multi-core architectures should verify this behavior.

Effectiveness of the parallelization approach used in this paper should be further studied. However, assessment of the efficiency in terms of execution time is shown in the next section.

7.2.3 Speedup

The importance of appropriate workload and synchronization overhead for the efficiency of parallel implementations of algorithms (and GRASP in particular) was already discussed in Section 6.3. In particular, general schemes for determining values of scheduling parameters *job_size_grasp* and *job_size_pr* were suggested. However, algorithm *GRASP_PAR* dose not make use of these schemes but rather schedules

dynamically only one iteration per thread. In this section, an improved parallel algorithm *GRASP_PARi* that uses schemes (6.3) and (6.4) is compared to its predecessor *GRASP_PAR* and to the sequential version *GRASP_FBR*. Achieved speedups are reported.

Algorithms *GRASP-FBR*, *GRASP-PAR* and *GRASP-PARi* are a sequential, parallel and an improved parallel version of the same algorithm. They were executed with the same parameters in the context of the experiment from Section 7.2.2 and average results are summarized in Table 7.5.

Probl.	Result	*GRASP-FBR*		*GRASP-PAR*		*GRASP-PARi*	
		Value	Time	Value	Time	Value	Time
*big*1	average	41.208	941	40.357	504	39.915	462
	speedup		1		1.867		2.037
*big*2	average	98.334	1138	97.453	690	97.724	552
	speedup		1		1.649		2.062
*big*3	average	247.72	1424	246.54	777	247.43	616
	speedup		1		1.832		2.312
*toy*1	average	438.75	107	438.75	58	438.75	57
	speedup		1		1.845		1.877
*toy*2	average	22.682	264	22.754	139	22.724	132
	speedup		1		1.899		2
*toy*3	average	31.715	264	32.054	149	31.549	152
	speedup		1		1.771		1.737
average speedup			1		1.810		2.004

Table 7.5 Results for comparison of parallel GRASP variations and assessment of achieved speedups. For each problem instance the average objective value and runtime (in sec.) are shown and speedups are computed.

Computational time of *GRASP-PAR* algorithm is better than the computational time of its sequential version. However, the achieved average speedup on two cores (1.81) is not linear. This shows that *GRASP-PAR* algorithm encounters synchronization problems. The improved parallel version *GRASP-PARi* successfully solves these problems and achieves the desired linear speedup (2.004). Still, speedups for some particular instances like *toy*1 and *toy*3 are not satisfactory. Nevertheless, the computational time of GRASP with Path Relinking is rather non-deterministic.

Quality of the new parallel version *GRASP-PARi* is very similar to the previous

parallel version and is often better than the quality of the sequential version. However, since average values are very close to each other, no particular conclusion can be made.

7.2.4 Quality Charts

Quality charts are provided for the purpose of better insight into the optimization process during execution. They contain close approximations of the optimization trajectory in every time period of the execution. Furthermore, the improvement plot shows how the quality of the found solutions develops with the time.

In particular, Figure 7.1 and Figure 7.2 show the optimization trajectories of two sample runs of *GRASP-FBR* and *GRASP-PAR* algorithms on *toy3* problem instance, respectfully. The sequential version takes more than 4 minutes to converge, while the parallel version executes in 1.9 minutes. Multiple restarts of the search happen all the time from initial solutions with average and above average objective function values. These solutions are generated either by the construction heuristic, or they are intermediate local path optima in generated paths during regular relinking or post-optimization.

The post-optimization with Path Relinking can be easily located in both charts at the last 25 seconds of the execution. Furthermore, restarts from initial solutions with low objective value dominate during this time. This indicates that solutions in the pool have very low objective values and thus, their combinations are also high quality schedules. However, in some cases intermediate path optimums have worse objective values which causes higher restarts. The reason for this behavior is that solutions in the pool are also diverse and paths that connect them may visit several other subregions. Because of this behavior, Path Relinking can be extremely efficient for optimization in highly multi-modal objective function landscapes.

7.2.5 Time to Target Value Plots

According to Aiex [98], GRASP and GRASP with Path Relinking have running time to (suboptimal) target value that fits a shifted exponential distribution. Time-to-target (TTT) plots show important information about the quality of stochastic local search procedures by visualizing their running time distributions. A TTT plot is generated by independently running an algorithm several times and measuring the time it takes to find a solution at least as good as a given target solution.

In this second experiment 50 independent runs of *GRASP-PAR* metaheuristic

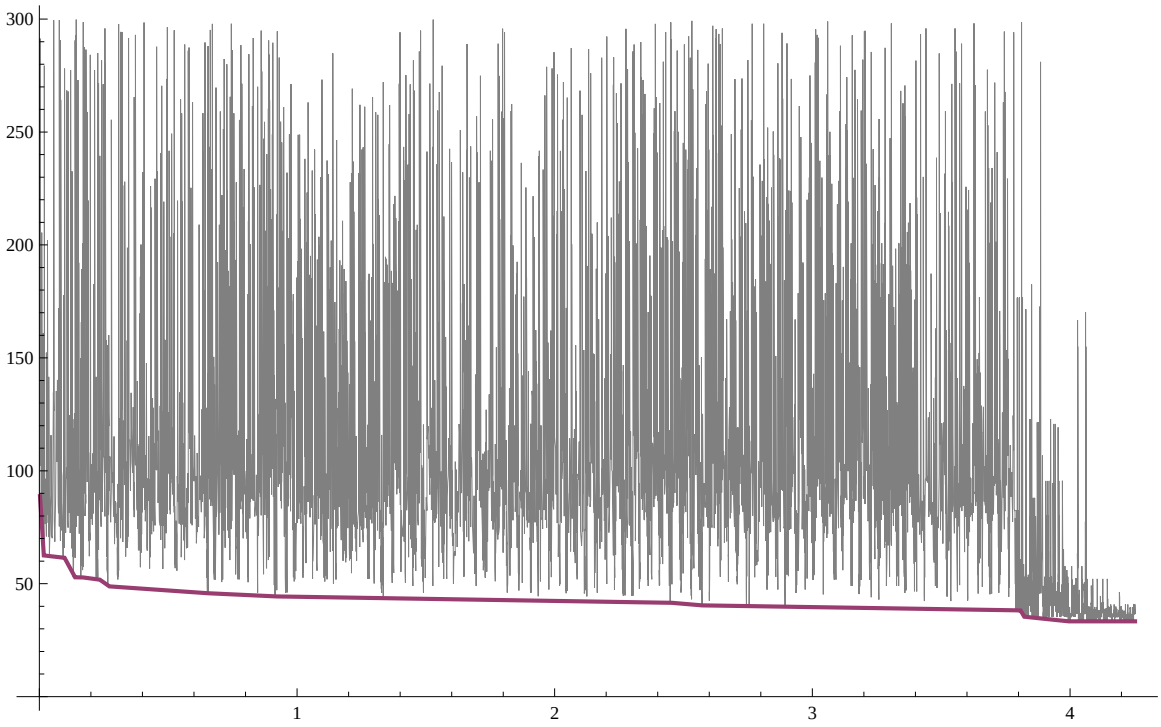


Figure 7.1 A *GRASP-FBR* optimization trajectory for *toy3* problem.

