Metaheuristics for Group Shop Scheduling

by

Christian Blum, Dipl.-Math.

Université Libre de Bruxelles, IRIDIA
Avenue Franklin Roosevelt 50, CP 194/6, 1050 Brussels, Belgium
cblum@ulb.ac.be

Supervised by

Marco Dorigo, Ph.D.

Maître de Recherches du FNRS
Université Libre de Bruxelles, IRIDIA
Avenue Franklin Roosevelt 50, CP 194/6, 1050 Brussels, Belgium
mdorigo@ulb.ac.be

May, 2002

A thesis submitted in partial fulfillment of the requirements of the *Université Libre de Bruxelles, Faculté de Sciences Appliquées* for the

DIPLOME D'ETUDES APPROFONDIES (DEA)

Abstract

The work presented in this thesis consists of two parts. The first part (Chapters 1 and 2) introduces a general formulation of Shop Scheduling problems, called the Group Shop Scheduling problem (GSP). This problem formulation covers among other Shop Scheduling problems the Job Shop Scheduling problem (JSP) and the Open Shop Scheduling problem (OSP). As both, the JSP and the OSP, are NP-hard combinatorial optimization problems, quite a lot of research has been devoted in the last 10-15 years to the development of metaheuristic methods to tackle them. Metaheuristic methods are approximate methods which combine basic heuristic methods in a higher level framework aimed at efficiently and effectively exploring a search space. Chapter 2 gives a state-of-the-art review on metaheuristic methods developed to tackle the JSP and the OSP.

The second part of the thesis summarizes the research results of the Metaheuristics Network [114] on the development and comparison of metaheuristics to tackle the GSP. The Metaheuristic Network is a Research Training Network funded by the Improving Human Potential program of the CEC. It aims at the comparison of metaheuristics on different combinatorial optimization problems. For each combinatorial optimization problem considered, five metaheuristics are implemented by different persons in different sites involved in the Metaheuristics Network. The five metaheuristics considered are: Ant Colony Optimization (ACO), Evolutionary Computation (EC), Iterated Local Search (ILS), Tabu Search (TS), and Simulated Annealing (SA). The main part of Chapter 3 gives a summary of the research results of the author on the development of ACO algorithms to tackle the GSP. These research results have been published or are about to be published in the following papers:

[17] C. Blum, A. Roli, and M. Dorigo. HC–ACO: The Hyper-Cube Framework for Ant Colony Optimization. In *Proceedings of MIC'2001 – Meta–heuristics International Conference*, volume 2, pages 399–403, Porto, Portugal, 2001. Also available as technical report TR/IRIDIA/2001-16, IRIDIA, Université Libre de Bruxelles.

[18] C. Blum and M. Sampels. Ant Colony Optimization for FOP Shop scheduling: A case study on different pheromone representations. In *Proceedings of the 2002 Congress on Evolutionary Computation, CEC'02 (to appear)*, 2002. Also available as technical report TR/IRIDIA/2002-03, IRIDIA, Université Libre de Bruxelles.

[95] M. Sampels, C. Blum, M. Mastrolilli, and O. Rossi-Doria. Metaheuristics for Group Shop Scheduling. Technical Report TR/IRIDIA/2002-07, IRIDIA, Université Libre de Bruxelles, 2002. Submitted to PPSN'02.

[19] C. Blum and M. Sampels. When Model Bias is Stronger than Selection Pressure. Technical Report TR/IRIDIA/2002-06, IRIDIA, Université Libre de Bruxelles, 2002. Submitted to PPSN'02.

[16] C. Blum. ACO applied to Group Shop Scheduling: A case study on Intensification and Diversification. Technical Report TR/IRIDIA/2002-08, IRIDIA, Université Libre de Bruxelles, 2002. Submitted to ANTS'2002.

The remaining part of Chapter 3 outlines the other metaheuristics developed to tackle the GSP and the comparison of these metaheuristics on various GSP instances.

Acknowledgments

First of all I want to express my thanks to my supervisor Marco Dorigo for giving me the opportunity to work in a very dynamic and stimulating environment. In this context I thank all my colleagues for making IRIDIA a nice place to work at.
Special thanks go to Michael Sampels with whom I cooperated closely on parts of the research results presented, and to Mark Zlochin with whom I had many fruitful discussions in the previous months. Furthermore I thank Andrea Roli for always being open for discussing ideas and for our continuous cooperation.
I'm also very grateful to my parents, sister and brother who are a safe haven for me to which to return to is always a great pleasure.
Last but not least I want to thank Maria Blesa who stands by my side to support me with her love.

Contents

1 Introduction **1**

1.1 Shop Scheduling: The JSP, the OSP and the GSP 2

1.2 Complexity 5

1.3 Benchmark instances 5

1.4 Algorithms 5

1.4.1 List scheduler algorithms 7

1.4.2 The shifting bottleneck procedure 8

1.4.3 Insertion techniques and beam search 9

1.4.4 A matching algorithm combined with packing and inserting 10

2 Metaheuristics for Job and Open Shop Scheduling **12**

2.1 Classification of metaheuristics 14

2.2 Evolutionary Computation 15

2.2.1 EC algorithms to tackle the JSP 16

2.2.1.1 Binary representation 16

2.2.1.2 Permutation representation 17

2.2.1.3 Algorithms without specific representation 18

2.2.1.4 Heuristically guided EC approaches 21

2.2.2 EC algorithms to tackle the OSP 22

2.3 Tabu Search 22

2.3.1 Tabu Search algorithms to tackle the JSP 24

2.3.2 Tabu Search algorithms to tackle the OSP 28

2.4 Simulated Annealing 29

2.4.1 SA algorithms to tackle the JSP 30

2.4.2 SA algorithms to tackle the OSP 32

2.5 Ant Colony Optimization 32

2.5.1 ACO algorithms to tackle the JSP 36

2.5.2 ACO algorithms to tackle the OSP 37
2.6 Other metaheuristic approaches 37
2.6.1 A large-step optimization method to tackle the JSP 38
2.6.2 A GRASP to tackle the JSP 38
2.6.3 A variable depth search method to tackle the JSP 39

3 Metaheuristics for Group Shop Scheduling 40
3.1 Common neighborhood and local search 41
3.2 Ant Colony Optimization 42
3.2.1 A new pheromone model $\mathsf{PH}_{\mathsf{rel}}$ 42
3.2.2 The Hyper-Cube Framework for Ant Colony Optimization 43
3.2.3 $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System for the GSP 44
3.2.4 Intensification and diversification strategies 48
3.2.5 Choice of an ACO algorithm for the comparison 51
3.3 Evolutionary Computation 54
3.4 Iterated Local Search 56
3.5 Simulated Annealing 56
3.6 Tabu Search 57
3.7 Comparison 57
3.8 Outlook to future work 61

List of Algorithms

1	The basic list scheduler algorithm for Shop Scheduling problems	7
2	Restrict(S) method by Giffler and Thompson	7
3	Restrict(S) method of the Non-Delay algorithm	8
4	The basic shifting bottleneck procedure	9
5	Framework for insertion techniques	10
6	The iterative packing method	11
7	Evolutionary Computation (EC)	16
8	MSXF crossover	19
9	Iterative Improvement	23
10	Tabu Search (TS)	23
11	Simulated Annealing (SA)	29
12	Ant System (AS)	33
13	Ant Colony Optimization (ACO)	34
14	$\mathcal{MM}$AS for the GSP	46
15	E-$\mathcal{MM}$AS for the GSP	50

Chapter 1

Introduction

Scheduling deals with the allocation of scarce resources to tasks over time. It is a decision making process with the goal of optimizing one or more objectives. Scheduling problems play important roles in most manufacturing and production systems as well as in most information–processing environments. The importance of the scheduling problem makes it one of the most studied combinatorial optimization problems in general. One of the difficulties encountered is that in practice not many scheduling problems fit into a common description model. This makes it very difficult to define a common framework for scheduling problems and also to find algorithms which can be applied (or adapted) to tackle a great variety of problems. In fact, a well working algorithm for a problem A might not work at all for a problem B being just a slight variation of problem A. From the great variety of problems, a few problem formulations emerged in the scientific area which are used since many years and which can be regarded as benchmark problems. The best known and most studied scheduling problems are certainly Shop Scheduling problems, and among them the *Job Shop Scheduling* problem (JSP) and the *Open Shop Scheduling* problem (OSP). It is well known that both problems are NP-hard[1], see [61, 47], and belong to the most intractable combinatorial optimization problems considered. This is dramatically illustrated by the fact that a JSP instance involving 10 jobs and 10 machines, 1963 proposed by Fisher and Thompson in [75], remained unsolved for more than a quarter of a century, even though every available algorithm was tried on it. The OSP seems even harder than the JSP. As an example, JSP instances are considered to be solvable to optimality nowadays for up to 100 operations, while there still remain unsolved instances of the OSP with less than 50 operations.[2] Many algorithms have been developed to tackle especially the JSP but also the OSP. In early years – beginning in the sixties – due to the hardness of the problem, the focus was on developing heuristic algorithms and more efficient complete algorithms like Branch and Bound methods. In the last 15 years, metaheuristic algorithms have been discovered to be useful and very efficient tools to tackle the JSP and the OSP. The state-of-the-art methods nowadays for tackling large JSP and OSP problems are nearly all attributed to the field of metaheuristics.

In this work we first introduce a way of describing Shop Scheduling problems. Based on this way of describing problems we formalize the JSP and the OSP. Then we introduce the

[1] This means that – assuming that $P \neq NP$ – no algorithm working in a time which is a polynomial of the problem parameters can be found to solve these problems.

[2] To put things into perspective: The famous TSP problem – also a NP-hard combinatorial optimization problem – is solvable to optimality nowadays for several thousand cities.

formulation of a more general Shop Scheduling problem, covering both the JSP and the OSP. This formulation of Shop Scheduling problems will be called *Group Shop Scheduling* problem (GSP). In Chapter 2 we survey existing metaheuristic methods for the JSP and the OSP, before in Chapter 3 we outline the metaheuristic algorithms for the GSP developed in the course of the Metaheuristics Network [114]. The focus in this part of the work is on Ant Colony Optimization (ACO). As the newly developed metaheuristic algorithms are working on problem instances of the GSP, they can be applied to both, JSP and OSP instances. Comparing these metaheuristic algorithms on problem instances from the whole range between the JSP and the OSP gives further insight into the differences between the JSP and the OSP.

1.1 Shop Scheduling: The JSP, the OSP and the GSP

In Shop Scheduling problems, *jobs* (items) are to be processed on *machines* with the objective of minimizing some function of the completion times of the jobs. Each machine can process only one job at a time. The processing of a job on a machine is called an *operation*; its *processing time* is fixed, and it cannot be interrupted (scheduling without preemption). In the JSP the processing of the jobs is subject to the constraints that the sequence of machines for each job is prescribed. These machine sequences are often called the *technological sequences*. In contrast, in the OSP a job can be processed in any order on the machines. In the following we give a more formal description of the problems. For this purpose we introduce the notation in Table 1.1.

Table 1.1: Notation

Notation	Meaning
n	number of jobs
m	number of machines
J_i	Job i
M_j	Machine j
G_l	Group l
O	set of operations
o, o'	operations
o_{ij}	operation on job J_i to be processed on machine M_j
$p(o)$	processing time of operations o
$m(o)$	machine on which o has to be processed
$t_s(o)$	starting time of operation o with respect to a solution s
$j(o)$	the job operation o belongs to
$g(o)$	the group operation o belongs to

A Shop Scheduling problem can be formalized as follows: We consider a finite set of operations O which is partitioned into subsets $M_1, \ldots, M_m$ (machines) where $\mathcal{M} = \bigcup_{j=1}^{m}\{M_j\}$ and into subsets $J_1, \ldots, J_n$ (jobs) where $\mathcal{J} = \bigcup_{i=1}^{n}\{J_i\}$. Also given is a partial order $\preceq \subseteq O \times O$ such that $\preceq \cap J_i \times J_j = \emptyset$ for $i \neq j$ (defining the technological sequences), and a function $p : O \rightarrow \mathbf{N}$. A feasible solution is a refined partial order $\preceq^* \supseteq \preceq$ for which the restrictions $\preceq^* \cap J_i \times J_i$ and $\preceq^* \cap M_k \times M_k$ are total $\forall\, i, k$ and the longest chain in $\preceq^*$ is of finite length.

In the course of this work, the cost of a feasible solution is defined by

$$C_{\max}(\preceq^*) = \max\{\sum_{o \in C} p(o) \mid C \text{ is a chain in } (O, \preceq^*)\} \ .$$

$C_{\max}$ is called the makespan of a solution. We aim at a feasible solution which minimizes $C_{\max}$. The Shop Scheduling problems we consider in this work are subject to the following constraints: (i) Each machine can process at most one operation at a time, (ii) operations must be processed without preemption, and (iii) operations belonging to the same job must be processed sequentially. This brief problem formulation covers among others the JSP and the OSP in the following way: The restriction $\preceq \ \cap J_i \times J_i$ is total in the JSP and trivial ($= \{(o, o) \mid o \in J_i\}$) in the OSP. In the JSP, $\preceq$ induces a total order on the operations of each job.

For the **Group Shop Scheduling problem (GSP)**, we consider a weaker restriction on $\preceq$ which includes the above scheduling problems by looking at a refinement of the partition $\mathcal{J}$ to a partition into *groups* $\mathcal{G} = \{G_1, \ldots, G_g\}$. We demand that $\preceq \ \cap G_i \times G_i$ has to be trivial and that for $o, o' \in J$ ($J \in \mathcal{J}$) with $o \in G_i$ and $o' \in G_j$ ($i \neq j$) either $o \preceq o'$ or $o \succeq o'$ holds. Note that the coarsest refinement $\mathcal{G} = \mathcal{J}$ (group sizes are equal to job sizes) is equivalent to the OSP and the finest refinement $\mathcal{G} = \{\{o\} \mid o \in O\}$ (group sizes of 1) is equivalent to the JSP.

Regarding the precedence constraints (the technological sequences), an *immediate predecessor* o' of an operation o with $j(o) = j(o')$ and $o' \neq o$ is denoted by $\preceq_{pred} (o)$ and the relation between them is denoted as $o' \preceq_{pred} o$. For $o \neq o'$ it holds: $o' \preceq_{pred} o \ \Leftrightarrow o' \preceq o$ and $\forall o'' \neq o, o'$ with $o' \preceq o''$ it holds that $o \preceq o''$.

Is is useful to represent Shop Scheduling problems on a *disjunctive* graph [93] $G = (V, A, E)$, with node set V, conjunctive arc set A, and disjunctive arc set E. The nodes of G correspond to operations, the arcs A[3] to precedence constraints, and the edges[4] (disjunctive arcs E) to pairs of operations to be performed on the same machine. G is node-weighted with node weights corresponding to the processing times of the associated operations. A formal description of the disjunctive graph for the Group Shop Scheduling problem is given as follows:

$$V = O \quad (1.1)$$

$$A = \{a_{o,o'} \mid o, o' \in V, o \preceq_{pred} o'\} \quad (1.2)$$

$$E = \{e_{o,o'} \mid o, o' \in V, m(o) = m(o') \vee g(o) = g(o')\} \quad (1.3)$$

Figure 1.1 shows the disjunctive graphs for the JSP version and the OSP version of a small Shop Scheduling problem (processing times are omitted in this example). Note that in the case of the OSP the arc set A is empty because no precedence constraints are given.

Figure 1.2 shows the disjunctive graph for one of the many possible GSP versions of the same Shop Scheduling problem.

A solution in terms of the disjunctive graph representation is a version of the disjunctive graph where every undirected arc in E has been given a direction such that the resulting graph is acyclic. The makespan of a solution is then associated with the length of a longest path in G (which corresponds to a maximal weight chain C in $(O, \preceq^*)$). Such a path will be called a

[3] Arcs are directed links in graphs denoted by $a_{v,v'}$ for $v, v' \in V$.

[4] Edges are undirected links in graphs denoted by $e_{v,v'}$ for $v, v' \in V$.

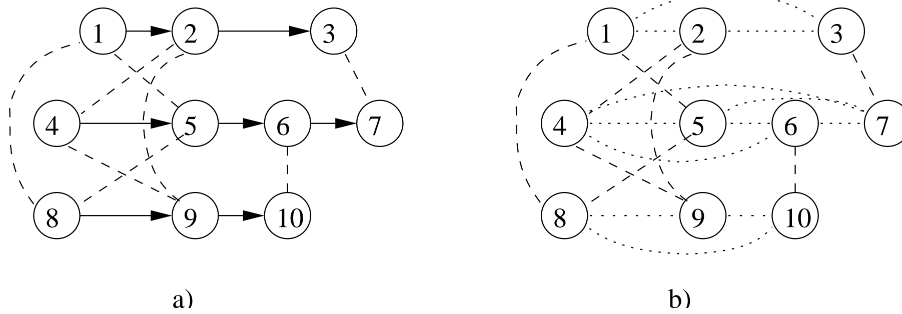

Figure 1.1: Disjunctive graph representation: a) The disjunctive graph corresponding to the JSP and b) the OSP version of the following problem: $O = \{1, ..., 10\}$, $\mathcal{J} = \{J_1 = \{1, 2, 3\},\ J_2 = \{4, ..., 7\},\ J_3 = \{8, 9, 10\}\}$, $\mathcal{M} = \{M_1 = \{1, 5, 8\},\ M_2 = \{2, 4, 9\}\ M_3 = \{3, 7\},\ M_4 = \{6, 10\}\}$ (processing times are omitted).

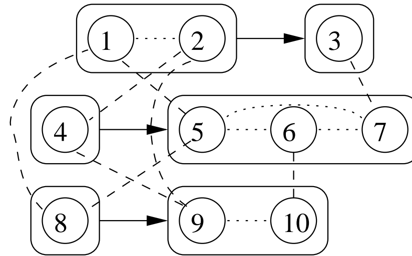

Figure 1.2: Disjunctive graph representation: The disjunctive graph corresponding to one of the GSP versions of the Shop Scheduling instance shown in Figure 1.1b). This GSP instance is obtained by a refinement of the job partition into the following group partition: $\mathcal{G} = \{G_1 = \{1, 2\},\ G_2 = \{3\},\ G_3 = \{4\},\ G_4 = \{5, 6, 7\},\ G_5 = \{8\},\ G_6 = \{9, 10\}\}$.

critical path in the course of this work. Note that in general a solution to a Shop Scheduling problem can also be represented as a permutation (henceforth called a sequence) of all the operations. Every sequence unambiguously defines the order of operations on machines and in jobs (groups). In general a solution can be defined by assigning a starting time $t(o)$ to every operation $o \in O$ such that operations don't overlap each other and the precedence constraints are respected. When we talk about sequences, we assume that every operation has been assigned the earliest possible starting time.

According to Fang [35], feasible schedules fall into four classes: Inadmissible, semi-active, active and non-delay schedules. Figure 1.3 illustrates the relationship between each of these types. There are an infinite number of *inadmissible* schedules, which contain excess idle time. *Semi-active* schedules contain no idle time. There might be *holes* in the schedule though, such that operations might be shifted to the left in machine or job sequences without delaying other operations. *Active* schedules contain no idle time, and furthermore, have no operations which can be completed earlier without delaying other operations. Optimal schedules are guaranteed to fall within the set of active schedules. *Non-delay* schedules are a subset of active schedules, in which operations are placed into the schedule such that no machine is ever kept idle if some

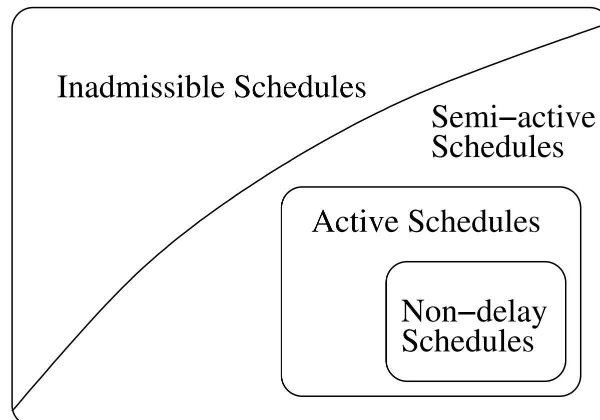

Figure 1.3: Types of feasible schedules for Shop Scheduling problems

operation is able to be processed on it.

1.2 Complexity

In general, both problems, JSP and OSP, are NP-hard and are considered to be among the most intractable combinatorial optimization problems. For the JSP this was proven by Lenstra et al. in [61]. Only a few particular cases are efficiently solvable. NP-hardness was proven for the OSP by Gonzales and Sahni in [47] for $m \geq 3$. As both, JSP and OSP, are special cases of GSP, we also can deduct the NP-hardness of the GSP.

1.3 Benchmark instances

A collection of benchmark instances which have been extensively used in the literature can be found in the OR-Library. The OR-Library is a collection of test data sets for a variety of Operations Research (OR) problems. These test data sets can be accessed via emailing to or.library@ic.ac.uk a message containing the name of the required file, or via the URL http://www.ms.ic.ac.uk/info.html. The OR-Library is maintained and described by Beasley in [11]. Table 1.2 lists benchmark instances for the JSP. For the OSP there are less benchmark instances. Taillard [101] provides 60 problems of varying size and a generator written in Pascal to generate them. However, these instances are commonly agreed to be quite easy to solve. Therefore, Brucker et al. [21] generated 18 harder instances.

1.4 Algorithms

A massive amount of literature about solving OSP and JSP has been published in the last 30 to 40 years. The methods proposed can be classified as either *complete* or *approximate* algorithms. Complete algorithms are guaranteed to find for every finite size problem instance

Table 1.2: Benchmark problems for the JSP

Problems	Source
abz5,...,abz9	Adams et al. [4]
ft6, ft10, and ft20	Fisher and Thompson in [75]
la01,...,la40	Lawrence [60]
orb01,...,orb10	Applegate and Cook [7]
swv01,...,swv20	Storer et al. [98]
yn1,...,yn40	Yamada and Nakano [109]
ta01,...,ta80	Taillard [101]

an optimal solution in bounded time (see [85, 78]). Yet, for many combinatorial optimization problems that are NP-hard such as Shop Scheduling problems, complete methods need exponential computation time in the worst-case and even for small problem instances these algorithms might take an amount of execution time too high for practical purposes. Therefore, the use of approximate methods to solve Shop Scheduling problems has been getting more and more attention in the last 30 years. In approximate methods we sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a significantly reduced amount of time.
Among the basic approximate methods we usually distinguish between *constructive* methods and *local search* methods. Constructive algorithms generate solutions from scratch by adding – to an initially empty partial solution – components, until a solution is complete. They are typically the fastest approximate methods, yet they often return solutions of inferior quality when compared to local search algorithms. Local search algorithms start from some initial solution and iteratively try to replace the current solution with a better solution in an appropriately defined neighborhood of the current solution, where the neighborhood is formally defined as follows:

Definition 1 *A* **neighborhood structure** *is a function $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ (where S is the set of all solutions to a combinatorial optimization problem) that assigns to every $s \in \mathcal{S}$ a set of neighbors $\mathcal{N}(s) \subseteq \mathcal{S}$. $\mathcal{N}(s)$ is also called the neighborhood of s.*

With the introduction of a neighborhood structure we can also define the concept of *locally* minimal solutions.

Definition 2 *A* **locally minimal solution (or local minimum)** *with respect to a neighborhood structure $\mathcal{N}$ is a solution $\hat{s}$ such that $\forall\ s \in \mathcal{N}(\hat{s}) : f(\hat{s}) \leq f(s)$. We call $\hat{s}$ a strict locally minimal solution if $f(\hat{s}) < f(s)\ \forall\ s \in \mathcal{N}(\hat{s})$*

In the last 20 years, a new approach to the design of approximate algorithms has emerged which basically tries to combine basic heuristic methods in a higher level framework aimed at efficiently and effectively exploring a search space. These methods are nowadays commonly called *metaheuristics*. In the following we shortly outline the most important constructive methods before giving – in the following chapter – an overview of the state-of-the-art in metaheuristic research for the JSP and the OSP.

1.4.1 List scheduler algorithms

List scheduler algorithms are probably the most frequently applied constructive heuristics for solving Shop Scheduling problems in practice. The reason for that is their simplicity in terms of implementation and their low time complexity. To construct a schedule, list scheduler algorithms build a sequence s containing all operations of O exactly once – starting with an empty sequence – by performing $|O|$ steps as shown in Algorithm 1. A sequence s unambiguously defines a solution[5] to an instance of a Shop Scheduling problem. For the problem instance depicted in Figure 1.2, the sequence $1-0-3-5-4-7-8-9-2-6$ defines group order $1 \preceq 0$ in group G_1, $5 \preceq 4 \preceq 6$ in group G_4 and $8 \preceq 9$ in group G_6. It also defines machine orders $0 \preceq 4 \preceq 7$, $1 \preceq 3 \preceq 8$, $2 \preceq 6$ and $5 \preceq 0$. In the following, partial sequences are denoted by $s_{x,y}$ where x is the actual length of the partial sequence and y is its final length. We also use the notation $t_{es}(o \mid s_{x,y})$ to denote the earliest possible starting time of an operation o with respect to the partial sequence $s_{x,y}$, $o \notin s_{x,y}$. Note that these earliest starting times are with respect to appending an operation to a partial sequence. Analogously, $t_{ec}(o \mid s_{x,y}) = t_{es}(o \mid s_{x,y}) + p(o)$ denotes the earliest possible completion time of an operation o with respect to the partial sequence $s_{x,y}$, $o \notin s_{x,y}$. In the course of this work the contents of the t-th position in a sequence s or a partial sequence $s_{x,y}$, $y \geq t$, is denoted by $s[t]$, $s_{x,y}[t]$ respectively.

