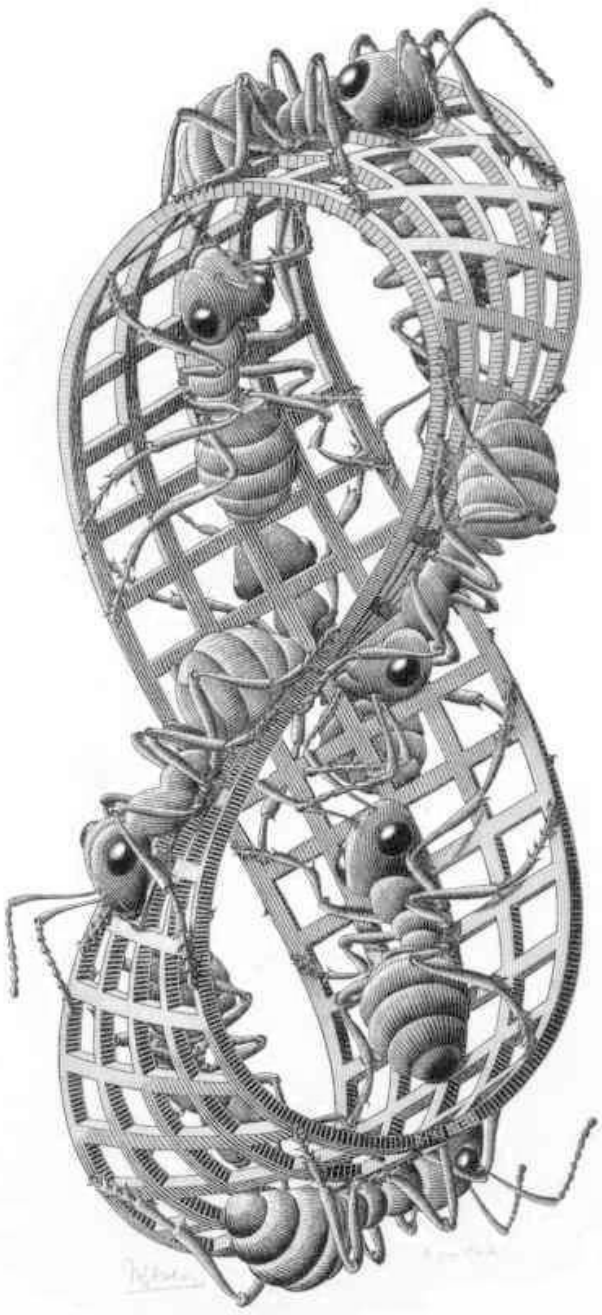
Matthijs den Besten



Ants for the single machine total weighted tardiness scheduling problem

Master's thesis in artificial intelligence, specialization autonomous systems, Faculty of Science, University of Amsterdam; under supervision of Marco Dorigo and Thomas Stützle at IRIDIA, Université Libre de Bruxelles, and Ben Kröse at the University of Amsterdam.

Master's thesis

Ants for the single machine total weighted tardiness scheduling problem

Matthijs den Besten

17th April 2000

Keywords: meta-heuristic, scheduling, ant colony optimization, local search

Cover art is *Band van Möbius II (Rode mieren)* by M. C. ESCHER (February 1963)

Preface

Go to the ant, you sluggard, watch her ways and get wisdom.

—Proverbs 6:6

Goal

This master's thesis concludes my studies in artificial intelligence (AI) at the University of Amsterdam. It reports on the work I have carried out during my stay at the Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle (IRIDIA) in Brussels from September 1999 to January 2000. The aim was to investigate whether the single machine total weighted tardiness scheduling problem (SMWTP) can be solved efficiently by algorithms based on the ant colony optimization (ACO) meta-heuristic. This research tries (i) to extend the range of problems which have been successfully attacked by algorithms based on the ACO meta-heuristic, (ii) to gain further insight into application issues of the ACO meta-heuristic, and (iii) to deepen the understanding of the nature of the problem under consideration.

Motivation

The SMWTP is a classical example of machine scheduling, which is a crucial form of decision–making in manufacturing as well as in service industries. For example, when a company has to meet the shipping date on which it has committed its products to the customers and the production time depends to a great extend on one resource, as is often the case, it is faced with the SMWTP. The SMWTP is the problem of how jobs' due dates can be met such that the cost of jobs being late, as measured by the weighted tardiness, is minimized. The ability to cope efficiently with this kind of problems will boost the company's competitiveness. This thesis helps to achieve that aim.

When it comes to solving $\mathcal{NP}$–hard combinatorial optimization problems like the SMWTP, algorithms based on meta-heuristics, general schemes to improve local search algorithms, often obtain impressive results. Ant colony optimization constitutes a recently defined and particularly successful meta-heuristic. Ant colony optimization has shown to be a versatile meta-heuristic that can easily be applied to a wide range of problems (see Section 1.3.4). Yet, like many other meta-heuristics, the ACO meta-heuristic is not a prefab recipe for successful application to a problem, many design issues still have to be decided when applying it to a particular problem. I can take advantage of the work

of Bauer (1998) who has applied the ACO meta-heuristic to the single machine total tardiness problem (SMTTP) and actually did a very straightforward application of ant colony system which is developed by Dorigo and Gambardella (1997b). The SMWTP is a generalization of the SMTTP that associates weights to the jobs' tardiness. Since the problems are related, algorithms that have success solving them can also be expected to have a lot in common. So, with relatively small modifications of Bauer's (1998) algorithm a successful algorithm for the SMWTP can be obtained, which is what I will pursue.

Work on meta-heuristics and local search receives increased attention in the AI community. Besides, the tasks AI researchers are concerned with are shifting from toy problems like the n-queens problem[1] to real world problems like route finding in dynamic networks or scheduling problems like the SMWTP.

Yet another development within AI is the emergence of agent-based computing as an overarching framework for bringing together the component sub-disciplines of AI that are necessary to design and build intelligent entities. Russell and Norvig (1995) go as far as to define AI as the study and construction of rational agents. Of all meta-heuristics, ant colony optimization fits in most perfectly with this paradigm. Agent-based computing is a software engineering methodology for the design of complex systems. Autonomous entities, called agents, form the building blocks of a multi-agent system. In agent-based terminology an ACO algorithm, which models the foraging behavior of ants, is a multi-agent system where the artificial ants act as autonomous agents. Much of the research in agent-based computing is spend on figuring out which communication schemes the agents should use. As such, the experience with the communication scheme used in ant colony optimization that is reported in this thesis can be of interest to the agent-based computing community as well.

Not only is ACO an agent-based AI algorithm, one could even say that it fits in with the ideas of Brooks (1990) which have gained popularity in robotics. Brooks states that what makes robots intelligent is not their internal model of the world, but their ability to interact with the environment they are placed in, their embodiment. Likewise, ACO's artificial ants are highly reactive, with knowledge of the world distributed over the graph environment with which the ants interact.

One thing that has remained unchanged ever since McCulloch and Pitts (1943) did the first work on artificial intelligence is the tension between approaches that try to achieve artificial intelligence through modeling of intelligent behavior found in real life on the one hand and approaches that employ some sort of clever engineering as means to this end on the other hand. By contrasting the nature inspired ant colony optimization with the more ad hoc iterated local search meta-heuristic, this thesis contributes to that debate. Some people are adherents of the hypothesis that there exists a connection between combinatorial optimization problems and the problems living beings face in nature. For example, the application of the Hopfield network to the traveling salesman problem (Hopfield and Tank 1985) was motivated by this view. One step further,

[1] The goal of the n-queens problem is to place n queens on a chess-board such that no queen attacks any other.

intuitively, things that work well in nature can also be expected to yield good performance on combinatorial optimization problems, as eons of evolution shaped the methods and planed them to perfection. So, the mechanisms which enable the ants to find the shortest path between their nest and a food source should also allow algorithms to solve combinatorial optimization problems efficiently, even though, as will become clear in this thesis, the artificial ants bear only little resemblance to their natural counterparts[2].

Overview

In order to determine whether ant colony optimization forms a suitable approach to the single machine total weighted tardiness scheduling problem, I will first analyze and improve Bauer's (1998) ACO algorithm of the SMTTP and then extend the improved algorithm to the more complex and interesting SMWTP. Subsequently, I will compare the algorithm against an algorithm which is based on a simple but for many problems effective meta-heuristic known as iterated local search. Finally, I will analyze the results and try to identify the relationship between problem characteristics and the algorithms' performance.

This Master's thesis is split into five chapters.

Chapter 1 (p. 1) provides background knowledge on single machine scheduling and meta-heuristics. It indicates when scheduling can be of practical interest, explains the SMWTP and the SMTTP in relation to other scheduling problems and it discusses standard strategies to solve scheduling problems. Next, local search, ant colony optimization and iterated local search are introduced. The generic algorithmic schemes are given along with the ideas that underlie the algorithms and pointers to related applications are provided. Finally, I will explain how the algorithms' performance will be assessed.

Chapter 2 (p. 17) assesses ants' intelligence solving the single machine total tardiness problem. After an overview of the SMTTP and research devoted to this problem, it introduces a first ACO approach of Bauer, Bullnheimer, Hartl and Strauss (1999a) discusses its configuration, and proposes improvements backed by computational results on a series SMTTP instances. The algorithm's performance is compared with that of an iterated local search algorithm and with the performance of the branch–and–bound algorithm of Swarc, Della Croce and Grosso (1998) and conclusions are drawn with respect to ACO's suitability for the SMTTP.

Chapter 3 (p. 33) shows how to tune the ant colony optimization and iterated local search in order to obtain a very well performing algorithm for the SMWTP and compares the performance of the two. Tuning involves the adoption of algorithm design choices and the algorithm's parameters to achieve the best possible performance on the problem under consideration, so that is what will be done. A crucial

[2]The same holds for other meta-heuristics. In fact natural paradigms are rather a source of new inspiration of developing algorithmic solutions to computationally hard problems which has to be adapted to efficiently solve these problems. In the process the resemblance will tend to faint.

component of both meta-heuristics is local search. I have implemented several variants of of local search algorithms and assess their effectiveness. Additionally, I investigate the influence of algorithm parameters like the size of the ant colony on the final performance. After the tuning the meta-heuristics' performance is contrasted against each other.

Chapter 4 (p. 71) analyzes the results given in Chapter 3. The aim of this analysis is to identify the factors that cause the hardness of a SMWTP instance. First, the correlation between certain aspects of the distribution of the due dates, the structure of an optimal solution, and the run time of the algorithm will be sought after. Next, the search space of individual instances is investigated. The search space is viewed as a fitness landscape and the landscape ruggedness and the distribution of local optima on the landscape are determined. Finally, I will try to relate problem characteristics to landscape topology.

Chapter 5 (p. 87) summarizes the results and gives a summary of the contributions of this thesis. It also critically reflects upon the achievements reached in this thesis and discusses directions to extend this work. The chapter ends with some concluding remarks.

Supplementary results are given in detail in the appendices, and an index and a list of acronyms should further help the reader to navigate through the report.

Acknowledgements

I am very grateful for the support given by numerous people during my work on this thesis. First of all, I would like to thank Marco Dorigo for granting me to do research at IRIDIA, Thomas Stützle for the daily discussions during my stay in Brussels and for the invaluable comments on early versions of this report, and all other people at the institute for their hospitality and kindness.

Next, I would like to thank Andreas Bauer for sending his master's thesis and answering questions on his approach, H. A. J. Crauwels for providing the benchmark instances for the SMTTP, Federico Della Croce and Bahar Yetiş Kara for making their algorithms' implementations available.

In addition, I would like to thank Ben Kröse who has been my supervisor at the University of Amsterdam, and Remko Scha who has been willing to form the graduation committee together with Ben Kröse and Thomas Stützle.

Finally, I would like to thank Herman Halfmouw, Nikos Massios, Hedderik van Rijn and Murat Eken for their valuable comments on this thesis.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

The concern of this thesis is to find efficient algorithms to solve $\mathcal{NP}$–hard scheduling problems. This chapter introduces the main concepts used in this thesis, starting with the problem to be solved, continuing with the methods employed to solve the problem and, finally, clarifying how these methods' performance will be compared.

1.1 Scheduling

The total weighted tardiness problem attacked in Chapter 3 and its non–weighted variant considered in Chapter 2 are scheduling problems. Before giving details on these two problems, I will explain why scheduling is of practical interest and how scheduling problems relate to each other. Standard methods to solve scheduling problems are discussed in the last part of this section.

1.1.1 Role of scheduling

A schedule is a detailed and timed plan that specifies how limited resources should be allocated to tasks in order to meet one or more objectives. Scheduling is the art of constructing such plans. The usable resources and tasks may take many forms. The resources may be machines in a factory, runways at an airport, or processing units in a computing environment, to name just a few. The tasks to be performed in these examples could be operations in a production process, take–offs and landings at an airport, and executions of computer programs, respectively. Each task may be characterized by the time it needs the resource, the earliest possible starting time, the date it is due and its priority relative to other tasks. The objectives to be met may also take many forms. They are typically related to the time needed to accomplish the tasks.

Scheduling is a decision–making process that can be found in most manufacturing and production systems as well as in most information–processing environments. Hence, finding out how to solve scheduling problems is of considerable practical interest.

1.1.2 Scheduling models

A large number of scheduling problems and models have been studied and analyzed in the literature. A neat introduction into the field is given by Pinedo (1995) who considers the most important and interesting problems and models. These models assume that a

finite number of jobs have to be scheduled on a finite number of machines[1].

Scheduling models can be classified as either deterministic or stochastic. *Deterministic* models suppose that the job data are exactly known in advance. In *stochastic* models, on the other hand, only the distribution of the data is known. The actual processing times, release dates and due dates are known only after the completion of the processing or after the actual occurrence of the release or due date.

This thesis is concerned with deterministic models. The large amount of research put into these models during the last four decades shows that they are already complex enough to be of strong interest. Besides, the static nature of deterministic models is not as restrictive as it may look on first sight, since many dynamic problems can actually be solved as a series of static[2] problems. Such a decomposition is only useful if the static sub-problems can be solved within a short period of time. One may have serious time constraints on the solution time of the static problems and therefore one needs fast algorithms to solve static problems. The acquisition of efficient algorithms needed in these situations is what this thesis' work hopes to achieve.

Deterministic models can be put in hierarchical order. In the commonly used representation scheme of Graham, Lawler, Lenstra and Rinnooy Kan (1979) each problem is specified by a three–field descriptor $\alpha|\beta|\gamma$ where α represents the machine environment, β defines the job characteristics and γ is the optimality criterion.

Machine environments

Different configurations of machines are possible. The problems treated in this thesis are all placed in an environment constituted by a single machine. This machine is available to process jobs at time zero and is continuously available as a resource. The three–field representation for problems concerning this configuration is $1|\beta|\gamma$. Other configurations of machines are left out of consideration for a number of reasons. Firstly, the single machine environment is very simple and forms a special case of other environments. Secondly, the results that can be obtained for single machine models provide a basis for heuristics for more complex machine environments. Thirdly, in many real life situations it is one machine that causes the bottleneck of the whole production environment. Production planning should then be oriented towards this single machine and optimize the schedule for that machine.

Job characteristics

The following pieces of data are associated with job j:

Processing time (p_{ij}): p_{ij} represents the processing time of job j on machine i. The subscript i will be omitted in the following since the problems in this thesis involve only one machine.

[1] The terms *job* and *machine* are common usage, they are substitutes for *task* and *resource* respectively.
[2] Problems whose topology and costs do not change while they are being solved are called *static*.

Release date (r_j): The release date r_j of job j is the time the job arrives at the system, that is, the earliest time at which job j can start its processing.

Due date (d_j): The due date d_j of job j represents the committed shipping or completion date, the date the job is promised to the customer. The completion of a job after its due date is allowed, but a penalty is incurred. When the due date must absolutely be met, it is referred to as a deadline.

Weight (w_j): The weight w_j of job j is basically a priority factor. It denotes the importance of job j relative to the other jobs in the system. For example, a weight may represent the actual cost of keeping the job in the system.

The problems dealt with in this thesis assume that all jobs are immediately available for processing, that setup times are sequence independent and added to the processing times, and that jobs are not subject to precedence constraints. Furthermore, the processing of a job cannot be interupted and the job has to be kept on the machine until completion, *preemption* is not allowed. As they only concern jobs characterized by the standard data, these problems are specified by an empty field in the tree–field representation: $1|\circ|\gamma$.

Objectives

The objective to be minimized is always a function of the completion times of the jobs. The *completion time* of job j depends on the schedule and is denoted by C_j. In addition, many objectives take due dates into account. The *lateness* of job j is defined as $L_j = C_j - d_j$, which is positive when job j is completed late and negative when it is completed early. The *tardiness* of job j is defined as $T_j = \max\{C_j - d_j, 0\} = \max\{L_j, 0\}$. The difference between tardiness and lateness lies in the fact that tardiness is never negative. Complementary to tardiness one can encounter *earliness*, $E_j = \max\{d_j - C_j, 0\}$. A possible objective function is the total weighted tardiness, $TW = \sum w_j T_j$. The minimization of this function for a sequence of jobs in a single machine environment is the goal of the single machine total weighted tardiness scheduling problem (SMWTP). The three–field notation for this problem type is $1||\sum w_j T_j$. It is investigated in depth in Chapters 3 and 4. Chapter 2 examines a variant of the SMWTP, that does not take the weights into account[3]. The problem thus obtained, the single machine total tardiness problem, is formally specified as $1||\sum T_j$.

An example

Consider an SMWTP instance of 5 jobs with the processing times, due dates and weights that are given in Table 1.1 on the following page. For the permutation of jobs given in Figure 1.1 on the next page, the completion time, the tardiness and the weighted tardiness of the jobs are given in Table 1.2 on the following page. The total tardiness of this permutation is $45 + 40 = 85$, the weighted total tardiness is $135 + 400 = 535$.

[3]Or where all weights have the same, non–zero, value.

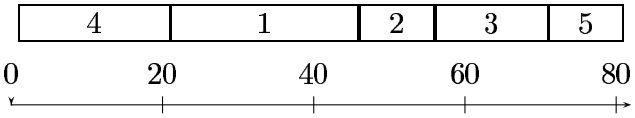


Figure 1.1: Gantt chart of a schedule

job	1	2	3	4	5
p_j	25	10	15	20	10
d_j	50	10	30	30	90
w_j	7	3	10	2	5

Table 1.1: A SMWTP instance

job	4	1	2	3	5
C_j	20	45	55	70	80
T_j	0	0	45	40	0
w_jT_j	0	0	135	400	0

Table 1.2: Evaluation of objectives for the SMWTP instances

Permutation $\pi = (2, 3, 1, 4, 5)$ is the optimal solution of this SMWTP instance with a weigthed total tardiness of 80.

1.1.3 Problem solving strategies

The task in scheduling problems is to find a permutation of jobs that meets the problem's objective best. Many scheduling problems, including those considered in this thesis, are known to be $\mathcal{NP}$–hard (Lawler, Lenstra, Rinnooy Kan and Shmoys 1993). That is, the time the best possible algorithm will need to solve the problem increases in the worst case exponentially with the size of the problem[4]. Despite being $\mathcal{NP}$–hard, these problems have to be solved efficiently in practice and a large number of different solution strategies have been proposed to achieve that goal.

Exact methods

A straightforward strategy is to enumerate all possible solutions and then pick the best one. Yet, this may take considerable time as there are $n!$ different sequences for a single machine problem of n jobs. Fortunately there exist more complex methods like branch–and–bound algorithms that allow to discard parts of the search space in which the optimal solution cannot be found. *Branch–and–bound*[5]. (Lawler and Wood 1966) is a backtracking–type algorithm that searches through the space of partial solutions. Search on a given branch is aborted and backtracking is done when the cost of the current partial solution plus a lower bound on the cost of the completion of the solution exceeds the best solution found to that point.