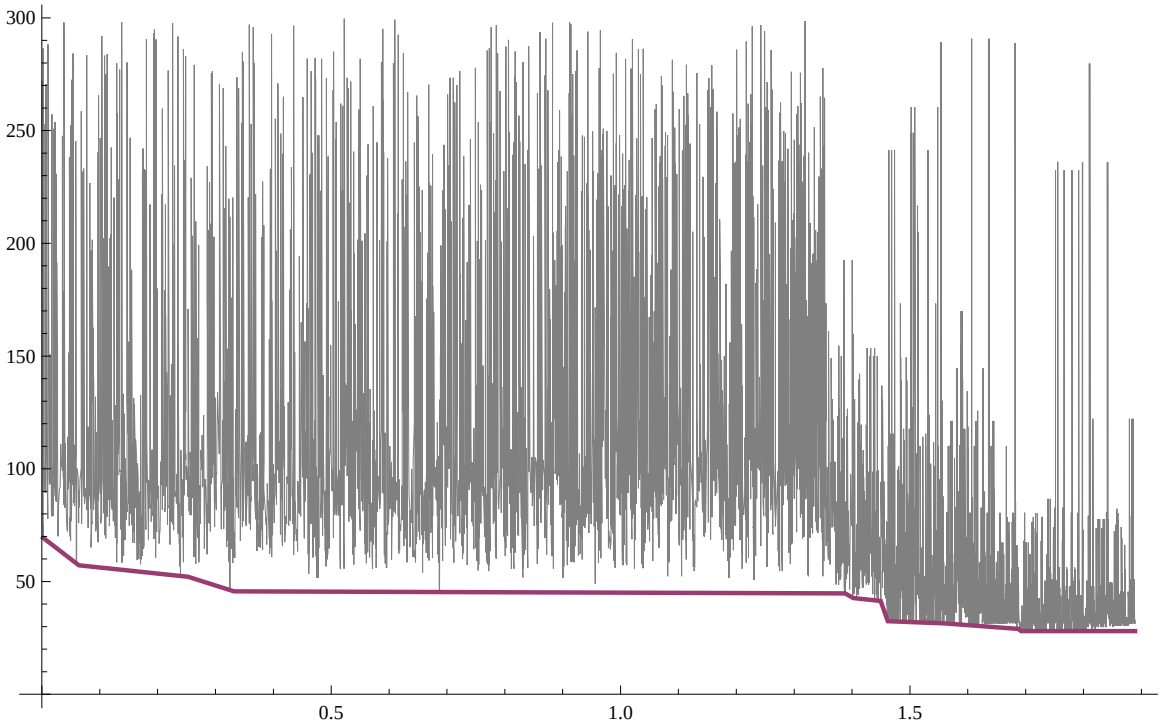


Figure 7.2 A *GRASP-PAR* optimization trajectory for *toy3* problem.

with 5 different randomization parameter configurations were executed on each of the instances of the problem. Two plots are generated for each of the instances - one for a suboptimal value that is reached before post-optimization and one for a better value that requires Path Relinking. The purpose is to study the time distributions for finding different values in different phases of the algorithm. The desired outcome is that a better combination of the two metaheuristics is elicited.

Plots are generated using Aiex [112] and are listed in separate figures 7.3 and 7.4 for each problem class, in order of the instances (two plots for each). They clearly show that better values are reached by Path Relinking in post-optimization phase, since the time distributions for reaching these values are shifted forward in time (see the right hand side figures). Exceptions are the plots for $big3$ and $toy1$ instances, where for each of them the two distributions are very similar. For $big3$ problem, both values are reached in the post-optimization phase, while for $toy1$ problem, values are reached early in the main iterations of GRASP.

The main observation is that, while the time distribution for reaching a suboptimal solution in the main GRASP iterations is indeed exponential, the time distribution for reaching a better solution in post-optimization is again exponential, but with different parameters. The probability for finding at least as good solution as the better solution with Path Relinking grows significantly faster with time. Therefore, Path Relinking contributes a major improvement in quality to the basic GRASP.

In this context, another possible combination of the two heuristics is to let Path Relinking impose on the time distribution to target value of Basic GRASP. While this is achieved by periodic relinking in the current version of the algorithm, another possibility would be to relink all elite solutions, regularly.

However, this approach did not give any significant improvement in preliminary testing. The reason is that when relinking is applied to all solutions from the pool too often, diversity in the pool disappears very fast. Thus, at the post-optimization phase no actual combinations can be obtained by path construction. Nevertheless, regular relinking is a promising direction for future research but it should be applied in combination with other diversification strategies.

Studying further the quality of the combined metaheuristics can give new insights and ideas for better incorporation of Path Relinking into GRASP.