Algorithm 1 The basic list scheduler algorithm for Shop Scheduling problems

$O_{rem} \leftarrow O$
$s_{0,|O|}$ is an empty sequence
for $t = 1, ..., |O|$ **do**
  $S \leftarrow \{o \in O_{rem} \mid \nexists o' \in O_{rem} \text{ with } o' \preceq_{pred} o\}$
  $S' \leftarrow \mathsf{Restrict}(S)$
  $o^* \leftarrow \mathsf{Choose}(S')$
  $s_{t-1,|O|}[t] = o^*$
  $O_{rem} = O_{rem} \setminus \{o^*\}$
end for

There exist two major ways of implementing $\mathsf{Restrict}(S)$ in Algorithm 1. The one proposed by Giffler and Thompson [42] works as shown in Algorithm 2. First the earliest possible completion times of all the operations in S are calculated. Then one of the machines M^* with minimal completion time t^* is chosen and set S' is the set of all operations in S which need to be processed on machine M^* and whose earliest possible starting time is $< t^*$. This way of restricting set S produces active schedules (see Figure 1.3). Algorithm 1 using the set restriction given by Algorithm 2 is usually called GT algorithm.

Algorithm 2 $\mathsf{Restrict}(S)$ method by Giffler and Thompson

Determine $t^* = \min\{t_{ec}(o \mid s_{t-1,|O|}) \mid o \in S\}$
Determine $\mathcal{M}^* = \{M \in \mathcal{M} \mid \exists\, o \in S \text{ with } m(o) = M \text{ and } t_{ec}(o \mid s_{t-1,|O|}) = t^*\}$
Choose $M^* \in \mathcal{M}^*$ randomly
$S' \leftarrow \{o \in S \mid m(o) = M^* \text{ and } t_{es}(o, s_{t-1,|O|}) < t^*\}$

[5] Note that the sequence to schedule mapping is a many to one mapping. Several sequences map to the same schedule.

The other major way of implementing Restrict(S) is called Non-Delay algorithm. It works as shown in Algorithm 3. First the earliest possible starting time t^* of all operations in S is determined. Then S' consists of all operations in S which can start at time t^*. As the name indicates, this algorithm produces non-delay schedules (see Figure 1.3).

Algorithm 3 Restrict(S) method of the Non-Delay algorithm

Determine $t^* = \min\{t_{es}(o \mid s_{t-1,|O|}) \mid o \in S\}$
$S' \leftarrow \{o \in S \mid t_{es}(o \mid s_{t-1,|O|}) = t^*\}$

Over the years quite a lot of research has been devoted to finding rules for choosing among the operations in set S' the one to be scheduled next. These rules are commonly called priority rules or dispatching rules. Table 1.3 shows a selection of them. We mention that sometimes these rules are used probabilistically (in a roulette-wheel-selection manner) instead of deterministically. None of these rules can be singled out to be labeled the “best performing” priority rule. Which rule performs best strongly depends on the structure of the problem instance to be solved. A good overview on priority rules can be found in [51].

Table 1.3: Priority Rules

Rule	Description
Random	An operation is randomly chosen
SPT	An operation with shortest processing time
LPT	An operation with longest processing time
MWR	An operation with most work remaining in the job
LWR	An operation with least work remaining in the job
LTW	An operation with least total work in the job
MTW	An operation with most total work in the job
MRO	An operation with most remaining operations in the job
LRO	An operation with least remaining operations in the job

1.4.2 The shifting bottleneck procedure

The shifting bottleneck procedure by Adams, Balas and Zawack [4] (for the JSP) and by Ramudhin and Marier [90] (for the OSP) is one of the most powerful iterative constructive procedures among heuristics for Shop Scheduling problems. For the following description of this method for the JSP we follow the description given in [15]. The idea is to solve for each machine a one-machine scheduling problem to optimality under the assumption that a lot of disjunctive arc directions in the optimal one-machine schedule coincide with an optimal job shop schedule. Consider all operations of a JSP instance that have to be scheduled on a machine M. In the (disjunctive) graph G corresponding to a partial schedule (node set is restricted to the operations already in the schedule) there exists a longest path of length r_o with operation o as the last node. Processing of operation o cannot start before time r_o which is called the *head* of operation o. There is also a longest path of length q_o with operation o as the start node. Obviously, when o is processed, it will take at least q_o time units to finish

the whole schedule. q_o is called the *tail* of operation o. Although the one-machine scheduling problem with heads and tails is NP-complete, there is a powerful branch and bound method proposed by Potts [88] and Carlier [23] which dynamically changes heads and tails in order to improve the order of operations on one machine.
As the name suggests, the shifting bottleneck heuristic always schedules bottleneck machines first. As a measure of the bottleneck quality of a machine M, the value of an optimal solution of a certain one-machine scheduling problem on machine M is used. The operation orders on scheduled machines are fully determined. Hence scheduling an additional machine probably results in a change of heads and tails of those operations whose machine order is still open. For all machines not scheduled, the maximum makespan of the corresponding optimal one-machine schedules, where the arc directions of the already scheduled machines are fixed, determines the bottleneck machine. In order to minimize the makespan of the JSP the bottleneck machine should be scheduled first. The basic shifting bottleneck procedure is shown in Algorithm 4.

Algorithm 4 The basic shifting bottleneck procedure

$\mathcal{M}' \leftarrow \emptyset$ is the set of already schedules machines
repeat
  for $M \in \mathcal{M} \setminus \mathcal{M}'$ **do**
    Compute head and tail for each operation $o \in M$
    Solve the one-machine scheduling problem for machine M to optimality. $C_{\max}^{M}$ denotes the resulting makespan.
  end for
  Let M^b be the bottleneck machine, i.e. $C_{\max}^{M^b} \geq C_{\max}^{M}$ $\forall M \in \mathcal{M} \setminus \mathcal{M}'$
  for $M \in \mathcal{M}'$ in the order of its inclusion **do**
    Unschedule M
    Compute head and tail for each operation $o \in M$
    Solve the one-machine scheduling problem for machine M to optimality
  end for
until $\mathcal{M} = \mathcal{M}'$

Adams et al. [4] also proposed an improvement of the basic procedure which consists basically of an enumeration tree where each path from the root to a leaf is similar to an application of Algorithm 4.

1.4.3 Insertion techniques and beam search

Insertion techniques have been successfully applied to many combinatorial optimization problems. Especially for permutation problems (like the TSP and also Shop Scheduling problems) insertion algorithms often generate good solutions. Similar to list scheduler algorithms, insertion algorithms successively complete a partial solution. Let's assume, we build a permutation as a sequence from left to right. Then the principle of insertion algorithms for Shop Scheduling problems can be stated as follows. Given a partial sequence $s_{x,y} = (s[1], s[2], ..., s[x])$[6] with $x < y$, an operation $o \notin s_{x,y}$ to be inserted next has to be determined. Operation o can now potentially be inserted at most at $x + 1$ positions in the partial sequence $s_{x,y}$. The feasible

[6]Remember that x denotes the actual length of the partial sequence whereas y denotes its final length.

ones[7] among the possibilities are evaluated (exact makespan or makespan approximations are possible) and the best insertion point is chosen. The pseudo-code for this mechanism is given in Algorithm 5.

Algorithm 5 Framework for insertion techniques

$O_{rem} \leftarrow O$
$s_{0,|O|}$ is an empty sequence
for $t = 1, ..., |O|$ **do**
  Choose operation $o \in O_{rem}$ to be inserted next
  Try to insert o in all t possible positions and evaluate the resulting feasible partial sequences.
  Chose the best insertion point and insert o.
  $O_{rem} = O_{rem} \setminus \{o\}$
end for

The main difference to list scheduler algorithms is that the insertion position for list scheduler algorithms is fix (a new operation is always appended to a partial sequence) and the crucial decision is to chose the next operation, whereas in insertion algorithms the order in which the operations are inserted is usually fix (e.g., ordered according to non-increasing processing times) and the crucial decision is where to insert the next operation in the partial sequence. Insertion techniques for OSP have been proposed by Bräsel [20]. An example for the JSP is the algorithm proposed by Werner et al. [108]. For the Permutation Flow Shop problem which is a special case of the JSP a very well working insertion algorithm has been proposed by Nawaz et al. [77].
Often insertion algorithms are combined with a technique called beam search [84]. The basic idea of this approach is to build a limited number of partial solutions in parallel. If a beamwith of k is applied, we select in each step of the (parallel) insertion algorithm the k best partial sequences (instead of just the best one). This selection of partial sequences is called the beam. This set of k partial sequences enters the next step of the insertion algorithm, and from the set of all partial sequences obtained by inserting an operation at some position in one of the partial sequences, we chose again the k best ones and proceed.

1.4.4 A matching algorithm combined with packing and inserting

In 1998, Guéret and Prins [49] proposed a heuristic method to solve the OSP which is based on algorithms for generating matchings in bipartite graphs. The heuristic works as follows. It partitions the set of operations into subsets, such that the operations in a subset can run simultaneously without violating any constraints. A subset might be interpreted as a partial schedule (a schedule slice). Obviously the length of a slice is the duration of its longest task. In a first phase, successive subsets are computed and the resulting slices are concatenated to generate a complete schedule. In a second phase, this preliminary schedule is improved by *packing* the slices, which means starting every operation as soon as possible.
Guéret and Prins proposed 3 different matching algorithms to generate differently weighted matchings in the first phase of the algorithm: (i) application of the well-known Hungarian algorithm generates min-weight resp. max-weight matchings, (ii) applying a variant of Dijk-

[7] Remember that a feasible (partial) schedule is characterized by an acyclic disjunctive graph representation.

stra's algorithm for computing paths of maximum capacity generates min-max resp. max-min matchings and (iii) an algorithm proposed by Martello et al. [66] provides balanced matchings[8].
For the second phase they proposed the iterative "packing" mechanism shown in Algorithm 6 which takes as input the partition of the operations produced by the matching algorithm of the first phase.

Algorithm 6 The iterative packing method

Let $\mathcal{P} = \{P_1, ..., P_k\}$ be the partition generated by the matching algorithm
Chose P^* with $|P^*| = \max\{|P| \mid P \in \mathcal{P}\}$
Schedule the operations $o \in P^*$ {all have starting time 0}
$\mathcal{P} = \mathcal{P} \setminus P^*$
$S[1] \leftarrow P^*$ {S is a sequence of partition elements}
while $\mathcal{P} \neq \emptyset$ **do**
  $gcp \leftarrow 0$
  for each $P \in \mathcal{P}$ **do**
    for $l = 1, ..., |S| - 1$ **do**
      Schedule the operations $o \in P$ between $S[l]$ and $S[l+1]$
      Compute the makespan of the new partial schedule, $C_{\max}^{before}$
      Pack the new partial sequence
      Compute the makespan of the new partial schedule, $C_{\max}^{after}$
      if $\frac{C_{\max}^{before}}{C_{\max}^{after}} > gcp$ **then**
        $gcp \leftarrow \frac{C_{\max}^{before}}{C_{\max}^{after}}$
        $P^* = P$
        $l^* = l$
      end if
    end for
  end for
  Schedule the operations $o \in P^*$ between $S[l^*]$ and $S[l^* + 1]$
  Add P^* to S at position l^* (while moving all partition elements P in S at positions $l \geq l^*$ one position to the right)
end while
Pack the final schedule

The algorithm outlined in this section was improved by an additional insertion technique and the application of a local search method for which we refer to [49].

[8] In a balanced matching the difference in length between the smallest and the longest task is minimized.

Chapter 2

Metaheuristics for Job and Open Shop Scheduling

Metaheuristic algorithms include[1] – but are not restricted to – Ant Colony Optimization (ACO), Evolutionary Computation (EC) including Genetic Algorithms (GA), Iterated Local Search (ILS), Simulated Annealing (SA), and Tabu Search (TS). Up to now there is no commonly accepted definition for the term metaheuristic. It is just in the last few years that some researchers in the field tried to propose a definition. In the following we quote some of them:

"A metaheuristic is a set of concepts that can be used to define heuristic methods that can be applied to a wide set of different problems. In other words, a metaheuristic can be seen as a general algorithmic framework which can be applied to different optimisation problems with relatively few modifications to make them adapted to a specific problem." This is the description of metaheuristics used in the Metaheuristics Network [114].

"A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions." This citation is taken from a metaheuristics bibliography by Osman and Laporte [83].

"A metaheuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method." This citation is taken from "Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization", by Voss, Martello, Osman and Roucairol [106].

"Metaheuristics are typically high-level strategies which guide an underlying, more problem specific heuristic, to increase their performance. The main goal is to avoid the disadvantages of iterative improvement and, in particular, multiple descent by allowing the local search to escape from local optima. This is achieved by either allowing worsening moves or generating

[1] in alphabetical order

new starting solutions for the local search in a more "intelligent" way than just providing random initial solutions. Many of the methods can be interpreted as introducing a bias such that high quality solutions are produced quickly. This bias can be of various forms and can be cast as descent bias (based on the objective function), memory bias (based on previously made decisions) or experience bias (based on prior performance). Many of the metaheuristic approaches rely on probabilistic decisions made during the search. But, the main difference to pure random search is that in metaheuristic algorithms randomness is not used blindly but in an intelligent, biased form." This citation is taken from the PhD thesis of Stützle [99].

Summarizing, we outline fundamental properties which characterize metaheuristics:

- Metaheuristics are strategies that "guide" the search process.
- The goal is to efficiently explore the search space in order to find (near-)optimal solutions.
- Techniques which constitute metaheuristic algorithms range from simple local search procedures to complex learning processes.
- Metaheuristic algorithms are approximate and non-deterministic.
- Metaheuristics incorporate mechanisms to avoid getting trapped in confined areas of the search space.
- The basic concepts of metaheuristics permit an abstract level description.
- Metaheuristics are not problem-specific.
- Metaheuristics make use of domain-specific knowledge as heuristics controlled by the upper level strategy.
- Metaheuristics often use search experience (memory) to guide the search.

In short we could say: Metaheuristics are high level concepts for exploring search spaces by using different strategies. These strategies should be chosen in such a way that a dynamic balance is given between the exploitation of the accumulated search experience (which is commonly called *intensification*) and the exploration of the search space (which is commonly called *diversification*). This balance is necessary on one side to quickly identify regions in the search space with high quality solutions and on the other side not to waste too much time in regions of the search space which are either already explored or don't provide high quality solutions.

The structure of the strategies is highly dependent on the philosophy of the metaheuristic itself. There are several different philosophies behind the existing metaheuristics. Some of them can be regarded as "intelligent" extensions of local search algorithms. The goal of this kind of metaheuristic is to escape from local minima to proceed in the exploration of the search space and to move on to find other hopefully better local minima. This is for example true for Tabu Search, Iterated Local Search, Variable Neighborhood Search, GRASP and Simulated Annealing. These metaheuristics (also called trajectory methods) work on one

or several neighborhood structure(s) imposed on the members (the solutions) of the search space.

We can find a different philosophy in algorithms like Ant Colony Optimization and Evolutionary Computation. They incorporate a learning component in the sense that they implicitly or explicitly try to learn correlations between solution components to identify high quality areas in the search space. This kind of metaheuristic performs in a sense a biased sampling of the search space. For instance, in Evolutionary Computation this is achieved by a recombination of solutions and in Ant Colony Optimization this is achieved by sampling the search space in every iteration according to a probability distribution.

2.1 Classification of metaheuristics

There are different ways to classify and describe metaheuristic algorithms. Depending on the characteristics selected to differentiate between them, several classifications are possible, each of them being the result of a specific viewpoint. We briefly summarize the most important ways of classifying metaheuristics.

Nature-inspired vs. non-nature inspired. Perhaps, the most intuitive way of classifying metaheuristics is based on the origins of the algorithm. There are nature-inspired algorithms, such as Genetic Algorithms and Ant Algorithms, and non nature-inspired ones such as Tabu Search and Iterated Local Search. In our opinion this classification is not very meaningful for the following two reasons. First, many recent hybrid algorithms do not fit either class (or, in a sense, they fit both at the same time). Second, it is sometimes difficult to clearly attribute an algorithm to one of the metaheuristics.

Population-based vs. single point search. Another characteristic which can be used for the classification of metaheuristics is the number of solutions used at the same time: Does the algorithm work on a population or on a single solution at any time? Algorithms working on single solutions are called *trajectory methods* and encompass local search-based metaheuristics, such as Tabu Search, Iterated Local Search and Variable Neighborhood Search. They all share the property of describing a trajectory in the search space during the search process. Population-based metaheuristics on the contrary perform search processes which describe the evolution of a set of points in the search space.

Dynamic vs. static objective function. Metaheuristics can also be classified according to the way they make use of the objective function. While some algorithms keep the objective function given in the problem representation "as it is", some others, such as Guided Local Search (GLS), modify it during the search. The idea behind this approach is to escape from local optima by modifying the search landscape. Accordingly, during the search the objective function is altered by trying to incorporate information collected during the search process.

One vs. various neighborhood structures. Most metaheuristic algorithms work on one single neighborhood structure. In other words, the fitness landscape which is searched doesn't change in the course of the algorithm. Other metaheuristics, like Variable Neighborhood Search (VNS), use a set of neighborhood structures which gives the possibility to diversify the search and tackle the problem jumping between different fitness landscapes.

Memory usage vs. memory-less methods. A very important feature to classify metaheuristics is the use they make of the search history, that is whether they use memory or not. Memory-less algorithms perform a Markov process, as the information they exclusively use is the current state of the search process to determine the next action. There are several different ways of making use of memory. Usually we differentiate between short term and long term memory structures. The first usually keeps track of recently performed moves, visited solutions or, in general, decisions taken. The second is usually an accumulation of synthetic parameters and indexes about the search. The use of memory is nowadays recognized as one of the fundamental elements of a powerful metaheuristic.

We find it most natural to describe metaheuristics following the single point vs. population-based search classification, which divides metaheuristics into trajectory methods and methods based on populations. This is motivated by the fact that this categorization permits a clearer description of the algorithms. Moreover, a current trend is the hybridization of methods in the direction of the integration of single point search algorithms in population-based ones.

Most of the metaheuristic algorithms iterate on a main loop. This process is stopped as soon as one or more termination conditions are met. Possible termination conditions include: maximum CPU time, a maximum number of iterations, a solution s with $f(s)$ less than a predefined threshold value is found, or the maximum number of iterations without improvements is reached.

In the following two sections, the state-of-the-art in metaheuristic research for JSP and OSP is presented.

2.2 Evolutionary Computation

Evolutionary Computation (EC) algorithms are inspired by nature's capability to evolve living beings well adapted to their environment. EC algorithms can shortly be characterized as computational models of evolutionary processes. In every iteration a number of operators is applied to the individuals of the current population to generate the individuals of the population of the next generation (iteration). Usually EC algorithms use operators to recombine two or more individuals to produce new individuals called *recombination* or *crossover* operators, and also operators which cause a self-adaptation of individuals called *mutation* or *modification* operators (depending on there structure). The driving force in evolutionary algorithms is the *selection* of individuals based on their *fitness* (this can be the value of an objective function or the result of a simulation experiment, or some other kind of quality measure). Individuals with a higher fitness have a higher probability to be chosen as members of the next iterations population (or as parents for the generation of new individuals). This corresponds to the principle of *survival of the fittest* in natural evolution. It is the capability of nature to adapt itself to a changing environment, which gave the inspiration for EC algorithms.

There has been a variety of slightly differing EC algorithms proposed over the years. Basically they fall into three different categories which have been developed independently from each other. These are Evolutionary Programming (EP) developed by Fogel et al. in 1966 [40] [41], Evolutionary Strategies (ES) proposed by Rechenberg in 1973 [92] and Genetic Algorithms initiated by Holland in 1975 [53] (see [46] and [73] for further literature). EP arose from the desire to generate machine intelligence. While EP originally was proposed to operate

on discrete representations of finite state machines, most of the present variants are used for continuous optimization problems. The latter also holds for most present variants of ES, whereas GAs are often used to tackle discrete optimization problems. Over the years there have been quite a few overviews and surveys about EC methods. Among those are the ones by Bäck [8], by Fogel [39], by Spears et al. [96] and by Michalewicz et al. [71]. Calégary et al. [22] tried to give a taxonomy of EC algorithms. A "combinatorial optimization"-oriented introduction into the field of EC methods is the overview work by Hertz et al. [52], which gives, in our opinion, a good overview of the different components of EC algorithms and of the possibilities to define them. Algorithm 7 shows the basic structure of every EC algorithm.

Algorithm 7 Evolutionary Computation (EC)

$P \leftarrow$ GenerateInitialPopulation()
Evaluate(P)
while termination conditions not met **do**
  $P' \leftarrow$ Recombine(P)
  $P'' \leftarrow$ Mutate(P')
  Evaluate(P'')
  $P \leftarrow$ Select($P'' \cup P$)
end while

In this algorithm, P denotes the population of individuals. A population of offspring is generated by *recombination* and *mutation* operators and the individuals for the next population are *selected* from the union of the old population and the offspring population. In the following we outline the most important EC algorithms to tackle the JSP and the OSP.

2.2.1 EC algorithms to tackle the JSP

EC algorithms have been widely used in the last 10 to 15 years to tackle the JSP. A wide variety of algorithms has been developed. A good overview (until 1997) is given by the work of Yamada and Nakano [112]. EC algorithms to tackle the JSP differ mainly in the representation used and the recombination operators applied. In the following we mainly focus on these two aspects. We split the different EC algorithms into sections depending on the solution representation used.

2.2.1.1 Binary representation

The first algorithms proposed were based on a *binary representation* which can be applied to any combinatorial optimization problem. As described in Section 1.1, a (semi-active) schedule can be obtained by assigning to all disjunctive arcs a direction such that the resulting graph remains acyclic. Therefore, by associating every disjunctive arc with a position in a bit string where the value at this position (0 or 1) indicates the direction, a schedule can be represented by a binary string[2]. An advantage of using the binary representation is that conventional recombination and mutation operators, such as one-point, multi-point or uniform crossover can be applied. However a resulting bit string might potentially be infeasible (the

[2]Usually the following convention holds: The value for a disjunctive arc $e_{o_{ij},o_{kj}}$ (where $i < k$) is 1 if the arc is directed from o_{ij} to o_{kj}. 0 otherwise.

corresponding disjunctive graph might contain cycles). There are two approaches to deal with this problem: one is to repair an illegal string and the other one is to add a penalty term to the objective function to penalize infeasible solutions.
An example for such a conventional EC method is the algorithm proposed by Nakano and Yamada [76]. They use a method which they call *harmonization* to repair infeasible bit strings. They also experiment with a mechanism called *forcing*. An illegal bit string produced by the operators can be considered as a genotype, and a feasible one after repair can be considered as a phenotype. Therefore, repairing a bit string does not necessarily mean replacing it with the repaired version, which would be called forcing. Nakano and Yamada experiment with limited forcing which improved convergence speed and solution quality. However, the results obtained (compared to the state-of-the-art) are not competitive.

2.2.1.2 Permutation representation

A solution to a JSP instance can be characterized by a permutation of the operations of each machine. Therefore, a schedule can be represented by the set of permutations (one for each machine) specifying the processing orders on machines. The advantage of this representation is that operators developed for permutation problems like the Traveling Salesman Problem (TSP) can be used on every one of the m partitions constituting a solution.
Kobayashi et al. [58] presented an algorithm using a recombination operator called *Subsequence Exchange Crossover* (SXX), a natural extension of the subtour exchange crossover for TSPs. A pair of subsequences both originating from the permutation representing the order on the same machine M_i, one from a solution s_0 and one from a solution s_1, is exchangeable if and only if the corresponding operations belong to the same jobs. Figure 2.1 shows an example. In case the resulting solution is not feasible, Kobayashi et al. use the GT algorithm (see Section 1.4.1) to produce a feasible schedule as similar to the infeasible schedule as possible.

	M1	M2	M3	M4
parent 1	5 8 1	9 4 2	3 7	6 10
parent 2	1 8 5	2 4 9	7 3	6 10
offspring 1	5 1 8	4 9 2	7 3	6 10
offspring 2	8 1 5	2 9 4	3 7	6 10

Figure 2.1: SXX crossover: An example for two solutions to the JSP instance shown in Figure 1.1,a).

A second permutation representation is based on a permutation with repetitions. For every job J_i, $i = 1, ..., n$ the number i occurs $|J_i|$ times in such a permutation with repetitions. By scanning the permutation from left to right the k-th occurrence of a job number refers to the k-th operation in the ordered sequence of operations belonging to this job (also called the technological order). An example on how to decode such a permutation is given in Figure 2.2. The well-known order crossover and the partially mapped crossover for the TSP are adapted for the application to this representation by Bierwirth et al. [12, 13]. They called the adapted crossover versions *Generalized Order Crossover* (GOX) and *Generalized Partially Mapped Crossover* (GPMX). They also proposed *Precedence Preservative Crossover* (PPX)

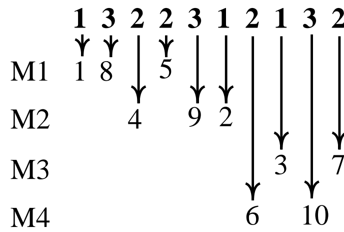

Figure 2.2: Decoding of a chromosome for the permutation with repetition representation: The chromosome encodes a solution to the JSP instance shown in Figure 1.1,a).

which respects the absolute order of genes in parental chromosomes. A randomly generated bitstring of the length of the chromosomes is used to determine for each gene from which parent to take it. An example is shown in Figure 2.3
None of the algorithms mentioned in this section belongs to the state-of-the-art nowadays.