Construction heuristics

Often solutions are needed very fast, for example when the problem is an element of a dynamic real world setting (see page 2). This requirement can generally not be met

[4] See Garey and Johnson (1979) for an introduction into complexity theory.

[5] Branch–and–bound is closely related to A* and other state–space search algorithms known in AI (Kumal, Nau and Kanal 1988).

by exact algorithms, especially when the problem is $\mathcal{NP}$–hard. Besides, not everyone is interested in the optimal solution. In many cases it is preferable to find a sub-optimal, but good, solution in a short time. Constructive algorithms generate solutions from scratch by adding solution components to an intially empty solution until it is complete. A common approach is to generate a solution in a greedy manner, where a *dispatching rule* decides heuristically which job should be added next to the sequence of jobs that makes up the partial solution. A survey of this type of priority rule based scheduling is given by Haupt (1989).

A distinction can be made between static and dynamic dispatching rules. *Static* rules are just a function of the a priori known job–data. An example of a static rule is earliest due date (EDD). This rule assigns the highest priority to the job with the earliest due date. *Dynamic* dispatching rules, on the other hand, depend on the partial solution constructed so far. An example of a dynamic rule is modified due date (MDD). The value MDD uses to determine the relative priority of the jobs is the job's completion time for a late job and the job's due date for a job that would finish early when processed next. The highest priority is assigned to the job with the lowest MDD–value.

A possibility to get still better performing dispatching policies is to combine simple rules like EDD or MDD. An example of a composite dispatching rule is the apparent urgency (AU) rule which is used for the SMWTP–problem (see Section 3.2.2).

Local search

When better solutions are required than those obtained by the application of dispatching rules and an exact algorithm takes too much time, then local search is often a good choice. Local search starts on some solution and tries to alter this solution slightly in order to obtain a better solution. Standard descent local search only considers improving modifications. The problem of this hill–climbing–like methodology is that it will easily get stuck in a local optimum before reaching the global best solution. Section 1.2 offers a more detailed introduction into local search.

Meta-heuristics

Apart from the standard descent local search, several general search schemes have been developed. These schemes are called meta-heuristics (Osman and Kelly 1996). A *meta–heuristic* is an iterative generation process which guides a subordinate heuristic by intelligent combination of different concepts of search space exploration and exploitation in order to find near–optimal solutions in an efficient way. Examples of meta-heuristics are simulated annealing (Kirkpatrick, Gelatt Jr. and Vecchi 1983), tabu search (Glover and Laguna 1997), genetic algorithms (Goldberg 1989), and of course iterated local search (Martin, Otto and Felten 1991) and ant colony optimization (Dorigo and Di Caro 1999) which will be treated in the next sections.

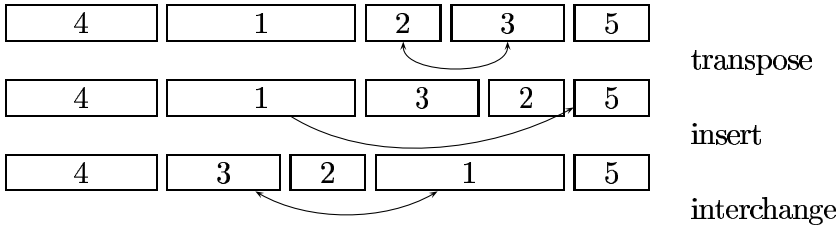


Figure 1.2: Moves that define a neighborhood

1.2 Local search

First applications of local search to $\mathcal{NP}$–hard problems stem from a time as early as the late fifties and the early sixties of the last century (Croes 1958, Lin 1965). Local search algorithms start from some initial solution and try to replace the current solution by a better solution in an appropriately defined neighborhood of the current solution. The most basic local search algorithm is iterated descent which only replaces the current solution with a better one and stops as soon as no better neighboring solutions can be found anymore.

Implementations of local search differ in the problem representation they adopt, the neighborhood they define on this representation, and the strategy they employ while searching through this neighborhood. The application of local search to machine scheduling is discussed in detail by Anderson, Glass and Potts (1997). This section only describes the local search algorithms that are used for the thesis–work.

1.2.1 Solution representation

Solutions are represented by stating the order, or sequence in which the jobs are processed, that is, a problem of n jobs is represented by a permutation of the integers $1, \ldots n$.

1.2.2 Neighborhoods

Three standard neighborhoods exist for the permutation representation. They are defined by the moves that are allowed to be applied to a sequence of jobs.

transpose: Swap two adjacent jobs. The current permutation $\pi = (\pi_1, \ldots, \pi_i, \pi_{i+1}, \ldots, \pi_n)$ is modified to $\pi' = (\pi_1, \ldots, \pi_{i+1}, \pi_i, \ldots, \pi_2)$.

interchange: Interchange is generalization of transpose. In an interchange–move $\pi = (\pi_1, \ldots, \pi_i, \ldots, \pi_j \ldots, \pi_n)$ is modified to $\pi' = (\pi_1, \ldots \pi_j, \ldots, \pi_i, \ldots, \pi_n)$.

insert: Let (i, j) be a pair of positions and $\pi = (\pi_1, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots, \pi_j, \pi_{j+1}, \ldots, \pi_n)$ the current permutation. The new permutation π' is obtained by removing the job π_i at position i and inserting it at position j. If $i < j$ a *right* insert is performed and we obtain $\pi' = (\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_j, \pi_i, \pi_{j+1}, \ldots, \pi_n)$ and if $i > j$ we get

Algorithm 1.1 Local Search: First improvement descent

Require: a solution s
Ensure: a solution s with a better or equal score of the objective function

```
1: repeat
2:   repeat
3:     pick a neighbor s_1 of s
4:     if f(s_1) < f(s_2) then /* improvement */
5:       s ← s_1
6:   until an improving move is found ∨ all neighbors of s have been considered
7: until no more improvement is made
```

$\pi' = (\pi_1, \ldots, \pi_{j-1}, \pi_i, \pi_j, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_n)$ by performing a *left* insert. Variants of this neighborhood allow only right insert moves or only left insert moves or combine these moves in another way.

There is no complete consensus on the terminology. Transpose is sometimes referred to as adjacent pairwise interchange, interchange is also called general pairwise interchange or swap, and insert is also known as shift. Figure 1.2 on the preceding page illustrates the moves on an example permutation.

There are some properties of neighborhoods which make them more or less suited for scheduling problems. First, neighborhoods differ in *size*, that is, in the number of neighbors for a single solution. Second, they differ in the *ease* at which new solution values can be computed. Third, the underlying *topology* of the neighborhood structure significantly affects the quality of solutions generated by a neighborhood search algorithm. These properties may help explain the empirically observed quality of certain neighborhoods in favor of others.

The sizes of transpose, insert and interchange are $n-1$, $(n-1)^2$ and $n(n-1)/2$, respectively, on a sequence of n jobs. Fortunately, all the neighborhood moves defined above can be evaluated rapidly for their effect on the solution fitness in case of SMWTP. Even without additional implementation tricks like those explained in Section 3.3.1 calculating the objective function value has a complexity of only $\mathcal{O}(n)$. The topology induced by these neighborhoods will be investigated in Chapter 4.

1.2.3 Pivoting rules

In a local search algorithm the *pivoting rule* determines which neighboring solution replaces the current one. The first–improvement and the best–improvement rule are most often used. In *first–improvement* local search, the neighborhood is scanned and the first found lower cost solution replaces the current solution. A skeleton of first improvement local search is provided in Algorithm 1.1. The *best–improvement* pivoting rule examines the whole neighborhood in each step and returns the best neighboring solution. If the best move is taken, an optimal solution is arrived at in less steps than following the first improvement would result in. On the other hand, determining which

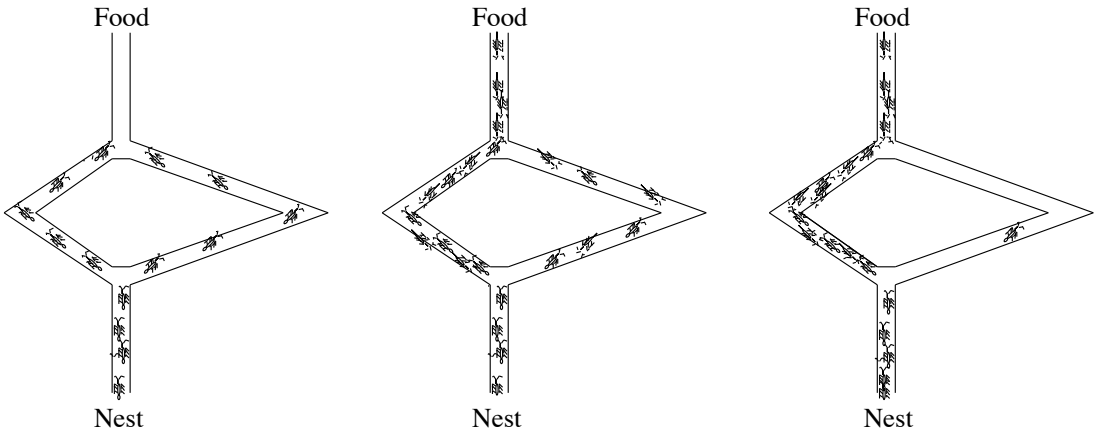


Figure 1.3: Experimental apparatus for the bridge experiment. Branches have different length. Ants move from the nest to the food source and back.

move is best requires more effort than accepting the first move that is improving. Finding out which pivoting rule to use requires experimental testing.

1.3 Ant colony optimization

One meta-heuristic in particular is employed in this thesis to solve scheduling problems: ant colony optimization (ACO). The ACO meta-heuristic (Dorigo and Di Caro 1999) is a framework that unifies approaches to combinatorial optimization problems which are inspired by the foraging behavior of real ant colonies. Algorithms that fall under the scope of the ACO meta-heuristic will be called ACO algorithms in the following.

1.3.1 The idea

The inspiring idea of ant colony optimization is the pheromone trail following behavior of real ant colonies. Real ants use this form of indirect communication as a means for cooperation. In particular, ants may deposit pheromone while walking and other ants may detect this pheromone and follow it. The more intense the strength of pheromone, the larger is the tendency to follow a particular pheromone trail.

Goss, Aron, Deneubourg and Pasteels (1989) carried out experiments with a laboratory colony of Argentine ants. Figure 1.3 illustrates how the experiment is set up. A nest is linked to a food source by a bridge with two branches of differing length. Ants that travel from the nest to the food and back must choose one of the branches. After a short transitory phase, most ants will use the shortest branch. This outcome, finding the shortest path, is explained by the pheromone following behavior of ants: at the beginning of the experiment there is no pheromone on the two branches and therefore the branches have equal probability to be chosen by an ant. The ants release their pheromone both on their way to the food and back. The shortest branch will collect more pheromone as ants that take this branch will reach the food earlier. When the first ants come back from

Algorithm 1.2 Ant colony optimization

```
1: set parameters, initialize pheromone trails
2: while termination criterion not met do
3:    construct solutions using trails
4:    apply local search /* optional */
5:    update trails
```

the food there is already more pheromone on the shortest branch and therefore they will tend to take that branch again. Ants that follow the 'right' branch can make the travel between food and nest more often and deposit their pheromone at a higher rate. In addition, more ants will tend to choose the shortest branch from the very moment it has a larger pheromone intensity. In the end the large majority of ants will end up using the shortest branch.

1.3.2 The algorithm

ACO algorithms make use of a mechanism derived from the pheromone following behavior of real ants. Artificial ants construct solutions. This construction process is guided by artificial pheromone and heuristic information. For many combinatorial optimization problems high quality heuristic information is available (consider EDD, MDD and AU introduced on page 5). This information can direct ants towards good solutions right from the start. Later on, as they reflect the ants' experience, the pheromone trails allow to improve upon these solutions. Some recent ACO algorithms achieve even better performance by applying local search to the solutions constructed by the ants.

The first ACO algorithms have been applied to the traveling salesman problem (TSP). The TSP is the problem a salesman faces when he wants to minimize the length of the tour needed to visit all his customers. Because the TSP has been the primary testbed for ant colony optimization and other ACO applications are often straight extensions upon the ACO algorithms for the TSP, I will explain the functioning of ACO algorithms at the example of the application to this classical problem. Often, the problem can be represented by a graph and the artificial ants' search can be interpreted as a constrained shortest path search on that graph. This is also the case here. The TSP is represented as a weighted graph with nodes as cities and edges as connections between the cities. Pheromone trails are associated with edges, that is, $\tau_{ij}(t)$ is the pheromone strength on edge (i, j) where t indicates the dependence of the pheromone trail on time. Additionally, the ant colony optimization!heuristic information coupled to the edges, which in all applications of ACO algorithms to the TSP is taken as $\eta_{ij} = \frac{1}{d_{ij}}$, makes the ant prefer close cities.

In general, all ACO algorithms for the TSP follow a specific algorithmic scheme which is outlined in Algorithm 1.2. Each cycle or iteration of the algorithm consists of three phases. The ants first construct solutions which can then be improved using local search. This is followed by a pheromone update to reflect the ants' search experience and to guide future ants. How exactly these phases are implemented differs between

the ACO algorithms, below I will present the particular choices taken in ant colony system (ACS), which was proposed by Dorigo and Gambardella (1997b).

Solution construction

Initially, each ant is put on some randomly chosen node. Ants choose the next node using the *pseudo–random–proportional action choice rule*: When located at node i, ant k moves with probability q_0 to node j for which $\tau_{ij}(t)^{\alpha} \cdot {\eta_{ij}}^{\beta}$ is maximal. α and β are paramters that determine the relative of pheromone trail and heuristic information. With probability $(1 - q_0)$ the ant chooses the next node probabilistically. In particular, the probability with which ant k, currently at node i chooses to go to node j at the tth iteration of the algorithm is:

$$p_{ij}^{k}(t) = \frac{\tau_{ij}(t)^{\alpha} \cdot {\eta_{ij}}^{\beta}}{\sum_{l \in \mathcal{N}_i^k} \tau_{il}(t)^{\alpha} \cdot {\eta_{il}}^{\beta}} \qquad \text{if } j \in \mathcal{N}_i^k \tag{1.1}$$

$\mathcal{N}_i^k$ is the set of cities which ant k has not yet visited. Thus q_0 is a parameter that sets the extent the ant will digress from the norm posed by the dominating trail and heuristic information in the graph.

Pheromone update

In ACS, only one ant, the one that found the best solution, is allowed to deposit pheromone. The trail update only applies to the edges of the global–best path and the new pheromone trail strength on these edges is given as:

$$\tau_{ij}(t+1) = (1 - \rho) \cdot \tau_{ij}(t) + \rho \cdot 1/L^{gb}, \tag{1.2}$$

where L^{gb} is the length of the best tour found since the start of the algorithm. Intuitively, $1/L^{gb}$ is the amount of pheromone added. In general, it is denoted by $\Delta\tau_{ij}^{gb}(t)$. ρ is a small constant value representing pheromone evaporation. *Pheromone evaporation* allows the ant colony to slowly forget its past history so that it can direct its search without being over-constrained by past decisions. In addition to the global updating rule, ants use a local update rule that they apply immediately after having crossed an edge during the solution construction:

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \rho \cdot \tau_0, \tag{1.3}$$

where τ_0 is the initial trail intensity, a small constant value.[6] Through the local update rule an already chosen edge is less desirable for the following ant. In this way a exploration of not yet visited edges is favored.

[6] In the direct predecessor of ACS, Ant-Q (Gambardella and Dorigo 1995) the term τ_0 corresponds to the discounted evaluation of the next state, a construct inspired by Q-learning. It was found that replacing this discounted evaluation by a small constant results in approximately the same performance and therefore ACS is preferred over Ant-Q. This small constant is given by $\tau_0 = 1/n \cdot L_{nn}$, where L_{nn} is the length of the tour constructed according to the nearest neighbor construction heuristic.

Local search

Recent research has indicated that, when applied to a wide variety of combinatorial optimization problems, ACO–algorithms perform best if the solutions constructed by the ants are improved by a local search algorithm. Such a combination may be very useful when constructive algorithms result in a relatively poor solution quality compared to local search algorithms. The hope is that the ants guide the local search by constructing promising initial solutions. The initial solutions generated by the ants are promising because the ants use with higher probability those edges which, according to the search experience, have more often been contained in better solutions.

In particular, for the TSP the combination of ACS with a local search algorithm proved to be such a favorable combination. Dorigo and Gambardella (1997b) have added a 3–opt algorithm to ACS to improve the ants' tours and could in that way achieve a much improved performance compared to ACS without local search.

1.3.3 Other ACO algorithms

Ant system

Ant system (AS) is the seminal ACO algorithm (Dorigo, Maniezzo and Colorni 1991, Dorigo 1992, Dorigo, Maniezzo and Colorni 1996). AS does not use local search, though it could easily be added. This makes AS a pure constructive algorithm. Ants in AS do not follow the pseudo–random–proportional , but instead they always choose the next node in the graph probabilistically according to Equation 1.1. Pheromone intensity $\tau_{ij}(t)$ is updated as

$$\tau_{ij}(t) = (1-\rho)\cdot\tau_{ij} + \sum_{k=1}^{m} 1/L_{ij}^{k} \tag{1.4}$$

In this equation, L_{ij}^{k} is the length of the tour constructed by ant k. The pheromone update applies to all edges in the graph. In AS, the ants do not apply local pheromone updating.

$\mathcal{MAX}$–$\mathcal{MIN}$ ant system

Stützle (1998b) has proposed some improvements over ant system resulting in $\mathcal{MAX}$–$\mathcal{MIN}$ ant system ($\mathcal{MM}$AS). The main modifications introduced by $\mathcal{MM}$AS with respect to AS are the following. First, to exploit the best solutions found, after each iteration only one ant (like in ACS) is allowed to add pheromone. Second, to avoid search stagnation, the allowed range of the pheromone trail strengths is limited to the interval $[\tau_{min}, \tau_{max}]$, that is, $\forall\tau_{ij}\ \tau_{min} \leq \tau_{ij} \leq \tau_{max}$. Last, the pheromone trails are initialized to the upper trail limit, which causes a higher exploration at the start of the algorithm. $\mathcal{MM}$AS and ACS perform comparably well on the TSP. $\mathcal{MM}$AS has also been applied to other combinatorial optimization problems like the quadratic assignment problem (Stützle 1997), the generalized assignment problem (Ramalhinho Lourenço and Serra 1998), and the flow shop problem (Stützle 1998a).

Algorithm 1.3 Iterated local search

1: generate initial solution s_0
2: apply local search $s_0 \rightarrow s$
3: **while** termination criterion not met **do**
4: modify solution $s \rightarrow s'$
5: apply local search $s' \rightarrow s''$
6: **if** accept **then**
7: $s \leftarrow s''$

1.3.4 Applications of ACO

Applications of ACO algorithms cover problems like the traveling salesman problem (Dorigo and Gambardella 1997a, Bullnheimer, Hartl and Strauss 1999b, Stützle and Dorigo 1999b), the quadratic assignment problem (Maniezzo and Colorni 1999, Stützle and Dorigo 1999a), the vehicle routing problem (Bullnheimer, Hartl and Strauss 1999a, Gambardella, Taillard and Agazzi 1999), the sequential ordering problem (Gambardella and Dorigo 1997). The use of ant colony optimization is not limited to static combinatorial optimization problems. For example, the ACO algorithms for network routing (Di Caro and Dorigo 1998, Schoonderwoerd, Holland, Bruten and Rothkrantz 1996) are a on competitive level and even outperform state–of–the–art routing methods in internet–like networks. An overview over these different applications of the ACO meta-heuristic is provided by Dorigo, Di Caro and Gambardella (1999).

1.4 Iterated local search

In this thesis, the effectiveness of ant colony optimization is assessed by comparing its performance with the performance of another meta-heuristic, iterated local search (ILS). ILS is a relatively simple algorithm but, despite this fact, it yields very good results on a variety of problems.