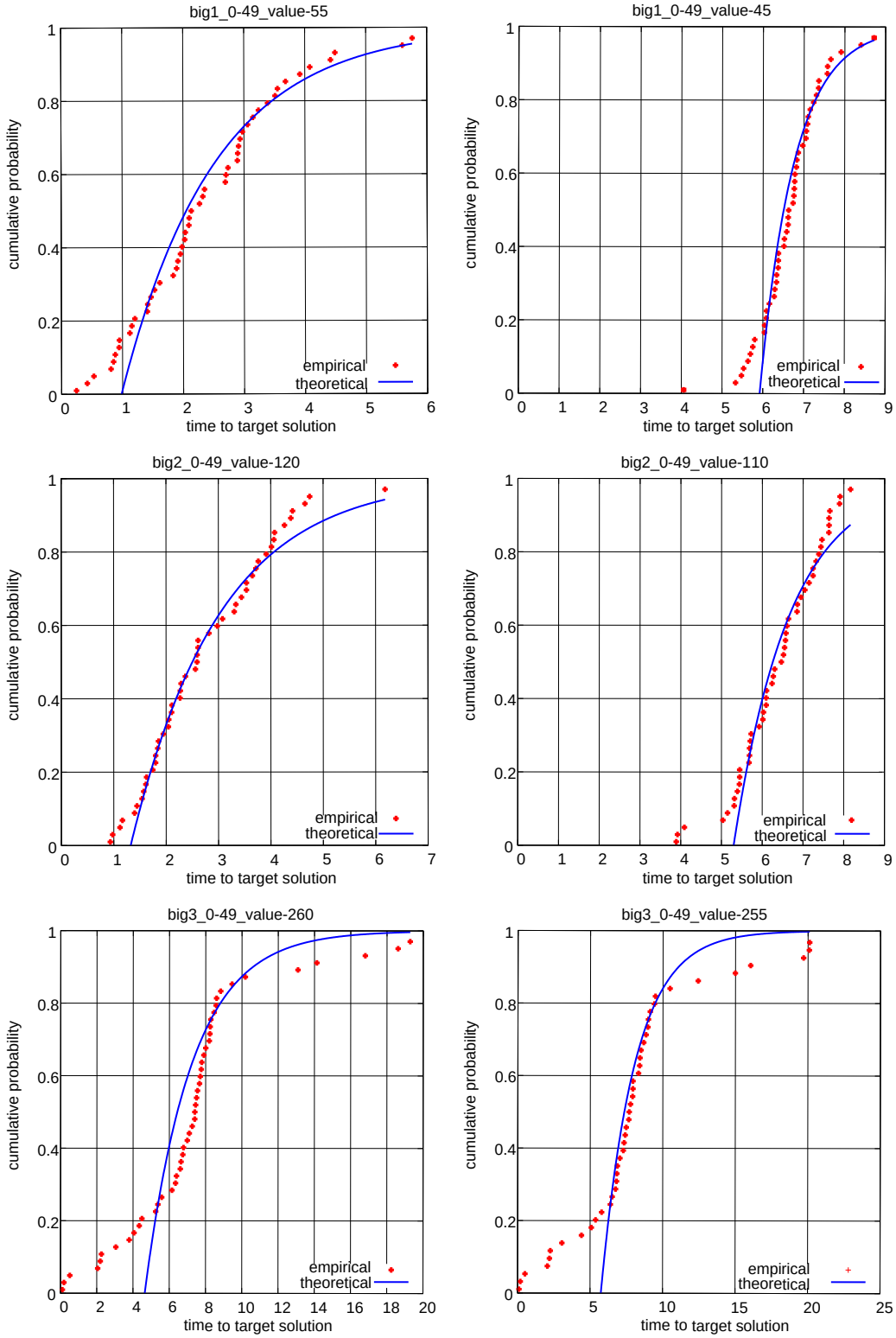


Figure 7.3 Time-to-target plots for instances of the Big problem.

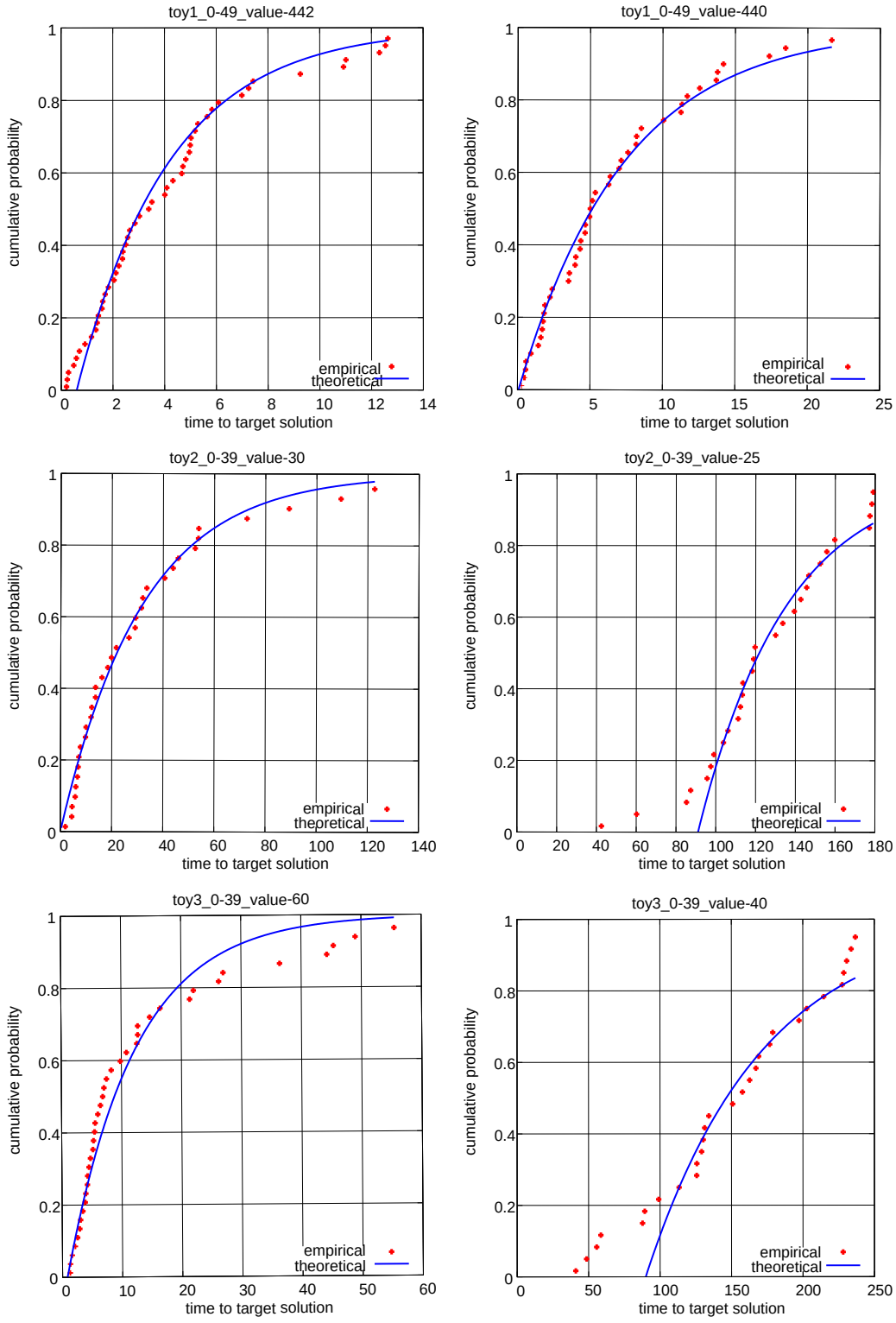


Figure 7.4 Time-to-target plots for instances of the Toy problem.

7.3 Comparison to Other Approaches

In this section, the proposed algorithm is compared to other approaches for assessing quality and robustness of the metaheuristic. These approaches were independently tuned or implemented for the particular problem configuration described in Chapter 3. Furthermore, results obtained with longer execution times are the subject for comparison.

7.3.1 Compared Algorithms

GRASP is compared on the same problem configurations and instances from the previous section against two other approaches: a Mixed Integer Programming (MIP) approach of a custom solver (XPRESMP) and a Tabu Search(TS) approach implemented by Jimborean et al. [6]. These techniques were already shortly presented in Section 2.2 and also, some results of their performance on other small-bucket lotsizing and scheduling problems were described in Section 2.2.4.

However, it is very hard to determine a common bases for comparison of these approaches. TS and GRASP are metaheuristics that produce good quality solutions relatively fast, but they do not provide any information about the global solution. On the other hand, MIP can find tight lower bounds for the optimal objective value and produce close-to-optimal solutions but it requires more execution time for these large problem instances.

In this third experiment the maximum execution time was set to 1 hour for all algorithms. *GRASP-MR* and *GRASP-PAR* variations of GRASP are used for the comparison. Nevertheless, it is very hard to adjust GRASP with Path Relinking so that post-optimization phase finishes always earlier than one hour. Alternatively, termination parameters of GRASP are set so that the runtime would not exceed one hour. They were empirically derived by trial-and-error and are shown in Table 7.6;

Parameter	Description	Toy	Big
max_iter	maximum number of iterations	5000	1200
max_term_iter	maximum number of non-improving iterations	5000	1200
pool_size	maximum size of the pool	30	30
relink_interval	parameter for periodic relinking	10	10

Table 7.6 Parameter values for determining less than 1 hour execution of GRASP with Path Relinking metaheuristic. These values were used for the purpose of comparison between GRASP, *TS* and *MIP*.