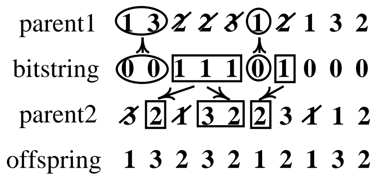

Figure 2.3: PPX crossover: An example for two solutions to the JSP instance shown in Figure 1.1,a).

2.2.1.3 Algorithms without specific representation

For some algorithms a specific representation is not necessary as the operators don't work on the chromosomes themselves, but rather work on the information which is contained in a solution. An example for such an algorithm is the GT crossover based EC algorithm proposed by Yamada and Nakano [109]. The GT crossover takes two solutions s_0 and s_1 as input and produces offspring as follows. For every machine M_j, $j = 1, ..., m$ we have a binary decision variable H_{jl} for all positions $l = 1, ..., |M_j|$. Values for these decision variables are randomly generated. Then the GT crossover – instead of using a priority rule to choose from the restricted set S' as the GT algorithm does – chooses the operation $o^* \in S'$ which is scheduled earliest on machine M_j in parent s_0 if $H_{jk} = 0$, or in parent s_1 if $H_{jk} = 0$ ($k - 1$ operations are already scheduled on machine M_j in the offspring). A second offspring is generated by reversing the values of the decision variables. The advantage of the GT crossover or similar operators is that they directly incorporate domain knowledge into the algorithm. By using the GT crossover the search process is limited to active schedules which is fine, because an optimal schedule must obviously be an active one. The results obtained by the algorithm proposed in [109] are quite good, but worse than the state-of-the-art.

An EC algorithm proposed by Ono et al. [81] applies a crossover operator called *Job-Order Based Crossover* (JOX) which is especially designed to preserve characteristics of the parent schedules. It works as follows. A solution to the JSP specifies for every machine the order

in which to process its operations. So, for every job we know the absolute position of its operations in these machine orders. For the crossover of two solutions, first for every job it is randomly decided if its operations keep their absolute positions in the machine orders given by the parents. Then the operations to keep their absolute positions are copied from parent 1 to offspring 1 and from parent 2 to offspring 2. After that for offspring 2 the missing operations for every machine are taken in the order given by parent 1 and copied in this order into the still free positions (respectively for offspring 1 and parent 2). We mention this method not because there is much evidence that the algorithm proposed in this paper is very effective, but rather for the fact that JOX is used in one of the state-of-the-art EC methods to be outlined later.

Among the best performing EC methods for the JSP are the ones using a recombination operator guided by probabilistic local search. The first recombination operator of that kind, which is called *Multi-Step Crossover Fusion* (MSXF), was proposed by Yamada and Nakano in [111]. The EC algorithm MSXF-GA based on MSXF belongs to the state-of-the-art among EC algorithms to tackle the JSP, therefore we outline it in more detail. The MSXF operator works as follows. Let s_0 and s_1 be the parent solutions, and let $d(s_0, s_1)$ be the DG distance[3] between s_0 and s_1. Then a probabilistic local search with starting solution s_0 and bias towards s_1 is applied as shown in Algorithm 8. This algorithm takes two parent solutions s_0 and s_1 as input and returns the best solution found during the probabilistic search.

Algorithm 8 MSXF crossover

$s_{best} \leftarrow s_0$, $s_{current} \leftarrow s_0$
repeat
  repeat
    Select $s \in \mathcal{N}_3(s_{current})$ with roulette wheel selection inverse proportional to the distances $d(s, s_1)$.
    if $C_{\max}(s) < C_{\max}(s_{current})$ **then**
      $s_{current} \leftarrow s$
    else
      Accept s with probability $p(s, s_{current}, T)$\{see text\}
    end if
  until s is accepted
  if $C_{\max}(s_{current}) < C_{\max}(s_{best})$ **then**
    $s_{best} \leftarrow s_{current}$
  end if
until some termination condition satisfied
Return s_{best}

The stochastic local search in Algorithm 8 applies a Simulated Annealing like acceptance criterion. If the new solution is worse than the current one it is still accepted with the following probability

$$p(s, s_{current}, T) = \exp\left(\frac{C_{\max}(s) - C_{\max}(s_{current})}{T}\right) \tag{2.1}$$

[3] If we regard two schedules in disjunctive graph form, then the DG distance between two solutions s_0 and s_1 is the number of disjunctive arcs which are of different direction.

The temperature parameter T is kept constant in the course of the recombination operation. The neighborhood structure used is the $\mathcal{N}_3$ neighborhood introduced by [27] and to be outlined in Section 2.3.1. Mutation is applied in MSXF-GA by applying Algorithm 8 with choosing among the neighbors of the current solution proportional – instead of inverse proportional – to the distances $d(s, s_1)$. This mutation operator is applied "instead" of the crossover version in case the two parent solution are too close to each other.
Another interesting component of MSXF-GA is the usage of the *reversed problem instance*. In general, a given problem instance of the JSP can be converted to the so-called reversed problem by reversing the precedence relations of operations in the jobs (the technological sequences). Obviously, the reversed problem instance is equivalent to the original problem instance in the sense that reversing the job orders of any schedule for the original problem instance results in a schedule for the reversed problem instance with the same makespan. An active schedule to the original problem instance might be improved by performing the following steps: (i) reverse the solution, (ii) make the reversed solution active, (iii) reverse the active reversed solution, (iv) make the re-reversed solution active again. To exploit the fact that sometimes the reversed problem instance is easier to solve than the original one, MSXF-GA starts with a starting population where half of the individual are (active) solutions to the original problem instance and the other half are (active) solutions to the reversed problem instance. Additionally, before crossover or mutation is applied, an operator reverses every individual to a certain probability.

Sakuma and Kobayashi [94] proposed an EC algorithm that combines the JOX crossover [81] and a crossover called *Extrapolation-Directed Crossover* (EDX). The EDX crossover is very similar to the MSXF crossover outlined above. In fact, the only difference to MSXF is that EDX incorporates a way of adjusting between interpolation behavior (in MSXF the crossover setting) and extrapolation behavior (the mutation version of MSXF). The results of this algorithm are comparable to the performance of MSXF-GA. They clearly improve on the algorithm proposed by Ono et al., which is only using JOX as a crossover operator.

The best EC algorithm for the JSP at the moment is a multi-population method called *Innately Split Model* (ISM) proposed by Ikeda and Kobayashi [54]. They analyzed two well known and difficult JSP instances to observe that good quality solutions are scattered all over the search space[4]. This, in general, makes it difficult or sometimes even impossible to detect good "building blocks" (parts good solutions have in common). This observation emphasizes the need for a good diversification mechanism in search algorithms. The performance of ISM suggests that an EC operating with multiple populations is a good way of attacking this problem. The multiple populations in ISM are handled as follows.

- ISM starts with a number of n_p populations with a number of n_i individuals each. A population is initialized by generating a random solution with the GT algorithm and by generating $n_i - 1$ mutations of this individual (JBSC proposed in [80] is used as mutation operator). The motivation is to initialize a population in a confined area of the search space.
- When two populations come too close to each other, one of them is removed from the search process and a new one is initialized.

[4]This was also observed by Mattfeld and Bierwirth in [68] who performed a search space analysis for the JSP.

- When a population doesn't improve during some fixed period of time, it is removed from the search process and a new one is initialized.

As crossover operator JOX together with the GT algorithm for forcing is used and crossover partners have to be chosen from the same population. In case the two parents chosen for crossover are of the same quality (not necessarily the same solutions) then mutation instead of crossover is applied. An additional feature is that – as proposed in [111] – the algorithm works on both, solutions to the original problem and solutions to the reversed problem.

2.2.1.4 Heuristically guided EC approaches

In contrast to the EC algorithms outlined in the previous sections, which are working on more or less[5] direct representations, the philosophy of heuristically guided EC approaches is a different one. These kind of algorithms work on indirect representations in the sense that a chromosome no more represents a solution itself, it rather provides instructions to a schedule builder on how to build a solution. This bears the advantage that all possible chromosomes result in feasible solutions.

The first algorithm of this kind was proposed by Fang et al. [36]. A chromosome is of length $|O|$ and the domain for every locus is $\{1, ..., n\}$. The schedule builder scans a chromosome from left to right. A number $a \in \{1, ..., n\}$ means: Schedule the first untackled task (according to the technological sequence) of the a-th job with unscheduled operations at the earliest possible place in the current partial schedule. The schedule builder keeps a circular list of uncompleted jobs to determine the a-th job with a modulo operation. To insert an operation in the partial schedule, the schedule builder (without changing the starting times of operations already scheduled) looks for a hole in the partial machine sequence where the operation can be scheduled without creating conflicts. If no hole is found, the operation is scheduled at the earliest time as last operation in the corresponding partial machine sequence. Examining the behavior of their algorithm, Fang et al. noticed that the convergence of the algorithm is characterized by an early convergence in the front parts of the chromosomes and a late convergence in the last parts of the chromosomes. This is reasonable as the meaning of the numbers at the end of a chromosome are dependent on the numbers in the front part of a chromosome. Based on this observation the crossover points for one-point crossover and the positions for mutation are determined by a mechanism which they call *Gene-Variance based Operator Targeting* (GVOT). This mechanism works by measuring the diversity of genes at each position of the chromosomes in a pool, and choosing the actual point of crossover or mutation via roulette-wheel-selection proportional to these variances. This mechanism improved the algorithm considerably. However, the results are worse than those of state-of-the-art algorithms. The performance of this algorithm might be improved by using local search to improve solutions after crossover.

Dorndorf and Pesch [34] proposed a different heuristically guided EC algorithm called *Priority Rule Based GA* (P-GA). Chromosomes are of length $n-1$. The domain for each position in the chromosomes consists of identifiers for different priority rules to be used by the GT algorithm which is used as the schedule builder. This means that the choice of an operation

[5] A direct representation where also infeasible solutions are allowed can also be regarded as an indirect representation.

from the restricted set S' for a position l in the sequence of operations build by the GT algorithm is made with the priority rule on position l of a chromosome.

Another heuristically guided EC algorithm – also proposed by Dorndorf and Pesch in [34] – called *Shifting Bottleneck Based GA* (SB-GA) controls the selection of nodes in the enumeration tree of the shifting bottleneck heuristic (see Section 1.4.2). Here an individual is represented by a permutation of machine numbers $1, ..., m$ where the entry in the i-th position of a chromosome represents the machine to be optimized in Algorithm 4. A cycle crossover operator is used as the crossover for this permutation representation.

2.2.2 EC algorithms to tackle the OSP

So far the research on EC algorithms specifically built to tackle the OSP is somewhat the fifth wheel on the wagon. The most well-known EC algorithms explicitly proposed to tackle the OSP are the 3 versions of a heuristically guided EC algorithm proposed by Fang et al. in [37]. In the first version called JOB+OP, a chromosome is of length $2|O|$. There are two loci associated with every construction step. The first one of them refers to an operation the second one to the job the operation has to be taken from. The domain for the first loci is $\{1, ..., m\}$ and the domain for every second locus is $\{1, ..., n\}$. The schedule builder scans a chromosome from left to right. For a construction step, the numbers $a \in \{1, ..., m\}$ in the first locus and $b \in \{1, ..., n\}$ in the second locus mean: Schedule the first a-th unscheduled operation (according to the technological sequence) of the b-th uncompleted job at the earliest possible place. Again like in the JSP version of this algorithm the schedule builder keeps a circular list of uncompleted jobs and for every uncompleted job a circular list of unscheduled operations. The a-th operation and the b-th job are determined by modulo operations. This version is a natural extension of the version to tackle the JSP proposed in [36].
A second version of the algorithm uses the same chromosomes as the JSP version of the algorithm and choses an unscheduled operations from the a-th uncompleted job by applying an apriori chosen priority rule (see Table 1.3 for a selection of priority rules). This version is called FH (for fixed heuristic).
The third version called EHC (for evolved heuristic choice) evolves the heuristic choice for every construction step. It uses the chromosomes of version JOB+OP except that the domain for the first loci for every construction step is now a set of identifiers for different priority rules. The results show that versions FH and EHC are superior to JOB+OP. In some cases best known solutions (generated by a Tabu Search approach by Taillard [101]) could be improved.

A technical report about EC methods applied to the OSP was prepared by Prins [89]. Finally we mention that quite a few EC algorithms developed for the JSP could be adapted to work for the OSP.

2.3 Tabu Search

Tabu Search is a metaheuristic method based on local search. Basic local search is usually called *iterative improvement*, since each move[6] within a given neighborhood structure $\mathcal{N}$ is

[6] A move is the transition from a solution s to a solution $s' \in \mathcal{N}(s)$ and usually defined by the modification which has to be done to s in order to generate s'.

only performed if the solution it produces is better than the current solution. The algorithm stops as soon as it finds a local minimum (see Section 1.4 for the definition of local and global minima and the neighborhood structure). The algorithmic framework for iterative improvement is sketched in Algorithm 9.

Algorithm 9 Iterative Improvement

```
s ← GenerateInitialSolution()
repeat
   s ← Improve(s,N(s))
until no improvement is possible
```

The function Improve(s,$\mathcal{N}(s)$) can either be a *first improvement*, or a *best improvement* function. The former scans the neighborhood $\mathcal{N}(s)$ and chooses the first solution which improves the objective function, the latter exhaustively explores the neighborhood and returns one of the solutions with the lowest objective function value. Both methods stop at local minima, therefore their performance strongly depends on the definition of S, f and $\mathcal{N}$. The performance of iterative improvement procedures on CO problems is usually quite unsatisfactory, thus several techniques have been developed to prevent algorithms from getting trapped in local minima, or to escape from them. One of these techniques is Tabu Search.

Tabu Search (TS) is among the most cited and used metaheuristics for CO problems. The basic ideas of TS were first introduced in [43] and independently sketched in [50]. A description of the method and its concepts can be found in [45]. TS explicitly uses the history of the search, both to avoid local minima and to implement an explorative strategy. The framework of the basic algorithm is given in Algorithm 10.

Algorithm 10 Tabu Search (TS)

```
s ← GenerateInitialSolution()
InitializeTabuLists(TL_1, ..., TL_r)
k ← 0
while termination conditions not met do
   AllowedSet(s, k) ← {z ∈ N(s) | no tabu condition is violated or at least one aspiration
   condition is satisfied}
   s ← ChooseBestOf(s,AllowedSet(s, k))
   UpdateTabuListsAndAspirationConditions()
   k ← k + 1
end while
```

The basic algorithm applies a best improvement local search as basic ingredient and uses a *short term memory* to escape from local minima and to avoid cycles. The short term memory is implemented as a set of *tabu lists* that store solution *attributes*. Attributes are usually components of solutions, moves, or differences between two solutions. Since more than one attribute can be considered, a tabu list is introduced for each of them. The set of attributes and related tabu lists define the *tabu conditions* which are used to filter the neighborhood of a solution and generate the *allowed set*, which is a subset of the set of neighbors. The use of tabu lists prevents the algorithm from returning to recently visited solutions, therefore it

prevents from infinite cycling[7] and forces the search to accept even uphill moves. The length l of the tabu list (*tabu tenure*) controls the memory of the search process. With small tabu tenures the search will concentrate on limited areas of the search space. On the opposite, a large tabu tenure forces the search process to explore larger regions, because it forbids revisiting a higher number of solutions. The tabu tenure can be varied during the search process.
Storing only attributes of recently visited solutions in the tabu lists introduces a loss of information, as forbidding a move means assigning the tabu status to probably more than one solution. Thus, it is possible that unvisited solutions of good quality are excluded from the allowed set. To overcome this problem, *aspiration criteria* are defined which allow to include a solution in the allowed set even if it is forbidden by tabu conditions. Aspiration criteria define the aspiration conditions that are used to construct the allowed set. The most commonly used aspiration criterion selects solutions which are better than the current best one.

2.3.1 Tabu Search algorithms to tackle the JSP

Tabu Search has been applied to the JSP since about 15 years and a lot of research has been devoted to the development of neighborhood structures, which are the crucial ingredient of any local search based method. All the neighborhood structures to be outlined in the following are defined on at least semi-active schedules. To be able to define the different neighborhood structures some extra notations are needed:

- Given an instance of the JSP, for an operation o, $jp(o)$ and $js(o)$ denote the immediate predecessor and successor of o in the technological sequence given for job $J = j(o)$ by the problem instance.
- Furthermore, given a solution s, mpo and mso denote the immediate predecessor and successor of an operation o in the machine sequence of machine $M = m(o)$.

We also require the following definition.

Definition 3 *Given a solution s to an instance of the JSP, a* **machine block** *is a maximal sequence (of size at least one) of operations on the same machine on a critical path of s. An operation o of a machine block is called* **internal** *operation, if it is neither the first nor the last operation on the machine block.*

Neighborhood $\mathcal{N}_0$: The first and most simple neighborhood $\mathcal{N}_0$ is defined as follows. A pair of operations o, o' are *swapable* with respect to a solution s, if $m(o) = m(o')$ and either $o = mp(o')$ or $o' = mp(o)$. In other words: o and o' are swapable if they are neighbors in the machine sequence given by solution s. A swapping step which consists in reversing the processing orders of two operations o and o' is depicted in Figure 2.4. Note that this neighborhood is relatively large and contains many neighbors which are infeasible. Another disadvantage of this neighborhood is that many neighbors of a solution are not of better quality. If for example the two swapped operations are not on a critical path, then this critical

[7] Cycles of higher periods are possible, since the tabu list has a finite length l which is usually smaller than the cardinality of the search space.

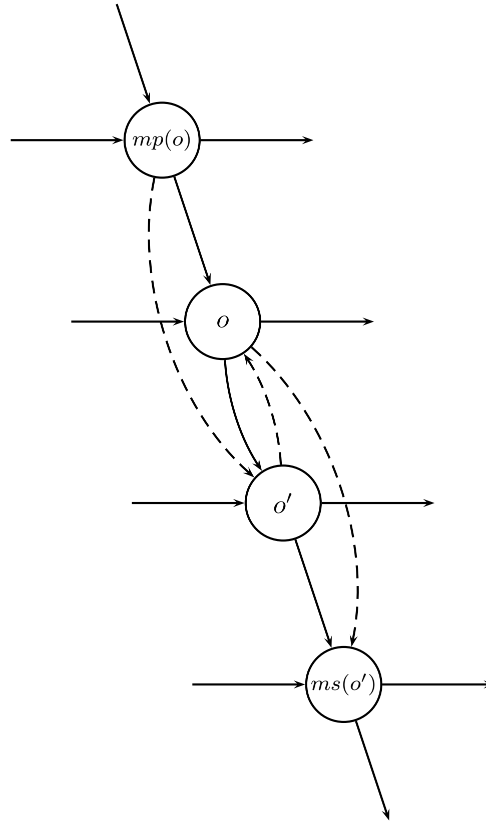

Figure 2.4: An example for reversing the processing orders of two operations o and o', where $o' = ms(o)$. The dashed arcs show the resulting processing orders.

path still exists in the obtained neighbor. To our knowledge, there is no well-working TS method based on this neighborhood.

Neighborhood $\mathcal{N}_{1a}$: This neighborhood outlined by Van Laarhoven et al. in [59] is based on the following observations:

- Reversing the processing orders of two swapable operations on a critical path with respect to a solution s can never lead to an infeasible solution.
- If the reversal of the processing orders of two swapable operations that are **not** on a critical path with respect to a solution s leads to a feasible solution s', then the makespan of s' can not be shorter than the makespan of s as the critical path in s still exists in s'.

The neighborhood $\mathcal{N}_{1a}$ of a solution s is defined by all solutions s' which can be generated by reversing the processing orders of two swapable operations in any machine block of a critical path in s. The advantage of this neighborhood is that it is connected.
A refinement $\mathcal{N}_{1b}$ of neighborhood $\mathcal{N}_{1a}$ was introduced by Matsuo et al. in [67]. This neighborhood is based on the following observation:

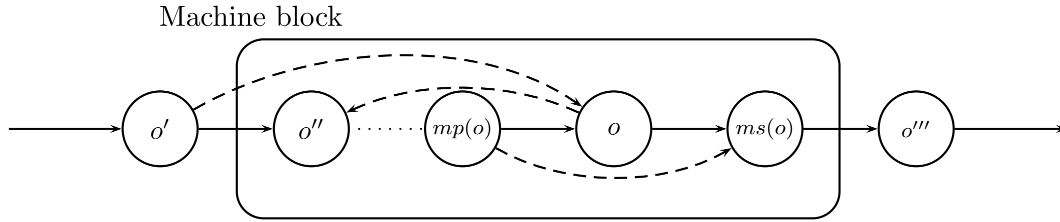

Figure 2.5: An example for neighborhood $\mathcal{N}_3$: An operation o is moved to the beginning of the machine block of which it is an internal operation. The dashed arcs show the resulting processing orders.

- Reversing the processing orders of two swapable, **internal** operations on a critical path with respect to a solution s can never lead to a better quality solution s'.

Therefore $\mathcal{N}_{1b}$ is defined as $\mathcal{N}_{1a}$ excluding the swapable, internal operations.
A further refinement $\mathcal{N}_{1c}$ of neighborhood $\mathcal{N}_{1b}$ is proposed by Nowicki and Smutnicki [79]. This neighborhood is based on the following observation:

- Reversing the processing orders of the first two operations of the first machine block on a critical path with respect to a solution s can never lead to a better quality solution s'. The same holds for the last two operations in the last machine block on a critical path with respect to a solution s.

Therefore $\mathcal{N}_{1c}$ is defined as $\mathcal{N}_{1b}$ excluding the first two operations in the first machine block of a critical path and the last two operations in the last machine block of a critical path.

Neighborhood $\mathcal{N}_2$: Dell'Amico and Trubian [27] proposed the following neighborhood. For any two swapable operations o and $o' = ms(o)$ at the beginning or the end of a machine block on a critical path with respect to a solution s neighbors are obtained by permuting the processing orders of the operations $mp(o)$, o and o' or the operations o, o' and $ms(o')$ such that o and o' are interchanged and the resulting solution is feasible.

Neighborhood $\mathcal{N}_3$: Another neighborhood proposed by Dell'Amico and Trubian [27] is defined as follows. Considered are machine blocks of size at least two. Neighbors are generated by positioning an operation o immediately in front of the first operation or after the last operation in its machine block. Only feasible neighbors are considered. See Figure 2.5 for an example.

Neighborhood $\mathcal{N}_4$: This neighborhood proposed in [67] re-orients at most three edges simultaneously. A neighbor is obtained by reversing the processing orders of two swapable operations o and $o' = ms(o)$ at the beginning or the end of a machine block in a critical path. And in addition by (if they exist) reversing the processing orders of operations $jp^t(o')$ and $mp(jp^t(o'))$ for some $t \geq 1$ and by reversing the processing orders of operations $js(o)$ and $ms(js(o))$. In this description $jp^t(o)$ of an operation o denotes its t-th job predecessor

$jp(...(jp(o)))$. The additional interchanges are only performed if certain additional conditions are met. We refer to [67] for details.

There are basically five different well-working TS algorithms to tackle the JSP. The TS algorithm proposed by Taillard [102] uses the neighborhood $\mathcal{N}_{1a}$. After the processing orders of two swapable operations o and $o' = ms(o)$ have been reversed, the reversal of o' and its machine successor is put in the tabu list. The size of the tabu list is variable. Every 15 iterations a new length for the tabu list is chosen uniformly random between 8 and 14. The strategy to search the neighborhood is the following. To save computation time, the quality of a neighbor is only estimated in such a way that the estimate is exact when both operations involved in the swap are still on a longest path, and that it is a lower bound otherwise. Then, from the allowed set, the schedule with minimum estimated makespan is selected.

The TS algorithm proposed by Barnes and Chambers [10] also uses neighborhood $\mathcal{N}_{1a}$. The length of the tabu list is kept fix in this algorithm. In case the allowed set is empty, the tabu list is emptied. The quality of the neighbors are calculated rather than estimated. A starting solution for this algorithm is obtained by taking the best from a set of schedules produced by the GT algorithm and the Non-Delay algorithm using different priority rules. The results obtained by this algorithm are comparable to the results obtained by the algorithm proposed by Taillard.

A TS algorithm proposed by Dell'Amico and Trubian [27] obtains even better results. This algorithm uses a union of neighborhoods $\mathcal{N}_2$ and $\mathcal{N}_3$. The items on the tabu list are forbidden re-reversals of processing orders. Depending on the type of neighbor, one or more such items are on the list. The length of the tabu list depends on how the quality of the current solution relates to the quality of the previous solution and the quality of the best solution found. Furthermore, the minimal and maximal allowable lengths of the tabu list are changed after a certain number of iterations. In case the allowed set is empty, a random neighbor is selected. A starting solution is obtained by a procedure called "bidir", which applies list scheduling simultaneously from the beginning and the end of the schedule.