1.4.1 The algorithm

The problem of the descent local search getting stuck at local optima is attacked in a very simple but effective way by iterated local search. It just kicks the hill-climber out of the local optimum leading it to a solution beyond the neighborhood searched in the local search algorithm in the hope that it will reach a better local optimum in the next local search application. More specifically, the idea is to modify the current solution s into an intermediate solution s' by a series of kick–moves and then apply the local search to s' to reach a new local optimal solution s''. An acceptance criterion determines whether the search is continued for s' or s'', or some other solution. This idea stems from Baum (1986), has been developed further by Martin et al. (1991) and is captured in Algorithm 1.3. The success of ILS is apart form local search dependent on the type and strength of the kick–move, and on the specification of the acceptance

criterion.

1.4.2 Applications of ILS

ILS algorithms are currently among the best performing approximation methods for many combinatorial optimization problems like the traveling salesman problem (Martin et al. 1991, Johnson and McGeoch 1997). ILS has also been applied with success to the flow shop problem (Stützle 1999), the job shop problem (Ramalhinho Lourenço 1995), and, most interesting, to the single machine total weighted tardiness scheduling problem (Congram, Potts and Van de Velde 1998), where iterated dynasearch, as the algorithm is called, is claimed to outperform all other meta-heuristics that have been applied to the problem. In the literature, iterated local search is often disguised under another name like "large step Markov chains" (Martin et al. 1991) or "simple variable neighborhood search" (Hansen and Mladenović 1999).

1.5 Experimental methodology

1.5.1 Empirical evaluation

In this thesis, the performance of ant colony optimization (ACO) is compared with that of iterated local search (ILS) on the single machine total tardiness problem (SMTTP) and the single machine total weighted tardiness scheduling problem (SMWTP). The comparison is accomplished by empirical evaluation of the meta-heuristics' performance. The guidelines put forward by Barr, Golden, Kelly, Resende and Steward (1995) are kept in mind in this process. These researchers state that any report on heuristics should reckon with at least the solution quality or effectiveness, the computational effort or efficiency and the robustness of the performance.

The quality of a solution is measured relative to the optimal or best known solution for the problem instance. The average solution quality produced by the ACO and ILS will be reported. Most of the time, however, efficacy is interpreted as finding the optimal, or best known, solution. In such cases, the estimated probability to find the optimal solution in dependence on the run–time will be reported.

The efficiency of the algorithms is measured in CPU–time. Both the ACO and ILS algorithm are implemented in the C++ programming language and tested on the same computer. Whenever possible the same data–structures are used for both algorithms.

The ability of the meta-heuristics to perform well over a wide range of problems, their robustness, is dealt with simply by testing them over a wide range of problems. The test problems are benchmark instances which have been used in earlier studies. These test problems cover the whole range of characteristics that the SMTTP and the SMWTP have to offer. In addition problems of different size are examined.

1.5.2 Run time distributions

In recent years, Hoos and Stützle (1998) have developed a novel type of evaluation methodology to analyze algorithms which involve random decisions during the search. This methodology, based on measuring run time distributions of algorithms applied to optimization problems, will be used in this thesis to analyze the behavior of ACO and ILS.

The run time required by an algorithm to achieve a specific goal, like finding an optimal solution to a given problem instance, is a random variable. The run time distribution (RTD) of this variable can give valuable information for the analysis and the characterization of an algorithm's behavior and can provide a basis for the comparison of algorithms. Knowledge on the RTD is obtained by running the algorithm many times on an instance and keeping track of the solution quality and run time whenever a new best solution is found. The empirical distributions can then be estimated a posteriori. The estimated cumulative run time distribution $\widehat{G_c}(t)$ to reach a solution quality bound c is computed as

$$\widehat{G_c}(t) = \frac{|\{j | rt(j) \leq t \wedge f(s_j) \leq c\}|}{k} \tag{1.5}$$

In this equation, k is the number of times the algorithm is run, c is required solution quality, $f(s_j)$ the best solution found so far, and $rt(j)$ the run-time of the j-th algorithm run. In practice, an algorithm may fail to reach a required solution within the available CPU–time. In that case a truncated RTD is measured.

The run time distribution gives information on the algorithm's behavior on specific problems instances. Often, similar behavior is observed on many of the instances and in that case the RTD allows to draw conclusions on the algorithm's conduct on this problem class. Particularly in Chapter 3 run time distributions are used to tune and evaluate the ACO and ILS algorithms.

1.6 Summary

In the foregoing, I have presented the two meta-heuristics that will compete on scheduling problems for the rest of this report. The problems under investigation in Chapter 2 and Chapter 3 are $\mathcal{NP}$–hard, so approximation algorithms are needed. Both ant colony optimization and iterated local search have a been successfully applied to a number of $\mathcal{NP}$–hard combinatorial optimization problems, however, it is hard to predict which of the algorithms will come out best.

A meta-heuristic will be successful if it can strike the right balance between exploitation of the accumulated search experience and the exploration of the search space to identify regions with high quality solutions. The meta-heuristics under consideration here differ in the tools they can employ to achieve that goal. More fundamental is the difference in philosophy underlying iterated local search and ant colony optimization. Ant colony optimization is based on behavior found in nature, iterated local search is a direct solution to the problem of descent local search that gets trapped in a local optimum. Similarly, the configuration of ant colony optimization tends to focus on the

creation of circumstances in which the ants will thrive, obtaining a graph representation of the problem on which ants can easily find the optimal solution, whereas the configuration of iterated local search tends to focus on the acquisition of a mechanism that is rich enough to help the descent procedure search through the landscape defined by the local search operators. In the following it will become clear which approach is the most viable.

2 Algorithms for the single machine total tardiness problem

2.1 Introduction

The single machine total tardiness problem (SMTTP) is a special case of the single machine total weighted tardiness scheduling problem that considers each job to be equally important. Since the priority factors attached to jobs are discarded, the resulting problems are less complex and, in practice, easier to solve. The SMTTP is also interesting in its own right and has attracted a large amount of attention throughout the years (Koulamas 1994, Russell and Holsenback 1997). As a result a wide range of algorithms has been developed that attempt to solve the problem. Most of these algorithms either perform an exhaustive search or construct solutions according to a dispatching policy. Only few applications of meta-heuristics to the SMTTP have been reported.

One of the algorithms that follow a meta-heuristic is the approach to the SMTTP which was recently proposed by Bauer, Bullnheimer, Hartl and Strauss (1999a, 1999b). These researchers applied ant colony system (ACS) to the problem and claim that their approach, ACS-SMTTP, outperforms all leading heuristics. This claim deserves further investigation, especially because it is not accompanied with information on the amount of computation needed to reach the reported results. The performance of ACS-SMTTP on the single machine total tardiness problem is a first test–case for the suitability of ant colony optimization for single machine sequencing problems. Re-implementation and further examination of this first ACO algorithm for the SMTTP will, in addition to forming a check of their implementation choices, at least hint whether ant colony optimization is appropriate for the single machine total weighted tardiness scheduling problem. To put the performance of the algorithm into perspective, ACS-SMTTP is compared with iterated local search.

The analysis performed in this chapter will remain rather shallow as it is intended to get a basic feel with main issues of concern in this thesis only. A more profound analysis of the algorithms is given in Chapter 3 which is concerned with the more complex and interesting single machine total weighted tardiness scheduling problem.

2.2 The single machine total tardiness problem

2.2.1 The problem

A single machine that can handle only one job at a time processes n jobs without preemption. Every job j is available for processing at time zero and requires a positive processing time p_j. The completion of the job is committed to a due date d_j. If C_j is the actual completion time of job j, then the job's tardiness T_j is defined as $T_j = \max\{0, C_j - d_j\}$, the time with which the job exceeds its due date. The objective of the single machine total tardiness problem (SMTTP) is to find a feasible processing sequence that includes all jobs ($j = 1, \ldots, n$) and minimizes the total tardiness $T = \sum T_j$.

2.2.2 Related research

The landmarks of SMTTP–research are set by Emmons (1969), Lawler (1977) and Du and Leung (1990). Thirty years ago Hamilton Emmons formulated dominance conditions that describe the circumstances under which jobs precede each other in at least one optimal sequence. A decade later Lawler showed that an SMTTP–instance is pseudo-polynomially solvable by deriving an $\mathcal{O}(n^4 \sum_j p_j)$ algorithm based on decomposition. The complexity of the SMTTP has remained open for many years. Only recently Du and Leung (1990) proved that the problem is $\mathcal{NP}$–hard.

Likewise, three classes of algorithms have emerged. First of all, there are algorithms that focus on getting a solution of reasonable quality without much computational effort. The WI heuristic (Wilkerson and Irwin 1971), the NBR heuristic (Holsenback and Russell 1992) and the PSK heuristic (Panwalkar, Smith and Koulamas 1993) are of this type. These algorithms incorporate dispatching rules to get a quick estimate of the desirable order of the jobs. A promising policy for the single machine total tardiness problem is to give priority to the earliest due date (EDD). The shortest processing time (SPT) dispatching rule works well if most of the jobs are delayed; the modified due date (MDD), defined as MDD $= \max\{C_j + p_j, d_j\}$, seems the most suited since it is directly based on the findings of Emmons (1969) and therefore meets the problem characteristics best.

Deterministic algorithms that focus on finding optimal solutions constitute the second class. Considerable progress has been made in this area. The first branch–and–bound approach (Emmons 1969) could only find optimal solutions on small problems. Potts and Van Wassenhove (1982) combined Lawler's decomposition approach with the dynamic programming algorithm of Schrage and Baker (1978) to solve instances with up to 100 jobs. State–of–the–art results are obtained by the algorithm of Swarc et al. (1998) which efficiently handles instances with up to 300 jobs.

After it was proven that there exist instances of the problem that are not solvable within polynomial time, interest has risen to apply nondeterministic meta-heuristics. They form the third class of algorithms. Ben-Daya and Al-Fawzan (1996), for example, report on a simulated annealing approach. The ACS-SMTTP belongs to this class, too.

2.2.3 Problem instances

Instances are generally classified by their tardiness factor (TF) and range of due dates (RDD). TF is given by

$$\mathrm{TF} = 1 - \frac{\sum_{j=1}^{n} d_j}{(n \sum_{j=1}^{n} p_j)} \tag{2.1}$$

It indicates how tight the due dates are. TF = 0 if due dates are on average as late as the completion time of the job that is processed last. TF = 1 if, on the other hand, all jobs are committed at the starting time of the system. RDD is the difference between the earliest and the latest due date normalized over the sum of all processing times:

$$\mathrm{RDD} = \frac{d_{\max} - d_{\min}}{C_{\max}} \tag{2.2}$$

RDD = 0 indicates that the jobs have a common due date. RDD = 1 is obtained if due dates differs as much as the sum of all jobs' processing times.

Many of the SMTTP–instances are generated randomly. Often, in any case for the instances I have applied, TF and RDD are used as parameters in the generation process. The processing time p_j is an integer generated randomly according to a uniform distribution $[1, 100]$. The integer due date d_j is generated randomly according to a uniform distribution in $[(1 - \mathrm{TF} - \frac{\mathrm{RDD}}{2}) \cdot \sum p_j, (1 - \mathrm{TF} + \frac{\mathrm{RDD}}{2}) \cdot \sum p_j]$. A range of values for RDD and TF is chosen to form a representative sample of the problem. The benchmark set of Potts and Van Wassenhove (1991), for example, consists of a group of 50–job problems and a group of 100–job problems. Each group contains five instances of all pairs of TF and RDD taken from the set $\{0.2, 0.4, 0.6, 0.8, 1.0\}$ resulting in a total of 2×125 instances in the benchmark set.

In this chapter, the algorithms' performance is reported for the 125 100–job instances. In addition, the algorithms are tested on problems of 150 to 400 jobs which were generated by Kara[1].

2.3 Using ant colony optimization

I already mentioned that an ACO algorithm has been developed for the SMTTP. In this section, the configuration of this algorithm, ACS-SMTTP, is described, and obscurities in the approach are pointed out.

2.3.1 The Vienna approach

Andreas Bauer at the University of Vienna was the first to apply ACO algorithms to the SMTTP. He adopted the ACO algorithms for the traveling salesman problem (TSP) which are described in Section 1.3.2 and applied them to the SMTTP.

When applied to the SMTTP, each ant starts with an empty sequence and then iteratively appends an unscheduled job to the partial sequence constructed so far. The desirability to add job j as ith element to the sequence is reflected in the value $\tau_{ij}(t)^{\alpha} \cdot \eta_{ij}{}^{\beta}$,

[1] Available on http://www.bilkent.edu.tr/~bkara/start.html.

where α and β are parameters. $\tau_{ij}(t)$ is the pheromone trail associated to the assignment of job j to position i, and η_{ij} is the heuristic information provided by dispatching rule. For example, $\eta_{ij} = 1/d_j$ if the earliest due date rule is used.

The solutions constructed by the ants are evaluated and local search is applied to the solution with the lowest total tardiness. Afterwards, a positive reinforcement $\Delta\tau_{ij} = 1/T^{gb}$ is given to the pheromone trail that corresponds to the best solution found so far (see Equation 1.2 on page 10). T^{gb} is the total tardiness of this solution. The initial trail intensity τ_0 is $\tau_0 = 1/(n \cdot T_{\text{EDD}})$, where T_{EDD} is the total tardiness for a sequence of jobs in non–decreasing EDD order and n is the number of jobs.

Four ACO variants, ant system, ant colony system, its predecessor Ant-Q, and $\mathcal{MAX}$–$\mathcal{MIN}$ ant system, were first tested on an SMTTP instance of 10 jobs posed by Holsenback and Russell (1992). The ACO algorithms were run with various heuristics. The combination of ant colony system (ACS) and the modified due date (MDD) heuristic proved best for this problem, $\mathcal{MAX}$–$\mathcal{MIN}$ ant system with MDD second best.

Next, ACS guided by MDD was combined with local search. For the local search, algorithms based on the transpose neighborhood ($\cdots \pi_i \pi_{i+1} \cdots$) and algorithms based on the interchange neighborhood ($\cdots \pi_i \cdots \pi_j \cdots$) were considered (see Section 1.2.2); additional tests were done to determine whether local search should be applied to all solutions found by the ants or only to the best solutions. These tests were performed on the 50–job and 100–job instances of Potts and Van Wassenhove (1991).

Finally, a number of settings for ACS parameters α, β, q_0, ρ and population size were tested on the instances. Based on the experimental results the following parameter–settings were chosen for ACS-SMTTP: $\alpha = 1$, $\beta = 2$, $q_0 = 0.9$, $\rho = 0.1$, 20 ants, heuristic information provided by MDD with local search based on the interchange neighborhood search and applied only to the iteration best solution of the colony.

2.3.2 Questions

The publications on the Vienna approach to SMTTP raise a number of questions. The first two questions relate to design choices, the last three are on comparison, efficiency and scaling — in that order.

1. *Which neighborhood is searched and what type of descent is used?* It is not entirely clear which neighborhood was ultimately chosen. Bauer et al. (1999a, Section 4.3) suggest a transpose neighborhood was used, while Bauer (1998, p. 58) reports better results for the interchange neighborhood. It is not made explicit either whether first improvement pivoting rule or best improvement pivoting rule is used to search the neighborhood. A look at the code (Bauer 1998, p. 91, 92) reveals that the first improvement rule is implemented. Yet, it could be that best improvement or local search based on the insert neighborhood performs significantly better.

2. *Wouldn't it be better to apply local search to all solutions?* Previous research on the extension of ant colonies with local search indicates that applying the local search to all solutions is often the best approach (Stützle 1998b). Surprisingly, here the opposite is done. The purpose of ants in ACO algorithms that make heavy use of

local search is to guide the local search out of the local optimum so that it can find a better optimum. It is by no means obvious that the solution that scores best on the objective function gives the best starting point for the local search. What's more, such is solution has a high likelihood of being the old local optimum or near to the old local optimum. So, if one wants to apply the local search to only one of the ants' solutions, then taking the worst solution is perhaps even a better choice.

3. *What is the quality of this approach relative to other more simple meta-heuristics like iterated local search?* Comparisons with results on Potts and Van Wassenhove's (1991) benchmark set are published by Bauer, Bullnheimer, Hartl and Strauss (1999a, 1999b). However, these are comparisons with simplistic algorithms like NBR and PSK and they are not based on similar computing times and therefore of little use.

4. *How long does it take to solve an instance? How hard are the benchmark instances anyway?* Knowledge on these issues is a prerequisite for comparisons with other algorithms. Still, so far nothing is reported on the computational effort ACS-SMTTP requires to solve an instance of the SMTTP. For any problem there exist instances which are easy to solve, but to assess ACS-SMTTP's performance one should identify hard instances and concentrate on them — on easy instances less complex algorithms suffice, as will be shown later.

5. *What about larger instances?* The parametric study on the 50–job set suggests that ACO is not useful for problems of that size (Bauer 1998, p. 61, 72). The best value for α is zero which indicates that the best performance is achieved if the pheromone mechanism is not used at all. On the set of 100 jobs ant colony optimization seems to be more effective, because here better performance is achieved with $\alpha = 1$ than $\alpha = 0$. One expects that on larger instances the need for a meta-heuristic like ant colony optimization is even more obvious: local search is more expensive on bigger search spaces and therefore guiding the search will be more necessary. This conjecture remains to be confirmed, though.

2.4 Implementation

In the next section, an ACO algorithm and an ILS algorithm will be compared on a number of SMTTP instances. Here details on the implementation of these algorithms are presented.

2.4.1 Local search

Initially, naive implementations of local search were used, that is, evaluation of moves was done by computing the tardiness for each of the jobs in the sequence. The neighborhood moves described in Section 1.2.2 were implemented. Preliminary tests, which will be discussed in a moment, singled out the interchange neighborhood as the most promising.

Based on these results an effort was made to implement a more efficient interchange neighborhood search. Less naive evaluation of interchange moves was obtained based on the observation that the tardiness only changes for jobs within positions i and j for the move $(\cdots \pi_i \cdots\cdots \pi_j \cdots)$; jobs outside these positions need not be evaluated. Both a local search using the best improvement pivoting rule and a local search using the first improvement pivoting rule were implemented.

Even more efficiency was gained through adoption of the speedups which Congram et al. (1998, Section 3.4) propose for the interchange local search on the single machine total weighted tardiness scheduling problem (SMWTP). These speedups are explained in depth in Section 3.3.1, since they were initially implemented for that problem and only later adopted for the SMTTP. Even though the speedups are designed for application to the SMWTP, they can be used for applications of local search to the SMTTP as well: take a constant value, say one, everywhere a weight is considered in the procedure.

2.4.2 Ant colony system

The ant colony system I use differs at some minor points from Bauer's implementation. At the initialization of the ant colony system, a starting solution is build by application of local search to a sequence of jobs that is constructed on the basis of the MDD rule. The purpose of this initialization is twofold. First of all, it prevents that the ants come into action for instances that can be solved with one local search application. Secondly, as they start with a high quality trail, ants are immediately guided towards regions of the search space where good solutions are found.

Superfluous computation is prevented by a somewhat ingenious initialization and maintenance of the values of $\tau_{ij}(t)^{\alpha} \cdot {\eta_{ij}}^{\beta}$. In a naive implementation each time an ant requests its environment for information on the desirability to put job j on position i, τ_{ij} would be looked up, η_{ij} would be computed — in case of EDD given as $1/d_j$ — and finally those values would be combined by applying the formula given above. When one uses static heuristic information and only alters pheromone–intensities of edges that are part of the iteration or global best solution, then it is better to fill a table with the values of $\tau_{ij}(t)^{\alpha} \cdot {\eta_{ij}}^{\beta}$ right at the start, use this table as a substitute, and update this table only when necessary.