7.3.2 Results

Results from the final comparison are shown in Table 7.7. The TS metaheuristic is deterministic and results are reproducible. GRASP metaheuristic is rather non-deterministic and information about average objective value and runtime for 3 runs are provided. MIP has only one run per instance, strictly for a duration of 1 hour. However, the lower bounds that *MIP* provides are not sufficient to prove optimality of the obtained solutions, except in the case of *toy1* instance where the plan with objective value of 438.75 have been proved to be a global optimum by MIP approach.

Probl.	Result	*MIP*		*TS*		*GRASP-MR*		*GRASP-PAR*	
		Value	Time	Value	Time	Value	Time	Value	Time
*big*1	avg.	42.539	60	38.625	65	39.327	47	38.954	25
	best					38.591	47	36.625	10
*big*2	avg.	96.183	60	96.784	60	98.413	44	96.201	23
	best					95.386	42	92.47	14
*big*3	avg.	240.61	60	292.483	69	246.71	42	247.54	20
	best					242.32	27	241.12	9
*toy*1	avg.	439.94	60	438.75	20	438.75	5.76	438.75	3.68
	best	438.75	120			438.75	1	438.75	0.17
*toy*2	avg.	24.909	60	21.922	10	23.525	11	22.208	6.12
	best					21.922	13	21.14	1.3
*toy*3	avg.	37.139	60	37.119	10	29.679	13	29.822	7
	best					28.011	15	28.011	2

Table 7.7 Results for comparison with other approaches. For each problem instance and each algorithm several runs for approximately one hour are summarized by their average objective value (Value) and runtime (Time) in minutes. Additionally, the best results obtained throughout testing for either less or more time are also provided.

Both a sequential and parallel versions are compared because the parallel version tends to give better results. Indeed, *GRASP-PAR* is better in quality than *GRASP-MR* in most of the cases, but there are some exceptions: *big*3 and *toy*3 instances. Additionally, the best values obtained by the corresponding algorithm throughout all experiments are specified.

GRASP-PAR is better than *MIP* in all other cases except for *big*2 and *big*3 instances. However, the average value of 96.201 for problem *big*2 is very close to the solution obtained by MIP. Furthermore, the best value for this instance shows that this is not the optimal solution. Undoubtedly, *MIP* finds a better solution for *big*3 problem.

GRASP-PAR and *TS* approaches yield production plans with very close objective values in average, except for instances *big*3 and *toy*3. For these instances *TS* gives significantly worse results. This indicates that in this particular cases Tabu Search cannot escape from local optima or it cannot direct the search into subareas with better solutions like the once found by GRASP.

Another aspect of this experiment is concerned with the longer computational time. It was already shown that GRASP with Path Relinking can give relatively good quality results for shorter computational time (< 20 min). These results are very close in average quality to the results obtained for less than 1 hour (≈ 40 min) and they even contain some of the best solutions found. Hence, there is no significant improvement when longer runtime is considered. One possible reason might be that execution parameters were not adjusted accordingly for longer executions. However, another possible explanation might be that longer executions need additional mechanisms for maintaining diversity in the pool.

In conclusion, GRASP with Path Relinking is a competitive and very robust methodology with respect to Tabu Search and MIP for the DLSP problem. In some cases it gives better schedules. Furthermore, shorter executions result in relatively good quality plans.

Chapter 8

Conclusion

This paper addresses a Discrete Lotsizing and Scheduling Problem (DLSP) on parallel machines with extremely large instances. In order to tackle the complexity of the problem, a mixed integer formulation with additional non-trivial constraints is introduced. A GRASP with Path Relinking approach is proposed for obtaining good quality solutions for large instances of the problem in short time. The algorithm is parallelized in Open-MP for the purpose of execution on modern multi-core personal computers. Results are competitive (and even better in some cases) with respect to a Mixed Integer Programming approach and a Tabu Search approach that were independently tuned and designed for the particular problem.

8.1 Achievements and Contributions

The initial problem considered in this paper (and introduced by the customer) consists of 21 machines, 272 items and 15 time periods. It is larger in size than all of the DLSP instances that can be observed in literature. Thus, a compromise with the problem specification that reduces the number of feasible solutions is suggested by our industrial advisers from "RISC Software Gmbh". Non-trivial constraints for limiting parallel production of an item on multiple machines and maximum number of settings in a time period are introduced. The resulting specific mixed integer formulation cannot be found in literature.

Although GRASP is a well known metaheuristic, its components need to be designed very carefully. Semi-greedy construction and local search are generic heuristics on their own that depend significantly on the choice of candidate elements for construction and on the neighborhood structure for improvement. Path Relinking is a

metaheuristic framework and thus, its overall quality also depends on the particular implementations of path construction, maintenance of the pool and path exploration. Furthermore, combining the two metaheuristics is a very challenging task and it requires special interpretation of the concept of adaptive memory.

In particular, the construction heuristic is implemented as a demand driven semi-greedy approach that successfully generates feasible solutions. The fact that demands are used as candidate elements that trigger production reduces the search space only into areas of relevant items. Furthermore, greediness is especially enforced in conformance to the non-trivial limiting constraints, resulting in initial solutions with quality better than the average. On the other hand, randomization is achieved according to a value based scheme for determining contents of both the candidate list (CL) and the restricted candidate list (RCL). Appropriate parametrization allows the metaheuristic to control more precisely randomized sampling of the search space.

Local search implementation benefits from a well structured framework of originally designed and implemented basic modifications and neighborhood operators for production plans. These specialized procedures represent the key steps in implementing a good trajectory based optimization heuristic. However, since generated transition neighborhoods are relatively large a randomized neighborhood exploration strategy is proposed. This strategy makes local search rather fast, but special care needs to be taken for choosing a termination criteria. For this purpose a new selection pressure concept is introduced for randomized local search that successfully prevents premature termination due to neighborhood structure.

Applying Path Relinking in combination with GRASP is a major enhancement of the basic GRASP heuristic. However, transition neighborhoods for path generation play a very important role for obtaining different combinations of elite solutions. Thus, the set of item, time and size fixes (of production plans), designed for introducing attributes of the guiding solution into the initial solution is also original. In addition, a new path exploration strategy which examines only "local path optima" is suggested. Relinking is performed both regularly an finally as a post-optimization step in combination with basic GRASP. Finally, maintenance of the pool of elite solutions throughout the execution of the heuristic requires an appropriate interpretation of adaptive memory.

A shared-memory parallel implementation of the enhanced metaheuristic is also implemented in Open-MP. The parallel version is especially designed for handling synchronization overhead more easily. Close to linear speedups are obtained for two

cores. Surprisingly, the parallel version often gives results that are better in quality. While the reason for this obscure observation is still unclear, one possible cause might be the parallel parametrization scheme of Mersenne Twister pseudo-random number generator.