One of the best TS methods proposed, which belongs to the state-of-the-art algorithms nowadays for the JSP, is the TS algorithm by Nowicki and Smutnicki [79]. The neighborhood used is $\mathcal{N}_{1c}$ with the restriction that only one critical path is regarded[8]. The items on the tabu list are forbidden re-reversals of processing orders. The length of the tabu list is fixed to 8. In case the allowed set is empty, the following mechanism is applied. If there is one neighbor only (which is tabu), then this one becomes the new current solution. Otherwise, the oldest item on the tabu list is removed iteratively until the allowed set contains at least one neighbor. A starting solution is obtained by the GT algorithm or an insertion technique. Another interesting feature of this algorithm is a backtracking scheme. The backtracking scheme forces the algorithm to restart from promising situations encountered in the recent past. The algorithm stores a fixed number of such situations in a FIFO list. Backtracking is applied in case the algorithm does not improve over a certain number of iterations. Also, complete restarts are applied, because neighborhood $\mathcal{N}_{1c}$ might not be connected.

A TS algorithm comparable (if not even better) than the one by Nowicki and Smutnicki is proposed by Pezella and Merelli in [86]. It is characterized by the following features. The initial solution is generated by the shifting bottleneck procedure (see Algorithm 4). Each

[8]Note that there might be several critical paths in a schedule.

time the algorithm finds a new best solution it is subject to a re-optimization based on the re-optimization cycle of the shifting bottleneck procedure. The algorithm uses three neighborhood structures: (i) a restriction of $\mathcal{N}_{1a}$ where the reversal of the processing orders of the first two and the last two operations is excluded. (ii) A modification of neighborhood $\mathcal{N}_3$: Considered are only machine blocks of size at least three. Neighbors are generated by positioning an operation o at the second position or the last but one position in its machine block. Only feasible neighbors are considered, and (iii) is the neighborhood $\mathcal{N}_{1c}$. Moves in the first two neighborhoods are called *internal moves* because they don't change the critical path. Moves in the third neighborhood do change the critical path and are called *external moves*. Moves in the first neighborhood are applied in an intensification phase of the algorithm whereas moves in the second and third neighborhood are applied in diversification phases. The tabu list applied is of dynamic size depending on the progress in the search process.

Recently a technical report has been released by Grabowski and Wodecki [48]. The results reported are even slightly better than the results by Pezella and Merelli. Furthermore the computation times are far below the computation times of the algorithm by Pezella and Merelli. It seems that this algorithm once published will be the state-of-the-art algorithm for tackling the JSP, both in speed and solution quality.

2.3.2 Tabu Search algorithms to tackle the OSP

Two TS methods have been proposed to tackle the OSP. The first one was proposed by Alcaide et al. [6]. The second one, which is currently the state-of-the-art algorithm for the OSP, was proposed by Liaw in [62]. Liaw generalized neighborhood structures defined for the JSP with the following more general definition of a *block*.

Definition 4 *Given a solution s to an instance of the OSP, a* **block** *is a maximal sequence (of size at least one) of operations on the same machine or in the same job on a critical path of s. An operation o in a block is called an* **internal** *operation if it is neither the first nor the last operation in this block.*

Then the following generalization of neighborhood $\mathcal{N}_{1c}$ was used.

Neighborhood $\mathcal{N}_{1c,OSP}$: Considering two operations o and o' where $m(o) = m(o')$, $o' = ms(o)$ and either o is the first operation of a block in a critical path or o' is the last operation of a block in a critical path, neighbors are obtained by any of the following four steps: (1) reverse the processing order of o and o' only, (2) reverse simultaneously the processing orders of o and o' and of $jp(o')$ and o', (3) reverse simultaneously the processing orders of o and o' and of o and $js(o)$, (4) reverse simultaneously the processing orders of o and o', of $jp(o')$ and o' and of o and $js(o)$. Accordingly consider these for steps (with js replace by ms and jp replaced by mp) if o and o' are from a job block instead of a machine block.

The algorithm works with a tabu list of fixed length and the initial solution is produced by a list scheduler algorithm applying a certain priority rule. Similar to the TS algorithm by Nowicki and Smutnicki, this algorithm uses restarts and a backtracking scheme. The backtracking scheme forces the algorithm to restart from promising situations encountered in the recent past. It is applied in case the algorithm does not improve over a certain number of iterations.

As a final note in this section we mention that most if not all of the neighborhoods defined in the previous section can be adapted by replacing the definition of a machine block (see Def. 3) by the definition of the more general block definition given in Def. 4. In this way the TS algorithms for the JSP can also be applied to the OSP.

2.4 Simulated Annealing

Simulated Annealing (SA) is commonly said to be the oldest among the metaheuristics and surely one of the first algorithms which had an explicit strategy to avoid local optima. The origins of the algorithm are in statistical mechanics (Metropolis algorithm) and it was first presented as a search algorithm for CO problems in [57] and [25]. The fundamental idea is to allow moves resulting in solutions of worse quality than the current solution (uphill moves) in order to escape from local minima. The probability of doing such a move is decreased during the search. The high level algorithm is described in Algorithm 11.

Algorithm 11 Simulated Annealing (SA)

```
s ← GenerateInitialSolution()
T ← T_0
while termination conditions not met do
   s' ← PickAtRandom(N(s))
   if f(s') < f(s) then
      s ← s' {s' replaces s}
   else
      Accept s' as new solution with probability p(T, s', s) {see text}
   end if
   Update(T)
end while
```

The algorithm starts by generating an initial solution (either randomly or heuristically constructed) and by initializing the so-called temperature parameter T. Then this cycle is repeated until the termination condition is reached. The instructions in the inner cycle are very simple: a solution $s' \in \mathcal{N}(s)$ is randomly sampled and it is accepted as new current solution depending on $f(s)$, $f(s')$ and T. s' replaces s if $f(s') < f(s)$ or, in case $f(s') \geq f(s)$, with a probability which is a function of T and $f(s') - f(s)$. The probability is generally computed following the Boltzmann distribution $exp(-\frac{f(s')-f(s)}{T})$.

The temperature T is decreased[9] during the search process, thus at the beginning of the search the probability of accepting uphill moves is high and it gradually decreases, converging to a simple iterative improvement algorithm. This process is analogous to the annealing processes of metals and glass, which assume a low energy configuration when cooled with an appropriate cooling schedule. Regarding the search process, this means that the algorithm is the result of two combined strategies: random walk and iterative improvement. In the first phase of the search, the bias toward improvements is low and it permits the exploration of the search space; this erratic component is slowly decreased thus leading the search to converge

[9] T is not necessarily decreased in a monotonic fashion. Elaborate cooling schemes also incorporate an occasional increase of the temperature.

to a (local) minimum. The probability of accepting uphill moves is controlled by two factors: the difference of the objective functions and the temperature. On the one hand, at fixed temperature, the higher the difference $f(s') - f(s)$, the lower the probability to accept a move from s to s'. On the other hand, the higher T, the higher the probability of uphill moves.

The choice of an appropriate cooling schedule is crucial for the performance of the algorithm. The cooling schedule defines the value of T at each iteration k, $T_{k+1} = Q(T_k, k)$, where $Q(T_k, k)$ is a function of the temperature at the previous step and of the iteration number. Theoretical results on non-homogeneous Markov chains [1] state that under particular conditions on the cooling schedule, the algorithm converges in probability to a global minimum for $k \to \infty$. More precisely:

$$\exists \Gamma \in I\!R \quad \text{s.t.} \quad \lim_{k \to \infty} Prob[\text{global minimum found after } k \text{ steps}] = 1$$
$$\text{iff} \quad \sum_{k=1}^{\infty} exp(\frac{\Gamma}{T_k}) = \infty$$

A particular cooling schedule which fulfills the hypothesis for the convergence is the one that follows a logarithmic law: $T_{k+1} = \frac{\Gamma}{\log(k+k_0)}$ (where k_0 is a constant). Unfortunately, cooling schedules which guarantee the convergence to a global optimum are not feasible in practice because they take infinite time. Therefore, faster cooling schedules are adopted in applications. One of the most used follows a geometric law: $T_{k+1} = \alpha T_k$, where $\alpha \in]0, 1[$, which corresponds to an exponential decay of the temperature.

The cooling rule can vary during the search, with the aim of tuning the balance between diversification and intensification. For example, at the beginning of the search, T might be constant or linearly decreasing, in order to sample the search space; then, T might follow a rule like the geometric one, to converge to a local minimum at the end of the search. More successful variants are *non-monotonic* cooling schedules (e.g., see [82, 65]). Non-monotonic cooling schedules are characterized by alternating phases of cooling and reheating, thus providing an oscillating balance between diversification and intensification.

The cooling schedule and the initial temperature should be adapted to the particular problem instance, since the cost of escaping from local minima depends on the structure of the search landscape. A simple way of empirically determining the starting temperature T_0 is to initially sample the search space with a random walk to roughly evaluate the average and the variance of objective function values. But also more elaborate schema can be implemented [55].

2.4.1 SA algorithms to tackle the JSP

SA algorithms have been applied to the JSP since about 10–15 years. They mostly differ in the neighborhood structure and in the cooling schedule applied. In the following we describe the most important ones from the literature.

The SA algorithm proposed by Matsuo et al. in [67] is a variant which incorporates iterative improvement local search (see Algorithm 9). Given a solution s, a neighbor $s' \in \mathcal{N}_4(s)$ (see Section 2.3.1 for the definition of $\mathcal{N}_4$) is randomly selected. The acceptance of s' as new current solution is decided by the usual acceptance criterion. In case s' is rejected, iterative improvement local search also in neighborhood $\mathcal{N}_4$ is applied and stopped at the local optimum

s''. If s'' improves s it is accepted as the new current solution. Their method also differs from most other implementations of SA in that the acceptance probability for a solution worse than the current solution is independent of the difference in makespan.

In [59] and [2] Aarts et al. describe and test two versions of an SA algorithm for the JSP. The first version uses $\mathcal{N}_{1a}$ as neighborhood structure and the second one uses $\mathcal{N}_4$ as neighborhood structure with $t = 1$ (see Section 2.3.1 for the definition of these neighborhood structures). Both algorithms use the same three-parameter cooling schedule. A first parameter ξ_0 defines the initial temperature, a second parameter ϵ_s defines the final temperature and a third parameter δ determines the decrement of the temperature.

In [97], Steinhöfel et al. report on two SA approaches which quite improve on the results of the approaches mentioned before. They introduce the following neighborhood definition.

Neighborhood $\mathcal{N}_5$: This neighborhood is an extension of the neighborhood $\mathcal{N}_{1a}$ defined in Section 2.3.1. A neighbor of a solution s is obtained by performing the following steps: (i) choose two operations o and o' from a machine block in a critical path with respect to s. Reverse all the processing orders of operation pairs $\langle o, ms(o)\rangle$, $\langle ms(o), ms^2(o)\rangle$, ... ,$\langle mp(o'), o'\rangle$. If $mp(o)$ is also on the critical path directly before o then introduce the processing order $mp(o) \rightarrow o'$. Similarly, if $ms(o')$ is also on the critical path immediately after o' then introduce the processing order $o \rightarrow ms(o')$.

Both SA versions proposed in [97] are based on this neighborhood. The two versions differ in the cooling schedule applied. The first cooling schedule is given by the simple relation $T(t+1) \leftarrow (1 - c_1) \cdot T(t)$ where $T(t)$ is the temperature at time t and $T(0)$ is appropriately defined (by taking into account upper and lower bounds for the makespan with respect to the problem instance to be tackled), and c_1 is a small positive constant. The second cooling schedule is governed by the hyperbolic function

$$T(t+1) \leftarrow \frac{T(0)}{1 + ((t+1) \cdot \phi(c_2) \cdot T(0))} \tag{2.2}$$

where $\phi : x \mapsto ln(1+x)/(C_{\max}^{UB} - C_{\max}^{LB})$ and c_2 is a positive constant.
Applying the first cooling schedule, relatively small problems could be solved quite quickly. The second cooling schedules produced better results (cooling down very slowly) on large problems and even improved the best known solutions on some instances introduced in [109]. In [5] a parallel implementation of the two SA approaches is described.

Yamada and Nakano [110] proposed an SA algorithm based on a work published earlier [113] which is a hybrid between SA and an improvement technique based on the shifting bottleneck procedure (see Algorithm 4). The algorithm published in [113] is a SA based on the neighborhood $\mathcal{N}_3$ (see Section 2.3.1). The main feature of this algorithm is that it jumps back to the best solution found after a number of steps in which no improvements could be found. The enhanced version in [110] has the following features. The disadvantage of $\mathcal{N}_3$ is that it might contain neighbors which are not feasible. Intending to preserve the idea of $\mathcal{N}_3$ the following neighborhood based on the GT algorithm was proposed.

Neighborhood $\mathcal{N}_6$: Let s be an active and feasible schedule. A neighbor is obtained by performing the following steps: (i) choose a machine block B on a critical path, (ii) chose an internal operation o on this path, (iii) apply the modified GT algorithm which creates a solution a) where o is moved as far left in B as possible while maintaining feasibility and b) a

solution where o is moved as far right in B as possible while also maintaining feasibility. For details on the modified GT algorithm we refer to [110].

Another feature of the algorithm is due to the fact that when the temperature drops situations might occur where all the neighbors are chosen several times before one of them finally is accepted. To avoid wasting time a mechanism is introduced which keeps track on which neighbors were already chosen and in case all of them were already chosen and none of them was accepted, a neighbor is chosen proportional to the relative acceptance probabilities and accepted as new current solution. A last mechanism added to the basic algorithm is called *BottleRepair*, which is a an improvement technique based on the shifting bottleneck procedure. With a solution s as input it works as follows: (i) reset all sequences on non-critical machines (machines which do not include any part of a critical path), (ii) re-optimize the still sequenced machines, (iii) solve a one-machine scheduling problem for each unsequenced machine and rank them by their makespans in descending order, (iv) add the machine sequences in order to the partial schedule and reoptimze every scheduled machine after adding one new machine. This mechanism is applied whenever a neighbor is not accepted. In case *BottleRepair* generates a solution s' better then the current solution, s' is accepted as new current solution. The results obtained with this algorithm are in terms of solution quality close to the state-of-the-art.

A general conclusion for SA based algorithms is that they can perform very well in tackling the JSP when run-time is of no concern. The results after long running times, especially for the SA methods of Steinhöfel et al. [97] and of Yamada and Nakano [110] are very good. But when good solutions are required quickly, the TS algorithms have a clear advantage.

2.4.2 SA algorithms to tackle the OSP

To our knowledge there has been no SA algorithm proposed to tackle the OSP. However, for SA the same holds as for TS. Existing algorithms to tackle the JSP can be applied to the OSP by extending the neighborhood definitions from machine blocks to general blocks as done in Def. 4.

2.5 Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic approach proposed by Dorigo [29] and later by Dorigo and colleagues [33, 31]. In the course of this section we follow the description of ACO given in [30].

The inspiring source of ACO is the foraging behavior of real ants. This behavior – described by Deneubourg et al. in [28] – enables them to find shortest paths between food sources and their nest. While walking from food sources to the nest and vice versa, ants deposit a substance called *pheromone* on the ground. When they decide about a direction to go they choose, in probability, paths marked by strong pheromone concentrations. This basic behavior is the basis for a cooperative interaction which leads to the emergence of shortest paths.

In ACO algorithms, an artificial ant incrementally constructs a solution by adding solution

components to a partial solution under consideration[10]. For doing that, artificial ants perform randomized walks on a completely connected graph $\mathcal{G} = (\mathcal{C}, \mathcal{L})$ whose vertices are the solution components $\mathcal{C}$ and the set $\mathcal{L}$ are the connections. This graph is commonly called a *construction graph*. The problem constraints Ω are built into the ants' constructive procedure in a way such that in every step of the construction process only feasible solution components can be added to the current partial solution. In most applications, ants are implemented to build feasible solutions, but sometimes it is unavoidable to work on infeasible solutions. Components $c_i \in \mathcal{C}$ and connections $l_{ij} \in \mathcal{L}$ can have associated a *pheromone value* τ (τ_i if associated to components, τ_{ij} if associated to connections), and a *heuristic value* η (η_i and η_{ij} respectively) representing *a priori* or run time information about the problem instance. These values are used by the ants to make probabilistic decisions on how to move on the construction graph. The probabilities involved in moving on the construction graph are commonly called *transition probabilities*.

Algorithm 12 Ant System (AS)

```
InitializePheromoneValues()
while termination conditions not met do
  for all ants a ∈ A do
    s_a ← ConstructSolution(τ,η)
  end for
  ApplyOnlineDelayedPheromoneUpdate()
end while
```

The first ACO algorithm proposed in the literature is called Ant System (AS) [33]. The pseudo-code for this algorithm is shown in Algorithm 12. In this algorithm, $\mathcal{A}$ denotes the set of ants and s_a denotes the solution constructed by ant $a \in \mathcal{A}$. After the initialization of the pheromone values, in every step of the algorithm every ant constructs a solution. These solutions are then used to update the pheromone values. The components of this algorithm are explained in more detail in the following.

InitializePheromoneValues(): At the beginning of the algorithm the pheromone values (τ_i and/or τ_{ij}) are initialized to the same small numerical value $ph > 0$.

ConstructSolution(τ,η): In the construction phase an ant incrementally constructs a solution by adding solution components to the partial solution constructed so far. The probabilistic choice of the next solution component to be added is done by using the transition probabilities, which in AS are determined by the following *state transition rule*:

$$p(c_r|s_a[c_l]) = \begin{cases} \frac{[\eta_r]^\alpha [\tau_r]^\beta}{\sum_{c_u \in J(s_a[c_l])} [\eta_u]^\alpha [\tau_u]^\beta} & \text{if} \quad c_r \in J(s_a[c_l]) \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

In this formula α and β are parameters to adjust the relative importance of heuristic information and pheromone values and $J(s_a[c_l])$ denotes the set of solution components which are allowed to be added to the partial solution $s_a[c_l]$ with c_l as the last component added (note

[10]Therefore, the ACO metaheuristic can be applied to any combinatorial optimization problem for which a constructive heuristic can be defined

that for the sake of simplicity in the above formula we are only dealing with pheromone on solution components).

ApplyOnlineDelayedPheromoneUpdate(): Once all ants have constructed a solution, the *online delayed pheromone update rule* is applied:

$$\tau_j \leftarrow (1-\rho) \cdot \tau_j + \sum_{a \in \mathcal{A}} \Delta\tau_j^{s_a} \tag{2.4}$$

where

$$\Delta\tau_j^{s_a} \quad = \quad \begin{cases} \frac{Q}{f(s_a)} & \text{if } c_j \text{ is a component of } s_a \\ 0 & \text{otherwise,} \end{cases} \tag{2.5}$$

where $f(s_a)$ is the quality of solution s_a, $0 < \rho < 1$ is a pheromone evaporation rate and Q is a parameter usually set to 1. This pheromone update rule leads to an increase of pheromone on solution components which were found in better quality solutions than other solution components (where the pheromone will decrease).

In the following we describe the more general ACO metaheuristic, which is based on the same basic principles as AS. The ACO metaheuristic framework shown in Algorithm 13 covers all the improvements and extensions of AS which have been developed over the years. It consists of three parts gathered in the ScheduleActivities construct. The ScheduleActivities construct does not specify how these three activities are scheduled and synchronized. This is up to the algorithm designer.

Algorithm 13 Ant Colony Optimization (ACO)

```
while termination conditions not met do
  ScheduleActivities
    ManageAntsActivity()
    EvaporatePheromone()
    DaemonActions() {optional}
  end ScheduleActivities
end while
```

ManageAntsActivities(): An ant builds constructively a solution to the problem by moving through nodes of the construction graph $\mathcal{G}$. Ants move by applying a stochastic local decision policy that makes use of the pheromone values and the heuristic values on components and/or connections of the construction graph (see the state transition rule of AS as an example). While moving, an ant keeps in memory the partial solution it has built in terms of the path it was walking on the construction graph. When adding a component c_j to the current partial solution, it can update the pheromone value(s) τ_i and/or τ_{ij} (in case the ant was walking on connection l_{ij} in order to reach component c_j). This kind of pheromone update is called *online step-by-step pheromone update*. Once an ant has build a solution, it can (by using the memory of the walked path) retrace the same path backward and update the pheromone values of the used components and/or connections according to the quality of the

solution it has built. This is called *online delayed pheromone update*.

EvaporatePheromone(): Pheromone evaporation is the process by means of which the pheromone intensity on the components decreases over time. From a practical point of view, pheromone evaporation is needed to avoid a too rapid convergence of the algorithm toward a sub-optimal region. It implements a useful form of *forgetting*, favoring the exploration of new areas in the search space.

DaemonActions(): Daemon actions can be used to implement centralized actions which cannot be performed by single ants. Examples are the use of a local optimization procedure applied to the solutions built by the ants, or the collection of global information that can be used to decide whether it is useful or not to deposit additional pheromone to bias the search process from an non-local perspective. As a practical example, the daemon can observe the path found by each ant in the colony and choose to deposit extra pheromone on the components used by the ant that built the best solution. Pheromone updates performed by the daemon are called *offline pheromone updates*.

Within the ACO metaheuristic framework as shortly described above the currently best working versions in practice are Ant Colony System (ACS) [32] and $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [100]. In the following we are going to outline the peculiarities of these algorithms shortly.

Ant Colony System (ACS): The ACS algorithm has been introduced to improve on the performance of Ant System (AS) [33]. ACS is based on AS but presents some important differences. First, the daemon updates pheromone values offline: At the end of an iteration of the algorithm – once all the ants have built a solution – pheromone is added to the arcs used by the ant that found the best solution from the start of the algorithm. Second, ants use a different decision rule to decide to which component to move next in the construction graph. The rule is called *pseudo-random-proportional* rule. With this rule, some moves are chosen deterministically (in a greedy manner), others are chosen probabilistically with the usual decision rule. Third, in ACS, ants perform only online step-by-step pheromone updates. These updates are performed to favor the emergence of other solutions than the best so far.

$\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS): $\mathcal{MM}$AS is again an extension of AS. First, the pheromone values are only updated offline by the daemon (the arcs that were used by the iteration best ant or the best ant since the start of the algorithm receive additional pheromone). Second, the pheromone values are restricted to an interval $[\tau_{min}, \tau_{max}]$ and the pheromone values are initialized to their maximum value τ_{max}. Putting explicit limits on pheromone values prevents the probability for any solution to be constructed to drop below a certain value greater than 0. This means that the chance of finding a global optimum never vanishes during the course of the algorithm.

Recently researchers have been dealing with finding similarities between ACO algorithms, EC algorithms and other probabilistic learning algorithms. An important step into this direction was the development of the Hyper-Cube Framework for Ant Colony Optimization (HC-ACO) proposed by Blum et al. [17]. In this framework the introduction of a kind of normalized online

delayed pheromone update rule enables to draw explicit connections to algorithms from EC like PBIL, or a simple GA using a recombination operator called Gene Pool Recombination [74].

2.5.1 ACO algorithms to tackle the JSP

Up until now there are two works on ACO algorithms to tackle the JSP. ACO algorithms mostly differ in four constituents of the algorithm. These are:

- The underlying constructive method which is used to construct solutions.
- The pheromone model which has the function of an adaptive memory used to make the construction mechanism probabilistic.
- The evaluation of the pheromone information. In other words: How are the transition probabilities defined?
- The pheromone update rule.

We will concentrate on these topics in order to outline the differences between the two approaches. The first ACO algorithm for the JSP was proposed by Colorni et al. in [26]. The constructive method used to probabilistically build solutions is the mechanism of list scheduler algorithms (see Algorithm 1). In every step $t = 1, ..., |O|$, the algorithm uses pheromone information and heuristic information to decide the next operation in the sequence s to be built. The pheromone model is the following.

Learning of a predecessor relation in s: In this model (called $\mathsf{PH_{suc}}$) we have a pheromone value τ_{o_i,o_j} on every pair of operations $o_i, o_j \in O$ and we have pheromone values τ_{o_i} $\forall o_i \in O$. The probabilities for operations $o_j \in S'$ to be scheduled in step t dependent on the partial schedule $s_{t-1,|O|}$ are as follows.

$$p(o_j|s_{t-1,|O|}) = \begin{cases} \frac{[\tau_{o_j}]\cdot[\eta_{o_j}]^\alpha}{\sum_{o_k \in S'}[\tau_{o_k}]\cdot[\eta_{o_k}]^\alpha} & : \text{ if } o_j \in S', t = 1 \\ \frac{[\tau_{o_i,o_j}]\cdot[\eta_{o_j}]^\alpha}{\sum_{o_k \in S'}[\tau_{o_i,o_k}]\cdot[\eta_{o_k}]^\alpha} & : \text{ if } o_j \in S', t > 1, s_{t-1,|O|}[t-1] = o_i \\ 0 & : \text{ otherwise} \end{cases} \tag{2.6}$$

where $S' = S$ is the set of operations which can be scheduled now (as defined in Algorithm 1). In this way of modeling the pheromones in every step of the construction phase (except for the first step) the next operation to be scheduled is dependent on the operation scheduled in the previous step.

As heuristic information several priority rules where tried in a randomized way. E.g., the SPT (shortest processing time) rule was used as $\eta_{o_j} = \frac{1}{p(o_j)}$, $\forall o_j \in O$. As a pheromone update rule the original Ant System global pheromone update rule is used. The results obtained with this algorithm are far off the state-of-the-art. There are mainly two reasons for that: 1) Recent research showed that local search is needed to improve solutions constructed by the ants to assist in guiding the search. 2) The algorithm proposed does not include any mechanisms for additional intensification and diversification of the search process which is

needed for problems like Shop Scheduling problems where good solutions don't necessarily have parts in common (see [68]). For the sake of completeness we mention that exactly the same algorithm was reinvented by van der Zwaan and Marques in [105].