2.4.3 Iterated local search

The implementation of ACS-SMTTP is compared against an algorithm that consists of multiple local search applications that are dependent as the initial solution of the local search procedure is the suitably perturbed solution of the previous local search application. The outline of algorithms of this type, known as iterated local search (ILS), is given in Algorithm 1.3 on page 12. The ILS algorithm will use the same local search as is employed by the ACO algorithm. Like in ant colony optimization, the initial solution is generated by the MDD construction heuristic. As long as the global optimum is not found, the current best solution is perturbed by six random interchange moves and then local search is applied to this solution. The search always continues by applying

	Δ_{avg}	Δ_{worst}	n_{opt}
insert	0.42	7.04	61
transpose	1.28	12.57	54
interchange	0.07	1.88	87

(a) local search neighborhoods

	Δ_{avg}	Δ_{worst}	n_{opt}	t_{avg}
first improvement	0.07	1.80	92	0.08
best improvement	0.06	1.53	90	0.08

(b) interchange local search search strategies

Table 2.1: Effectiveness of local search for the SMTTP. Each variant is run once on each of the 125 100–job instances. For a solution π the deviation Δ to the optimal solution π^{gb} is defined as $\Delta = 100 \cdot (f(\pi) - f(\pi^{gb})/f(\pi^{gb})$ where f is the objective function of the SMTTP. Δ_{avg} is then the average deviation and Δ_{worst} is the largest deviation. n_{opt} is the number of optimal solutions found (out of 125) and t_{avg} is the CPU time averaged over the 125 instances.

kick–moves to the best solution found so far. This ILS configuration is merely based on hunches on what might yield good results, a more thorough investigation of the design choices and parameter settings for iterated local search is done in Section 3.4 on the configuration of ILS algorithms to minimize total weighted tardiness.

2.5 Results

In this section, I discuss the computational results which hint how the questions posed in Section 2.3.2 should be answered. These questions concern both the configuration of ACS-SMTTP and its exact performance. To make comparison with the original results easier, I will test my re-implementation on the 100–job SMTTP instances that were used by Bauer, and kindly provided by H. A. J. Crauwels. In addition, more rigorous performance assessment is made on some of the instances of over 100 jobs available at http://www.bilkent.edu.tr/~bkara/start.html. The initial tests were run on a Pentium II (266 MHz) with 320 MB RAM, the other tests on problems of 150 and more jobs were run on a Pentium III (450 MHz) with 256 MB RAM.

2.5.1 Local search configuration

Local search is one of the most important components for both ant colony optimization and iterated local search and differences in neighborhoods and implementation of local search may have a large effect on the meta-heuristic's performance. To ascertain which neighborhood works best and what pivoting rule should be employed, I have implemented a series of local search variants and run them on the 125 100–job SMTTP instances. Table 2.1 summarizes the results.

In Table 2.1(a) the transpose, the insert, and the interchange neighborhood are compared with respect to the average deviation from the optimum (Δ_{avg}), the worst deviation from the optimum (Δ_{worst}) and the number of optimal solutions (n_{opt}) found after one run on each of the 125 instances. The local search checks the neighborhood according to the best improvement pivoting rule. Undoubtedly, local search with the

	i_{avg}	Δ_{worst}	n_{opt}	t_{avg}
ILS	18.63	0	625	1.25
ACS-SMTTP	7.07	0.03	623	3.71
ACS-SMTTP′	0.72	0	625	1.85

Table 2.2: Effectiveness of ant colony optimization and iterated local search for the SMTTP. The algorithms are run five times with a computation bound of 100 seconds on the 100–job benchmark set. The average number of iterations, i_{avg}, the worst deviation to the optimal solutions, Δ_{worst}, the number of runs terminated successfully, n_{opt}, and the CPU time are averaged over the 5×125 tries, t_{avg} is given.

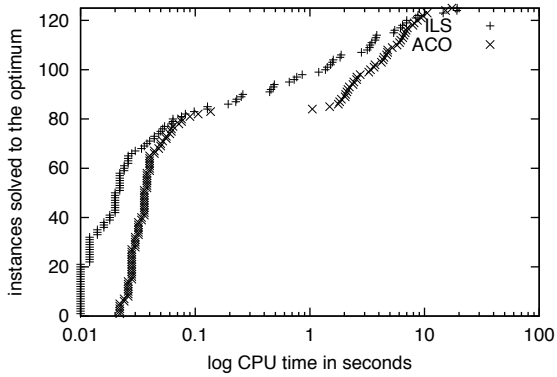


Figure 2.1: Cumulative distribution of run times of ACS-SMTTP′(×) and ILS(+) over the 125 benchmark instances. The algorithms were run 5 times on each instance with a CPU limit of 100 seconds. The average run times per instance are sorted. A point in the plot depicts the average run time on which the nth instance is solved.

interchange neighborhood reaches the highest solution quality: it has the lowest average and worst case deviation combined with the highest amount of solved instances.

The results in Table 2.1(b) do not clearly indicate which pivoting rule is best used. In the end, I decided to use best improvement local search.

2.5.2 Local search application

Now that the a local search configuration is settled, it should be determined how the local search is best employed. The first concern is whether the local search should be applied to all the ant's solutions, as is done in ACS-SMTTP′, or only to the best ones which is done in the original ACS-SMTTP. Of later concern is the question whether ant colony optimization or iterated local search provides the best environment for application of local search.

Table 2.2 summarizes the results for five runs of ACS-SMTTP, ACS-SMTTP′, and ILS on the 125 100–job SMTTP instances. The table's data confirm the supposition that it is better to apply the local search to all the ants' solutions: on average ACS-SMTTP′ finds the optimal solution two times faster than ACS-SMTTP and while ACS-SMTTP′ always solves the instances, ACS-SMTTP does not succeed 2 times out of the 5×125 tries. Interestingly, ACS-SMTTP′ needs on average less than one iteration to solve an instance. The explanation for this is that often instances are solved already after the first local search — in the zero*th* iteration, while only a few require more iterations. In the 0.72 iterations of ACS-SMTTP′ containing 20 ants an equivalent of $20 \cdot 0.72 = 14.4$ local search applications are performed. This is less than ILS goes through on average.

ACS-SMTTP needs even less local search applications: 7.07 — one per iteration. Yet, the overhead required to achieve this smaller number of local search runs is more costly than the savings it results in.

A cumulative distribution[2] of the average run time over the 125 instances is given in Figure 2.1 on the facing page for the ILS algorithm and for ACS-SMTTP′. The distribution of ACS-SMTTP′'s average run times shows a gap between 0.1 and 1 seconds. During this period the algorithm is occupied with first the application of a local search to each of the ants' solutions. The approximately 90 instances that fall before the gap are solved by the local search application at the algorithms' initialization. This number of instances corresponds to the n_{opt} values reported in Table 2.1 on page 23. The fact that initialization of ACS-SMTTP′ is more costly than the ILS initialization is reflected in the difference in the distribution of the easiest 90 instances.

Because ILS needs less time for one iteration it is able to solve more instances during the first seconds. An iteration of ACS-SMTTP′ is in fact a sort of multi–start local search. Multiply applying local search a slightly varying starting solutions is usually outperformed by an iterated application. So, it is not surprising that iterated local search has an advantage during the first iterations. Yet, in the end, the hardest instances require the same run time of both ILS and ACS-SMTTP′. So, apparently, the ACO solution construction process can catch up with the ILS solution modification mechanism for more difficult SMTTP instances.

2.5.3 Hard instances

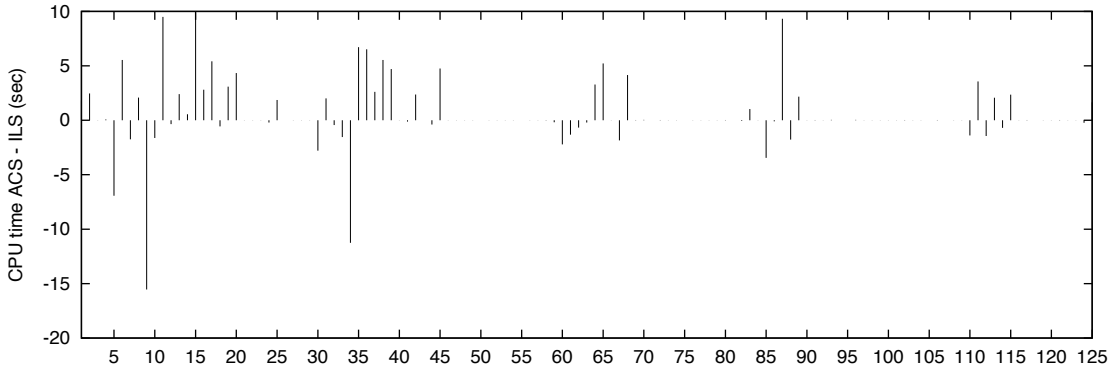
In the foregoing only global results were presented. It turned out that only a few instances require more than one or two local search applications. Focusing on those instances might provide more insight, therefore I shift the attention towards the individual instances.

Figure 2.2 on the following page indicates how instances and the algorithms' run times relate to each other. More specifically, Figure 2.2(a) compares the run times of ILS with the run times needed by ACS-SMTTP′ and Figure 2.2(b) plots for each of the instances the corresponding average ILS run times. Figure 2.2(c) gives the values for the parameters that were used in the generation of the instances. It helps to differentiate among the instances.

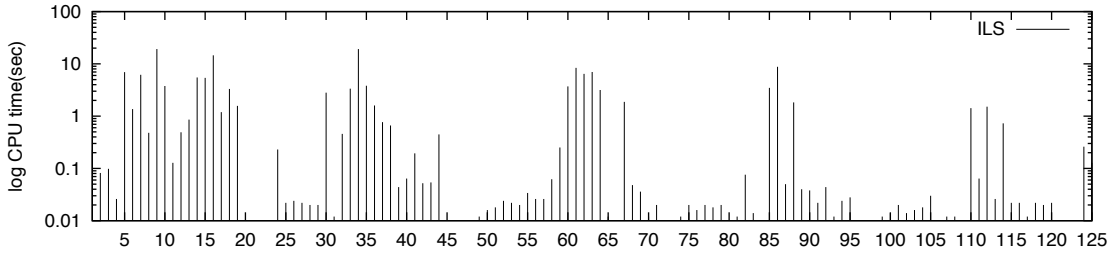
Figure 2.2(a) shows that for most instances it makes no difference whether ILS or ACS-SMTTP′ is applied. Still, there are considerably less instances where ILS needs more time to find the optimum than ACS-SMTTP′.

It is risky to draw conclusions based on the results in Figure 2.2(b). Clearly, the run times vary per instance. Yet it is not clear whether this is due to randomness in the algorithm's behavior or due to differences between the instances. Figure 2.2(b) shows a kind of periodic behavior that suggests a link with the tardiness factor and range of due dates of the job. According to Holsenback and Russell (1992) instances with a small

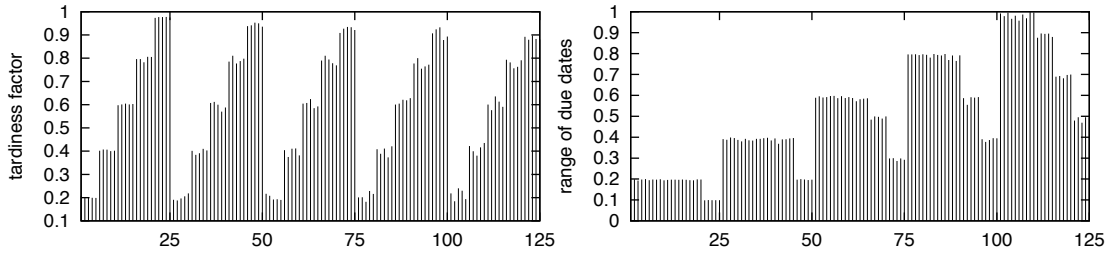
[2] This is *not* a run time distribution (RTD). An RTD considers the distribution of the algorithm's run times on a single run. Here, on the contrary, it is the distribution of instance solving time over the benchmark set that is investigated.



(a) difference between the average run time of ACS-SMTTP/ and ILS. A negative time indicates the amount that the average run time of ILS exceeds the average run time of ACS-SMTTP/. A positive time indicates how much ILS is faster.



(b) log of the average CPU–time ILS needs to solve an instance



(c) tardiness factor (TF) and range of due dates (RDD) corresponding to each instance

Figure 2.2: Detailed results on the run times of the algorithms that were run five times on the 100–job benchmark set with a 100 second computation bound.

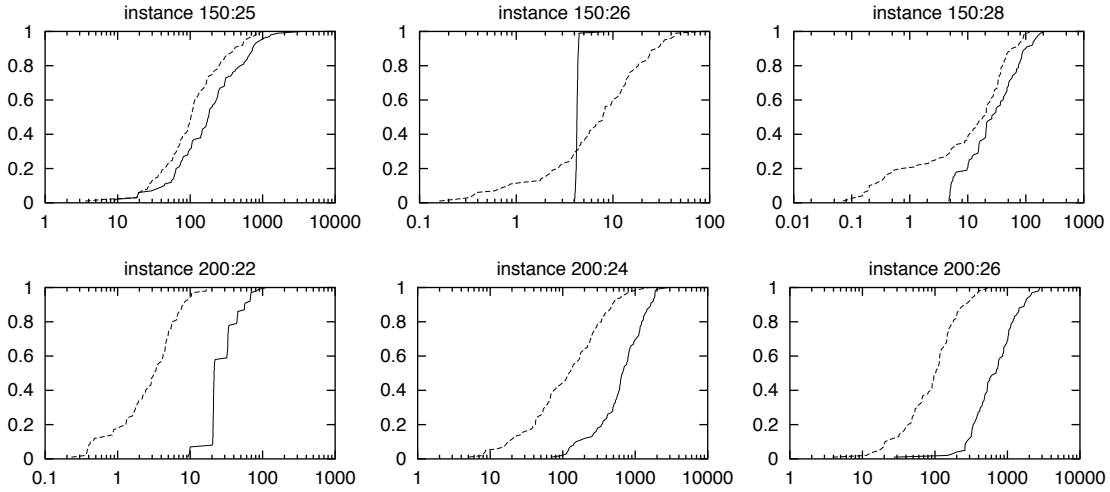


Figure 2.3: Run time distributions giving the empirical cumulative probability of finding the optimal solution for ACS-SMTTP/ (solid line) and iterated local search (dotted line) on SMTTP instances of 150 and 200 jobs with TF = 0.6 and RDD = 0.2. The algorithms are run 100 times with a computation time of 20 minutes on each of the instances. The empirical solution probability is given on the y-axis, the log CPU time in seconds on the x-axis.

range of due dates and a tardiness factor between 0.6 and 0.8 are especially difficult to solve. A similar observation can be made from Figure 2.2(b).

2.5.4 Large instances

The 100–job problems alone cannot provide a sound basis for comparison of iterated local search and ant colony optimization, as most instances are solved after one ACO iteration. On instances that contain more jobs, the local search will be more costly, the search space much larger and more complex, and therefore the difference in performance of the meta-heuristics may become more marked. On http://www.bilkent.edu.tr/~bkara/start.html one can find SMTTP instances which are generated in the same way as the 100–job benchmark set and contain more jobs. Problem instances were generated with 150, 200, 300, 400, and 500 jobs. It was found, again, that the instances with TF = 0.6 and RDD = 0.2 were the most difficult to solve, so I tested the algorithms only on these instances.

Figure 2.3 depicts the distribution of the run times the algorithms need to solve instances of 150 and 200 jobs[3]. The corresponding numerical results are given in Table 2.4 on page 29 and Table 2.5 on page 30. The names of the instances correspond to their original names on Kara's web-page[4]. Clearly, the run times of ILS, given as a dotted line in the figure, are generally shorter than ACO's run times, given as solid lines. Though

[3]See Section 1.5.2 for an introduction into run time distributions.

[4]The instances are numbered consecutively. Instances 1 to 10 have RDD = 0.2 and TF = 0.2, instances 11 to 20 RDD = 0.2 and TF = 0.4, etcetera.

	21	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*
150	2	1	3	2	3	1	2	9	3	1
200	4	18	39	54	7	31	13	27	16	40
300	172	219	320	310	221	206	418	340	113	264
400			1365	3029			2824			

Table 2.3: Run times of Swarc et al.'s (1998) algorithm for the SMTTP measured on a Pentium III (450 Mhz) with 250 MB RAM under Windows 95 ©.

the instances have the same tardiness factor and range of due dates, they differ strongly in the run times needed to solve them: the 200–job instances generally require more time due to problem size, yet some 150–job instance take longer to solve on average than some 200–job instances. In the tables, the average run time for ACS-SMTTP′ on the 150–job instances ranges from 4.29 to 1203.21 seconds, for ILS from 0.37 to 748.77 seconds. In addition, a run can be a factor 100 slower or faster than the another run with the same algorithm on the same instance. Hence, even on a single instance run times differ very strongly.

2.5.5 Comparison to exact methods

Table 2.3 gives the run times of Swarc et al.'s (1998) state–of–the–art algorithm for the SMTTP. On problems up to 300 jobs this exact algorithm is clearly preferable over the approximation algorithms devised in this chapter. At the same time, the meta-heuristics are less severely affected by the increase of jobs. The run times of the exact algorithm appear to scale exponentially with the increase of the number of jobs. This is clearly not the case for the approximation algorithms (see Table 2.4 and Table 2.5). So, on problems of 500 and more jobs it might well be that an ILS or ACO method is more efficient than the algorithm of Swarc et al. (1998).

2.5.6 Summary of results

In Section 2.3.2, I formulated five questions on Bauer et al.'s (1999a) approach to the SMTTP. They can now be answered.

1. The first question concerned the configuration of local search. The results of Section 2.5.1 indicate that in ACS-SMTTP an interchange local search should have been used to achieve the best performance. Whether the neighborhood check was governed by the best improvement or first improvement pivoting rule does not make a big difference.