Concerning the computational evaluation of the algorithm, both internal comparison of variations of GRASP and external comparison to Mixed Integer Programming and Tabu Search approaches are provided. Six test instances were provided by "RISC Software Gmbh" company and they contain real data from a customer in the automotive supply chain industry. GRASP with Path Relinking behaves very competitively to both TS and MIP for longer runtime ($\approx 1h$) and in some cases even better schedules are derived. However, the quality of results obtained with shorter runtime ($\approx 10min$) is very close to the one for longer runtime. This indicates that a possible application of the algorithm can be an acceptance criteria for new orders (i.e. additional demands).

In summary, the most important contributions of this paper are:

- obtaining competitive results to TS and MIP approaches;
- appropriate design and implementation of specific local changes and neighborhood operators for local search and path generation;
- suitable parametrization of the metaheuristic components that allows for more fine-tuned control;
- introducing a new concept of selection pressure for Randomized Local Search;
- applying an original path exploration strategy in Path Relinking.

8.2 Difficulties and Limitations

Although GRASP with Path Relinking is relatively independent from the specific heuristics its results may be influenced by the randomization parameters. A lot of effort was spent for finding appropriate initial values for parameters α, β and β_t, β_v of the construction heuristic. However, Pitsoulis [77] reports that a GRASP with a fixed RCL parameter value may not converge (asymptotically) to a global optimum. Therefore, in our implementation these parameter values are randomly varied in appropriate epsilon regions around their initial values during execution.

Nevertheless, the need for adjusting initial values of randomization parameters is one of the main limitations in the project.

Reactive GRASP strategy offers a mechanism of self-adjusting α parameter (Resende [79]). The value of α is chosen from a small discrete set of possible values. During execution, the probabilities that these values get selected are adaptively changed according to the solutions objective function values they tend to produce. Thus, the resulting adaptive memory mechanism can be used by the metaheuristic to incorporate both intensification and diversification in the search process. This technique may be very beneficial to the current implementation of GRASP by eliminating the disadvantage of fixed initial values for randomization parameters.

Alternatively, quality and efficiency of the construction heuristic as well as the improvement heuristic may be increased by isolating effective candidate elements and moves (Glover et al. [55]). Appropriate candidate list strategies may have a major impact on the overall performance of the metaheuristic. However, recency-based memory is a specific mechanism of Tabu Search, but frequency-based memory can be used for both construction and improvement in GRASP. Constructive neighborhoods may improve quality of the initial solutions by introducing elements of consistent variables already at the construction.

Another difficulty was experienced when the algorithm had to be adjusted for longer runtime ($\approx 1h$). The number of iterations of basic GRASP and the pool size had to be adjusted, accordingly. In this case, basic GRASP has to terminate soon enough so that post-optimization with Path Relinking will have time to do the actual improvement in quality. Improper selection of these parameters might be one of the reasons for the non-significant prevalence of solution quality obtained from longer runs to the quality obtained from shorter runs. Additional testing with larger pool sizes and different update schemes of the pool may further prove the robustness of the method in longer runtime.

Testing the parallel version on more than two cores would have been rather advantageous for evaluating scalability in terms of speedup. Furthermore, results from higher scale parallel executions would have justified our observation about the increased effectiveness. Experiments with other parallel schemes like Leapfrog or Sequence Splitting and different PRNG providers would clarify this behavior.

Finally, a great limitation of the methodology itself is that no formal indication of how far the global optimum might be can be obtained. Undoubtedly, the capability of Branch-and-Cut and Column Generation methods that use Lagrangean Relaxation

to provide lower bounds for the global optimum is a major advantage. Nevertheless, the heuristics used to provide upper bounds and concrete solutions in these methods may take longer computational time in their way of proving the global optimum.

8.3 Future Research Directions

Most of the simple heuristics presented in this paper introduce a relatively large number of parameters. However, we argue that appropriate parametrization of simple heuristics can be used by guiding metaheuristics to leverage the interplay of intensification and diversification. In particular Reactive GRASP can control randomized sampling of the search space by parameters of the construction heuristic.

In the same direction, another possibility that is a subject for future research is an Evolutionary GRASP strategy that makes use of the new concept of selection pressure in Randomized Local Search. Variable selection pressure can be used by the higher level metaheuristic to gain more time for diversification in the beginning of the search and to intensify into promising areas in latter stages of the search. In this context, the usefulness of relinking suboptimal solutions can be studied when the solutions are very similar or very dissimilar to each other. GRASP with evolutionary Path Relinking have been interpreted for the purpose of controlling diversity in the pool (Andrade [95]) but it haven't been interpreted in the context of variable selection pressure.

Distributed parallel versions of GRASP with Path Relinking are well documented in literature (Aiex [93]). However, combinations of independent or cooperative distributed strategies and shared-memory parallelizations are a promising new area. Particularly, the interplay between a cooperative strategy where distributed machines are allowed to exchange information about their pools and an evolutionary (shared-memory) Path Relinking strategy is of special interest.

Finally, a formal study of the effect of PRNGs on GRASP and GRASP with Path Relinking is an attractive area for future research. Parallel schemes of PRNGs are especially promising in this context as they might introduce better effectiveness. However, these are topics more concerning stochastic Monte Carlo simulations.

Bibliography

[1] A. Drexl and A. Kimms, "Lot sizing and Scheduling - Survey and Extensions," European J. of Operational Research **99(2),** 221–235(15) (1997).

[2] R. Stainko, "RISC Software Gmbh," Personal communication (2008-2009).

[3] S. V. Hoesel and A. Kolen, "A linear description of the discrete lot-sizing and scheduling problem," European J. of Operational Research **75(2),** 342–353(11) (1994).

[4] C. L. Monma and C. N. Potts, "On the Complexity of Scheduling with Batch Setup Times," Operations Research **37(5),** 798–804(6) (1989), dOI: 10.1287/opre.37.5.798.

[5] T. C. E. Cheng and Z.-L. Chen, "Parallel machine scheduling with batch setup times," Operations Research **42(6),** 1171–1174(3) (1994).

[6] A. Jimborean, "Solving Production Planning Problems using the Tabu Search Metaheuristic (MS's Thesis)," ISI-Hagenberg, Hagenberg im Mülkreis, Upper Austria (2009).

[7] A. Allahverdia, C. T. Ng, T. C. E. Cheng, and M. Y. Kovalyov, "A survey of scheduling problems with setup times or costs," European J. of Operational Research **187(3),** 985–1032(47) (2008).

[8] R. Jans and Z. Degraev, "Meta-heuristics for dynamic lot sizing: A review and comparison of solution approaches," European J. of Operational Research **177(3),** 1855–1875(20) (2007).

[9] C. Gicquel, M. Minoux, and Y. Dallery, "Capacitated Lot Sizing models: a literature review," Technical report, Laboratoire Gènie Industriel (LGI), Ecole Centrale Paris (2008) , hAL: hal-00255830.

[10] M. Salomon, L. G. Kroon, R. Kuik, and L. N. V. Wassenhove, "Some Extensions of the Discrete Lotsizing and Scheduling Problem," Management Science **37(7),** 801–812(11) (1991).

[11] W. Bruggemann and H. Jahnke, "Remarks on: "Some Extensions of the Discrete Lotsizing and Scheduling Problem"," Management Science **43(1),** 122 (1997).