A second ACO approach for the JSP was proposed by Teich et al. in [104]. This algorithm also applies the mechanism of list scheduler algorithms to probabilistically construct solutions. The difference is in the pheromone model and the generation of the transition probabilities. Teich et al. applied the following pheromone model which was introduced by Merkle and Middendorf in [69] for permutation problems.

Learning of absolute positions in s: This model can be regarded as a standard in permutation type problems. To every operation $o_j \in O$ and every position i in a sequence s we have associated a pheromone value $\tau_{o_j,i}$. Teich et al. use two different strategies to generate the probabilities for the operations in set S' of the ant construction phase to be chosen by the ant (called transition probabilities). The first evaluation strategy is the standard evaluation:

$$p(o_j|s_{t-1,|O|},t) = \begin{cases} \frac{[\tau_{o_j,t}]\cdot[\eta_{o_j}]^\alpha}{\sum_{o_k\in S'}[\tau_{o_k,t}]\cdot[\eta_{o_k}]^\alpha} & : \text{ if } \quad o_j \in S' \\ 0 & : \text{ otherwise} \end{cases} \tag{2.7}$$

With this pheromone representation the algorithm tries to learn absolute positions of operations in a sequence s. The second evaluation strategy – called summing evaluation – was introduced in [69] and further tested in [70]. The transition probabilities are:

$$p(o_j|s_{t-1,|O|},t) = \begin{cases} \frac{\sum_{l=1}^{t}[\tau_{o_j,l}]\cdot[\eta_{o_j}]^\alpha}{\sum_{o_k\in S'}\sum_{l=1}^{t}[\tau_{o_k,l}]\cdot[\eta_{o_k}]^\alpha} & : \text{ if } \quad o_j \in S' \\ 0 & : \text{ otherwise} \end{cases} \tag{2.8}$$

In this way of evaluating the transition probabilities, if an operation is by some stochastical error not placed at a position in s where it should have been placed, the probability remains high to schedule it closely afterward. In the following we denote this pheromone model combined with standard evaluation by $\mathsf{PH}_{\mathsf{abs}}$ and combined with summing evaluation by $\mathsf{PH}_{\mathsf{sum}}$.

As heuristic information, Teich et al. used the SPT priority rule in the same way as outlined above. However, this algorithm neither reaches state-of-the-art performance.

2.5.2 ACO algorithms to tackle the OSP

To our knowledge there has been no ACO algorithm proposed to tackle the OSP. The algorithms proposed for the JSP can easily be adapted to work for the OSP though.

2.6 Other metaheuristic approaches

Except for the metaheuristic approaches mentioned in the previous sections there have been a few more approaches which we outline in the following.

2.6.1 A large-step optimization method to tackle the JSP

The large-step optimization method proposed by Lourenco [63] fits into a metaheuristic framework which is nowadays known as Iterated Local Search (ILS) [99, 64]. ILS applies local search to an initial solution until it finds a local optimum; then it perturbs the solution and it starts again a local search. On the basis of an acceptance criterion it is then decided whether the new local optimum is accepted as the new current solution. The importance of the perturbation is obvious: a too small perturbation might not enable the system to escape from the basin of attraction of the local optimum just found. On the other side, a too strong perturbation would make the algorithm similar to a random restart local search.
To tackle the JSP, Lourenco [63] introduced a combination of small step moves based on the neighborhood $\mathcal{N}_{1a}$ (this corresponds to applying iterative improvement local search) and large step moves in order to reach new areas in the search space (this corresponds to the perturbation mechanism). The large steps considered are as follows. Randomly select two machines and remove them from the current schedule. Then solve the one-machine problem for each of the two machines and insert the obtained machine sequences into the current schedule. Starting solutions are generated through the application of a randomized GT algorithm. The results obtained are not state-of-the-art.

2.6.2 A GRASP to tackle the JSP

The Greedy Randomized Adaptive Search Procedure (GRASP), see [38, 87], is a simple metaheuristic approach which combines constructive heuristics and local search. GRASP is an iterative procedure, composed of two phases: solution construction and solution improvement. In the construction phase, a feasible solution is built, one element at a time. At each construction iteration, the next element to be added is determined by ordering all elements in a candidate list with respect to a greedy function that measures the benefit of selecting each element. The adaptive component of GRASP arises from the fact that the benefits associated with every element are updated at each iteration of the construction phase to reflect the changes caused by the selection of previous elements. The probabilistic component of GRASP is characterized by the random choice of an element from a restricted candidate list in every construction step.
Binato et al. [14] propose a GRASP algorithm to tackle the JSP. Their algorithm is characterized by a construction phase based on list scheduler algorithms and by a local search phase based on the $\mathcal{N}_{1a}$ neighborhood (see Section 2.3.1). The construction steps are influenced by a set of elite solutions found in the course of the search process. Solution components to be found in elite solutions are preferred in a randomized way while constructing a solution. Another interesting feature of this algorithm is the *Proximate Optimality Principle* (POP) introduced by Glover and Laguna in [44]. This principle states, that good partial solutions on a number of $x < |O|$ operations are similar to good partial solutions when one operation is added to the x operations. This principle in mind, iterative improvement based on neighborhood $\mathcal{N}_{1a}$ is applied to partial solutions at certain stages of the construction process.
This algorithm obtains good solutions for relatively easy problem instances. For difficult problem instances the results are quite far off the best known solutions though.

2.6.3 A variable depth search method to tackle the JSP

One of the most powerful and certainly one of the state-of-the-art algorithms to tackle the JSP is a method called *Guided Local Search*[11] (GLS) proposed by Balas and Vazacopoulos in [9]. Their algorithm is based on a neighborhood where a varying number of processing orders is reversed to obtain the neighbors. The neighborhood is defined as follows.

Neighborhood $\mathcal{N}_7$: A neighbor s' of a solution s is obtained by an interchange of two operations o and o' in the same machine block on a critical path of s. Either operation o' is the last one in the block and there is no directed path in the disjunctive graph corresponding to s connecting $js(o)$ to o', or, operation o is the first one in the block and there is no directed path connecting o to $jp(o')$.

This neighborhood is further restricted by applying experience in the form of so-called *guide-posts* gathered during the search process. For details we refer to [9]. GLS works by building up an incomplete enumeration tree (called neighborhood tree). Each node of such a tree corresponds to a solution. An edge of a tree joins two solutions s and s' where the child s' is obtained through applying a $\mathcal{N}_7$ move to s involving two operations o and o' as outlined above. The reversed processing order of o and o' is preserved in all nodes in the subtree rooted in s'. The children of a node are ranked by their evaluations. The number of children is limited by a decreasing function of the depth in the neighborhood tree. Finally, besides fixing of certain processing orders and restricting the number of children, the depth of a tree is limited by a logarithmic function of the number of operations on the tree's level. Altogether the size of the neighborhood tree is bounded by a linear function of the number of operations. GLS works by iteratively constructing neighborhood trees. If a tree contains the best solution found so far, it is accepted as the next tree root. Otherwise, a random solution in the current neighborhood tree is chosen as the root of the next tree. The algorithm starts with a solution constructed by the GT algorithm using the MWR (most work remaining) priority rule.
In the same paper, Balas and Vazacopoulos outline some combinations of GLS and the shifting bottleneck procedure. The idea is to replace the re-optimization cycle of the shifting bottleneck procedure with applying GLS instead. So, whenever there are a number of m_0 fixed machine sequences defining a partial solution s_p, GLS is applied for a certain number of neighborhood tree generations. The root of the first tree is defined by the partial solution s_p. The machine sequences defined by the best solution found by GLS are then used to continue the shifting bottleneck procedure. This enhancement of the shifting bottleneck procedure works extremely well for a lot of benchmark problems, both in time and in solution quality.

[11] The name of this algorithm is not to be mistaken with the general concept metaheuristic Guided Local Search introduced by Voudouris and Tsang in [107].

Chapter 3

Metaheuristics for Group Shop Scheduling

The Metaheuristic Network [114] is a Research Training Network funded by the Improving Human Potential program of the CEC. It aims at the comparison of metaheuristics on different combinatorial optimization problems. For each combinatorial optimization problem considered, five metaheuristics are implemented by different people in different sites involved in the Metaheuristics Network. The five metaheuristics considered are: Ant Colony Optimization (ACO), Evolutionary Computation (EC), Iterated Local Search (ILS), Tabu Search (TS), and Simulated Annealing (SA). This chapter is devoted to the research conducted in the course of the Metaheuristics Network on the Group Shop Scheduling problem (GSP). As the author of this thesis developed and implemented the ACO metaheuristic to tackle the GSP, the focus of this chapter is on the ACO metaheuristic and the research results published in the following papers:

- C. Blum, A. Roli and M. Dorigo. HC-ACO: The Hyper-Cube Framework for Ant Colony Optimization. In *Proceedings of the 2001 Metaheuristics International Conference, MIC'01*, 2002 [17].
- C. Blum and M. Sampels. Ant Colony Optimization for FOP Shop scheduling[1]: A case study on different pheromone representations. In *Proceedings of the 2002 Congress on Evolutionary Computation, CEC'02 (to appear)*, 2002 [18].
- C. Blum and M. Sampels. When Model Bias is Stronger than Selection Pressure. Submitted to the *2002 Conference on Parallel Problem Solving in Nature, PPSN'02*, [19].
- M. Sampels, C. Blum, M. Mastrolilli and O. Rossi-Doria. Metaheuristics for Group Shop Scheduling. Submitted to the *2002 Conference on Parallel Problem Solving in Nature, PPSN'02*, [95].
- C. Blum. ACO applied to Group Shop Scheduling: A case study on Intensification and Diversification. Submitted to *ANTS2002*, [16].

[1] For historical reasons, the GSP was called FOP Shop Scheduling in [18]. However, the name Group Shop Scheduling fits better to the structure of the problem.

The structure of this chapter is as follows. First, the developed metaheuristics are described. Then we compare these metaheuristics on GSP instances from the range between JSP and OSP – the two extreme cases of the GSP – in order to further improve the understanding of the differences between OSP and JSP. Often when reading research papers from the literature, it becomes obvious that comparisons of metaheuristics are not done in a fair way. Usually, researchers compare their results with results from the literature. There are several pitfalls when doing that: (i) the paper where the results are taken from might be pulished much earlier such that the computation times are not comparable, because the advance in computing power from year to year is tremendous, (ii) the compared algorithms might be implemented on completely different data structures, such that again the results are not comparable, (iii) the compared algorithms might be implemented in different programming languages. In the Metaheuristics Network we tried to eliminate these factors in metaheuristic comparison by fixing the framework for the comparison in the following way:

- The programming language was decided to be C++. The compiler chosen was the GNU C++ compiler gcc, version 2.95.3.
- The basic data structures to hold the problem data and to represent solutions were provided. Furthermore, a neighborhood structure to be outlined in the following section was provided. Every algorithm based on neighborhood structures (such as TS, SA, or local search to be used inside the population-based metaheuristics) had to be based on this neighborhood structure.
- The metaheuristics were all tested on a beowulf cluster consisting of 8 PCs with AMD Athlon 1100 Mhz CPU under Linux.

By fixing this framework we created the basis for a fairer comparison of metaheuristics than what is done usually.

3.1 Common neighborhood and local search

In Def. 3 of Section 2.3.1 and in Def. 4 of Section 2.3.2 we gave the definitions of machine blocks and more general blocks (including machine blocks and job blocks) on critical paths of solutions to the JSP and the OSP. In the following we generalize these definitions to obtain a definition of blocks for the GSP.

Definition 5 *Given a solution s to an instance of the GSP, a* **block** *is a maximal sequence (of size at least one) of operations on the same machine or of the same group on a critical path of s. An operation o in a block is called an* **internal** *operation if it is neither the first nor the last operation in this block.*

According to this block definition we generalize the neighborhood structure $\mathcal{N}_{1c}$ – introduced by Nowicki and Smutnicki in [79] for the JSP – for the application to the GSP.

Neighborhood structure $\mathcal{N}_{1c,GSP}$: The neighborhood of a solution s is defined by all solutions s' which can be generated by reversing the processing orders of the first two operations

or the last two operations in a block of a critical path with respect to a solution s. Excluded are the first two operations in the first block and the last two operations in the last block.

By providing the neighborhood structure to be used by any local search method we make sure that every metaheuristic searches the same landscape, which might enable us to draw conclusions on which metaheuristic searches this landscape in a more effective way than others.

3.2 Ant Colony Optimization

The ACO algorithm outlined in this section[2] is characterized by a new pheromone model combined with a new evaluation strategy. Furthermore, it is implemented in the Hyper-Cube Framework, which is a certain way of implementing ACO algorithms proposed by Blum et al. in [17]. Based on the Hyper-Cube Framework, additional intensification and diversification mechanisms are introduced into the search process by means of a list of elite solutions found in the course of the search.

3.2.1 A new pheromone model $\mathsf{PH}_{\mathsf{rel}}$

As outlined in Section 2.5, there have been two different ACO approaches to tackle the JSP. They use three different "pheromone-model"-"pheromone-evaluation-strategy" combinations. These are (i) learning of successor relations in sequences to be built combined with the standard evaluation ($\mathsf{PH}_{\mathsf{suc}}$), (ii) learning of absolute positions of operations in the sequences to be built combined with the standard evaluation ($\mathsf{PH}_{\mathsf{abs}}$), and (iii) learning of absolute positions combined with the summing evaluation ($\mathsf{PH}_{\mathsf{sum}}$). However, all three combinations seem to model the GSP, and Shop Scheduling problems in general, in a somewhat artificial way. Combination (i) is artificial because by using this combination the algorithm tries to learn successor relationships among the operations in sequences s to be generated by the algorithm. This means that the algorithm potentially learns relationships between operations which are not related at all (which means that they are not to be processed on the same machine or that they are not in the same group). Combinations (ii) and (iii) are artificial because an algorithm using these "pheromone-model"-"pheromone-evaluation-strategy" combinations learns absolute positions in the sequences s generated by the algorithm and does only implicitly take into account the relations among operations. By introducing the following pheromone model in [18] we intended to introduce a more natural modeling of Shop Scheduling problems.

Learning of relations among operations: In this new pheromone model – which we called $\mathsf{PH}_{\mathsf{rel}}$ in [18] – we assign pheromone values to pairs of *related* operations. We call two operations $o_i, o_j \in O$ *related* if they belong to the same group, or if they have to be processed on the same machine. Formally, a pheromone value τ_{o_i,o_j} for a pair of operations $o_i, o_j \in O$, where $o_i \neq o_j$, exists iff $g(o_i) = g(o_j)$ or $m(o_i) = m(o_j)$.

The meaning of a pheromone value τ_{o_i,o_j} is that if τ_{o_i,o_j} is high then operation o_i should be scheduled before operation o_j. The choice of the next operation to be scheduled is handled as follows. If in step t of the construction mechanism (which is the list scheduler algorithm) there is an operation $o_i \in S'_t$ (remember that S'_t is the restricted set of operations which can be scheduled in step t of the construction phase) with no related and unscheduled operations

[2] This ACO algorithm was developed at IRIDIA, Université Libre de Bruxelles, Belgium, by Christian Blum.

left, it is chosen. Otherwise we choose among the operations of set S'_t with the following transition probabilities:

$$p(o \mid s_{t-1,|O|}) = \begin{cases} \frac{\min_{o_r \in S_o^{rel}} \tau_{o,o_r}}{\sum_{o_k \in S'_t} \min_{o_r \in S_{o_k}^{rel}} \tau_{o_k,o_r}} & : \text{ if } \quad o \in S'_t \\ 0 & : \text{ otherwise} \end{cases}$$

where $S_o^{rel} = \{o' \in O \mid m(o') = m(o) \vee g(o') = g(o),\ o' \text{ not scheduled yet}\}$, and $s_{t-1,|O|}$ is a partial sequence of length $t-1$ and final length $|O|$. The meaning of this rule for computing the transition probabilities is the following: If at least one of the pheromone values between an operation $o_i \in S'_t$ and a related operations o_r that is not scheduled yet is low, then the operation o_i probably should not be scheduled now. By using this pheromone model the algorithm tries to learn relations between operations. The absolute position of an operation in the sequence s is not important anymore. Of importance, is the relative position of an operation with respect to the related operations.

As shown in [18], $\mathsf{PH_{rel}}$ appears to be superior to the other "pheromone-model"-"pheromone-evaluation-strategy" combinations proposed for Shop Scheduling type problems. In particular – as shown in [19] – unlike other pheromone models, $\mathsf{PH_{rel}}$ does not introduce an overly strong model bias potentially leading the search process to low quality areas in the search space.

3.2.2 The Hyper-Cube Framework for Ant Colony Optimization

In most ACO implementations the hyperspace for the pheromone values used by the ants to construct solutions is only implicitely limited. In contrast, the Hyper-Cube Framework – introduced in [17] – provides a way of implementing ACO algorithms such that the hyperspace for the pheromone values is explicitly know to be $[0,1]^z$ where z is the number of pheromone values. In the following we describe the Hyper-Cube Framework on the pheromone model introduced in the previous section. The Hyper-Cube Framework is characterized by a pheromone update rule where the relative difference between the qualities of the solutions produced in an iteration are important rather than the absolute qualities of these solutions. In general, any pheromone update rule can be adapted to work in the Hyper-Cube Framework. In order to present the idea, we choose the usual Ant System global pheromone update rule as proposed in [33]. For the pheromone values τ_{o_i,o_j} of pheromone model $\mathsf{PH_{rel}}$ introduced in the previous section the usual Ant System global pheromone update rule is the following one.

$$\tau_{o_i,o_j} \leftarrow (1-\rho) \cdot \tau_{o_i,o_j} + \sum_{l=1}^{k} \Delta^{s_l} \tau_{o_i,o_j} \tag{3.1}$$

where

$$\Delta^{s_l} \tau_{o_i,o_j} \quad = \quad \begin{cases} \frac{1}{C_{\max}(s_l)} & \text{if } o_i \text{ before } o_j \text{ in } s_l, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

In the Hyper-Cube Framework a normalization of the contribution of every solution used for updating the pheromone values is done in the following way.

$$\tau_{o_i,o_j} \leftarrow (1-\rho) \cdot \tau_{o_i,o_j} + \rho \cdot \sum_{l=1}^{k} \Delta^{s_l} \tau_{o_i,o_j} \tag{3.3}$$

where

$$\Delta^{s_l}\tau_{o_i,o_j} = \begin{cases} \frac{\frac{1}{C_{\max}(s_l)}}{\sum_{r=1}^{k}\frac{1}{C_{\max}(s_r)}} & \text{if } o_i \text{ before } o_j \text{ in } s_l, \\ 0 & \text{otherwise.} \end{cases} \tag{3.4}$$

where we multiply the sum of normalized contributions with the evaporation rate ρ. The Hyper-Cube Framework allows a nice interpretation of the pheromone update rule. In the following we regard the set of all pheromone values as a vector $\vec{\tau}$ where each position is associated with exactly one of the pheromone values[3]. Remember that the meaning of each pheromone value τ_{o_i,o_j} is the following: If τ_{o_i,o_j} is high – which means close to 1 – then o_i should be scheduled before o_j, and vice versa. This means that we can regard a solution s as a vector $\vec{s}$ of the same size as $\vec{\tau}$ where a position associated to pheromone value τ_{o_i,o_j} is set to 1, if o_i is scheduled before o_j in s, and 0 otherwise. Now we transform equations (3.3) and (3.4) to obtain the following pheromone update rule.

$$\tau_{o_i,o_j} \leftarrow \tau_{o_i,o_j} + \rho \cdot \left(\left(\sum_{l=1}^{k} \frac{\frac{1}{C_{\max}(s_l)} \cdot \delta(o_i, o_j, s_l)}{\sum_{r=1}^{k}\frac{1}{C_{\max}(s_r)}} \right) - \tau_{o_i,o_j} \right) \tag{3.5}$$

where

$$\delta(o_i, o_j, s) = \begin{cases} 1 & \text{if } o_i \text{ before } o_j \text{ in } s, \\ 0 & \text{otherwise.} \end{cases} \tag{3.6}$$

By defining

$$\vec{d} \leftarrow \sum_{l=1}^{k} \left(\frac{\frac{1}{C_{\max}(s_l)}}{\sum_{r=1}^{k}\frac{1}{C_{\max}(s_r)}} \right) \cdot \vec{s_l} \tag{3.7}$$

we can rewrite equation (3.5) in vector form as

$$\vec{\tau} \leftarrow \vec{\tau} + \rho \cdot \left(\vec{d} - \vec{\tau} \right) \tag{3.8}$$

where $\vec{d}$ is the weighted sum of solution vectors $\vec{s_l}$, $l = 1, ..., k$. This means that the Ant System global pheromone update rule in the Hyper-Cube Framework shifts the vector $\vec{\tau}$ with stepsize ρ toward the weighted sum of solution vectors $\vec{s_l}$. This also implies that if the algorithm starts with pheromone values from the interval $[0, 1]$ then they will have a lower bound 0 and an upper bound 1. If the algorithm even starts with an initial pheromone vector in the convex hull of feasible solution vectors, then it will never leave this convex hull. This view is graphically presented in Figure 3.1.

3.2.3 $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System for the GSP

$\mathcal{MM}$AS was proposed by Stützle and Hoos as an improvement of the original Ant System (AS) proposed by Dorigo et al. in [33]. $\mathcal{MM}$AS differs from AS by applying a lower and an upper bound, $\tau_{\min}$ and $\tau_{\max}$, on the pheromone values. The lower bound (a small positive

[3] This means that the vector $\vec{\tau}$ has as many positions as the cardinality of the set of pheromone values.

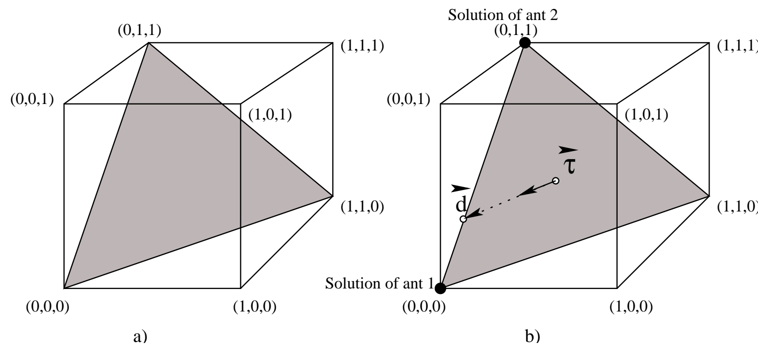

Figure 3.1: The Hyper-Cube Framework: An example for an artificial problem instance modeled by three pheromone values. The length of the pheromone vector $\vec{\tau}$ and the length of the solution vectors $\vec{s}$ is therefore 3. We assume to have three different feasible solutions: $(0,0,0)$, $(1,1,0)$ and $(0,1,1)$. The gray shaded area is the convex hull of the feasible solutions, which corresponds to the hyperspace in which the pheromone vector moves. In b) a situation is depicted where the algorithm has produced two solutions $(0,0,0)$ and $(0,1,1)$. We assume $(0,0,0)$ to have the shorter makespan. Therefore, vector $\vec{d}$, which is the weighted sum of these two solutions, is closer to $(0,0,0)$ than to $(0,1,1)$. The update rule will then shift vector $\vec{\tau}$ toward this vector $\vec{d}$.

constant) is preventing the algorithm from converging[4] to a solution. Another important feature of $\mathcal{MM}$AS is that – due to a quite aggressive pheromone update – the algorithm concentrates quickly on an area in the search space. When the system is stuck in an area of the search space[5] the best solution found since the start of the algorithm is used to update the pheromone values in every iteration until the algorithm gets stuck again. After that, a restart is performed. The reason for doing that is the hope of finding a better solution between the restart best solution and the best solution found since the start of the algorithm. This mechanism is clearly an intensification mechanism. Additional diversification is reached by the original $\mathcal{MM}$AS by restarting the algorithm with equal pheromone values. This way of diversifying the search process is not guided by any search history.
The framework of our $\mathcal{MM}$AS algorithm for the GSP is shown in Algorithm 14. The most important features of this algorithm are explained in the following.

In Algorithm 14, $\tau = \{\tau_1, ..., \tau_l\}$ is a set of pheromone values, n_a is the number of ants, and s_j is a solution to the problem – a sequence containing all operations – constructed by ant j, where $j = 1, ..., n_a$. Furthermore, s_{ib} denotes the best solution constructed in an iteration, s_{rb} denotes the best solution found in a *restart phase* of the algorithm (a phase where $glob_conv == FALSE$), and s_{gb} denotes the best solution found since the start of the algorithm.

InitializePheromoneValues(τ): In the basic version of the algorithm we initialize all the pheromone values to the same positive constant 0.5.

[4]In the course of this work we refer to convergence of an algorithm as a state where only one solution has a probability greater than 0 to be generated.
[5]This is usually determined by some convergence measure.