2. Next, it was asked if it wouldn't be better to apply the local search to all the solutions. The results of ACS-SMTTP and ACS-SMTTP′ in Table 2.2 on page 24 strongly suggest the latter configuration should be used, contrary to the findings of Bauer (1998).[5]

[5]Bauer (1998, Section 3.3.2) has tested whether local search should be applied to all ants' solutions or

	t_{min}	t_{max}	t_{avg}	t_{stddev}	i_{min}	i_{max}	i_{avg}	i_{stddev}	n_{opt}
150:21	0.64	48.61	9.08	8.44	4	273	48.72	45.35	100
150:22	0.22	62.63	7.70	9.18	1	311	37.53	44.93	100
150:23	6.22	3579.62	748.77	664.12	33	18985	3884.48	3445.62	95
150:24	0.13	1.19	0.37	0.23	1	6	1.90	1.19	100
150:25	3.57	998.55	172.06	192.80	23	5697	983.45	1102.95	100
150:26	0.16	86.79	11.89	13.05	1	481	65.77	71.66	100
150:27	0.43	98.33	21.60	17.43	3	501	113.93	91.47	100
150:28	0.07	118.13	24.95	26.70	1	628	130.84	139.34	100
150:29	0.24	4.60	1.43	0.88	1	27	7.42	4.91	100
150:30	0.12	42.06	6.79	7.76	1	214	34.88	39.87	100
200:21	2.56	307.86	58.45	45.32	7	677	136.62	102.48	100
200:22	0.23	18.09	4.04	3.50	1	48	10.46	8.74	100
200:23	3.88	305.65	74.99	58.30	9	629	154.43	119.97	100
200:24	5.03	1535.40	210.77	251.38	10	3086	425.60	505.01	100
200:25	0.58	57.73	10.03	11.15	2	142	24.31	26.47	100
200:26	3.89	540.71	125.49	104.58	8	1177	273.25	227.62	100
200:27	4.00	598.02	102.96	90.89	8	1293	232.93	200.13	100
200:28	1.20	176.59	41.39	31.53	3	440	103.68	79.24	100
200:29	1.23	57.47	18.93	14.02	3	139	44.59	32.88	100
200:30	0.59	76.21	18.75	16.37	1	154	38.47	33.74	100
300:21	7.24	352.29	86.61	59.11	5	255	61.31	41.98	100
300:22	22.71	992.83	234.53	177.05	20	849	202.10	156.03	100
300:23	10.46	1001.92	212.29	212.90	9	681	139.62	143.29	100
300:24	22.04	3388.42	567.00	526.27	14	2282	388.86	361.30	100
300:25	22.61	760.22	300.16	215.41	17	694	255.71	186.75	83
300:26	59.47	2979.03	1141.27	839.70	42	2210	858.27	632.66	81
300:27	19.62	1275.16	280.03	225.39	18	843	197.02	154.91	100
300:28	164.91	3423.47	1179.71	820.34	107	2541	851.86	600.55	73
300:29	14.79	3200.84	905.45	755.36	12	2400	675.10	567.24	96
300:30	1.66	151.48	31.72	27.89	1	102	21.93	18.70	100
400:23	96.43	2940.67	748.34	601.92	27	766	201.85	159.32	79
400:24	248.07	3522.89	1550.16	865.78	76	1109	487.98	271.80	64
400:27	85.68	2171.13	753.92	483.07	28	695	239.31	154.11	100

Table 2.4: Iterated local search for large SMTTP instances. The algorithm is run 100 times per instance with a bound of 1 hour. Given are, out of 100 tests, the shortest run time (t_{min}), the longest run time (t_{max}), the average run time (t_{avg}) to solve the instance,and the run times' standard deviation (σ_t). In addition these statistics are given for the number of iterations, that is, i_{min} for the shortest number of iteration, i_{max} for the maximum, i_{avg} the average, and σ_t the standard deviation. Finally, the number of tests that resulted in a globally optimal solution (n_{opt}) is given.

	t_{min}	t_{max}	t_{avg}	t_{stddev}	i_{min}	i_{max}	i_{avg}	i_{stddev}	n_{opt}
150:21	8.19	72.60	23.46	12.37	1	12	3.50	2.22	100
150:22	4.30	74.58	17.87	14.98	0	13	2.40	2.70	100
150:23	111.19	3204.78	1203.21	890.85	20	587	221.52	163.82	91
150:24	4.30	10.67	9.98	0.63	0	1	0.99	0.10	100
150:25	9.11	3569.71	321.37	449.04	1	699	62.63	88.35	100
150:26	3.93	9.18	4.29	0.50	0	1	0.01	0.10	100
150:27	9.34	366.17	91.24	73.03	1	68	16.25	13.53	100
150:28	4.68	207.60	44.19	44.53	0	38	7.07	8.07	100
150:29	7.81	24.54	11.85	2.43	1	3	1.13	0.37	100
150:30	9.98	106.59	26.34	19.83	1	19	3.99	3.69	100
200:21	47.24	2244.82	451.67	387.60	3	174	34.26	30.08	99
200:22	9.37	113.63	31.36	18.53	0	9	1.87	1.59	100
200:23	52.68	2347.87	660.12	411.47	3	169	45.97	29.31	100
200:24	69.53	2867.66	817.37	544.39	4	197	55.53	37.69	100
200:25	21.16	128.42	39.07	22.39	1	9	2.40	1.84	100
200:26	27.08	2800.46	832.91	591.76	1	199	58.42	42.26	100
300:21	82.53	3392.89	1038.29	827.49	1	74	22.03	18.14	99

Table 2.5: Ant colony optimization for large SMTTP instances. The algorithm is run 100 times with a bound of one hour. See Table 2.4 on the preceding page for a description of the entries.

3. The third question inquired after the quality of the ACS-SMTTP relative to other algorithms for the SMTTP. The improved ACS-SMTTP, ACS-SMTTP′ performs worse than iterated local search (see Figure 2.2(a) on page 26 and Figure 2.3 on page 27) and, at least for SMTTP instances of less than 300 jobs, exact algorithms are able to outperform approximation algorithms like ILS and ACO (see Table 2.3 on page 28). However, more investigation in this last issue is needed and there is still room for improvement of the implementations of ILS and ACO.

4. The efficiency of ACS-SMTTP was the worry of the fourth question. For an efficient PC–implementation, Bauer, Bullnheimer, Hartl and Strauss (1999b, p. 14) estimate computation times of 10 seconds on the 100–job problems. This has proven to be a correct estimate, yet, experience also shows that the instance are rather easy to solve by standard local search. One local search already outperforms all

only to the best. He ran both configurations on the 125 50–job instances. Although the deviation to the optimum was smaller when local search was applied to all solutions, he choose the employ local search only once per iteration on the ground that this last variant required less time to run and could find the global optimum for the same amount of instances. Yet, considering that the 100–job instances could on average be solved within one ACO iteration of twenty local search applications (Section 2.5.2), tests on even smaller problems cannot possibly convey anything meaningful on the application of local search in ant colony optimization. As noted, on such problems ACS-SMTTP′ with twenty ants is just some kind of multi-start local search, no wonder it will be outperformed by the more iterated ACS-SMTTP.

"leading heuristics" with which Bauer et al. (1999b, Table 2) compare ACS-SMTTP. It is not clear how important the ant's contribution is, hence one could question the value of that comparision.

5. The fifth and last question's anxiety was on the performance of ACS-SMTTP on larger problem instances. From Section 2.5.4 and Section 2.5.5 it can be concluded that ACO, and ILS scale well to larger problems. Yet, as pointed out earlier, additional research is needed to get a definite picture of ACO's scaling behavior.

2.6 Summary

The Vienna publications clearly demonstrate that ant colony optimization can be applied to the SMTTP. Yet, the much simpler iterated local search which is described in this chapter could achieve the same effectiveness in less time. Therefore one can at least doubt whether ants are useful at all for the SMTTP. On the other hand, this chapter has also shown that the proposed configuration of ACS-SMTTP is not optimal, which suggests that there is still potential for further improvements.

I'll leave the exploration of the SMTTP to future research. For now, I will concentrate on the more challenging single machine total weighted tardiness scheduling problem which is known to be much harder to solve.

3 Algorithms for the single machine total weighted tardiness scheduling problem

3.1 Introduction

This chapter presents a number of algorithms for the problem of scheduling a single machine to minimize total weighted tardiness. The algorithms are variants of local search, iterated local search, or ant colony optimization. They are mainly extensions of the algorithms for the total tardiness problem presented in the previous chapter.

The adoption of a meta-heuristic's components and parameter settings to the problem to be solved is indispensable for arriving at a high–performing algorithm. In particular, it is unlikely that a generic implementation of a meta-heuristic will outperform a problem–specific and carefully tuned[1] implementation of another meta-heuristic. At the same time, not all meta-heuristics are equally easy to tune. As tuning cannot last infinitely, the initial solution–quality of a generic meta-heuristic combined with its tuning potential decide for the worth of this meta-heuristic for the problem at hand. Documentation of the process that consists of modification of the generic algorithms presented in Chapter 1 is indeed the main concern here. After an explanation of the problem and an overview of algorithms applied to it (Section 3.2), Section 3.3, 3.4, and 3.5 describe the tuning of local search, iterated local search, and ant colony optimization, respectively. In Section 3.6 the resulting ant colony optimization algorithm is compared against the iterated local search algorithm and Section 3.7 concludes with a summary of the findings of this chapter.

3.2 The single machine total weighted tardiness problem

3.2.1 The problem

The single machine total weighted tardiness scheduling problem (SMWTP) is a generalization of the single machine total tardiness problem. The objective is to minimize the weighted sum of the jobs' tardiness–values, $T = \sum w_j T_j$. Recall that the tardiness T_j of a job j with completion time C_j and due date d_j is defined as $T_j = \max\{C_j - d_j, 0\}$. If all job weights are the same, then actually the SMWTP is equivalent to the single machine total tardiness problem, which was attacked in the previous chapter. The SMWTP is strongly $\mathcal{NP}$–hard (Lawler 1977, Lenstra, Rinnooy Kan and Bruckner 1977)

[1] Tuning involves the adoption of algorithmic design choices and the algorithm's parameter settings to achieve the best possible performance on the problem under consideration.

3.2.2 Related research

Abdul-Razaq, Potts and Van Wassenhove (1990) performed a computational comparison of several state–of–the–art exact algorithms for the SMWTP. Each of the available algorithms requires substantial computation time and/or memory, especially when the number of jobs is more than about 50. Consequently, computer resources are not always adequate to obtain exact solutions and therefore the problem comes within the realm of approximation algorithms.

Congram et al. (1998) report remarkable results with iterated dynasearch. It is currently the best known algorithm for the SMWTP, being superior to the simulated annealing algorithm of Matsuo, Suh and Sullivan (1987), to descent and simulated annealing algorithms of Potts and Van Wassenhove (1991), and to the algorithms developed by Crauwels, Potts and Van Wassenhove (1998) which comprise simple descent local search, simulated annealing, threshold accepting, tabu search and genetic algorithms. *Iterated dynasearch* is an iterated local search algorithm that uses a carefully designed local search algorithm for the SMWTP, called dynasearch.

Congram et al. (1998) employ a commonly used heuristic, apparent urgency, to construct the initial solutions for the iterated dynasearch algorithm. The apparent urgency (AU) rule of Morton, Rachamadugu and Vepsalainen (1984) performs quite well relative to other simple heuristics (Potts and Van Wassenhove 1991). AU is a dynamic dispatching rule that selects an unscheduled job j with the highest AU_j value to occupy the first unfilled position of the sequence, where AU_j is defined by

$$AU_j = \frac{w_j}{p_j} \exp\left(-\frac{\max\{d_j - C_j, 0\}}{k\overline{p}}\right) \tag{3.1}$$

In this expression, $\overline{p}$ is the average of the processing times of the remaining jobs, and k is a scaling parameter that is subject to tuning. AU combines the weighted shortest processing time (WSPT) dispatching rule, which is optimal when all due dates are zero, with the minimum slack (MS) heuristic, which is optimal when all due dates are sufficiently loose and spread out. If k is very large, AU_j reduces to $\frac{w_j}{p_j}$, the WSPT–value, and for small k the MS–value, $\max\{d_j - C_j, 0\}$, is predominant. In my implementation, $k = 0.5$ for TF ≤ 0.3, $k = 0.9$ for $0.3 <$ TF ≤ 0.4 and $k = 2.0$ for TF > 0.4, where TF $= 1 - \overline{d}/C_{\max}$ is the tardiness factor of the problem instance. These settings for k were also used by Congram et al. (1998) who follow Potts and Van Wassenhove (1991) in their AU implementation.

3.2.3 Problem instances

The instances I use are taken from ORLIB[2]. The sets with instances of 40 jobs, 50 jobs and 100 jobs are generated in the same manner as those of the previous chapter, that is, for each job j processing time p_j is randomly drawn according to a uniform distribution on the interval $[1, 100]$, due dates are randomly drawn according to a uniform distribution on the interval $[(1 - \text{TF} - \frac{\text{RDD}}{2}) \cdot \sum p_i, (1 - \text{TF} + \frac{\text{RDD}}{2}) \cdot \sum p_i]$, where tardiness factor (TF)

[2]They can be found at `http://mscmga.ms.ic.ac.uk/`.

and range of due dates (RDD) are parameters. There are five instances for each pair of TF and RDD from the set $\{0.2, 0.4, 0.6, 0.8, 1.0\}$. Finally, the weights, randomly drawn according to a uniform distribution of integers between 1 and 10, are added. A branch–and–bound algorithm for the SMWTP (Potts and Van Wassenhove 1985) determined the optimal solutions of the 40–job and 50–job instances. For the 100–job instances the effectiveness of the algorithms is measured against the best known solutions known to date which are given along with the instances in ORLIB.

3.3 Local search

Iterated local search (ILS) and ant colony optimization (ACO) have one component in common, local search. Local search algorithms transform solutions into better solutions by a series of small alterations. The local search neighborhood defines which alterations may be considered. A good local search configuration is a necessary condition for high performance of ILS and ACO algorithms.

The use of the dynasearch neighborhood search is the feature that distinguishes iterated dynasearch from other incarnations of iterated local search. Dynasearch takes advantage of dynamic programming techniques allowing it to search a larger neighborhood than standard descent methods, thereby enabling it to perform better. In spite of this, I will not use dynasearch as a component in my ILS and ACO implementations. Standard descent methods are easier to implement and applicable to a wider variety of combinatorial optimization problems. Moreover, the gulf with dynasearch can possibly be bridged by applying simple ideas to improve local search efficiency. Below I will discuss some of these ideas.

3.3.1 Configuring local search

In general, the configuration of local search involves finding the right solution representation, neighborhood, and search strategy (see Section 1.2). Here, the permutation representation is chosen. Interchange ($\cdots \pi_i \cdots \pi_j \cdots$) as well as left insert ($\cdots \pi_i \cdots \pi_j \cdots$) and right insert ($\cdots \pi_i \cdots \pi_j \cdots$) moves on this representation are considered. Transpose ($\cdots \pi_i \pi_{i+1} \cdots$) is not considered here, because results with this neighborhood were already discouraging for the application to the single machine total tardiness problem in the previous chapter. The neighborhoods are searched either according to the first improvement search strategy or according to the best improvement search strategy.

Search control

Sometimes it can be beneficial not to search the whole neighborhood. The two techniques that I have tested are don't look bits and a restriction on the number of moves evaluated per step. Their usage is illustrated in Algorithm 3.1 on the next page, which is a specification of the inner loop in the local search skeleton introduced in Section 1.2 (Algorithm 1.1 on page 7).

Algorithm 3.1 Local Search: Ways to shrink the neighborhood

Require: a permutation π of n jobs;
$\kappa \in \mathbb{N}$;
an array of *don't look* bits initialized to *false* which are associated with positions in π and reset to false whenever jobs on the positions are involved in a move;

```
1: for i = 0 to n do
2:   if ¬don't–look(i) then
3:     for j = i to min{n, i + κ} do
4:       evaluate–move(i, j, π)
5:     if ¬improvement then
6:       don't–look(i) ← true
7:     else
8:       don't–look(j) ← false
```

Don't look bits, first proposed by Bentley (1992) for the traveling salesman problem, are used in the outer loop of the algorithm. Each position in the permutation has a don't look bit associated to it. Initially all don't look bits are set to zero. The ith don't look bit is set to one if, for a job i which is checked in the outer loop (Line 1 in Algorithm 3.1), there exist no moves involving the corresponding position that yield improvement. If the don't look bit is one that position is not considered in the outer loop of the local search algorithm. The don't look bit is reset to zero if the job has been part of some improving move. Don't look bits brand positions that are not worth looking at, considering earlier experience. With this simple memory the local search can concentrate on more promising moves and therefore save computation time.

Positions to be considered in the inner loop (Line 3 in Algorithm 3.1) can be limited in two ways. Firstly, the inner loop candidate jobs can be limited to positions later in the permutation than the position of the outer loop job. The idea is to prevent double evaluations as jobs $1, \ldots, i$ have already been considered as outer loop candidates. Secondly, the parameter κ allows to specify the range in which the move should take place. By setting κ to a small value, the algorithm is forced to concentrate on jobs that have a close position in the permutation. In this way the potential to disturb the current solution is diminished and more energy is spend on exploring solutions near the current one.

The ideas that are illustrated in Algorithm 3.1 rarely appear in that way. Their usefullness depends, among others, on the pivoting rule: placing a bound on first improvement descent does not help much as this bound is either to low or never reached. In case of best improvement descent don't look bits often have a devastating effect on the solution quality which is not by far countered by the increase in speed.

Directing the neighborhood and simplifying evaluation

The blunt methods described in the previous section are not the only way to limit the search effort. It is often possible to devise heuristics which allow to reduce the *checkout*

Algorithm 3.2 Local search: Preconditions prior to evaluation

Require: positions i and j, $i < j$;
permutation π;
required improvement Δ;
a partition of π into *runs* of *late* and *non–late* jobs;

Let: *run* $\equiv$ partial sequence of jobs that fulfill the same criterion;
$C_j \equiv \sum_{i=1}^{j} p_{\pi_i}$;
run is *late* $\equiv \forall \pi_j \in$ *run* $(C_j > d_{\pi_j})$;
$V_j \equiv \sum_{i=1}^{j} (w_{\pi_i} \cdot \max\{C_i - d_{\pi_i}, 0\})$;
$W_j \equiv \sum_{i=0}^{j} (w_{\pi_i} \cdot U_i)$, where $U_i \equiv \begin{cases} 1 \text{ if } C_i > d_{\pi_i} \\ 0 \text{ otherwise} \end{cases}$;

time, the time needed to examine to whole neighborhood, without losing effectiveness of search. These heuristics cheaply filter out non–improving moves so that less moves have to be evaluated per checkout. The moves that still have to be evaluated could be computed naively by summation of the tardiness calculated for all separate jobs in the sequence, but, fortunately, less expensive methods have been invented for the SMWTP.

The evaluation algorithm presented here has been introduced by Congram et al. (1998) who implemented an interchange descent procedure to compare their dynasearch against. Algorithm 3.3 on the next page specifies the speedups they propose. The algorithm combines heuristics to direct the search towards promising moves with quite cunning evaluation of the moves.

Some preprocessing allows for efficient implementation of the speedups (see Algorithm 3.2). The partial sums of the processing times (denoted as C_j for the jth job in the permutation), the partial sums of the weighted tardiness values (V_j), and the partial sums of the weights for late jobs (W_j) are computed in advance. Additionally, the sequence is partioned into *runs*, subsequences of jobs that share a property of all being late or all being non–late. The run data-structure has three components: a bit indicating whether it contains late or non–late jobs, an integer for the start of the run in the permutation and an integer that gives the position of the last job belonging to the run. The use of this data-structure will become clear in a moment.

First (Line 1), the combined weighted tardiness δ is computed for an interchange move $(\cdots \pi_i \cdots\cdots \pi_j \cdots)$ that involves jobs π_i and π_j. On the basis of the value of δ it is decided whether the move is worth further evaluation. If job π_j requires more time to process than job π_i (Line 2), then δ is the best improvement one can expect from the move, as the completion time for all jobs between i and j increases by $p_{\pi_j} - p_{\pi_i}$. Consequently, also the weighted tardiness could get longer. If, on the other hand, job π_j needs less processing than job π_i (Line 9), then the best that could happen to the jobs on positions $i+1, \ldots, j-1$ is that they cease to be tardy, or, at least, that their tardiness will decrease with the time $p_{\pi_i} - p_{\pi_j}$ they can start earlier (Line 10). Only if the estimated reduction δ is better than the required improvement Δ, the effect of the move has to be computed precisely (Line 3, 10). What improvement is demanded depends on the search strategy. If Δ is zero any improvement passes the test. When

Algorithm 3.3 Local Search: Interchange evaluation

Require: preprocessing of Algorithm 3.2 on the page before
Ensure: reduction δ due to exchange of job π_i with job π_j

```
1:  δ ← V_i − V_{i−1} + V_j − V_{j−1}
2:  if p_{π_j} > p_{π_i} then
3:    if δ < Δ then
4:      for all runs within positions i + 1, ..., j − 1 do
5:        if late then
6:          δ ← δ + (p_{π_j} − p_{π_i}) · Σ_{π_k ∈ run} w_{π_k}
7:        else
8:          δ ← δ + Σ_{π_k ∈ run} (w_{π_k} · max{C_k − d_{π_k}, 0})
9:  else /* p_{π_j} ≤ p_{π_i} */
10:   if δ < min{V_j − V_i, (p_{π_i} − p_{π_j}) · (W_j − W_i)} − Δ then
11:     for all late runs within positions i + 1, ..., j − 1 do
12:       δ ← δ + Σ_{π_k ∈ run} (w_{π_k} · max{C_k − d_{π_k}, 0})
```

Algorithm 3.4 Local Search: Left insert evaluation

Require: preprocessing of Algorithm 3.2 on the page before
Ensure: reduction δ due to insertion of job π_j to position i;

```
1:  δ ← w_{π_j} · (max{C_j − d_{π_j}, 0} − max{C_{i−1} + p_{π_j} − d_{π_j}, 0})
2:  if δ < Δ then
3:    for all runs within positions i, ..., j − 1 do
4:      if late then
5:        δ ← δ + p_{π_j} · Σ_{π_k ∈ run} w_{π_k}
6:      else
7:        δ ← δ + Σ_{π_k ∈ run} (w_{π_k} · max{C_k − d_{π_k}, 0})
```

the best improvement rule is followed, the Δ is adapted during the search so that only moves which give a possibly better improvement than the best improvements found in the current checkout are taken into consideration.