[12] S. Webster, "Remarks on: "Some Extensions of the Discrete Lotsizing and Scheduling Problem"," Management Science **45(5),** 768–769(1) (1999).

[13] J. F. Shapiro, in *Mathematical Programming Models and Methods for Production Planning and Scheduling*, Vol. 4 of *Handbooks in Operations Research and Management Science: Logistics of Production and Inventory*, S. C. Graves, A. H. G. R. Kan, and P. H. Zipkin, eds., (Elsevier Science Publishers B. V., Amsterdam, 1993), pp. 371–443(72).

[14] T. E. Vollman, W. L. Berry, and D. C. Whybark, *Manufacturing Planning and Control Systems*, 4-th ed. (Irwin/McGraw-Hill, New York, 1997), p. 836, iSBN: 9780786312092.

[15] S. C. Graves, in *Manufacturing Planning and Control*, *Handbook of Applied Optimization*, P. Pardalos and M. Resende, eds., (Oxford University Press, New York, 2002), p. 18.

[16] W. W. Trigeiro, L. J. Thomas, and J. O. McClain, "Capacitated Lot Sizing with Setup Times," Management Science **35(3),** 353–366(13) (1989).

[17] B. P. Dzielinski and R. E. Gomory, "Optimal Programming of Lot Sizes, Inventory and Labor Allocations," Management Science **11(9),** 874–890(16) (1965).

[18] S. Kreipl and N. Pinedo, "Planning and Scheduling in Supply Chains," Production and Operations Management **13(1),** 7792(15) (2004).

[19] D. C. Chatfield, "The economic lot scheduling problem: A pure genetic search approach," Computers and Operations Research **34(10),** 2865–2881(16) (2007).

[20] K. Huang and S. Ahmed, "A stochastic programming approach for planning horizons of infinite horizon capacity planning problems," European J. of Operational Research (In Press), doi:10.1016/j.ejor.2008.12.009.

[21] C. S. Sung, "A single-product parallel-facilities production-planning model," International J. of Systems Science **17(7),** 983989(6) (1986).

[22] B. Karimi, S. M. T. F. Ghomi, and J. M. Wilson, "The capacitated lot sizing problem: A review of models and algorithms," OMEGA **31(5),** 365378(14) (2003).

[23] S. C. Graves, "A Review of Production Scheduling," Operations Research **29(4),** 646–675(29) (1981).

[24] M. C. V. Nasimento, M. G. C. Resende, and F. M. B. Toledo, "GRASP Heuristic with Path-relinking for the Multi-plant Capacitated Lot Sizing Problem," European J. of Operational Research (In Press), doi:10.1016/j.ejor.2009.01.047.

[25] G. D. Eppen and R. K. Martin, "Solving Multi-Item Capacitated Lot-Sizing Problems Using Variable Redefinition," Operational Research **35(6),** 832–848(16) (1987).

[26] P. Stadelmeyer, "RISC Software Gmbh," Personal communication (2008-2009).

[27] X. Y. Zhu and W. E. Wilhelm, "Scheduling and lot sizing with sequence-dependent setup: a literature review," IIE Transactions **26(2),** 987–1007(20) (2006).

[28] H. C. Bahl, L. P. Ritzman, and J. N. D. Gupta, "Determining lot sizes and resources requirements: a review," Operations Research **35(3),** 329345(16) (1987).

[29] F. Sahling, L. Buschkuhl, H. Tempelmeier, and S. Helber, "Solving a multi-level capacitated lot sizing problem with multi-period setup carry-over via a fix-and-optimize heuristic," Computers and Operations Research **36(9),** 2546–2553(7) (2009).

[30] A. Jones and L. C. Rabelo, "Survey of Job Shop Scheduling Techniques," Md, NISTIR, National Institute of Standards and Technology, Gaithersburg (1998) .

[31] M. Salomon, M. M. Solomon, L. N. V. Wassenhove, Y. Dumas, and S. Dauzere-Peres, "Solving the discrete lotsizing and scheduling problem with sequence dependent set-up costs and set-up times using the Travelling Salesman Problem

with time windows," European J. of Operational Research **100(3),** 494–513(19) (1997).

[32] R. Jans, "Solving Lotsizing Problems on Parallel Identical Machines Using Symmetry Breaking Constraints," Erim report series reference no. ers-2006-051-lis, Erasmus University Rotterdam (EUR) - RSM Erasmus University, HEC Montreal (2006) , available at SSRN: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1101146 (Accessed on 18 June, 2009).

[33] C. Gicquel, N. Miegeville, M. Minoux, and Y. Dallery, "Discrete lot sizing and scheduling using product decomposition into attributes," Computers and Operations Research **36(9),** 2690–2698(8) (2009).

[34] B. Fleischmann, "The discrete lot-sizing and scheduling problem," European Journal of Operational Research **44(3),** 337–348(11) (1990).

[35] G. Belvaux and L. A. Wolsey, "Lot-sizing problems: modelling issues and a specialized branch-and-cut system BC-PROD," CORE Discussion Papers 1998049, Universite catholique de Louvain, Center for Operations Research and Econometrics (CORE), (1998) , http://ideas.repec.org/p/cor/louvco/1998049.html (Accessed June 3, 2009).

[36] C. Jordan and A. Drexl, "Discrete Lotsizing and Scheduling by Batch Sequencing," Management Science **44(5),** 698–713(15) (1998).

[37] N. Balakrishnan, J. J. Kanet, and S. V. Sridharan, "Early/tardy scheduling with sequence dependent setups on uniform parallel machines," Computers & Operations Research **26(2),** 127–141(14) (1999).

[38] N. Absi and S. Kedad-Sidhoum, "The multi-item capacitated lot-sizing problem with setup times and shortage costs," European J. of Operational Research **185(3),** 1351–1374(23) (2008).

[39] A. Klosea and S. Gortz, "A branch-and-price algorithm for the capacitated facility location problem," European J. of Operational Research **179(3),** 1109–1125(16) (2007).

[40] M. Diaby, H. C. Bahl, M. H. Karwan, and S. Zionts, "Capacitated lot-sizing and scheduling by Lagrangean relaxation," European J. of Operational Research **59(3),** 444–458(14) (1992).

[41] D. Cattrysse, M. Salomon, R. Kuik, and L. N. V. Wassenhove, "A Dual Ascent and Column Generation Heuristic for the Discrete Lotsizing and Scheduling Problem with Setup Times," Management Science **39(4),** 477–486(10) (1993).

[42] R. Jans and Z. Degraev, "Meta-heuristics for dynamic lot sizing: A review and comparison of solution approaches," IIE Transactions **36(1),** 47–58(11) (2004).

[43] A. Hiroaki, M. Susumu, and I. Jun, "A Column Generation Approach for Discrete Lotsizing and Scheduling Problem on Identical Parallel Machines," Journal of Japan Industrial Management Association **55(2),** 69–76(7) (2004).

[44] L. Tang and G. Liu, "A mathematical programming model and solution for scheduling production orders in Shanghai Baoshan Iron and Steel Complex," European J. of Operational Research **182(3),** 1453–1468(15) (2007).