Algorithm 14 $\mathcal{MM}$AS for the GSP

```
s_gb ← NULL
s_rb ← NULL
cf ← 0
glob_conv ← FALSE
InitializePheromoneValues(τ)
while termination conditions not met do
  for j = 1 to n_a do
    s_j ← ConstructSolution(τ)
    LocalSearch(s_j)
  end for
  s_ib ← argmin(C_max(s_1), ..., C_max(s_{n_a}))
  Update(s_ib, s_rb, s_gb)
  if glob_conv == FALSE then
    ApplyOnlineDelayedPheromoneUpdate(τ,s_rb)
  else
    ApplyOnlineDelayedPheromoneUpdate(τ,s_gb)
  end if
  cf ← ComputeConvergenceFactor(τ)
  if cf ≥ 0.99 AND glob_conv == TRUE then
    ResetPheromoneValues(τ)
    s_rb ← NULL
    glob_conv ← FALSE
  else
    if cf ≥ 0.99 AND glob_conv == FALSE then
      glob_conv ← TRUE
    end if
  end if
end while
```

ConstructSolution(τ): An important part of an ACO algorithm is the constructive mechanism used to probabilistically construct solutions. We used the mechanism of a list scheduler algorithm as outlined in Algorithm 1 in Section 1.4.1. To summarize, the list scheduler algorithm builds a sequence s of all operations – starting with an empty sequence – by performing $|O|$ steps as follows:

1. Create a set S_t (where t is the step number) of all operations that can be scheduled next.

2. Use a heuristic to produce a set $S'_t \subseteq S_t$.

3. Use a heuristic to pick an operation $o \in S'_t$ to be scheduled next. Add operation o to the partial sequence $s_{t-1,|O|}$.

Remember that a sequence s of all operations is a total order on all operations that specifies a total order on the operations of each group and of each machine. This unambiguously defines

a solution to an instance of the problem. The construction mechanism applied by the ants chooses $S'_t = S_t$ in step 2, and in step 3 it chooses among the operations in S'_t probabilistically. The probabilities for the operations in S'_t (called transition probabilities) to be chosen, depend on the pheromone model. For our algorithm we choose the pheromone model $\mathsf{PH_{rel}}$ proposed by Blum et al. in [18] and outlined in the previous section. The transition probabilities – biased by heuristic information – are generated as follows:

$$p(o \mid s_{t-1,|O|}) = \begin{cases} \frac{\left(\min_{o_r \in S_o^{rel}} \tau_{o,o_r}\right) \cdot \eta_o{}^{\alpha}}{\sum_{o_k \in S'_t} \left(\min_{o_r \in S_{o_k}^{rel}} \tau_{o_k,o_r}\right) \cdot \eta_{o_k}{}^{\alpha}} & : \text{ if } \quad o \in S'_t \\ 0 & : \text{ otherwise,} \end{cases}$$

where $S_o^{rel} = \{o' \in O \mid m(o') = m(o) \vee g(o') = g(o), o' \text{ not scheduled yet}\}$. As heuristic information η_o we use the inverse of the earliest starting time of an operation o with respect to the current partial schedule $s_{t-1,|O|}$[6], and α is a parameter to adjust the influence of the heuristic information η_o.

$\mathsf{LocalSearch}(s_j)$: To every solution s_j constructed by the ants, a best improvement local search based on neighborhood structure $\mathcal{N}_{1c,GSP}$ as outlined in Section 3.1 is applied.

$\mathsf{Update}(s_{ib}, s_{rb}, s_{gb})$: This function updates solutions s_{rb} and s_{gb} with the iteration best solution s_{ib}. s_{rb} is replaced by s_{ib}, if $C_{\max}(s_{ib}) < C_{\max}(s_{rb})$. The same holds for s_{gb}.

$\mathsf{ApplyOnlineDelayedPheromoneUpdate}(\tau, s)$: The algorithm is implemented in the Hyper-Cube Framework described in Section 3.2.2. $\mathcal{MM}$AS algorithms usually apply a pheromone update rule which (depending on some convergence measure) uses the iteration best solution s_{ib}, the restart best solution s_{rb} and s_{gb}, the best solution found since the start of the algorithm, to update the pheromone values. Our algorithm only uses the restart best solution and the global best solution. The reason for that is the different structure of the scheduling problems covered by the GSP. Preliminary experiments showed that for OSP instances a much higher selection pressure is needed to make the algorithm converge than for JSP instances. The implications in practice are that the use of the iteration best solution s_{ib} for updating the pheromone values would have to be fine-tuned depending on the problem instance structure. To avoid this we decided against using the iteration best solution.
As our algorithm – at any time – only uses one solution (s_{rb} or s_{gb}) for updating the pheromone values, we can specify the pheromone updating rule for our algorithm deriving it from equation (3.5) as follows.

$$\tau_{o_i,o_j} \leftarrow f_{mmas}\left(\tau_{o_i,o_j} + \rho \cdot (\delta(o_i, o_j, s) - \tau_{o_i,o_j})\right) \tag{3.9}$$

where the delta-function is as defined in equation (3.6), and

$$f_{mmas}(x) \quad = \quad \begin{cases} \tau_{\min} & \text{if } x < \tau_{\min}, \\ x & \text{if } \tau_{\min} \le x \le \tau_{\max}, \\ \tau_{\max} & \text{if } x > \tau_{\max}. \end{cases} \tag{3.10}$$

[6] We add 1.0 to all earliest starting times in order to avoid division by 0.

We set the lower bound $\tau_{\min}$ for the pheromone values to 0.001 and the upper bound[7] $\tau_{\max}$ to 0.999. Therefore, after applying the pheromone update rule above, we set the pheromone values that exceed the upper bound or are below the lower bound back to the respective bound.

ComputeConvergenceFactor(τ): To measure the "extent of being stuck" in an area in the search space we compute after every iteration a so–called *convergence factor* cf. We compute this factor as follows.

$$cf \leftarrow \left(\left(\frac{\sum_{o_i \neq o_j,\text{ related}} \max\{\tau_{\max} - \tau_{o_i,o_j}, \tau_{o_i,o_j} - \tau_{\min}\}}{\sum_{o_i \neq o_j,\text{ related}} \tau_{\max} - \tau_{\min}} \right) - 0.5 \right) \cdot 2.0 \tag{3.11}$$

From the formula above it becomes clear that when the algorithm is initialized (or restarted) with pheromone values all 0.5, cf is 0.0 and when all pheromone values are either equal to $\tau_{\min}$ or equal to $\tau_{\max}$, cf is 1.0.

ResetPheromoneValues(τ): In the basic version of our algorithm we reset all the pheromone values to the same positive constant 0.5.

This concludes the description of the basic $\mathcal{MM}$AS for the GSP (henceforth identified by U for uniform initialization and resetting of pheromone values). As mentioned before, Shop Scheduling problems are in general multimodal problems in the sense that good solutions are scattered all over the search space. Therefore we expect to be able to improve the basic algorithm presented in this section with additional intensification and diversification mechanisms.

3.2.4 Intensification and diversification strategies

Intensification and diversification of the search process are quite unexplored topics in ACO research. There are just a few papers explicitly dealing with the topic. The mechanisms already exisiting can be divided into two different categories. The first one consists of mechansims changing in some way the pheromone values, either on-line (e.g., [32, 91]) or by resetting the pheromone values (e.g., [100, 103]). The second category consists of algorithms applying multiple colonies and exchanging information between them in some way (e.g., [72, 56]). In contrast to that, most of the intensification and diversification mechanisms to be outlined in the following are based on a set of elite solutions found by the algorithm in the history of the search process.
As a first strategy to introduce more intensification and diversification into the the search process performed by Algorithm 14, we changed the functions to initialze and reset the pheromone values. In an algorithm henceforth identified by R (for random pheromone setting) we use a pheromone value initialization and resetting function that generates for every pheromone a value uniformly random between 0.25 and 0.75[8]. This introduces more intensification, because right from the start of a restarting phase the algorithm is more focused on a certain

[7] Note that in contrast to the usual way of implementing ACO algorithms, in the Hyper–Cube Framework also the upper bound for pheromone values is necessary to avoid convergence to a solution.

[8] Note that the Hyper–Cube framework facilitates a strategy like that, as we explicitly know the space in which the pheromone values move to be $[0, 1]^z$ where z is the number or pheromone values.

area in the search space given by the randomly chosen values for the pheromones. On the other side more diversification is introduced, because with every restart the algorithm focuses probably on a different area of the search space.
Examining the behaviour of the $\mathcal{MM}$AS algorithm presented in the previous section we noticed that the algorithm wastes time by always – at the end of a restart phase – moving towards the best solution found since the start of the algorithm. After some while there are no improvements to be found around this solution. The strategy of always moving toward the best found solution might work well for problems like the Traveling Salesman Problem with a fitness-distance correlation[9], for problems without a fitness-distance correlation other strategies seem to be required. To change this strategy we keep a list L_{elite} of elite solutions found during the search. Henceforth, we call the solutions s_{rb} – the best solutions found in a restart phase of the algorithm – elite solutions. L_{elite} works as a FIFO list and has a maximum length l. The framework of the algorithm using this list of elite solutions, henceforth called E-$\mathcal{MM}$AS, is given in Algorithm 15.

In Algorithm 15, if *loc_conv* == FALSE and *glob_conv* == FALSE we say the algorithm is in a *restart phase*. In case *loc_conv* == TRUE and *glob_conv* == FALSE we say the algorithm is in the *phase of local convergence*, and in case *loc_conv* == FALSE and *glob_conv* == TRUE we say the algorithm is in the *phase of global convergence*. In Algorithm 15, the phase of convergence to the best solution found since the start of the algorithm as shown in Algorithm 14 is replaced by consecutive phases of local and global convergence. In the phase of local convergence the system moves toward a solution $s_{closest}$, which is closest to s_{rb} in L_{elite} with respect to the following distance measure:

$$d_h(s_1, s_2) = \sum_{o_i \neq o_j,\ \text{related}} \delta(o_i, o_j, s_1, s_2), \tag{3.12}$$

where $\delta(o_i, o_j, s_1, s_2) = 0$ if the processing order between o_i and o_j is the same in both solutions s_1 and s_2, and $\delta(o_i, o_j, s_1, s_2) = 1$ otherwise. By doing that the algorithm tries to improve solution $s_{closest}$. In the phase of global convergence, the system moves toward a solution s_{best}, which is the best quality solution among the solutions in L_{elite}. The components of Algorithm 15 different or additional to the components of Algorithm 14 are outlined in the following.

AddToEliteList(L_{elite},s_{rb}): Every solution s_{rb} at the end of a restart phase is added to L_{elite}. At the beginning of the algorithm the list is empty. In a situation where the length of L_{elite} is smaller than l, the new solution is just added. In case the length of L_{elite} is l, additionally the first element of L_{elite} is removed.

UpdateEliteList(L_{elite},s,s_{rb}): At the end of the phases of local or global convergence it is checked if $s_{closest}$, respectively s_{best}, was improved. If this is the case, $s_{closest}$, respectively s_{best}, is removed from the list L_{elite} and the improved solution s_{rb} is added at the end of L_{elite}. This mechanism implies that a solution, once added to L_{elite}, has $|L_{elite}|$ (the length of

[9]Informally, the expression *fitness-distance correlation* denotes the existence of the following property: The higher the quality of a solution, the higher is the probabilitiy that it has many parts in common with the global optimum.

Algorithm 15 E-$\mathcal{MM}$AS for the GSP

$s_{gb} \leftarrow$ NULL, $s_{rb} \leftarrow$ NULL, $s_{closest} \leftarrow$ NULL, $s_{best} \leftarrow$ NULL, $cf \leftarrow 0$
$loc_conv \leftarrow$ FALSE, $glob_conv \leftarrow$ FALSE
InitializePheromoneValues(τ)
while termination conditions not met **do**
  for $j = 1$ to n_a **do**
    $s_j \leftarrow$ ConstructSolution(τ)
    LocalSearch(s_j)
  end for
  $s_{ib} \leftarrow$ argmin($C_{\max}(s_1), ..., C_{\max}(s_{n_a})$)
  Update(s_{ib}, s_{rb}, s_{gb})
  ApplyOnlineDelayedPheromoneUpdate(τ,s_{rb})
  $cf \leftarrow$ ComputeConvergenceFactor(τ)
  if $cf \geq 0.99$ AND $glob_conv$ == FALSE AND loc_conv == FALSE **then**
    AddToEliteList(L_{elite},s_{rb})
    $s_{closest} \leftarrow \text{argmin}\{d_h(s_{rb}, s) \mid s \in L_{elite}, s \neq s_{rb}\}$ {*see text for definition of* $d_h(.,.)$}
    $s_{rb} \leftarrow s_{closest}$
    $loc_conv \leftarrow$ TRUE
  else
    if $cf \geq 0.99$ AND $glob_conv$ == FALSE AND loc_conv == TRUE **then**
      UpdateEliteList(L_{elite},$s_{closest}$,s_{rb})
      $s_{best} \leftarrow \text{argmin}\{C_{\max}(s) \mid s \in L_{elite}\}$
      $s_{rb} \leftarrow s_{best}$
      $loc_conv \leftarrow$ FALSE
      $glob_conv \leftarrow$ TRUE
    else
      if $cf \geq 0.99$ AND $glob_conv$ == TRUE AND loc_conv == FALSE **then**
        UpdateEliteList(L_{elite},s_{best},s_{rb})
        ResetPheromoneValues(τ)
        $s_{rb} \leftarrow$ NULL, $s_{closest} \leftarrow$ NULL, $s_{best} \leftarrow$ NULL
        $glob_conv \leftarrow$ FALSE
        $loc_conv \leftarrow$ FALSE
      end if
    end if
  end if
end while

L_{elite}) restart phases of the algorithm to be improved. If it is improved it has again the same time to be improved again. If a solution in L_{elite} is not improved within $|L_{elite}|$ restart phases of the algorithm, it is dropped from L_{elite} and its neighborhood is regarded as an explored region of the search space.

The intuition behind the usage of a list of elite solutions as described above is the following. Instead of only on one area – as done in the usual $\mathcal{MM}$AS – our algorithm works on several areas in the search space trying to improve the best solutions found in these areas. If it can not improve them in a certain amount of time they are discarded even if they contain the best solution found since the start of the algorithm. This prevents the algorithm from wasting time and acts as a diversifying component. This mechanism also incorporates a strong intensifying component by applying the phase of local convergence and the phase of global convergence as described above.
Algorithm E-$\mathcal{MM}$AS using the random pheromone initialization and resetting as described at the beginning of this section is henceforth identified by ER. We also developped two more ways of resetting the pheromone values in Algorithm E-$\mathcal{MM}$AS. The first one is aiming at a diversification of the search depending on the elite solutions in L_{elite}. In this scheme the pheromone values are reset as follows.

$$\tau_{o_i,o_j} \leftarrow f\left(\frac{\sum_{s \in L_{elite}} \delta(o_i,o_j,s)}{|L_{elite}|}\right) \text{ where } f(x) = \begin{cases} 0.5 + x & \text{if } x < 0.25, \\ 1.0 - x & \text{if } 0.25 \leq x \leq 0.75, \\ x - 0.5 & \text{if } x > 0.75 \end{cases} \tag{3.13}$$

The delta-function is the same as defined in (3.6). The intuition of this setting is that we want to reset the pheromone values in order to concentrate the search in areas of the search space different from the current set of elite solutions. However, the more agreement is found among the elite solutions about an order between two related operations o_i and o_j, the more we rely on the accuracy of this order and the resetting of the corresponding pheromone values is approximating equal chance for both directions (cases $x < 0.25$ and $x > 0.75$ in (3.13)). Algorithm E-$\mathcal{MM}$AS using for the first three restart phases the random setting of pheromone values and afterwards this diversification scheme is henceforth identified by ED.
Another scheme for resetting pheromone values aims at an intensification of the search process. First the best solution s_{best} among the elite solutions is chosen, then another one s_{second} different from s_{best} is chosen from L_{elite} uniformly random. Using these two solutions, the resetting of pheromone values is done as follows.

$$\tau_{o_i,o_j} \leftarrow f_{mmas}\left(\frac{\delta(o_i,o_j,s_{best}) + \delta(o_i,o_j,s_{second})}{2.0}\right) \tag{3.14}$$

where the delta-function is as defined in (3.6) and the function f_{mmas}, which keeps the pheromone values in their bounds, is as defined in (3.10). Algorithm E-$\mathcal{MM}$AS using for the first three restart phases the random setting of pheromone values and fliping a coin for all consecutive restart phases to choose among the diversification setting of pheromone values described above and the intensification setting is henceforth identified by EDI.

3.2.5 Choice of an ACO algorithm for the comparison

In order to choose one of the ACO algorithms for the metaheuristics comparison, we ran experiments on the five different algorithm options outlined in the previous sections. These

are: U, R, ER, ED and EDI. Additionally, we tested a further enhencement of EDI where we incorporated a short Tabu Search run based on the neighborhood structure $\mathcal{N}_{1c,GSP}$ applied to the best ant per iteration. We fixed the length of this Tabu Search run to $|O|/2$. This version of the algorithm is henceforth identified by EDI_{TS}. A summary of the characteristics of the different algorithm options is given in Table 3.1. We tested these six versions on three

Table 3.1: Different versions of the ACO algorithm to tackle the GSP

Identifier	Characteristics
U	$\mathcal{MM}$AS using uniform setting of pheromone values
R	$\mathcal{MM}$AS using random setting of pheromone values
ER	E-$\mathcal{MM}$AS using random setting of pheromone values
ED	E-$\mathcal{MM}$AS using random setting for the first three restart phases, and after that the diversification setting of pheromone values
EDI	E-$\mathcal{MM}$AS using random setting for the first three restart phases, and after that fliping a coin in every restart phase to choose among diversification setting and intensification setting
EDI_{TS}	The same as EDI with an additional short Tabu Search run for the best ant in every iteration

problem instances. We chose the first problem tai_15_15_1_jsp with 15 jobs and 15 machines introduced by Taillard in [101] for the JSP, and the first problem tai_15_15_1_osp with 15 jobs and 15 machines also introduced in [101] for the OSP. The optimal solutions for these problems are known to be 1231, and 937 respectively. As a third problem we chose whizzkids97 which is a very difficult GSP instance on 197 operations introduced for a competition held at the TU Eindhoven in 1997. The optimal solution for this problem is 469. We ran each of the six versions of our algorithm 10 times for 18000 seconds on each problem instance. The results are shown in Figures 3.2, 3.3 and 3.4. There are several observations to be mentioned. The short Tabu Search run on the iteration best ant improves the algorithm considerably. Version EDI_{TS} finds the optimal solutions for the JSP and the OSP instance by Taillard and produces a solution which is only 2.77% above the optimal solution for the instance whizzkids97. Furthermore, the results on the whizzkids97 instance show that the versions R, ER, ED and EDI clearly improve on the basic version U of our algorithm. Among these four improved versions, version EDI has a clear advantage over the other three versions. These observations are supported by the result of the pairwise Wilcoxon rank sum test. This test produced probabilities between 89% and 98% for version EDI to be different from the other versions. The results concerning EDI are the same – even if not with the same statistical significance – on the two instances by Taillard. The lack of statistical significance might be caused by the fact that these two instances are easier to solve than the whizzkids97 instance. The diversification setting alone (version ED) and the random settings (versions R and ER) do not seem to improve the basic version U on the Taillard instances. Therefore we choose version EDI_{TS} as the ACO algorithm to enter the comparison of metaheuristics to tackle the GSP.

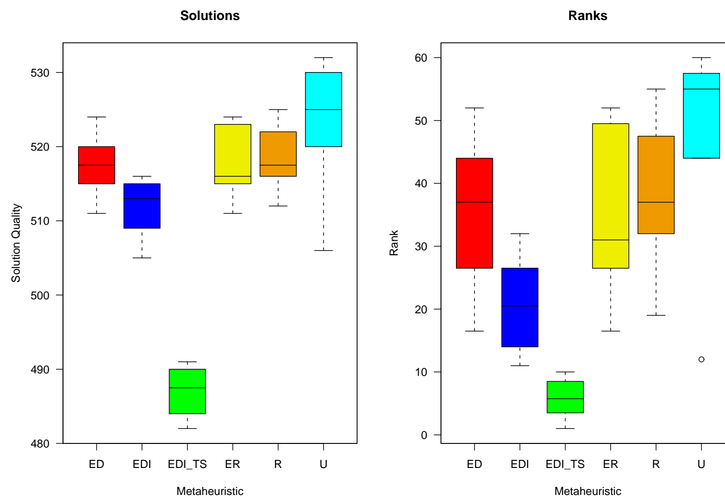

Figure 3.2: Comparison of the ACO versions summarized in Table 3.1 on the whizzkids97 instance. The absolute values of the solutions generated by each ACO version (left) and their relative rank in the comparison among each other (right) are depicted in two boxplots. A box shows the range between the 25% and the 75% quantile of the data. The median of the data is indicated by a bar. The whiskers extend to the most extreme data point which is no more than 1.5 times the interquantile range from the box. Extreme points are indicated as circles.

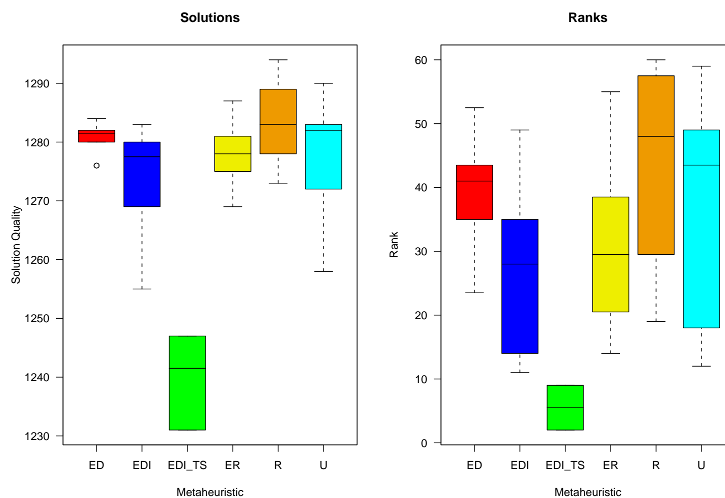

Figure 3.3: Comparison of the ACO versions summarized in Table 3.1 on the tai_15_15_1_jsp instance. The absolute values of the solutions generated by each ACO version (left) and their relative rank in the comparison among each other (right) are depicted in two boxplots. A box shows the range between the 25% and the 75% quantile of the data. The median of the data is indicated by a bar. The whiskers extend to the most extreme data point which is no more than 1.5 times the interquantile range from the box. Extreme points are indicated as circles.

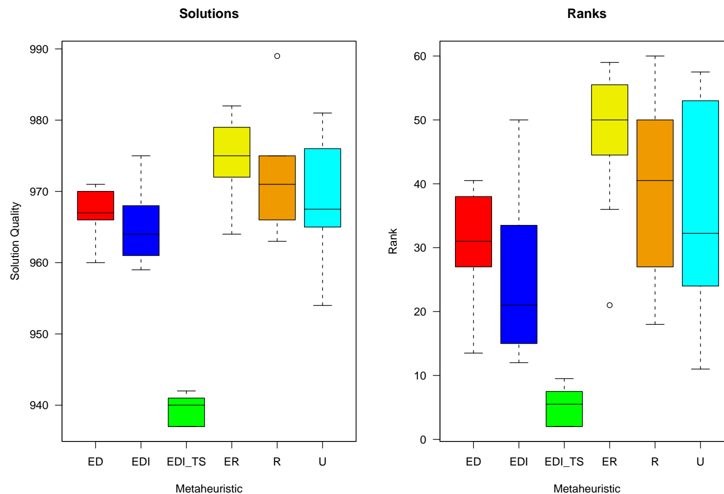

Figure 3.4: Comparison of the ACO versions summarized in Table 3.1 on the tai_15_15_1_osp instance. The absolute values of the solutions generated by each ACO version (left) and their relative rank in the comparison among each other (right) are depicted in two boxplots. A box shows the range between the 25% and the 75% quantile of the data. The median of the data is indicated by a bar. The whiskers extend to the most extreme data point which is no more than 1.5 times the interquantile range from the box. Extreme points are indicated as circles.

3.3 Evolutionary Computation

The EC algorithm[10] developed for the GSP is characterized by a steady-state evolution process. To improve the offspring after crossover, we use a best improvement local search on the neighborhood structure $\mathcal{N}_{1c,GSP}$ defined in Section 3.1. Tournament selection is used to choose which individuals reproduce at each generation and a "replace if better policy" is used to decide whether or not to accept the offspring for the new population. The initial population is built by using the Non-Delay algorithm (see Algorithm 3 in Section 1.4.1). The population size is set to 50. A solution is represented by a sequence (total order on O), which induces a total order on each machine $M \in \mathcal{M}$ and each group $G \in \mathcal{G}$.

The crossover that is applied by this algorithm is a kind of uniform order based crossover respecting group precedence relations. It generates an offspring from two parents as follows:

1. Produce a partial child sequence where each position is either filled with the content of the first parent or left free, with equal probability.

2. Insert the missing operations in the partial list in the order in which they appear in the second parent.

3. If there is a free position between the last operation of the previous group and the first of the next one, put the current operation there.

[10]This algorithm was developed at the Napier University, Edinburgh, UK, by Olivia Doria-Rossi.