The evaluation that has to be carried out if the pretesting is passed exploits a number of properties of the SMWTP. First, since start times for jobs only alter within position i to j, only those jobs have to be considered (Line 4, 11). Next, jobs that are already late and have to start later as a result of the move will have their tardiness increased proportionally to their weight (Line 5, 6). When, on the other hand, the start time is earlier, which is only possible when $p_{\pi_j} < p_{\pi_i}$, non–late jobs can be ignored (Line 11). In other cases, the increase in tardiness of the jobs still has to be computed (Line 8, 12). Finally, sequences of high quality will typically consist of a few long runs of jobs that are all late or non–late (see Figure 4.4 on page 75). With help of the run data-structure such subsequences can be evaluated at once rather than having to evaluate each job individually.

The speedup tricks for the interchange local search are also applicable for the insert

Algorithm 3.5 Local Search: Right insert evaluation

Require: preprocessing of Algorithm 3.2 on page 37
Ensure: reduction δ due to insertion of job π_i to position j;
1: $\delta \leftarrow w_{\pi_i} \cdot (\max\{C_{j-1} - p_{\pi_i} - d_{\pi_i}, 0\} - \max\{C_i - d_{\pi_i}, 0\})$
2: **if** $\delta < \min\{V_j - V_i, p_{\pi_i} \cdot (W_j - W_i)\} - \Delta$ **then**
3: **for all** *late runs* within positions $i+1, \ldots, j$ **do**
4: $\delta \leftarrow \delta - \sum_{\pi_k \in run}(w_{\pi_k} \cdot \min\{p_{\pi_k}, C_k - d_{\pi_k}\})$

algorithms. An insert move is equivalent to an interchange move that involves one empty[3] job. So, the left insert evalution procedure in Algorithm 3.4 on the preceding page is obtained by substituting p_{π_i} and $V_i - V_{i-1}$ in Algorithm 3.3 on the facing page with zero and Algorithm 3.5 describes the behavior Algorithm 3.3 would have when p_{π_j} is zero and consequently the interchange would have been reduced to a right insert move.

Combining neighborhoods

In addition to the ideas discussed in the previous section it can often be useful to combine descent neighborhoods. Solutions that represent a local optimum in one neighborhood are not necessary local optima in another neighborhood as well (Reeves 1997). Combining neighborhoods can be done in a number of different ways. Most straightforward is to evaluate all implemented neighborhood moves and then take the best one, or first run one neighborhood search until it reaches a local optimum and then use this local optimum as a starting solution for another neighborhood search. The systematic combination of different neighborhoods in local search is actually the topic of the recently introduced variable neighborhood search[4] meta-heuristic (Hansen and Mladenović 1997, Mladenović and Hansen 1997, Hansen and Mladenović 1999).

3.3.2 Computational results

This section reports on the performance of a number of local search configurations. The results are obtained on the 3 sets of 125 instances of 40 jobs, 50 jobs and 100 jobs that are available via ORLIB. The results are summarized in Tables 3.1–3.5. For each set of 125 instances the following data are given: Δ_{avg}, the average percentage deviation from the local optimum, n_{opt}, the number of instances for which a global, or best known, optimum is found, and t_{avg}, the average run time of the local search.

Initialization

Table 3.1 on the following page gives indicative results of the effectiveness of dispatching rules. It compares the earliest due date (EDD) and modified due date (MDD) dispatch-

[3] An *empty job* is a job that has no processing time and a due date beyond the worst completion time.
[4] Yes, I did say (on page 13) that variable neighborhood search is just another name for iterated local search. It really is ..., it is only that the tuning concentrates on neighborhoods rather than kick strengths and acceptance criteria.

	40 jobs			50 jobs			100 jobs		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
AU	12	26	0.0013	13	20	0.0020	21	21	0.0075
EDD	139	22	< 0.0001	155	20	< 0.0001	135	24	0.0004
MDD	66	22	< 0.0001	65	20	0.0002	62	24	0.0010
AU + LS	1.01	56	0.0070	0.84	52	0.0122	0.92	28	0.0675
EDD + LS	0.81	62	0.0174	0.75	54	0.0325	0.62	33	0.2327
MDD + LS	0.91	64	0.0124	0.91	53	0.0215	0.65	38	0.1323

Table 3.1: Comparison of the effectiveness of earliest due date (EDD), modified due date (MDD), and apparent urgency (AU) on SMWTP–instances of 40, 50, and 100 jobs. Results are presented for the initial solution as well as the solution obtained after applying a best improvement interchange local search that implements the evalution method of Algorithm 3.3 on page 38. Results are averaged over 125 instances; Δ_{avg} is the average percentage deviation from the optimum, n_{opt} the number of optima found (out of 125), t_{avg} the average CPU–time in seconds.

ing rule that were also used in the previous chapter with the apparent urgency (AU) dispatching rule introduced on page 34. The AU heuristic results in the smallest average deviation from the optimal solution. When local search is applied to this solution, it will reach a local optimum much faster than it would if started with a worse solution — like, for example given by EDD. On the other hand, with local search, the EDD and MDD heuristics appear to be slightly better: more global optima are found and a lower overall deviation from the global optima is attained. Still, in the following tests the initial solutions will be generated by the AU dispatching rule since this rule is the most used in literature and leads to the shortest run times for the local search.

Neighborhoods and pivoting rules

Local search is characterized by the way in which the problem is represented, the neighborhood structure defined on this representation and the way the neighborhood is searched. In Table 3.2 on the next page results are given for a comparison between four basic neighborhoods that are defined on the permutation–based representation for the SMWTP: interchange, left insert, right insert, and an insert[5] neighborhood that takes the best move of left insert and right insert. The neighborhoods are tested in combination with the best improvement pivoting rule as well as the first improvement pivoting rule. The local search procedures are implemented taking advantage of the SMWTP–specific speedups described in Section 3.3.1.

A first observation is that small differences in neighborhood definition can have a

[5]The first improvement insert local search checks the neighborhood like in Algorithm 3.1 on page 36 but without don't–look bits and with $\kappa \to \infty$. The method 'evaluate–move(i, j, π)' (Line 4) evaluates both a left insertion of job π_j on position i and the right insertion of job π_i on position j. The move that is most improving, is at all, is subsequently carried out. If no improvement is reported by 'evalutate–move(i, j, π)', the neighborhood scan is continued. When the best improvement pivoting rule is used, the local search move is, as one would expect, the best out of the left and right insert neighborhoods.

	interchange			left insert			right insert			insert		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
40	1.07	57	0.005	9.43	48	0.01	1.61	58	0.004	0.62	70	0.02
50	1.00	48	0.008	10.34	28	0.02	3.08	34	0.009	1.35	49	0.05
100	0.90	29	0.057	13.49	24	0.27	3.40	28	0.139	0.68	38	1.60

(a) first improve descent

	interchange			left insert			right insert			insert		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
40	1.01	56	0.004	9.24	53	0.01	1.36	59	0.003	0.56	85	0.01
50	0.84	52	0.008	10.22	34	0.02	2.15	42	0.007	1.03	63	0.02
100	0.92	28	0.040	13.19	26	0.22	1.81	29	0.051	0.56	42	0.26

(b) best improve descent

Table 3.2: Comparison of neighborhoods' and pivoting rules' effectiveness for the SMWTP. The local search procedures are applied to a solution generated by AU. Results are given for the 40–job, 50–job and 100–job instances. Results are averaged over 125 instances; Δ_{avg} is the average percentage deviation from the optimum, n_{opt} the number of optima found (out of 125), t_{avg} the average CPU–time in seconds.

large impact. Left insert can only slightly improve on the initial solution and needs a large amount of computation time to do so. Right insert can obtain a much reduced average deviation from the global optimum in less time. The best local search in terms of solution quality, insert, evaluates both left and right insert moves and then chooses the best of these.

The combined left+right insert local search is the most effective. Unfortunately, it is the most time–consuming as well. The average CPU time it spends before halting at a local optimum is more than the sum of the times required by just left or right insert local search. There is a number of explanations for this phenomenon. First of all, its neighborhood is effectively twice the size of the left or right insert neighborhood and to check such a larger neighborhood simply requires more time. Secondly, being in possession of a richer set of operators, the local search is less prone to get stuck in a poor local optimum and is therefore able to make more moves. Finally, in a less benign scenario, a big neighborhood allows for long detours. Especially when the first, not necessarily steepest, descending move is followed, the richer set of operators offers a bigger opportunity to make detours. This might explain the average of 1.60 seconds required by insert local search on the 100–job instances (Table 3.2(a)) to reach a local optimum.

For all the neighborhoods that are considered, it is better to use the best improvement pivoting rule (Table 3.2(b)) than the first improvement pivoting rule (Table 3.2(a)). Best improvement local search gives better solution quality and is faster.

	40 jobs			50 jobs			100 jobs		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
full search	1.07	57	0.005	1.00	48	0.008	0.90	29	0.057
don't look bit	1.43	49	0.002	1.39	37	0.003	1.18	27	0.017

Table 3.3: Search control in first improvement interchange local search applied to a solution generated by AU. See Table 3.2 on the page before for a description of the entries.

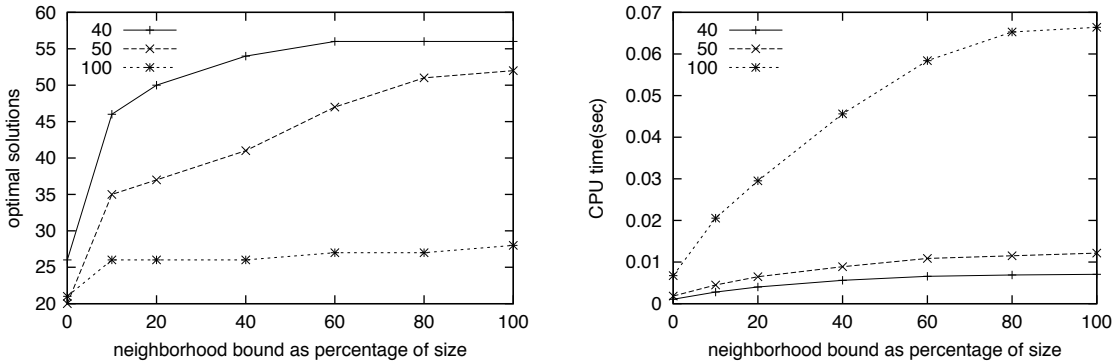


Figure 3.1: Shrinking the neighborhood: Best improvement interchange with the search bound to $\kappa\% \cdot n$. On the left side the number of optimal solutions is plotted against the neighborhood size, on the right the average CPU time is given. Results are obtained from the 3×125 instances of 40, 50 and 100 jobs.

Search control

In general, a full search of the neighborhood is likely to yield a better solution quality than a partially searched neighborhood. It is still interesting to investigate into the effect of narrowing down the search space. Especially when the local search is going to be applied many times, as is the case in the ILS and ACO algorithms that will be considered in Section 3.4 and Section 3.5 respectively, one might prefer a local search that is not necessarily the most effective after one run, but requires much less computational effort and can therefore be run more often.

Table 3.3 summarizes the effectiveness of a local search algorithm that puts the idea of don't look bits into practice. A look at this table reveals that don't look bits enable the algorithm to become about a factor 2 – 3 faster without effecting the efficacy too much. Unfortunately don't look bits are hard to incorporate successfully into local search with a non–first improvement pivoting rule or local search that operates on the insert neighborhood. I have tried it, but results were very bad.

That it can be advantageous to exchange checkout time for solution quality is also illustrated in Figure 3.1. Here a best improvement pivoting rule is used on an interchange neighborhood, which is not entirely checked, but restricted by parameter κ (see Algorithm 3.1 on page 36). There is a clear tradeoff between solution quality and computational effort: when κ is bound to a small percentage of the problem size, the local search is performed much faster, but less optimal solutions are found as well. The ideal

	first improve			best improve		
	L	*R*	*L&R*	*L*	*R*	*L&R*
40	3.3	7.9	4.3	4.0	6.4	4.8
50	3.3	9.4	4.8	3.9	7.6	5.2
100	4.5	15.1	6.0	5.3	9.7	6.5

(a) naive → cunning

	first improve			best improve		
	L	*R*	*L&R*	*L*	*R*	*L&R*
40	1.4	2.0	1.6	1.2	2.3	1.5
50	1.4	2.1	1.7	1.3	2.0	1.7
100	1.3	3.0	1.7	1.3	3.5	1.8

(b) cunning → guiding

Table 3.4: Effect of programming effort on computation time. Local search using the insert neighborhood is applied to an initial AU solution. Left insert (L) as well as right insert (R) and "best of both" ($L\&R$) are considered. Given two local search implementations α and β, $\alpha \rightarrow \beta$, each entry represents $t^{\alpha}_{avg}/t^{\beta}_{avg}$ where t_{avg} is the average CPU time(sec) over the 125 instances of n jobs. *Naive* means evaluating move by simply computing the objective function of the resulting permutation, *cunning* is Algorithm 3.4 on page 38 or Algorithm 3.5 on page 39 without the pretesting on Line 2. When this last element is added, *guiding* is obtained.

value of κ has yet to be determined when combined with the ACO or ILS meta-heuristic.

Efficient evaluation

Quite some effort has been devoted to the implementation of the speedup techniques introduced in Section 3.3.1. In the following, I give hints on their usefulness and show that the programming effort is worthwhile.

Luckily, the effort the programmer invests in constructing ingenious implementations does pay off with increased efficiency of the algorithm. In Table 3.4 the efficiency gain is given for the insert[6] neighborhood with the first improvement and the best improvement pivoting rule. The table entries represent the decrease in CPU–time averaged over the 3 sets of 125 instances of 40, 50, and 100 jobs by the introduction of the particular techniques indicated. In Table 3.4(a) the transition from a naive evaluation to a more ingenious evaluation is considered. Table 3.4(b) shows the improvement obtained by usage of heuristics to filter out non–improving moves. For both transitions the improvement obtained increases with the instance size. This is what one would expect as more moves need to be evaluated on these problem instances. The gain obtained by guiding the search is rather independent with respect to the pivoting rule that is used to control the search.

Combining neighborhoods

It will often be the case that one descent method can improve the solutions found by another descent method if the neighborhood structure explored by the descent methods

[6]For the interchange neighborhood approximately the same results are obtained. However, a major distinction between the interchange and the insert neighborhoods' results is that the more ingenious implementation has no effect on the solution quality of local search based on the insert neighborhood, whereas it does make a difference for interchange–based local search. Small differences in, for example, the search strategy, might cause the aberrant behavior of the interchange local search.

	inter. + right ins.			inter. + insert			right ins. + inter.			insert + inter.		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
40	0.52	78	0.005	0.32	90	0.008	0.46	79	0.006	0.15	96	0.013
50	0.67	64	0.008	0.37	79	0.014	0.73	62	0.010	0.31	87	0.028
100	0.72	35	0.047	0.59	46	0.102	0.66	37	0.071	0.21	49	0.273

Table 3.5: Best improvement interchange local search followed by best improvement insert local search and the other way around applied to a solution generated by AU. See Table 3.2 on page 41 for a description of the entries.

is different. Two neighborhood descent runs will, in general, also require more time than one. It depends on the way in which the local search is employed by the meta–heuristic whether the combination is worth the effort.

The results shown in Table 3.5 confirm the expectations in that combined neighborhoods result in better quality solutions, but take more time to search than single neighborhoods. The right half of the table gives results for interchange local search which is directly followed by an right insert or left+right insert local search. The interchange local search acts as an initilization mechanism that prevents costly insert moves and in addition enables the insert local search to achieve better solution quality. The computational effort required by this combined local search is actually less than the effort Table 3.2 on page 41 reports for a single insert local search.

The most effective however is a local search procedure that first searches the insert neighborhood and then performs an interchange neighborhood search on the solution it arrived at. This algorithm solves most instances in the benchmark set and comes closest to the optimal solutions of instances it cannot yet solve. Unfortunately it is the most CPU–consuming as well. Tests in the context of iterated local search and ant colony optimization will show whether this is a serious impediment.

3.3.3 Related work

It would be nice to determine if the descent methods defined in this section can compete with the dynasearch method. Alas, this it not possible. Congram et al. (1998) only report results on multi–start and iterated local search employment of the dynasearch neighborhood, and it is only in the iterated version that dynasearch is substantially better than traditional interchange descent local search.

The generic idea of *dynasearch* is to consider a neighborhood of the type used in a traditional descent local search algorithm, but to allow several moves of a certain type to be made in a single iteration. Dynamic programming is used to find the best combination of moves. More specifically, the dynasearch for the SMWTP considers an interchange neighborhood consisting of all solutions that can be obtained from a permutation π by a series of pairwise independent[7] interchange moves. This neighborhood has size $2^{n-1}-1$. Because a dynasearch interchange move is equivalent to a best series of independent swaps, dynasearch uses effectively a best improvement search strategy.

[7]Two moves that interchange job π_i with π_j and job π_k with job π_l respectively are said to be *independent* if $\max\{i,j\} \leq \min\{k,l\} \vee \min\{i,j\} \geq \max\{k,l\}$.

Iterated dynasearch owes its performance to the much larger neighborhood size it can search compared with standard neighborhood search. In the foregoing I have investigated into another method to obtain a larger neighborhood: combination of standard neighborhoods. Given the results with iterated dynasearch, an iterated combined neighborhood search has much perspective, precisely how much perspective it has will be investigated in the following.

Other ways to improve descent local search are discussed by Crauwels et al. (1998). The authors propose a binary representation for the SMWTP that encodes a wealth of knowledge about the problem and present computational results for local search methods that use this representation and methods that represent the solution as a permutation of jobs. The comparison shows that the permutation–based methods have a higher likelihood of generating an optimal solution, but are less robust in that some poor solutions are obtained. Based on these results the permutation representation is to be preferred for the ILS and ACO algorithms. The algorithms will perform a series of local search applications and the fact that some of the applications might result in poor solutions forms not the least hindrance. Besides the permutation representation is much simpler. It is also for simplicity sake that the ILS and ACO algorithms will contain 'just' descent local search and not some simulated annealing or tabu search algorithm proposed in literature.

3.4 Iterated local search

In this chapter, ant colony optimization (ACO) is compared against iterated local search (ILS). There already exists a very successful ILS algorithm for the SMWTP, iterated dynasearch of Congram et al. (1998). The ACO algorithm is not directly compared against this algorithm since it was not made publicly available by the authors and its re-implementation is not trivial. Besides, the comparison with a ILS algorithm which is implemented by myself is more fair in the sense that the same programming skills and data-structures used for the algorithms' implementation.

3.4.1 Configuring iterated local search

The three main components of ILS are local search, the modification mechanism, and the acceptance criterion (see Algorithm 1.3 on page 12). Section 3.3 already discussed the local search, here the other two components are examined.

Solution modification

The purpose of the solution modification is to alter the current solution in such a way that the subsequent neighborhood search can find a new local optimum. One simple possibility for solution modification is to apply a number of random neighborhood moves to the current solution. Interchange moves as well as insert moves are considered. Intuitively, the kick–move should operate on another neighborhood than the one used in the local search to prevent that the kicks are immediately reversed and the local search

halts at the old local optimum again. For the same reason, the kick should be sufficiently large to move to a solution outside the neighborhood of the local optimum. On the other hand, the kick should not be too large, or else the good characteristics of the previous local optimum are lost, and the procedure is then effectively close to a multi–start algorithm which is known to perform bad. The number of kicks need not necessarily be constant. One possibility is to vary the kick–strength in a certain range. If an iteration does not result in a better local optimum, the number of kicks is increased. If a better local optimum is found, the kick–strength is reset to a minimum value. This last idea was first proposed in the context of the variable neighborhood search meta-heuristic by Mladenović and Hansen (1997).