[45] H. Sural, M. Denizel, and L. N. V. Wassenhove, "Lagrangean relaxation based heuristics for lot sizing with setup times," European J. of Operational Research **194(1),** 51–63(12) (2009).

[46] P. J. Billington, J. O. McClain, and L. J. Thomas, "Mathematical Approaches to Capacity-Constrained MRP Systems: Review, Formulation and Problem Reduction," Management Science **29(10),** 1126–1141(15) (1983).

[47] M. Caserta and E. Q. Rico, "A cross entropy-Lagrangean hybrid algorithm for the multi-item capacitated lot-sizing problem with setup times," Computers and Operations Research **36(2),** 530–548(18) (2009).

[48] K. Akartunali and A. J. Miller, "A heuristic approach for big bucket multi-level production planning problems," European J. of Operational Research **193(2),** 396–411(15) (2009).

[49] E.-H. Aghezzaf, "Production planning and warehouse management in supply networks with inter-facility mold transfers," European J. of Operational Research **182(3),** 1122–1139(17) (2007).

[50] F. Glover and M. Laguna, *Tabu Search* (Kluwer Academic Publishers, Boston, 1997), p. 404, iSBN: 9780792381877.

[51] L. Ozdamar and S. Birbil, "Hybrid heuristics for the capacitated lot sizing and loading problem with setup times and overtime decisions," European J. of Operational Research **110(3),** 525–547(22) (1998).

[52] S. Minner, "A comparison of simple heuristics for multi-product dynamic demand lot-sizing with limited warehouse capacity," International J. of Production Economics **118(1),** 305–310(5) (2009).

[53] F. Glover, "Tabu search - Part I," ORSA Journal on Computing **1(3),** 190–206(16) (1989).

[54] F. Glover, "Tabu search - Part II," ORSA Journal on Computing **2(1),** 4–32(28) (1990).

[55] F. Glover, in *Advances in metaheuristics, optimization and stochastic modeling technologies*, *Interfaces in Computer Science and Operations Research*, R. S. Barr, R. Helgason, and J. L. Kennington, eds., (Kluwer Academic Publishers, Boston, MA, 1996), Chap. Tabu search and adaptive memory programming: Advances, applications and challenges, p. 175(75).

[56] F. Carvalho, A. S. Pereira, M. Constantino, and J. P. Pedroso, "Tabu Search for a Discrete Lot Sizing and Scheduling Problem," In , Proceedings of the 4th Metaheuristics International Conference (MIC2001) (Porto, Portugal, 2001).

[57] A. Pereira, E. Carvalho, M. Constantino, and J. P. Pedroso, in *Metaheuristics: computer decision-making*, *Applied Optimization* (Kluwer Academic Publishers, Norwell, MA, USA, 2004), Chap. Random start local search and tabu search for a discrete lot-sizing and scheduling problem, pp. 575–600(25), iSBN: 1-4020-7653-3.

[58] U. Buschera and L. Shen, "An integrated tabu search algorithm for the lot streaming problem in job shops," European J. of Operational Research (In Press), doi:10.1016/j.ejor.2008.11.046.

[59] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," Science **220(4598),** 671–680(9) (1983).

[60] R. Kuik, M. Salomon, L. N. van Wassenhove, and J. Maes, "Linear programming, simulated annealing and tabu search heuristics for lotsizing in bottleneck assembly systems," IIE Transactions **25(1),** 6272(10) (1993).

[61] A. Kimms and A. Kimms, "A Genetic Algorithm for Multi-Level, Multi-Machine Lot Sizing and Scheduling," Computers & Operations Research **26,** 829–848(19) (1996).

[62] D. Jeffcoat and R. Bulfin, "Simulated annealing for resource-constrained scheduling," European J. of Operational Research **70(1),** 43–51(8) (1993).

[63] D. Ferreira, P. M. Franca, A. Kimms, R. Morabito, S. Rangel, and C. F. M. Toledo, "Heuristics and meta-heuristics for lot sizing and scheduling in the soft drinks industry: a comparison study," Computational Intelligence: Metaheuristics for Scheduling in Industrial and Manufacturing Applications **128,** 169–210(41) (2008).

[64] H. Meyr, "Simultaneous lot sizing and scheduling on parallel machines," European J. of Operational Research **139(2),** 277–292(15) (2002).

[65] S. A. Torabi, S. M. T. F. Ghomi, and B. Karimi, "A hybrid genetic algorithm for the finite horizon economic lot and delivery scheduling in supply chains," European J. of Operational Research **173(1),** 173–189(16) (2009).

[66] C. Gicquel, M. Minoux, and Y. Dallery, "On the discrete lot-sizing and scheduling problem with sequence-dependent changeover times," Operations Research **37(1),** 32–36(4) (2009).

[67] A. Drexl and C. Jordan, "Discrete Lotsizing and Scheduling by Batch Sequencing," Management Science **44(5),** 698–713(15) (1998).

[68] M. R. Paula, M. G. Ravetti, G. R. Mateus, and P. M. Pardalos, "Solving parallel machines scheduling problems with sequence-dependent setup times using variable neighbourhood search," IMA Journal of Management Mathematics **18(2),** 101–115(14) (2007), doi:10.1093/imaman/dpm016.

[69] H. Tempelmeier and L. Buschkuhl, "Dynamic multi-machine lotsizing and sequencing with simultaneous scheduling of a common setup resource," International J. of Production Economics **113(1),** 401–412(11) (2008).

[70] K. Brockmann and T. Decker, "A parallel tabu search algorithm for short term lot sizing and scheduling in flexible flow line environments," In , Proceedings of the 16th International Conference on CAD/CAM, Robotics and Factories of the Future (2000).

[71] T. A. Feo and M. G. C. Resende, "Greedy randomized adaptive search procedures," J. of Global Optimization **6,** 109–133(24) (1995).

[72] M. G. C. Resende, "Metaheuristic hybridization with GRASP," At&t labs research technical report, AT&T Labs Research, Florham Park, NJ (2008) .

[73] T. A. Feo, J. F. Bard, and S. D. Holland, "A GRASP for scheduling printed wiring board assembly," IIE transactions **28(2),** 155–165(10) (1996), iSSN 0740-817X.

[74] S. Binato, W. J. Hery, D. M. Loewenster, and M. G. C. Resende, "A greedy randomized adaptive search procedure for job shop scheduling," Technical report, A&T Labs Research, Florham Park, NJ (2000) .

[75] P. L. Rocha, M. G. Ravetti, G. R. Mateus, and P. M. Pardalos, "Exact algorithms for a scheduling problem with unrelated parallel machines and sequence and machine-dependent setup times," Computers and Operations Research **35(4),** 1250–1264(14) (2008).

[76] S. Rojanasoonthon and J. Bard, "A GRASP for Parallel Machine Scheduling with Time Windows," INFORMS J. on Computing **17(1),** 32–51(19) (2005).

[77] L. Pitsoulis and M. G. C. Resende, in *Handbook of Applied Optimization*, P. M. Pardalos and M. G. C. Resende, eds., (Oxford University Press, 2002), Chap. Greedy randomized adaptive search procedures, p. 168181(13).