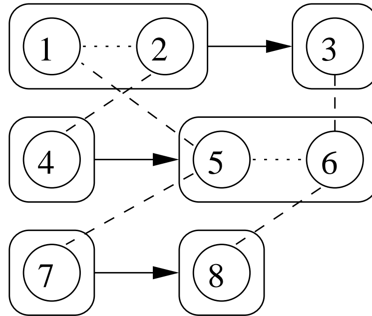

Figure 3.5: An example for a GSP instance on 8 operations: $O = \{1,\dots,8\}$, $\mathcal{J} = \{J_1 = \{1,2,3\}, J_2 = \{4,5,6\}, J_3 = \{7,8\}\}$, $\mathcal{M} = \{M_1 = \{1,5,7\}, M_2 = \{2,4\}, M_3 = \{3,6,8\}\}$, $\mathcal{G} = \{G_1 = \{1,2\}, G_2 = \{3\}, G_3 = \{4\}, G_4 = \{5,6\}, G_5 = \{7\}, G_6 = \{8\}\}$, $G_1 \prec G_2, G_3 \prec G_4, G_5 \prec G_6$, processing times are ommitted.

4. Otherwise, put it just before the first operation of the next group and shift the list to fill the first free position.

In order to demonstrate this mechanism, we consider the GSP instance depicted in Figure 3.5. The two parents below are both feasible schedules for the described instance. A partial child sequence is produced and its free positions are filled as follows:

Parent1	2	1	4	3	5	6	7	8
Parent2	7	4	8	1	5	2	6	3
Partial child	$*$	1	$*$	3	$*$	6	7	$*$
– – – – – – – – – – – – – –	–	–	–	–	–	–	–	–
Child	4	1	5	2	3	6	7	8

The first operation of the second parent that is not yet in the partial sequence, in this case operation 4, is inserted in the first free position; then operation 5 is inserted in the next free position checking that it occurs after 4, which is the last operation of the previous group in job 2; now operation 8 has to be inserted, but since it has to be scheduled after operation 7, it ends up in the next free position after 7; finally operation 2 has to be scheduled before operation 3, and since there is no free position in the list before 2, it is inserted immediately before 2 and the list is shifted forward until the next free position is reached.

As modification (mutation) operator we implemented a variable neighborhood search (VNS) based on the neighborhood structure $\mathcal{N}_{1c,GSP}$ described in Section 3.1 for $\mathcal{N}_k$, $k = 1,\dots,10$, where $\mathcal{N}_k(\preceq^*) = \underbrace{\mathcal{N}_{1c,GSP}(\mathcal{N}_{1c,GSP}(\cdots\mathcal{N}_{1c,GSP}}_{k}(\preceq^*)))$. That means that a random solution in $\mathcal{N}_{1c,GSP}$ is chosen first, then the local search is applied, and if no improvement is found, a random solution in $\mathcal{N}_2$ is chosen followed by local search, then a random solution in $\mathcal{N}_3$ and so on until a better solution is found. The mutation rate is set to be 0.5.

3.4 Iterated Local Search

ILS, in spite of its simplicity, is a powerful metaheuristic that applies a local search algorithm iteratively to modifications of the current solution. A detailed description of ILS algorithms can be found in [64]. ILS roughly works as follows. First an initial locally optimal solution, with respect to the given local search, is constructed. A good starting point can be important, if high-quality solutions are to be reached quickly. Then, more importantly, a perturbation has to be defined, that is a way to modify the current solution to an intermediate state to which the local search can be applied next. Finally, an acceptance criterion is used to decide from which solution to continue the search process.

The implementation described here[11] for the GSP works with a best improvement local search based on neighborhood structure $\mathcal{N}_{1c,GSP}$. The initial solution is generated using the Non-Delay algorithm (see Section 1.4.1). The idea used for the perturbation is to slightly modify the definition of the problem instance data and apply the local search for this modified instance to the current solution regarded as a solution of the new instance; the result is the perturbed solution in the original problem instance. In the GSP the processing times of the operations, unlike group or machine data, can be easily modified such that a solution to one problem instance can be regarded as a solution to the other. For a percentage α of operations the processing time is therefore increased or decreased (with the same probability) by a certain percentage β of its value; then the local search within the modified problem instance is run for the current solution and finally the resulting locally optimal solution to the modified instance, regarded as a solution to the original instance, is the perturbed solution. Note that this solution is not necessarily a local optimum for the original instance. Now the local search can be applied to the intermediate perturbed solution to reach a locally optimal solution.

Finally the acceptance criterion tells us whether to continue the search from the new local optimum or from our previous solution. Random walk, the "accept-only-if-better" strategy, and SA type acceptance criteria have been tested along with different values for α and β. The random walk acceptance criterion with $\alpha = 40$ and $\beta = 40$ has been selected as it gives the best performance.

3.5 Simulated Annealing

SA is a metaheuristic based on the idea of annealing in physics [3]. The algorithm starts out with some initial solution and moves from neighbor to neighbor. If the proposed new solution is equal to or better than the current solution, it is accepted. If the proposed new solution is worse than the current solution, it is even then accepted with some positive probability. For the SA implemention[12] for the GSP the latter probability is

$$P_{accept} = \exp(-\frac{\Delta}{T}) = \exp(-\frac{(C_{\max}(\preceq^{*\prime}) - C_{\max}(\preceq^{*}))/C_{\max}(\preceq^{*})}{T}),$$

where $\preceq^*$ denotes the current solution, $\preceq^{*\prime}$ denotes the the proposed next solution, Δ is the percent cost change, and the temperature T is simply a control parameter. Ideally, when local optimization is trapped in a poor local optimum, Simulated Annealing can "climb" out of the

[11] This algorithm was developed at the Napier University, Edinburgh, UK, by Olivia Doria-Rossi.

[12] This algorithm was developed at IDSIA, Lugano, CH, by Monaldo Mastrolilli.

poor local optimum. In the beginning the value of T is relatively large so that many cost-increasing moves are accepted in addition to cost-decreasing moves. During the optimization process the temperature is decreased gradually so that fewer and fewer cost-increasing moves are accepted.

The selection of the temperature is done as follows. We set the initial temperature such that the probability to accept a move with $\Delta = \delta = 0.01$ is $P_{start} = 0.9$. Moreover, at the end of the optimization process, we would like that the probability to accept a move with $\Delta = \delta = 0.01$ is $P_{end} = 0.1$. With these requirements, we constraint the temperature at time x to be $T = r^x \tau_{\max}$, where $\tau_{\max} = -\delta / \ln P_{start}$, $r = \sqrt[t_{\max}]{\delta / (\ln(1/P_{end}) \cdot \tau_{\max})}$, and where $t_{\max}$ denotes the maximum time allowed for computation.

3.6 Tabu Search

In the TS algorithm implemented for the GSP[13], according to the neighborhood structure $\mathcal{N}_{1c,GSP}$, a move is defined by the exchange of certain adjacent critical operation pairs. We forbid the reversal of the exchange of a critical operation pair by recording the iteration number on which the exchange was performed and requiring that this number plus the current length T be strictly less than the current iteration number.

The tabu status length T is crucial to the success of the TS procedure, and we propose a self-tuning procedure based on empirical evidence. T is dynamically defined for each solution. It is equal to the number c of operations of the current critical path divided by a suitable constant d (we set $d = 5$). We choose this empirical formula since it summarizes, to some extent, the features of the given problem instance and those of the current solution. For instance, there is a certain relationship between c and the instance size, between c and the quality of the current solution. In order to diversify the search it may be unprofitable to repeat the same move often if the number of candidate moves is "large" or the solution quality is low, in some sense, when c is a "large" number.

With the aim of decreasing the probability of generating cycles, we consider a variable neighbor set: every non tabu move is a neighbor with probability 0.8. Moreover, in order to explore the search space in a more efficient way, TS is usually augmented with some aspiration criteria. The latter are used to accept a move even if it has been marked tabu. We consider a tabu move as a neighbor with probability 0.3, and perform it only if it improves the best known solution. To summarize, the proposed TS considers a variable set of neighbors and performs the best move that improves the best known solution (aspiration), otherwise performs the best non tabu move chosen among those belonging to the current variable neighborhood set.

3.7 Comparison

We tested the proposed metaheuristics on the whizzkids97 instance. This is a GSP instance that was used for a mathematics competition in The Netherlands in 1997 [115]. It consists of 197 operations on 15 machines and 20 jobs which are subpartitioned into 124 groups. As this is the only established GSP instance, we derived further GSP instances from JSP instances.

[13] This algorithm was developed at IDSIA, Lugano, CH, by Monaldo Mastrolilli.

The most prominent problem instance for JSP is a problem with 10 machines and 10 jobs which was introduced 1963 by Fisher and Thompson in [75]. It had been open for more than twenty years before the optimality of a solution was proved by Carlier and Pinson [24]. Another famous series of 80 problem instances for JSP and 60 OSP instances was generated by Taillard [101]. We use the Fisher-Thompson instance ft10 and Taillard's first JSP instance tai_15_15_1_jsp on 15 jobs and 15 machines to generate 10 respectively 15 new benchmark instances for the GSP. For both problems, we refined the job partition into a group partition by subdividing each $J_i = o^i_1 \preceq \cdots \preceq o^i_{j_i}$ into b groups of fixed length $g = 1, \ldots, 10$ respectively $g = 1, \ldots, 15$ (and possibly one last group of shorter length):

$$\{o^i_1, \ldots, o^i_g\}, \{o^i_{g+1}, \ldots, o^i_{2g}\}, \ldots, \{o^i_{(b-1)g+1}, \ldots, o^i_{j_i}\} \; (b = \lceil j_i/g \rceil) \ .$$

We tested the five developed metaheuristics: ACO[14], EC, ILS, SA and TS as described in the previous sections. From the whizzkids97 GSP instance we derived three problem instances: (i) The original GSP instance in the following denoted by whizzkids97_gsp, (ii) the OSP version whizzkids97_osp, and (iii) the JSP version whizzkids97_jsp where we fixed the technological sequences as they appear in the file. We tested each metaheuristic for 10 trials of 18000 seconds each (1800 seconds for the OSP version). The OSP version of this problem turned out to be very easy to solve (see Figure 3.6) and all metaheuristics, except for the EC algorithm which is performing worse, show a similar performance. However, ACO is the only algorithm which always finds the best solution found. The original GSP instance (see Figure 3.7) seems to be much harder to solve, especially for the population based methods. TS, although not finding the best solutions found by SA, showed the overall best performance, followed by ILS, SA, ACO and further behind the EC algorithm. The JSP instance (see Figure 3.8) again shows TS in the advantage over the other algorithms. ILS, ACO and SA show a similar performance. SA shows a higher variance in the qualities of the solutions produced. Again the performance of the EC algorithm is worse than the performance of the other metaheuristics.

We further tested our metaheuristics on the 10 GSP instances derived from ft10 for a time limit of 60 seconds for each of the 10 trials per metaheuristic. A summary of these results showing the mean ranks in a mean rank based comparison is given in Figure 3.9. Again, TS showed the best results for most group sizes, except for group size 8 where ACO has a slight advantage. ACO is altogether the second best algorithm. The results in Figure 3.10 show that ACO, EC and TS find the optimal solution 930 to the original JSP version of ft10 within 60 seconds. The third algorithm in the ranking is ILS which performs poorly for group size 1, but then ranks between 2 and 4 for the other group sizes. EC shows a quite particular behaviour. It performs quite well for small and large group sizes[15], but shows decreasing performance for the medium group sizes. SA ranks last for most of the groups sizes. It might be that for SA 60 seconds is not enough for reaching good solutions.

We observed a similar behavior for the 15 instances derived from tai_15_15_1_jsp, which we tested for running times of 600 and 1800 seconds for each of the 10 trials per metaheuristic (see Figures 3.11 and 3.12). It turned out that with increasing running time the difference between TS and the other metaheuristics became smaller, and TS was outperformed by ACO and ILS for some medium group sizes (7, 9 and 10). ACO even outperforms TS slightly on groups sizes 2 and 4 for the longer running times. For longer running times ACO is for nearly

[14]Version EDI_{TS} described in Section 3.2.

[15]For instances close to JSP and close to OSP.

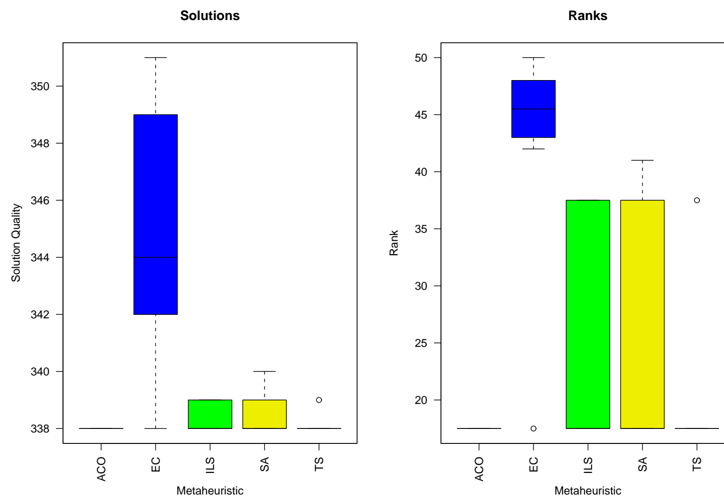

Figure 3.6: Comparison of the metaheuristics on the whizzkids97_osp problem, which is the OSP instance derived from whizzkids97. The absolute values of the solutions generated by each metaheuristic (left) and their relative rank in the comparison among each other (right) are depicted in two boxplots. A box shows the range between the 25% and the 75% quantile of the data. The median of the data is indicated by a bar. The whiskers extend to the most extreme data point which is no more than 1.5 times the interquantile range from the box. Extreme points are indicated as circles.

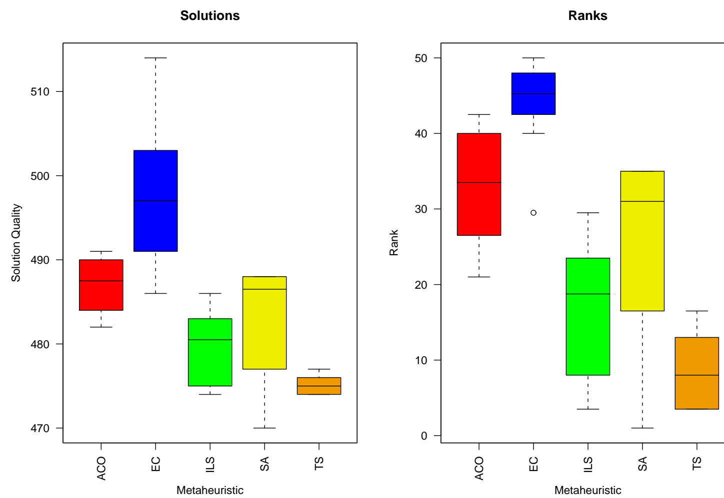

Figure 3.7: Comparison of the metaheuristics for the original whizzkids97_gsp problem. The absolute values of the solutions generated by each metaheuristic (left) and their relative rank in the comparison among each other (right) are depicted in two boxplots. A box shows the range between the 25% and the 75% quantile of the data. The median of the data is indicated by a bar. The whiskers extend to the most extreme data point which is no more than 1.5 times the interquantile range from the box. Extreme points are indicated as circles.

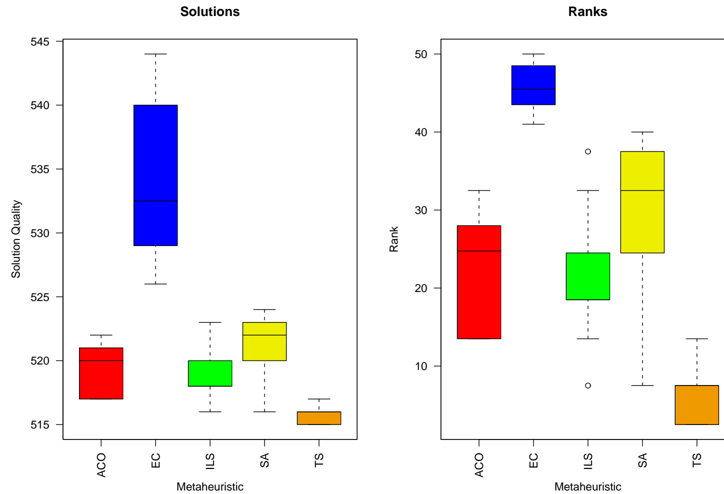

Figure 3.8: Comparison of the metaheuristics on the whizzkids97_jsp problem, which is the JSP instance derived from whizzkids97. The absolute values of the solutions generated by each metaheuristic (left) and their relative rank in the comparison among each other (right) are depicted in two boxplots. A box shows the range between the 25% and the 75% quantile of the data. The median of the data is indicated by a bar. The whiskers extend to the most extreme data point which is no more than 1.5 times the interquantile range from the box. Extreme points are indicated as circles.

all group sizes quite close to the performance of the TS. However, the TS is the only algorithm which finds (even within 600 seconds) the optimal solution 1231 for the original JSP version of tai_15_15_1_jsp (see Figure 3.13). EC again performs rather poorly on medium group sizes and performs well on small group sizes (where the VNS might help to find good solutions) and big group sizes (where the Non-Delay algorithm to produce the initial solution helps to find very good starting solutions).

Summarizing we can say: TS overall shows the best performance, which indicates the power of the neighborhood structure defined in Section 3.1. ACO shows overall the second best performance even outperforming the TS on some group sizes, followed by the ILS whose performance is for a few exceptions consistently slightly worse than the performance of the ACO. The EC approach performs well on small and big group sizes, whereas for medium group sizes the relative performance of the algorithm is rather poor. SA can perform very well when long running times are allowed. For example, SA found the best solution among the metaheuristics for the difficult original whizzkids97 problem when given 18000 seconds of running time. Therefore, we conclude that among the population based metaheuristics the ACO implementation shows a clear advantage over the EC implementation, and among the trajectory methods in general the TS implementation outperforms the ILS implementation and the SA implementation. However, we suggest that there is no "best metaheuristic" with which to tackle the GSP. Our fair comparison showed that depending on the position of the problem instance between JSP and OSP different metaheuristic techniques show advantages. The GSP might best be tackled by a hybrid metaheuristic approach that combines the elements of the algorithms described in this work according to the results of our analysis.

3.8 Outlook to future work

There are several possible directions for future work. Concerning the ACO algorithm, different heuristic information should be tested. An interesting result might be that different heuristic information leads the algorithm to different areas in the search space which would suggest the usefulness of using different heuristic information in different restart phases. Also, other construction mechanisms for ACO could be considered. For example, an insertion based construction algorithm might be computionally more expensive, but might improve the performance.

Concerning problem instances, a generator should be developed which generates GSP instances not only with 0 variance in the group sizes, but rather providing a way of adjusting the desired variance in the group sizes of the problem instance to be generated.

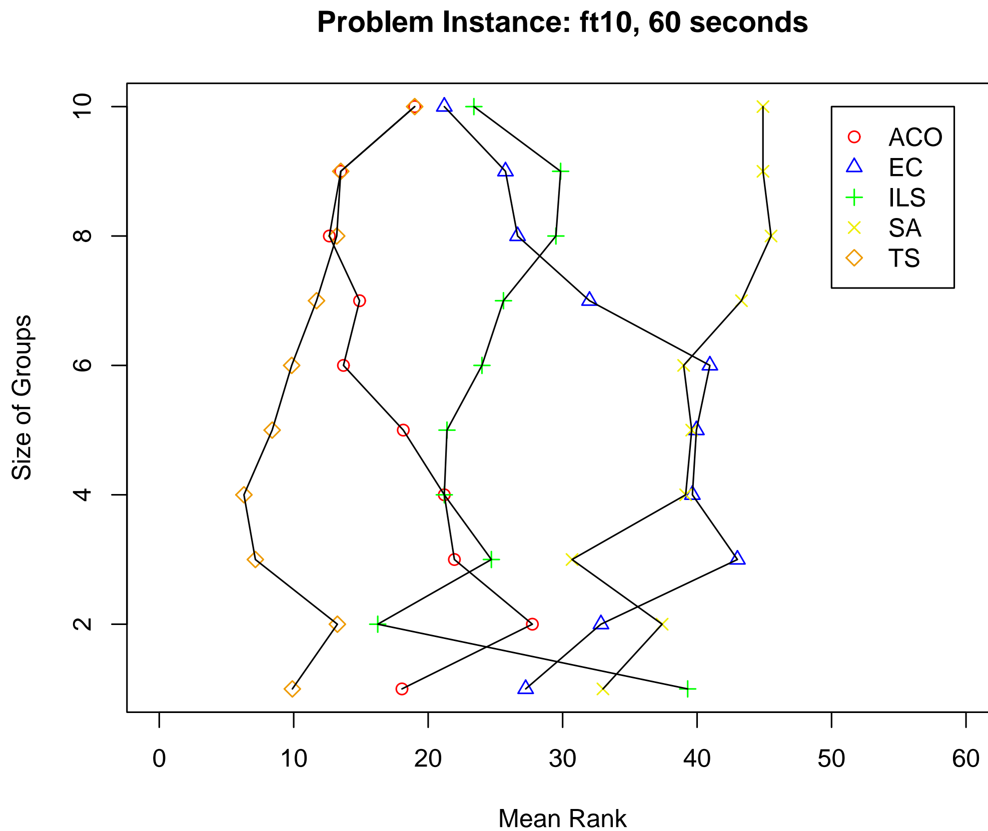

Figure 3.9: Mean ranks (x-axis) of the solutions generated by the metaheuristics to the 10 GSP instances derived from ft10. The group size (y-axis) varies from 1 to 10. The metaheuristics were tested 10 times each on every derived problem, for a time of 60 seconds per run. Note that the best mean rank a metaheuristic can achieve is 5 in case the solutions produced in its 10 trials are all better than the 40 solutions generated by the other 4 metaheuristics. Accordingly the worst mean rank is 45.

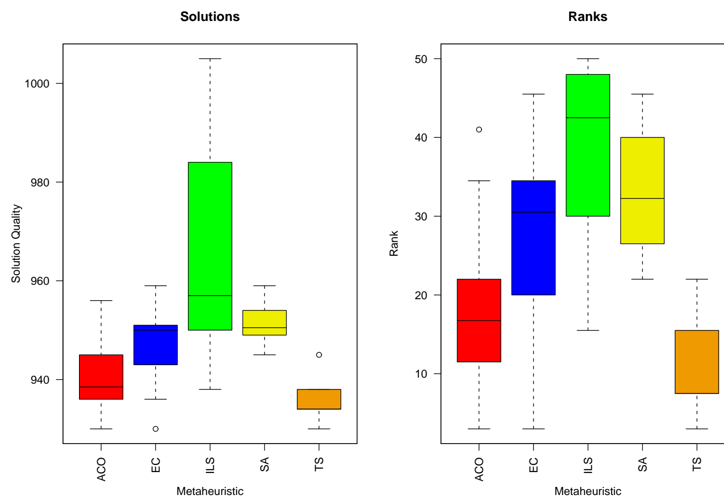

Figure 3.10: Comparison of the metaheuristics on the original ft10 problem, which is a JSP instance. The absolute values of the solutions generated by each metaheuristic (left) and their relative rank in the comparison among each other (right) are depicted in two boxplots. A box shows the range between the 25% and the 75% quantile of the data. The median of the data is indicated by a bar. The whiskers extend to the most extreme data point which is no more than 1.5 times the interquantile range from the box. Extreme points are indicated as circles.

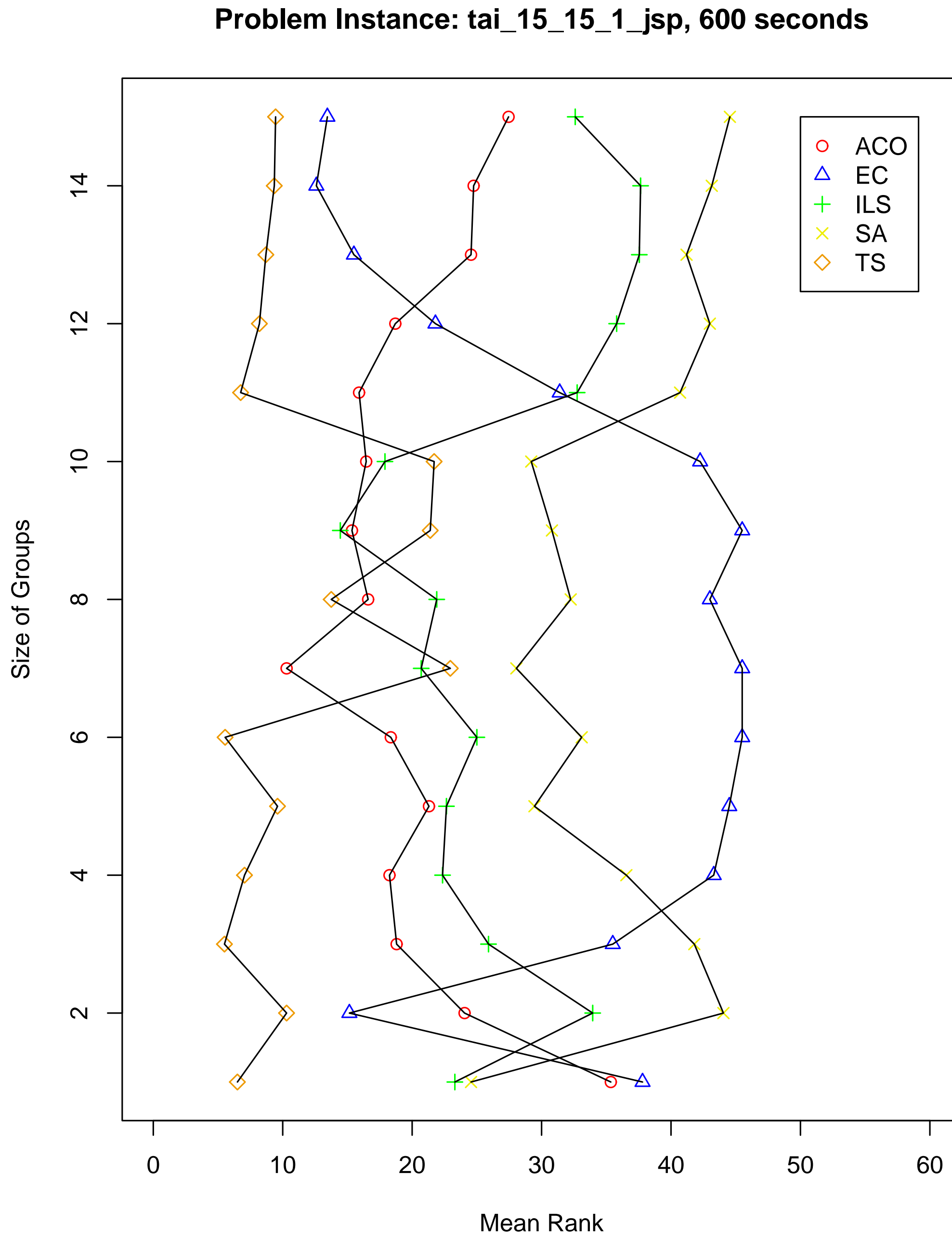

Figure 3.11: Mean ranks (x-axis) of the solutions generated by the metaheuristics to the 15 GSP instances derived from tai_15_15_1_jsp. The group size (y-axis) varies from 1 to 15. The metaheuristics were tested 10 times each on every derived problem, for a time of 600 seconds per run.