Acceptance criterion

The acceptance criterion determines to which solution the next modification is applied. How exactly the acceptance criterion is specified could affect the performance of iterated local search. Congram et al. (1998) propose some sort of backtracking mechanism, Martin et al. (1991) adopt a simulated annealing acceptance criterion, and Johnson (1990) suggests to accept only solutions that have a better objective value than the current solution. This last and most simple choice is actually applied in many other ILS applications (Hansen and Mladenović 1999, Johnson and McGeoch 1997, Martin and Otto 1996), and will also be applied in this ILS application to the SMWTP.

3.4.2 Computational results

The results discussed in this section are obtained on the 125 100–job instances of ORLIB. The 40–job and 50–job instances are not considered since they were already almost solved by local search alone (see Section 3.3.2). Much of the testing is performed on 10 instances that proved to be relatively hard on preliminary tests. The discussion of the results subsequently focuses on 2–4 of these instances, in Appendix A summary results for all 10 instances can be found.

Initialization

In general, I found that for the SMWTP the construction heuristic which is used to initialize the iterated local search does not matter too much: In preliminary tests starting with an AU–solution or an MDD–solution did not show any difference when averaged over the 125 100-job instances. The run time distributions of the algorithm indicated, however, that for some instances MDD–initialization causes shorter run–times whereas for other instances AU should be used. Therefore the ILS will be initialized as follows: run a local search on the solution constructed by MDD; subsequently run a local search on the solution constructed by AU and take the better of these two solutions as starting solution.

	full search			don't look bit		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
i	0.11	49	6.56	0.07	62	5.77
l	0.09	62	5.78	0.09	65	5.42
r	0.19	50	6.37	0.26	43	6.89

(a) iterated first improve interchange

	full search			$\kappa = 50$ bound		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
i	0.11	50	6.41	0.10	55	6.17
l	0.01	98	3.20	0.76	81	4.43
r	0.19	50	6.37	0.20	49	6.62

(b) iterated best improve interchange

	left insert			right insert			insert		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
i	7.27	27	8.35	0.31	43	7.06	0.04	64	5.75
l	10.62	26	8.38	0.05	62	5.69	0.08	56	5.95
r	0.18	55	6.88	0.46	42	6.97	0.09	60	5.99

(c) iterated best improve insert

	insert+interch.		
	Δ_{avg}	n_{opt}	t_{avg}
i	0.073	69	5.41
l	0.017	81	4.62
r	0.041	69	5.73

(d) iterated combined

	full search			don't look bit		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
i	0.021	80	4.5	0.008	89	3.8
l	0.007	92	3.8	0.009	93	3.6
r	0.018	81	4.6	0.001	95	3.5

(e) iterated first improve interchange + best improve insert

	full search			$\kappa = 50$ bound		
	Δ_{avg}	n_{opt}	t_{avg}	Δ_{avg}	n_{opt}	t_{avg}
i	0.016	87	3.8	0.015	89	3.9
l	0.001	115	2.2	0.007	114	2.4
r	0.011	87	4.5	0.001	98	3.5

(f) iterated best improve interchange + insert

Table 3.6: Effectiveness of local search configurations for ILS on the SMWTP. Results are obtained by testing a local search configuration once on each of the 125 100–job instances with a CPU bound of 10 seconds. Three types of kicks are tried: i for the interchange kick, l for the left insert kick and r for the right insert kick. Solution modification consists of applying a kick 6 times. Results are obtained by averaging over the 125 instances. Δ_{avg} is the average deviation from the optimum, n_{opt} is the number of instances solved to the optimum, t_{avg} is the average CPU time in seconds.

Local search

Table 3.6 on the page before gives the average deviation, number of optima found, and average CPU–time for ILS that is run once on each of the 125 instances for a time limit of 10 seconds. A number of local search configurations are tried in combination with three possibilities to modify the local optima produced by the local search: the first method, denoted by i, is to apply six random interchange moves. In the second method, denoted by l, six random left insert moves are applied. The third method, denoted by r, employs right insert moves. In general, method l seems to be the best way to modify solutions since it leads to the least deviation from the best known solutions and computation time combined with the highest number of best known solutions found.

Neighborhoods and pivoting rules Table 3.6(a) compares two versions of iterated first improvement interchange local search. The full search on the left is the standard version, the version on the right uses the don't look bits introduced on page 35. From the table it is not entirely clear whether the don't look bits should be used. It may yield better results in combination with the interchange and almost the same with left insert kicks, but worse performance is obtained in combination with right insert kicks. However, this balance might easily change again if other elements, like the kick–strength, are modified.

More or less the same holds for Table 3.6(b). In this table a standard iterated version of best improvement interchange local search is compared with a version that puts a bound on the neighborhood with use of parameter κ. An explanation on the meaning and usage of κ can be found on page 36. The value of 50 for κ is based on preliminary tests with a series of values for κ with modification method s. There, $\kappa = 50$ came out best. Especially in combination with left insert kicks, the results of best improvement local search in this table are better than those obtained with first improvement local search.

In Table 3.6(c), the insert neighborhoods are compared. The differences between these neighborhoods, which were so manifest after a single run (Table 3.2(b) on page 41), are much less clear for the iterated version. Left insert is still performing badly, in general, and the combination of left and right insert is still the most effective. However, right insert local search outperforms the combined left+right insert when left insert kicks are used to modify the solution and if right insert kicks are used, left insert local search is not that bad at all. Yet, the performance of the ILS variants is not really surprising: it is in line with the configuration guidelines that state that the solution modification should preferably not be done on the same neighborhood as the local search[8].

Combining neighborhoods The combination of interchange and insert is investigated in Table 3.6(d), 3.6(e), and 3.6(f). For a single–run of local search the most important effect of combining neighborhoods was that the faster interchange neighborhood provided

[8] Still, one might wonder why the combination of left and right insert neighborhoods works so well. The left insert operator is the inverse of the right insert operator. Kicks by one operator can therefore be easily reversed by search with the other operator, it seems. Apparently, this is not the case. Probably because more than one kick is performed for a solution modification.

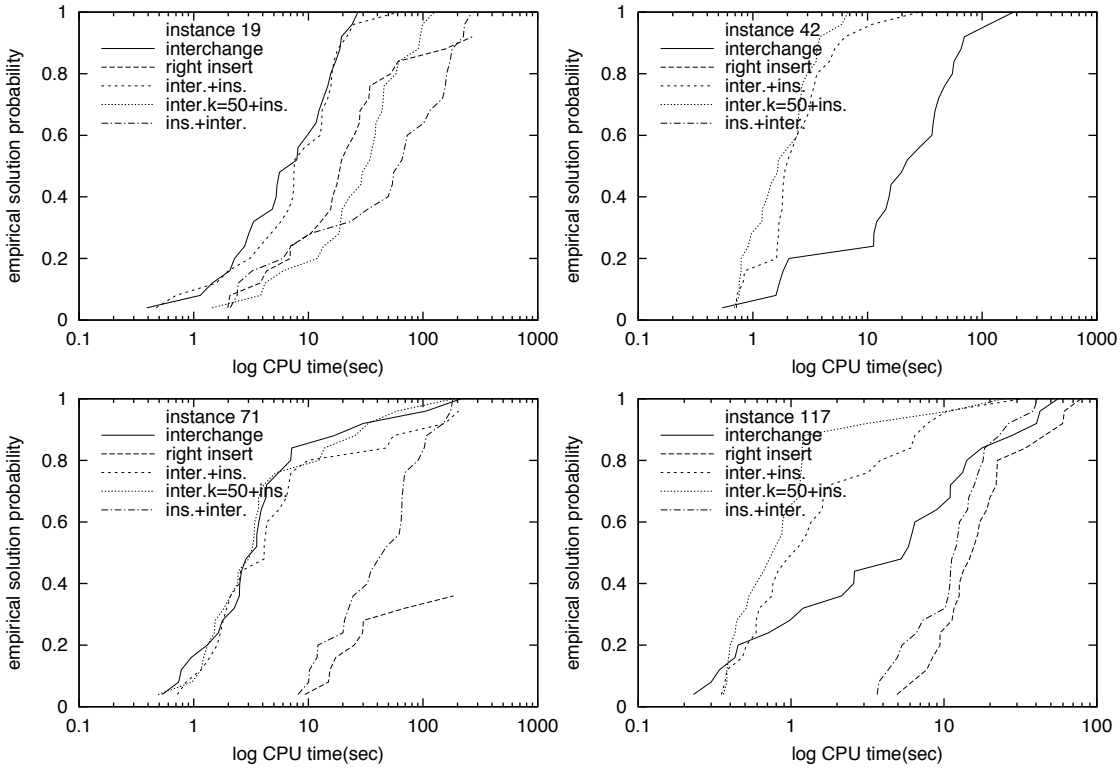


Figure 3.2: Local search for ILS. Solutions are modified with 6 left insert kicks. The run time distributions are based on 25 tries with a CPU time bound of 300 seconds.

an intelligent initialization for the more costly insert neighborhoods thus allowing to save computational effort (Table 3.5 on page 44). The biggest gain due to intelligent initialization is made in the first application of the local search. Yet, the positive effect of combining neighborhoods also occurs in Table 3.6 on page 47. So, apparently the run time of the iterated local search are sped up as well as a result of this neighborhood combination.

The best way to combine neighborhoods is *not* by starting with insert local search (Table 3.6(d)). Still, if one focuses on the number of optima obtained and the effort required, the iterated employment of this local search is not worse than the employment of its constituent local searches — except for the combination of full iterated best improvement interchange local search with left insert kicks (Table 3.6(b) on page 47). The reduction of the checkout of the interchange neighborhood by means of don't look bits (Table 3.6(e)) or the κ parameter (Table 3.6(f)) has, in general, a positive effect on the performance of the combined local search except if the left insert kicks are used. The best performance is achieved with an iterated local search algorithm that combines standard best improvement interchange local search followed by best improvement insert local search with six left insert kicks to modify the solution.

Run time distributions Table 3.6 on page 47 helps selecting the right local search for the ILS algorithm. Yet, it is a rather coarse filter. It provides no information about the algorithm's performance on individual instances and, even worse, it does not reflect the variation in the algorithm's behavior which is bound to occur due to the randomness built in into the modification mechanism of ILS. Experience in the previous chapter already indicated that this global measure is not sufficient to base gross conclusions upon.

In order to acquire a better picture of the performance of the iterated local search configurations, I have analyzed the run time distributions of these algorithms for 10 100–job SMWTP instances which were chosen because they showed to be relatively hard in preliminary tests. Instances are numbered consecutively and the instance with number i is the ith of the 125 available. An introduction into run time distributions is given in Section 1.5.2. Here, an estimation of the distributions is made after running the algorithm configuration twenty five times with a computation bound of five minutes.

Figure 3.2 on the page before gives the run time distributions for ILS algorithms that use interchange, right insert, insert preceded by interchange, insert preceded by restricted interchange, or interchange preceded by insert. All local searches follow the best improvement pivoting rule in their neighborhood check. None of the variants performs best on all the instances. That is, the left most curve in the plots does not belong to one and the same local search algorithm used by ILS. In the plots of instance 19 and 71 in the left half of Figure 3.2 on the preceding page, the curve represents the run time distribution of the interchange variant, in the other plots it is the variant that applies insert local search after it has applied interchange local search which requires the least effort to find an optimal solution.

Note also the huge variation in run times required by the algorithm. In 'lucky' runs, it can find the optimal solutions for an instance within one second, but sometimes it might take more than a minute on the same instance before finding the optimal solution.

Still, the big picture that one can derive from the run time distributions on the ten instances confirms what the data of Table 3.6 on page 47 already suggested: the local search that starts with interchange and then proceeds with insert is the best choice. Alternative search control, like limiting evaluation to κ moves, yields no clear advantage and therefore the insert preceded by interchange local searches will check their neighborhoods in the standard best improvement fashion in the following. Besides, without κ, the algorithm has a greater simplicity.

Solution modification

The investigation into solution modification concerns both the type of random moves and the intensity of the perturbation.

Kick neighborhood The influence of the neighborhood structure which the kick–moves use is investigated on the 10 instances as well. Figure 3.3 on the facing page gives the run time distributions for two of the 10 instances. As can be observed, the quality of the kick–move operator depends not only on the local search neighborhood but on

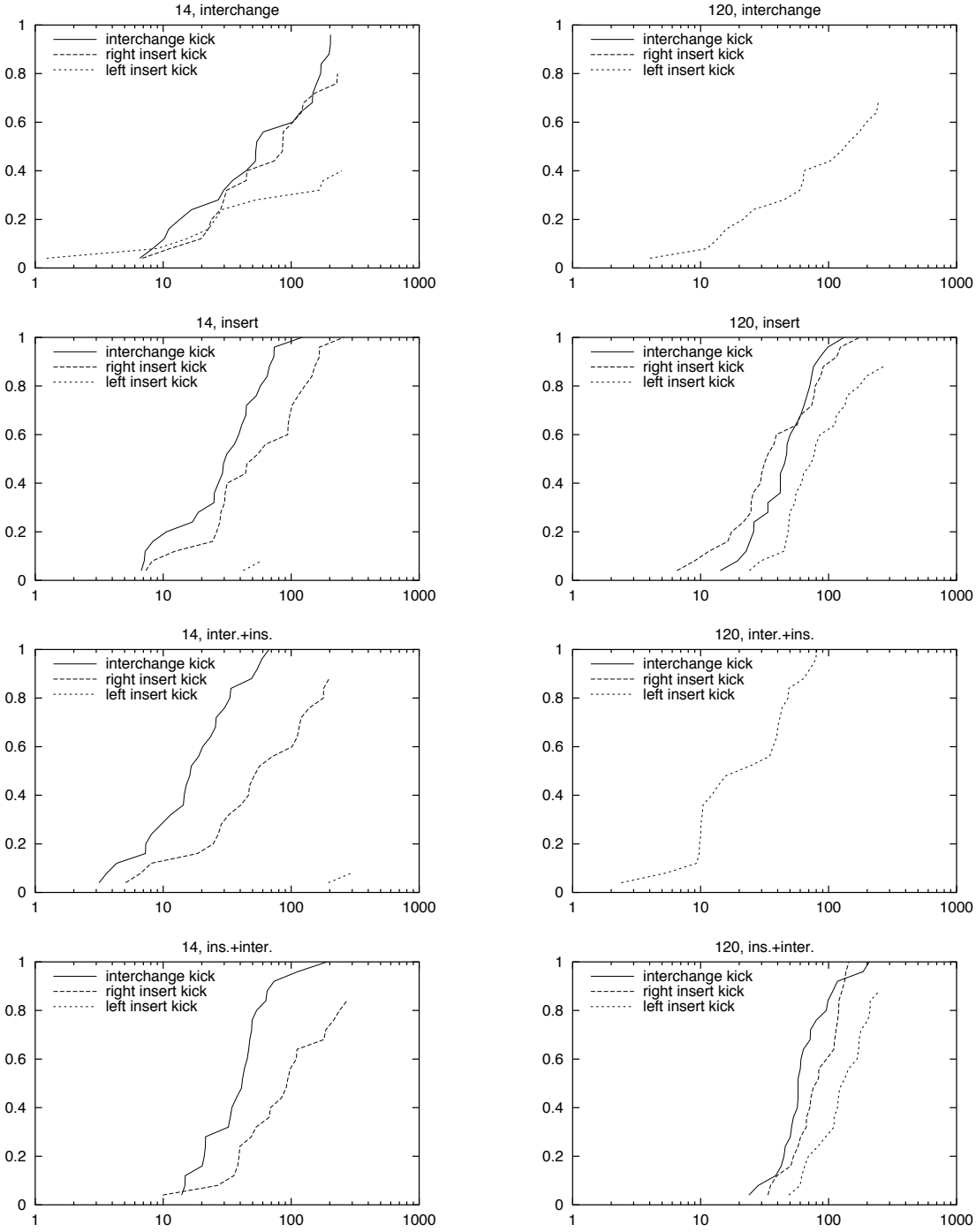


Figure 3.3: Neighborhoods for local search and solution modification. The empirical solution probability (y-axis) against the log CPU time in seconds (x-axis). For instance 14 and 120 the performance of iterated *interchange*, *(left&right) insert*, *interchange+insert* and *insert+interchange* local search modified by 6 *interchange*, *left insert*, or *right insert* kicks is given. The run time distributions are based on 25 runs with a bound of five minutes.

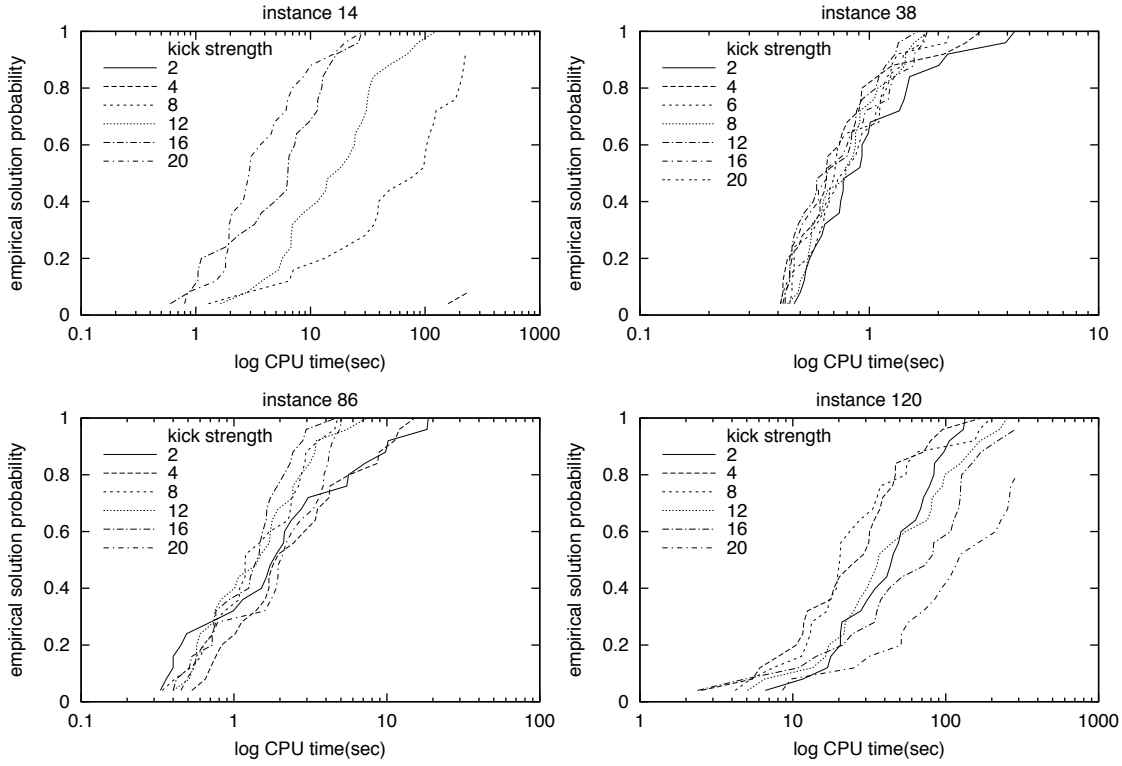


Figure 3.4: Kick strength for ILS on the SMWTP: run time distributions for 25 tries with a bound of 300 seconds. The local search used is best improvement interchange + insert. The kick is a random left insert move.

particular instances as well. On instance 14 an interchange kick (depicted by a solid line) consistently gives the best performance. Yet, on instance 120 interchange and right insert kicks lead to the entrapment of the algorithm in suboptimal local optima when the local search is (first) done on an interchange neighborhood. The run time distributions do not unequivocaly single out one of the neighborhoods as most suited modification mechanism. However, the results of Table 3.6 on page 47 clearly indicate that in general the best performance is achieved if random left insert kicks are used. So, I will stick to that in the following.