[78] T. A. Feo and M. G. C. Resende, "A probabilistic heuristic for a computationally difficult set covering problem," Operations Research Letters **8,** 6771(4) (1989).

[79] M. G. C. Resende and C. C. Ribeiro, in *Handbook of Metaheuristics*, *Operations Research & Management Science*, F. Glover and G. Kochenberger, eds., (Kluwer Academic Publishers, 2003), Chap. Greedy randomized adaptive search procedures, p. 219249(30).

[80] I. Rechenberg, *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution* (Reprinted by Frommann-Holzboog (1973), 1971).

[81] H.-P. Schwefel, *Numerische Optimierung von Computer Modellen mittels der Evolutionsstrategie (PhD thesis)* (Reprinted by Birkhäuser Verlag (1977), Basel, Stuttgart, 1974).

[82] M. Affenzeller, "Transferring the Concept of Selective Pressure from Evolutionary Strategies to Genetic Algorithms," Proceedings of the 14th International Conference on Systems Science **2,** 346–353(7) (2001).

[83] M. Affenzeller and S. Wagner, "A Self-Adaptive Model for Selective Pressure Handling within the Theory of Genetic Algorithms," Computer Aided Systems Theory: EUROCAST 2003 pp. 384–393(9) (2003), lecture Notes in Computer Science 2809.

[84] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement Learning: A Survey," Journal of Artificial Intelligence Research **4,** 237–285(48) (1996).

[85] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, Cambridge, MA, 1998), p. 342, iSBN: 978-0-262-19398-6.

[86] J. B. Atkinson, "A greedy randomised search heuristic for time-constrained vehicle scheduling and the incorporation of a learning strategy," J. of the Operational Research Society **49,** 700–708(8) (1998).

[87] C. Fleurent and F. Glover, "Improved constructive multistart strategies for the quadratic assignment problem using adaptive memory," INFORMS J. on Computing **11,** 198–204(6) (1999).

[88] E. D. Taillard, L. M. Gambardella, M. Gendreau, and J.-Y. Potvin, "Adaptive memory programming: A unified view of metaheuristics," European J. of Operations Research **135,** 1–16(16) (2001).

[89] F. Glover, M. Laguna, and R. Marti, "Fundamentals of Scatter Search and Path Relinking," Control and Cybernetics **39(3),** 653–684(21) (2000).

[90] M. G. C. Resende and C. C. Ribeiro, in *Metaheuristics: Progress as Real Problem Solvers*, *Operations Research/Computer Science Interfaces Series*, T. Ibaraki, K. Nonobe, and M. Yagiura, eds., (Springer, 2005), Chap. GRASP with path-relinking: Recent advances and applications, p. 2963(34), iSBN: 978-0-387-25382-4.

[91] M. Boudia, M. A. O. Louly, and C. Prins, "A reactive GRASP and path relinking for a combined productiondistribution problem," Computers & Operations Research **34(11),** 34023419(17) (2007).

[92] R. M. Aiex, S. Binato, and M. G. C. Resende, "Parallel GRASP with path-relinking for job shop scheduling," Parallel Computing **29,** 393430(37) (2003).

[93] R. M. Aiex and M. G. C. Resende, "Parallel strategies for GRASP with path-relinking," Technical report, Internet and Network Systems Research Center, AT&T Labs Research, Florham Park, NJ (2003) .

[94] C. Fleurent and F. Glover, "Improved constructive multistart strategies for the quadratic assignment problem using adaptive memory," INFORMS Journal on Computing **11,** 198–204(6) (1999).

[95] D. V. Andrade and M. G. C. Resende, "GRASP with Evolutionary Path-relinking," At&t labs research technical report, AT&T Labs Research, Florham Park, NJ (2007) .

[96] M. G. C. Resende and R. F. Werneck, "A hybrid heuristic for the p-median problem," J. of Heuristics **10,** 5988(29) (2004).

[97] P. Festa and M. G. C. Resende, in *Essays and Surveys on Metaheuristics*, C. C. Ribeiro and P. Hansen, eds., (Kluwer Academic Publishers, 2002), Chap. GRASP: An annotated bibliography, pp. 325–367(42).

[98] R. M. Aiex, M. G. C. Resende, and C. C. Ribeiro, "Probability distribution of solution time in GRASP: An experimental investigation," Journal of Heuristics **8,** 343373(30) (2002).

[99] M. Süs, "Breaking Out of Loops in OpenMP," http://www.thinkingparallel.com/2007/06/29/breaking-out-of-loops-in-openmp/ (Accessed June 10, 2009).

[100] M. Süss and C. Leopold, in *Euro-Par 2006 Parallel Processing*, Vol. 4128/2006 of *Lecture Notes in Computer Science* (Springer Berlin / Heidelberg, 2006), Chap. Implementing Irregular Parallel Algorithms with OpenMP, pp. 635–644(9).

[101] D. E. Knuth, *Seminumerical Algorithms*, Vol. 2 of *Art of Computer Programming*, 3 ed. (Addison-Wesley Professional, 1997), p. 784, iSBN: 978-0-201-89684-8.

[102] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP* (McGraw-Hill Professional, 2004), p. 529, iSBN 978-0-072-82256-4.

[103] P. D. Coddington and S.-H. Ko, "Techniques for empirical testing of parallel random number generators," In , G. Egan, R. Brent, and D. Gannon, eds., Proceedings of the 12th international conference on Supercomputing pp. 282–288(6) (ACM, New York, USA, 1998).

[104] H. Bauke, "Tinas Random Number Generator Library," http://trng.berlios.de/trng.pdf, p. 120 (2009), accessed June 14, 2009.

[105] G. Marsaglia, "Random Numbers Fall Mainly in the Planes," Proceedings of the National Academy of Science (USA) **61(1),** 25–28 (1968).

[106] H. Bauke and S. Mertens, "Random numbers for large scale distributed Monte Carlo simulations," Physical Review E **75(6),** (14) (2007).

[107] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," ACM Trans. on Modeling and Computer Simulation **8(1),** 3–30 (1998).

[108] M. Saito and M. Matsumoto, "SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator," In , Monte Carlo and Quasi-Monte Carlo Methods 2006 pp. 607–622(15) (Springer, 2008).

[109] M. Matsumoto and T. Nishimura, "Dynamic Creation of Pseudorandom Number Generators," In , Monte Carlo and Quasi-Monte Carlo Methods 1998 pp. 56–69(13) (Springer, 2000).

[110] A. Fog, "Pseudo random number generators: uniform and non-uniform distributions," http://www.agner.org/random/ (Accessed June 14, 2009).

[111] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation," ACM Transactions on Mathematical Software **26,** 436–461(25) (2000).

[112] R. M. Aiex, M. G. C. Resende, and C. C. Ribeiro, "TTTPLOTS: A Perl program to create time-to-target plots," Optimization Letters **1,** 355–366(11) (2007).

List of Figures

List of Algorithms

Index

Eidesstattliche Erklärung

Ich erkläre an Eides statt, das ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäßentnommenen Stellen als solche kenntlich gemacht habe.