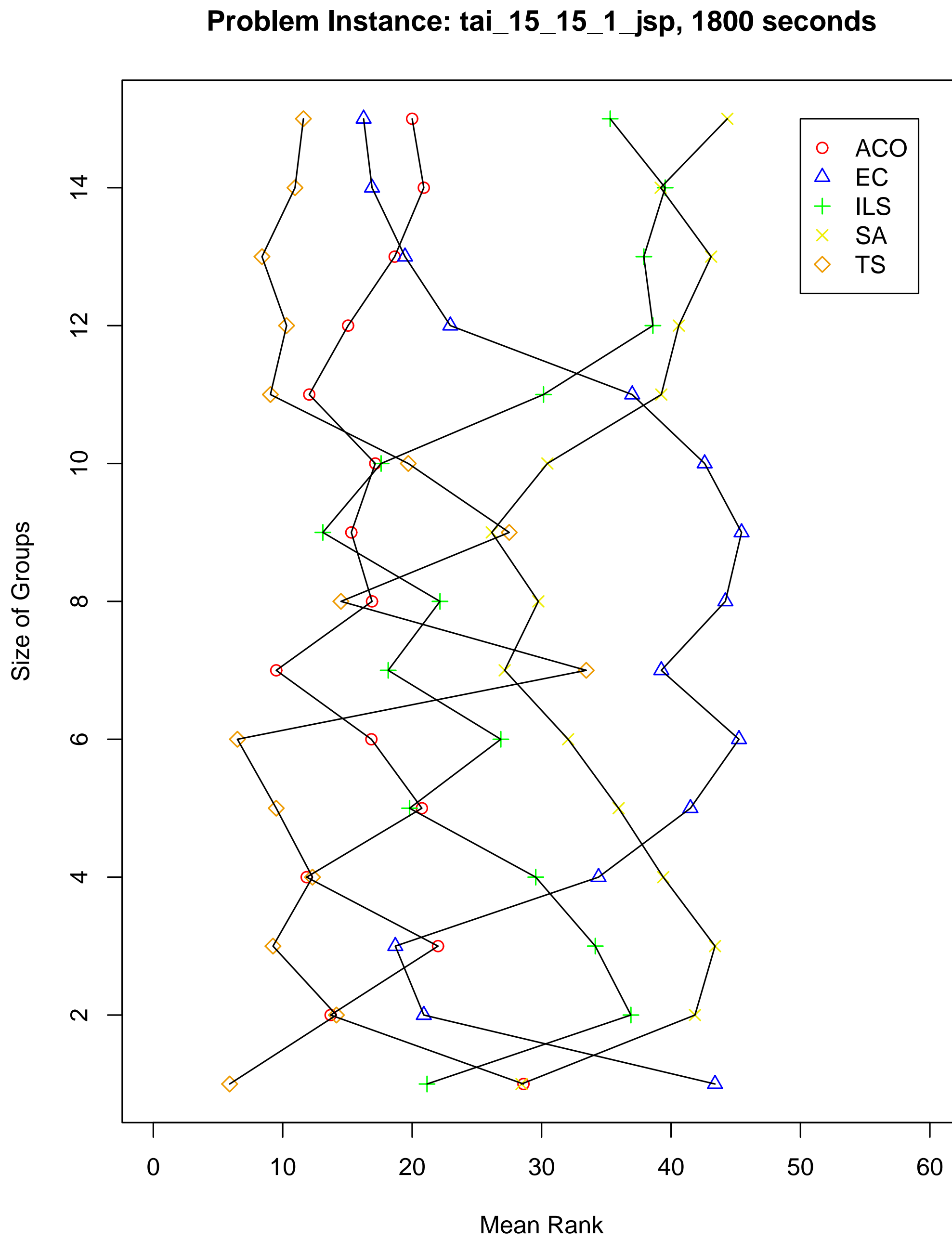

Figure 3.12: Mean ranks (x-axis) of the solutions generated by the metaheuristics to the 15 GSP instances derived from tai_15_15_1_jsp. The group size (y-axis) varies from 1 to 15. The metaheuristics were tested 10 times each on every derived problem, for a time of 1800 seconds per run.

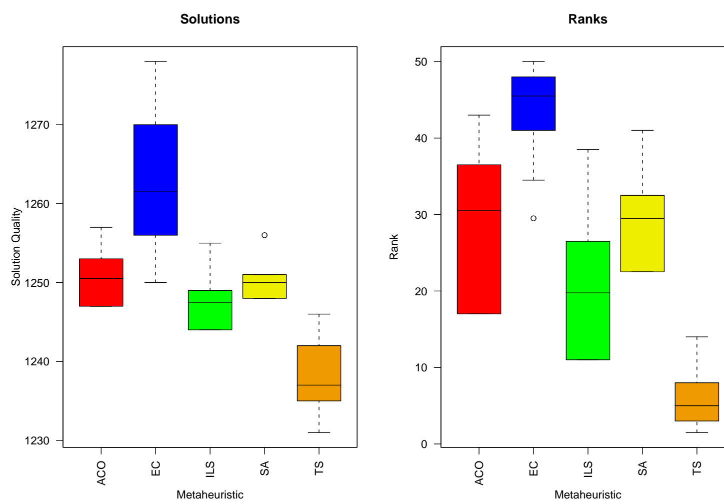

Figure 3.13: Comparison of the metaheuristics on the original tai_15_15_1_jsp problem, which is a JSP instance. The absolute values of the solutions generated by each metaheuristic (left) and their relative rank in the comparison among each other (right) are depicted in two boxplots. A box shows the range between the 25% and the 75% quantile of the data. The median of the data is indicated by a bar. The whiskers extend to the most extreme data point which is no more than 1.5 times the interquantile range from the box. Extreme points are indicated as circles.

Bibliography

[1] E.H.L. Aarts, J.H.M. Korst, and P.J.M. van Laarhoven. Simulated annealing. In Emile H. L. Aarts and Jan Karel Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 91–120. Wiley-Interscience, Chichester, England, 1997.

[2] E.H.L. Aarts, P.J.M. Van Laarhoven, J.K. Lenstra, and N.L.J. Ulder. A Computational Study of Local Search Algorithms for Job Shop Scheduling. *ORSA Journal on Computing*, 6(2):118–125, 1994.

[3] E.H.L. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Wiley-Interscience, 1997.

[4] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for Job Shop Scheduling. *Management Science*, 34(3):391–401, 1988.

[5] A. Albrecht, U. Der, K. Steinhöfel, and C.-K. Wong. Distributed simulated annealing for job shop scheduling. In *Proceedings of the 6th Conference on Parallel Problem Solving in Nature, PPSN'00*, LNCS 1917, pages 243–252, 2000.

[6] D. Alcaide, J. Sicilia, and D. Vigo. A tabu search algorithm for the open shop problem. *TOP: Trabajos de Investigación Operativa*, 5(2):282–296, 1997.

[7] D. Applegate and W. Cook. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing*, 3:149–156, 1991.

[8] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[9] E. Balas and A. Vazacopoulos. Guided Local Search with Shifting Bottleneck for Job Shop Scheduling. *Management Science*, 44(2):262–275, 1998.

[10] J.W. Barnes and J.B. Chambers. Solving the Job Shop Scheduling Problem Using Tabu Search. *IIE Transactions*, 27:257–263, 1995.

[11] J.E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[12] C. Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *European Journal of Operational Research*, 17:87–92, 1995.

[13] C. Bierwirth, D. Mattfeld, and H. Kopfer. On permutation representations for scheduling problems. In *Proceedings of the 4th Conference on Parallel Problem Solving in Nature, PPSN'96*, 1996.

[14] S. Binato, W.J. Hery, D.M. Loewenstern, and M.G.C. Resende. A GRASP for Job Shop Scheduling. Technical Report 00.6.1, AT&T Labs Research, 2000.

[15] J. Blażewicz, W. Domschke, and E. Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93:1–33, 1996.

[16] C. Blum. ACO applied to Group Shop Scheduling: A case study on Intensification and Diversification. Technical Report TR/IRIDIA/2002-08, IRIDIA, Université Libre de Bruxelles, 2002. Submitted to ANTS'2002.

[17] C. Blum, A. Roli, and M. Dorigo. HC–ACO: The hyper-cube framework for Ant Colony Optimization. In *Proceedings of MIC'2001 – Meta–heuristics International Conference*, volume 2, pages 399–403, Porto, Portugal, 2001. Also available as technical report TR/IRIDIA/2001-16, IRIDIA, Université Libre de Bruxelles.

[18] C. Blum and M. Sampels. Ant Colony Optimization for FOP Shop scheduling: A case study on different pheromone representations. In *Proceedings of the 2002 Congress on Evolutionary Computation, CEC'02 (to appear)*, 2002. Also available as technical report TR/IRIDIA/2002-03, IRIDIA, Université Libre de Bruxelles.

[19] C. Blum and M. Sampels. When Model Bias is Stronger than Selection Pressure. Technical Report TR/IRIDIA/2002-06, IRIDIA, Université Libre de Bruxelles, 2002. Submitted to PPSN'02.

[20] H. Bräsel. *Lateinische Rechtecke und Maschinenbelegung.* PhD thesis, TU Magdeburg, 1990.

[21] P. Brucker, J. Hurink, B. Jurisch, and B. Wöstmann. A branch & bound algorithm for the open-shop problem. *Discrete Applied Mathematics*, 76:43–59, 1997.

[22] P. Calegari, G. Coray, A. Hertz, D. Kobler, and P. Kuonen. A taxonomy of evolutionary algorithms in combinatorial optimization. *Journal of Heuristics*, 5:145–158, 1999.

[23] J. Carlier. The one machine sequencing problem. *European Journal of Operational Research*, 11:42–47, 1982.

[24] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, pages 164–176, 1989.

[25] V. Cerny. A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[26] A. Colorni, M. Dorigo, V. Maniezzo, and M. Trubian. Ant system for Job-shop scheduling. *Belgian Journal of Operations Research, Statistics and Computer Science*, 34:39–53, 1994.

[27] M. Dell'Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.

[28] J.-L. Deneubourg, S. Aron, S. Goss, and J.-M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behaviour*, 3:159–168, 1990.

[29] M. Dorigo. *Optimization, Learning and Natural Algorithms* (in Italian). PhD thesis, DEI, Politecnico di Milano,Italy, 1992. pp. 140.

[30] M. Dorigo and G. Di Caro. The Ant Colony Optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, 1999. Also available as Technical Report IRIDIA/99-1, Université Libre de Bruxelles, Belgium.

[31] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2):137–172, 1999.

[32] M. Dorigo and L.M. Gambardella. Ant colony system: A cooperative learning approach to the travelling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[33] M. Dorigo, V. Maniezzo, and A. Colorni. Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics - Part B*, 26(1):29–41, 1996.

[34] U. Dorndorf and E. Pesch. Evolution based learning in a job shop scheduling environment. *Computers in Operations Research*, 22:25–40, 1995.

[35] H.-L. Fang. *Genetic Algorithms in Timetabling and Scheduling.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.

[36] H.-L. Fang, P. Ross, and D. Corne. A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA'93*, pages 375–382. Morgan Kaufmann, 1993.

[37] H.-L. Fang, P. Ross, and D. Corne. A Promising Hybrid GA/heuristic Approach for Open-Shop Scheduling Problems. In *Proceedings of the 11th European Conference on Artificial Intelligence, ECAI'94*, pages 590–594. John Wiley & Sons, Ltd., 1994.

[38] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[39] D.B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1994.

[40] L.J. Fogel. Toward inductive inference automata. In *Proceedings of the International Federation for Information Processing Congress*, pages 395–399, Munich, 1962.

[41] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence through Simulated Evolution.* Wiley, 1966.

[42] B. Giffler and G.L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8:487–503, 1960.

[43] F. Glover. Future paths for integer programming and links to artificial intelligence. *Comp. Oper. Res.*, 13:533–549, 1986.

[44] F. Glover and M. Laguna. *Modern Heuristic Techniques for Combinatorial Problems*, chapter Tabu search, pages 70–141. Blackwell Scientific Publications, Oxford, 1993.

[45] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[46] D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, Reading, MA, 1989.

[47] T. Gonzales and S. Sahni. Open shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery*, 23(4):665–679, 1976.

[48] J. Grabowski and M. Wodecki. A new very fast tabu search algorithm for the job shop problem. Technical Report 21/2001, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2001.

[49] C. Guéret and C. Prins. Classical and new heuristics for the open-shop problem: A computational evaluation. *European Journal of Operational Research*, 107:306–314, 1998.

[50] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. In *Congress on Numerical Methods in Combinatorial Optimization*, Capri, Italy, 1986.

[51] R. Haupt. A survey of priority rule-based scheduling. *OR Spektrum*, 11:3–16, 1989.

[52] A. Hertz and D. Kobler. A framework for the description of evolutionary algorithms. *European Journal of Operational Research*, 126:1–12, 2000.

[53] J.H. Holland. *Adaption in natural and artificial systems*. The University of Michigan Press, Ann Harbor, MI, 1975.

[54] K. Ikeda and S. Kobayashi. GA Based on the UV-Structure Hypothesis and Its Application to JSP. In *Proceedings of the 6th Conference on Parallel Problem Solving in Nature, PPSN'00*, LNCS 1917, pages 273–282, 2000.

[55] L. Ingber. Adaptive simulated annealing (ASA): Lessons learned. *Control and Cybernetics – Special Issue on Simulated Annealing Applied to Combinatorial Optimization*, 25(1):33–54, 1996.

[56] H. Kawamura, M. Yamamoto, K. Suzuki, and A. Ohuchi. Multiple Ant Colonies Algorithm Based on Colony Level Interactions. *IEICE Transactions on Fundamentals*, E83–A(2):371–379, 2000.

[57] S. Kirkpartick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[58] S. Kobayashi, I. Ono, and M. Yamamura. An efficient genetic algorithm for job shop scheduling problems. In *Proceedings of the 4th International Conference on Genetic Algorithms, ICGA'91*, pages 506–511, 1995.

[59] P.J.M. Van Laarhoven, E.H.L. Aarts, and J.K. Lenstra. Job Shop Scheduling by Simmulated Annealing. *Operations Research*, 40:113–125, 1992.

[60] S. Lawrence. Resource Constraint Project Scheduling: an Experimental Investigation of Heuristic Scheduling Techniques (Supplement). Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, 1984.

[61] J.K. Lenstra and R.H.G. Kan Rinnooy. Computional complexity of discrete optimization problems. *Annals of Discrete Mathematics*, 4:121–140, 1979.

[62] C.-F. Liaw. A tabu search algorithm for the open shop scheduling problem. *Computers and Operations Research*, 26:109–126, 1999.

[63] H. Ramalhino Lourenco. Job-shop scheduling: Computational study of local search and large-step optimization methods. *European Journal of Operational Research*, 83:347–364, 1995.

[64] H. Ramalhino Lourenco, O. Martin, and T. Stützle. Iterated Local Search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*. To appear. Also available at http://www.intellektik.informatik.tu-darmstadt.de/˜tom/pub.html.

[65] M. Lundy and A. Mees. Convergence of an annealing algorithm. *Mathematical Programming*, 34(1):111–124, 1986.

[66] S. Martello, W.R. Pulleyblank, P. Toth, and D. de Werra. Balanced optimization problems. *Operations Research Letters*, 3(5):275–278, 1984.

[67] H. Matsuo, C.J. Suh, and R.S. Sullivan. A Controlled Search Simulated Annealing Method for the General Jobshop Scheduling Problem. Technical Report 03-04-88, Graduate School of Business, University of Texas, Austin, 1988.

[68] D.C. Mattfeld, C. Bierwirth, and H. Kopfer. A Search Space Analysis of the Job Shop Scheduling Problem. *Annals of Operations Research*, 86:441–453, 1999.

[69] D. Merkle and M. Middendorf. Ant ant algorithm with a new pheromone evaluation rule for total tardiness problems. In *Proceedings of the EvoWorkshops 2000*, volume 1803 of *Lecture Notes in Computer Science*, pages 287–296. Springer, 2000.

[70] D. Merkle and M. Middendorf. On the behaviour of ACO algorithms: Studies on simple problems. In *Proceedings of MIC'2001 – Meta-heuristics International Conference*, volume 2, pages 573–577, Porto – Portugal, 2001.

[71] Z. Michalewicz and M. Michalewicz. Evolutionary computation techniques and their applications. In *Proceedings of the IEEE International Conference on Intelligent Processing Systems*, pages 14–24, Beijing, China, 1997. Institute of Electrical & Electronics Engineers, Incorporated.

[72] M. Middendorf, F. Reischle, and H. Schmeck. Information Exchange in Multi Colony Ant Algorithms. In *Proceedings of the Workshop on Bio-Inspired Solutions to Parallel Processing Problems*, LNCS 1800, pages 645–652. Springer Verlag, 2000.

[73] M. Mitchell. *An introduction to genetic algorithms.* MIT press, Cambridge, MA, 1998.

[74] H. Mühlenbein and H.-M. Voigt. Gene Pool Recombination in Genetic Algorithms. In I.H. Osman and J.P. Kelly, editors, *Proc. of the Metaheuristics Conference*, Norwell, USA, 1995. Kluwer Academic Publishers.

[75] J.F. Muth and G.L. Thompson. *Industrial Scheduling.* Prentice Hall, Englewood Cliffs, NJ, USA, 1963.

[76] R. Nakano and T. Yamada. Conventional genetic algorithm for job shop problems. In *Proceedings of the 4th International Conference on Genetic Algorithms, ICGA'91*, pages 474–479, 1991.

[77] M. Nawatz, E.E. Enscore, and I. Ham. A heuristic algorithm for the m-machine, n-job flow shop sequencing problem. *OMEGA*, 11:91–95, 1983.

[78] G.L. Nemhauser and A.L. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, New York, 1988.

[79] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job-shop problem. *Management Science*, 42(2):797–813, 1996.

[80] I. Ono and S. Kobayashi. An Evolutionary Algorithm for Job-Shop Scheduling Problems Using the Inter-Machine Job-Based Order Crossover. *Journal of Japanese Society for Artificial Intelligence*, 13(5):780–790, 1998.

[81] I. Ono, M. Yamamura, and S. Kobayashi. A Genetic Algorithm for Job-shop Scheduling Problems Using Job-based Order Crossover. In *Proceedings of the IEEE International Conference on Evolutionary Computation, CEC'96*, pages 547–552, 1996.

[82] I.H. Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41:421–451, 1993.

[83] I.H. Osman and G. Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63:513–623, 1996.

[84] P.S. Ow and T.E. Morton. The single machine early/tardy problem. *Management Science*, 35:177–191, 1989.

[85] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization - Algorithms and Complexity.* Dover Publications, Inc., New York, 1982.

[86] F. Pezzella and E. Merelli. A tabu search method guided by shifting bottleneck for the job-shop scheduling problem. *European Journal of Operational Research*, 120:297–310, 2000.

[87] L.S. Pitsoulis and M.G.C. Resende. Greedy Randomized Adaptive Search procedures. Technical report, AT&T Labs Research, 2001.

[88] C.N. Potts. Analysis of a heuristic for one machine sequencing with release dates and delivery times. *Operations Research*, 28:1436–1441, 1980.

[89] C. Prins. Competitive genetic algorithms for the open shop scheduling problem. Technical Report 99/1/AUTO, École des Mines de Nantes, 1999.

[90] A. Ramudhin and P. Marier. The Generalized Shifting Bottleneck Procedure. *European Journal of Operational Research*, 93:34–48, 1996.

[91] M. Randall and E. Tonkes. Intensification and Diversification Strategies in Ant Colony System. Technical Report TR00-02, School of Information Technolony, Bond University, 2000.

[92] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog, 1973.

[93] B. Roy and B. Sussmann. Les problémes d'ordonnancement avec constraints dijonctives. Technical Report Note DS 9 bis, SEMA, Paris, France, 1964.

[94] J. Sakuma and S. Kobayashi. Extrapolation-Directed Crossover for Job-shop Scheduling Problems: Complementary Combination with JOX. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'00*, pages 973–980. Morgan Kaufmann, 2000.

[95] M. Sampels, C. Blum, M. Mastrolilli, and O. Rossi-Doria. Metaheuristics for Group Shop Scheduling. Technical Report TR/IRIDIA/2002-07, IRIDIA, Université Libre de Bruxelles, 2002. Submitted to PPSN'02.

[96] W.M. Spears, K.A. De Jong, T. Bäck, D.B. Fogel, and H. de Garis. An overview of evolutionary computation. In Pavel B. Brazdil, editor, *Proceedings of the European Conference on Machine Learning (ECML-93)*, volume 667, pages 442–459, Vienna, Austria, 1993. Springer Verlag.

[97] K. Steinhöfel, A. Albrecht, and C.K. Wong. Two Simulated Annealing-Based Heuristics for the Job Shop Scheduling Problem. *European Journal of Operational Research*, 118(3):524–548, 1999.

[98] R.H. Storer, S.D. Wu, and R. Vaccari. New search spaces for sequencing instances with application to job shop scheduling. *Management Science*, 38:1495–1509, 1992.

[99] T. Stützle. *Local Search Algorithms for Combinatorial Problems - Analysis, Algorithms and New Applications.* DISKI - Dissertationen zur Künstliken Intelligenz. infix, 1999.

[100] T. Stützle and H. H. Hoos. $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[101] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[102] E. Taillard. Parallel Taboo Search Techniques for the Job Shop Scheduling Problem. *ORSA Journal on Computing*, 6(2):108–117, 1994.

[103] E.-G. Talbi, O. Roux, C. Fonlupt, and D. Robillard. Parallel Ant Colonies for the quadratic assignment problem. *Future Generation Computer Systems*, 17:441–449, 2001.

[104] T. Teich, M. Fischer, A. Vogel, and J. Fischer. A new Ant Colony Algorithm for the Job Shop Scheduling Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'01*, page 803, 2001.

[105] S. van der Zwaan and C. Marques. Ant Colony Optimization for Job Shop Scheduling. In *Proceedings of the 3rd Workshop on Genetic Algorithms and Artificial Life, GAAL'99*, 1999.

[106] S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors. *Meta-Heuristics - Advances and Trends in Local Search Paradigms for Optimization.* Kluwer Academic Publishers, 1999.

[107] C. Voudouris and E. Tsang. Guided Local Search. Technical Report CSM–247, Department of Computer Science, University of Essex, 1995.

[108] F. Werner and A. Winkler. Insertion techniques for the heuristic solution of the job shop problem. *Discrete Applied Mathematics*, 58:191–211, 1995.

[109] T. Yamada and R. Nakano. A genetic algorithm applicable to large-scale job shop problems. In *Proceedings of the 2nd Conference on Parallel Problem Solving in Nature, PPSN'92*, pages 281–290, 1992.

[110] T. Yamada and R. Nakano. Job-Shop Scheduling by Simulated Annealing Combined with Deterministic Local Search. In *Meta-heuristics: theory & applications*, pages 237–248. Kluwer Academic Publishers, MA, USA, 1996.

[111] T. Yamada and R. Nakano. Scheduling by Generic Local Search with Multi-Step Crossover. In *Proceedings of the 4th Conference on Parallel Problem Solving in Nature, PPSN'96*, pages 960–969, 1996.

[112] T. Yamada and R. Nakano. *Genetic algorithms in engineering systems*, chapter Job-shop scheduling, pages 134–160. IEE Control Engineering 55. The Institution of Electrical Engineers, 1997.

[113] T. Yamada, B.E. Rosen, and R. Nakano. A Simulated Annealing Apprach to Job Shop Scheduling using Critical Block Transition Operators. In *Proceedings of the IEEE International Conference on Neural Networks, ICNN'94*, 1994.

[114] http://www.metaheuristics.net/, 2000.

[115] http://www.win.tue.nl/whizzkids/1997 , 1997.