Kick–strength In Section 3.4.1 it was noted that, in addition to the neighborhood the kick operates upon, also the number of kicks applied in the modification procedure, the *kick–strength*, is a crucial factor in the design of iterated local search. Figure 3.4 gives for four instances the run time distributions for iterated local search variants that use 2, 4, 8, 12, 16, or 20 random left insert moves to modify solutions found by an interchange local search succeeded by an insert local search. The figure shows that though for some SMWTP instances the strength of the kick has only little effect on the algorithm's performance, on other instances the number of kicks applied does make a

big difference. For example, the interchange+insert local search performed very badly on the $14th$ instance when solutions were modified by six random left insert kicks (see Figure 3.3 on page 51), but it is able to find the best known solution very fast when 16 or 20 kicks are applied.

The run time distributions on instance 120 neatly illustrate the maxim that the number of kicks should not be too large and not too small either. A modification consisting of six random kicks strikes the golden mean and applying only two kicks or as many as twenty kicks leads to bad performance. Yet, on instance 38 the kick–strength has virtually no effect. This is probably caused by the ease of the instance: it can on average be solved within 5 iterations while on instance 120 about 500 iteration are required.

Since there is no kick–strength that is optimal for each of the instances, the final ILS will vary the kick strength between 4 and 20 random kicks. In the beginning only four kicks are applied. If an ILS iteration does not result in the better solution the number of kicks is incremented with one. Incrementing stops if a maximum of twenty kicks is reached and the kick strength is reset to the minimum of four if a better solution is found.

Acceptance criterion

The acceptance criterion used until now stated that the starting solution for the modify–search–accept cycle is the best solution currently known to the algorithm. Yet, also other acceptance criteria could be imagined which accept solutions of equal quality or, with a certain probability, solutions of worse quality. I have tried different such acceptance criteria in some preliminary tests. However, the increase of search space exploration achieved by making the acceptance criterion more tolerant seemed not to improve the ILS algorithm and, hence, I stay with the standard acceptance criterion.

3.4.3 Comparison with iterated dynasearch

Congram et al. (1998) were the first to develop an ILS algorithm for the SMWTP. Their iterated local search has the following features:

- Three local search variants were considered: of course dynasearch and, for comparison, best improvement interchange descent and first improvement interchange descent.
- The solution modification is done by application of six interchange kicks. Kicks based on random exchanges of three jobs were also tried. A modification comprising two 3–exchanges proved a reasonable alternative to six random interchanges, but was nevertheless not employed in the final configuration.
- Every new solution, improving or not, is accepted to be the current solution, except for every five iterations when backtracking occurs and the current solution is set to the best solution found thus far.

- For the first 100 iterations, a series of transposes of adjacent non–late jobs that are not in EDD order is performed, starting at the beginning of the sequence. After the first 100 iterations, a similar procedure transposes adjacent non–late jobs, either if they are not currently in EDD order, or with probability 1/3 if they are currently in EDD order and their interchange would not cause either one to become late. All transposes are performed prior to the kick. Their purpose is to reduce the computation time for the dynasearch descents in the beginning and to diversify the search in the end.

In the light of the experience in Section 3.4.2 some of these features arouse wonder. For example, the transpose of adjacent jobs according to EDD criteria looks sophisticated, but what is exactly its contribution to the performance of the algorithm and why is it only used in combination with the dynasearch neighborhood? And what is the use of the backtracking mechanism? It allows for more exploration of the search space, but is that really needed here? As for the modification mechanism, probably better performance could be achieved with a variable number of left insert kicks, the mechanism proposed in Section 3.4.2.

It might well be that dynasearch becomes redundant in the light of the combination of interchange and insert neighborhoods which showed so successful in the previous section. iterated dynasearch was outperformed iterated best improvement interchange descent, but the interchange neighborhood is not the strongest standard neighborhood available and it can be doubted whether the ILS configuration it was embedded in gets the most out of this neighborhood.

3.5 Ant colony optimization

In the foregoing, we saw how local search can be configured and how it can be employed in the framework of iterated local search. The hypothesis of this thesis is that ants have more sophistication to guide the local search than this framework can offer. The ultimate test of this hypothesis is described in Section 3.6. First, the ant colony optimization (ACO) algorithm has to be tuned. This process is described below.

3.5.1 Configuring ant colony optimization

The tuning process starts with the improved ant colony system for the single machine total tardiness problem (SMTTP) (ACS-SMTTP/) which is described in Chapter 2. Though this algorithm is configured for the SMTTP, it can be applied to the SMWTP virtually without changes. The main changes concern the different objective function, different heuristic and different local search.

The parameters to be tuned in ant colony system (ACS) are α, β, q_0 and the population size (see Section 1.3.2). The performance of ACS and ant colony optimization, in general, is known to be rather robust with respect to the exact settings of these parameters. α is usually set to one, β is set to two, q_0 should normally be high and ρ low (Dorigo and Gambardella 1997b, Gambardella and Dorigo 1997). Together with the size of the

colony these parameters determine the ratio of exploitation and exploration. There is a number of other ways one can try to improve the ACO–algorithm. Naturally, the choice of the local search and the meaning of the pheromone trails matters. Supplementary, the modifications presented below can be considered.

Directing the population

The population of ants is directed by the pheromone on the edges of the graph and by a heuristic value provided by a dispatching rule. More specifically, in ACS for the SMWTP jobs are scheduled using the following action choice rule.

- With probability q_0, the unscheduled job j that maximizes $\tau_{ij}(t)^{\alpha} \cdot {\eta_{ij}}^{\beta}$ is put in the current sequence at position i. Here, $\tau_{ij}(t)$ is the pheromone trail associated to the assignment of job j to position i, t indicates the dependence of the pheromone trail on the time, η_{ij} is the heuristic desirability of assigning job j to position i, and α and β determine the relative influence of pheromone trail and heuristic information, respectively.
- With probabiliy $1 - q_0$, job j is chosen randomly with a probability given by

$$p_{ij}^{k}(t) = \frac{\tau_{ij}(t)^{\alpha} \cdot {\eta_{ij}}^{\beta}}{\sum_{l \in \mathcal{N}_i^k} \tau_{il}(t)^{\alpha} \cdot {\eta_{il}}^{\beta}} \qquad \text{if } j \in \mathcal{N}_i^k \tag{3.2}$$

  $\mathcal{N}_i^k$ is the set of jobs which ant k has not yet scheduled.

Because only the pheromone trail of edges of the global-best solution the ants have found so far is updated, it is likely that only few edges receive pheromone and that therefore differentiation among edges is difficult. For a long time many edges will not collect pheromone at all. The heuristic information provides then the only means to differentiate among these edges, while perhaps other factors, like neighboring an edge with a strong pheromone intensity, might make further differentiation possible. Below I will discuss a number of ideas whose implementation should further help the ants to pick the right edges.

Candidate list The candidate list is a trick that makes the pseudo–random–proportional action choice rule more greedy. It is a standard speedup technique in algorithms for the traveling salesman problem (Martin et al. 1991, Reinelt 1994, Johnson and McGeoch 1997). Usually, with a probability $1 - q_0$ all jobs not already scheduled are considered to be added to the partial solution. With the candidate list instead an 'elite' of the most promising jobs is chosen first. Jobs not belonging to this elite are excluded from selection by the ants. In the implementation used here, the candidate list is formed by the first c jobs that belong to the best solution found so far and are not already scheduled. By restricting the set of jobs to the most promising ones, it can be expected that, especially for large SMWTP instances, the ants are able to concentrate their search faster on good solutions. The candidate list is used in a number of ACO algorithms to achieve better performance (e.g. Stützle 1998a).

Pheromone diffusion There is a high likelyhood that jobs which are, based on current information, best scheduled on position i would also do well on position $i+1$ or $i-1$. Pheromone for putting a job on position i should then also have a positive effect on the choice of putting a job on a neighboring position j. There are a number of ways to achieve this effect, here I consider two of them: changing the pheromone trail update and changing the action choice rule.

Equation 3.3 is an alternative for the update rule of Equation 1.2 on page 10. It causes the pheromone to diffuse over the graph because the neighbored positions get some pheromone in the update.

$$\tau_{i+k,j}(t+1) = (1-\rho) \cdot \tau_{i+k,j}(t) + \rho \cdot \Delta\tau_{ij}^{gb}/2^k \tag{3.3}$$

In this equation $\Delta\tau_{ij}^{gb}$ is the amount of pheromone that is added. Hence, if job j is put on position i in the best solution found so far, neighboring nodes get exponentially less pheromone.

In Equation 3.4 des_{ij} represents the desirability to put job j on position i. des_{ij} also takes account of the desirability to put jobs at near by positions when choosing a job for position i.

$$des_{ij} = \sum_{k=0}^{k=n} \frac{\tau_{i+k,j}(t)^{\alpha} \cdot {\eta_{i+k,j}}^{\beta}}{2^k} \tag{3.4}$$

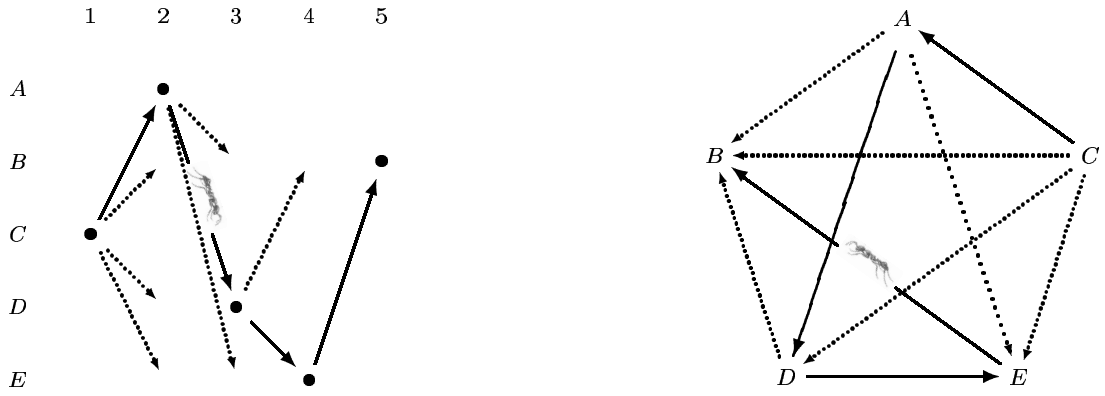
When $\tau_{ij}(t)^{\alpha} \cdot {\eta_{ij}}^{\beta}$ is substituted by des_{ij} in the action choice rule described on the page before, then jobs will be more likely to be put on positions near to their best known position.

Problem representation

ACS-SMTTP uses a graph representation where nodes correspond to jobs to be put on a position in the solution sequence. This is the *absolute* representation illustrated in Figure 3.5(a). The pheromone value τ_{ij} is interpreted as the desirability to assign job j to position i. Such a representation is not a bad choice since often the absolute position of a job is particularly important in scheduling applications. In contrast, ACO applications to the TSP use a *relative* representation (Figure 3.5(b)). That is, each node is associated to a city and a path through the graph directly maps to a tour from city to city. Perhaps such a representation would fit the SMWTP better than the absolute representation. This would indicate that predecessor–successor relationship of jobs is more important for the minimization of the total weighted tardiness than their position in the permutation.

Heuristic information

The heuristic information η_{ij} used by the ants is based on either the earliest due date (EDD), the modified due date (MDD), or the apparent urgency (AU) dispatching rule. η_{ij} will be $\eta_{ij} = 1/d_j$ or $\eta_{ij} = 1/\max\{C_j + p_j, d_j\}$ or $\eta_{ij} = 1/((w_j/p_j) \cdot \exp(-\max\{d_j - C_j, 0\}/k\overline{p}))$, respectively. It specifies the desirability to put a job j with processing time



(a) absolute representation

(b) relative representation

Figure 3.5: ACO's graph representation for SMWTP. Ants' solution (C, A, D, E, B) is a path connecting all jobs. A node in the left graph associates a job to a certain position, while a node in the right graph represents the relative position of jobs in a predecessor–successor relationship. Arrows are the edges the ant might choose during the construction of the solution — solid arrows are chosen.

p_j, due date d_j and weight w_j on position i in the schedule. C_j is the job's completion time, $\overline{p}$ the average processing time of the remaining jobs, and k is a scaling parameter which is set as proposed on page 34. I have also considered modifications of EDD and MDD that take account of the job's weights. However, the modified heuristic information resulted in dissappointingly bad performance in preliminary tests, so I have not pursued this direction any further.

Other ACO–variants

ACS is by no means the only incarnation of ant colony optimization which could work on the SMWTP. $\mathcal{MAX}$–$\mathcal{MIN}$ ant system ($\mathcal{MM}$AS) is the other ACO algorithm that performed well on the traveling salesman problem. It could also be effective for the SMWTP.

In fact, I have implemented the $\mathcal{MM}$AS and tested it both with and without use of ACS's greedy action choice rule (p. 10). In both cases $\mathcal{MM}$AS could not outperform the ACS implementation. This is not surprising: $\mathcal{MM}$AS tends to be better than ACS on problems that require relatively much exploration of the search space, yet, as we have seen and will see, for most SMWTP instances fully exploiting the current best solution is enough, so further exploration is not needed.

3.5.2 Computational results

Like for ILS, the results presented here are based on the run time distributions for algorithm configurations on 10 relatively hard SMWTP instances. Each configuration

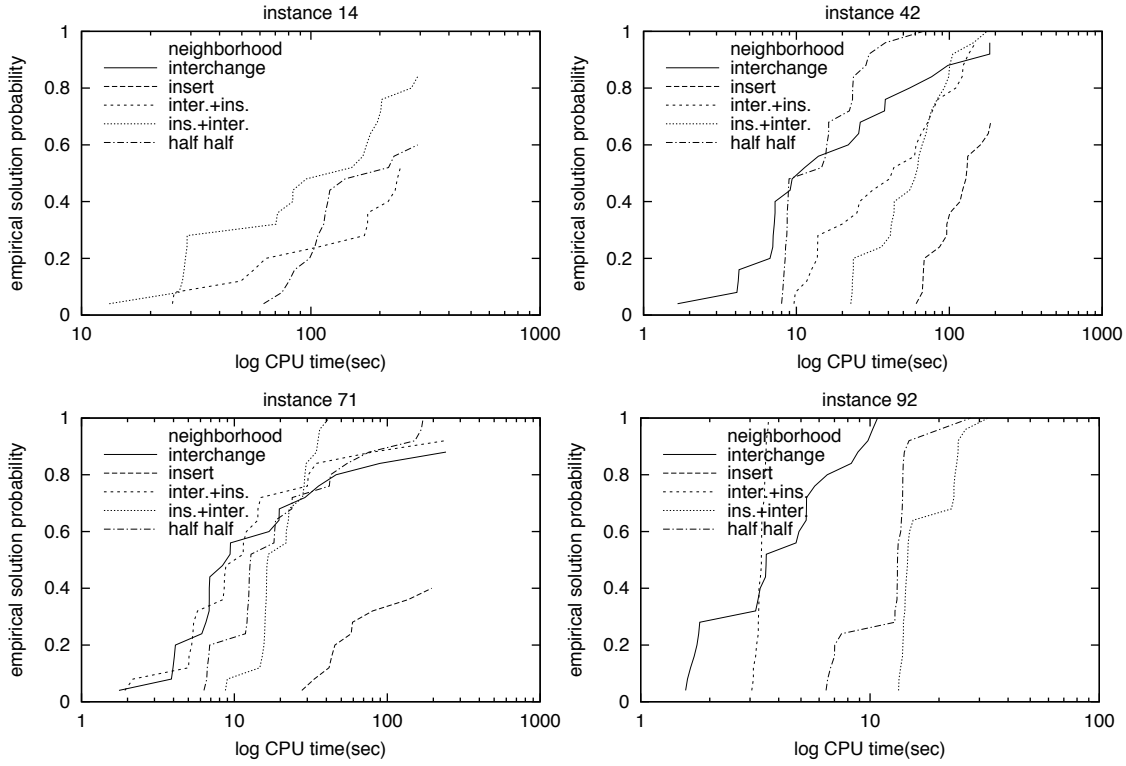


Figure 3.6: Run time distributions for the combination of ACS with different local search algorithms. The algorithm performs 25 tries with CPU–time bounded to 300 seconds. The colony contains 20 ants.

is run 25 times on an instance while limited to spend at most five minutes per instance. Summary results are given in Appendix B.

Initial configuration

The initial ACO algorithm for the SMWTP has the following parameter settings: $\alpha = 1$, $\beta = 2$, $\rho = 0.1$, $q_0 = 0.9$. The colony contains 20 ants. Local search is applied to all solutions these ants construct and the AU dispatching rule provides the heuristic information. The initial solution is formed by a sequence sorted in non–decreasing AU order on which a local search is applied. Further details on the implementation are given in Section 2.4.2.

In the following, I will successively investigate which local search is best used, whether one should change the size of the colony, whether the ants' solution construction can be improved, what the effect is of the pheromone trail, and, finally, what source of heuristic information is most valuable for the ants.

Local Search

Figure 3.6 on the facing page plots the run time distributions for five ACO configurations on instance 14, 42, 71 and 92 of the 125 instance from the benchmark set. The algorithms differ in the local search neighborhoods used. All local search algorithms use the best improvement pivoting rule and a permutation solution representation.

Typically, the combined *interchange+insert* local search that first performs a local search on an interchange neighborhood and then a local search on an insert neighborhood and its counterpart *insert+interchange* local search, perform better than just a *insert* or *interchange* local search. Which of these combined neighborhoods is best is instance dependent. For example, on instance 14 *insert+interchange* finds 50 % of its runs on optimal solution within 100 seconds, while *interchange+insert* finds a solution only 20 % of the runs. On the other hand, all runs of *interchange+insert* solve instance 92 within 10 seconds, while *insert+interchange* has not solved a single run in this time-span. So, perhaps both combined neighborhoods should be used. This last idea is incarnated in *half/half* which denotes an ACO algorithm that uses a heterogeneous colony of ants, in which half of the ants apply *interchange+insert* and the other half apply *insert+interchange* local search[9]. On most instances the run time of this algorithm will be somewhere between the homogeneous colonies which apply one type of combined local search. Sometimes, like on instance 42, the mix of ants performs even better.

Size of the colony

Many ACO run time distributions in Figure 3.6 on the preceding page approximate a step function. This is most prominent on instance 92. Such distributions emerge because the termination criterion is only checked after a whole iteration has finished, so after local search has been applied to all the ant's solutions and the pheromone trails have been updated. So, twenty applications of *interchange+insert* cost slightly more than three seconds on instance 92, while twenty applications of *insert+interchange* require ten extra seconds. Since the algorithms have a high probability to find the optimal solution of an instance within a few iterations, it is worth investigating if the time needed per iteration could be shorter. Most straightforward this is done by using less ants.[10]

Figure 3.7 on the following page gives the run time distributions for colonies of 2, 6, 10 and 20 ants on instance 42, 67, 92 and 117. Three effects of shrinking the colony are: more frequent opportunities to terminate the algorithm, reduced exploration of the search space, and increased ability to 'learn' about the problem through more frequent pheromone update. The first effect, increased frequency of updating the information,

[9] In addition some test were carried out with a variant of *half/half* that insert local search is applied on the first half of the ants solutions and interchange local search on the second half. This variant was less succesful than the *half/half* that employes combined neighborhoods.

[10] Another strategy would be to gradually increase the number of ants (Stützle 1998b, Section 4.3.4). Of course, checking if the termination criterion is met more often, e.g. after each local search application, will yield better results as well. However, note that the differences between the amount of ants are not just an artifact of checking after each iteration.

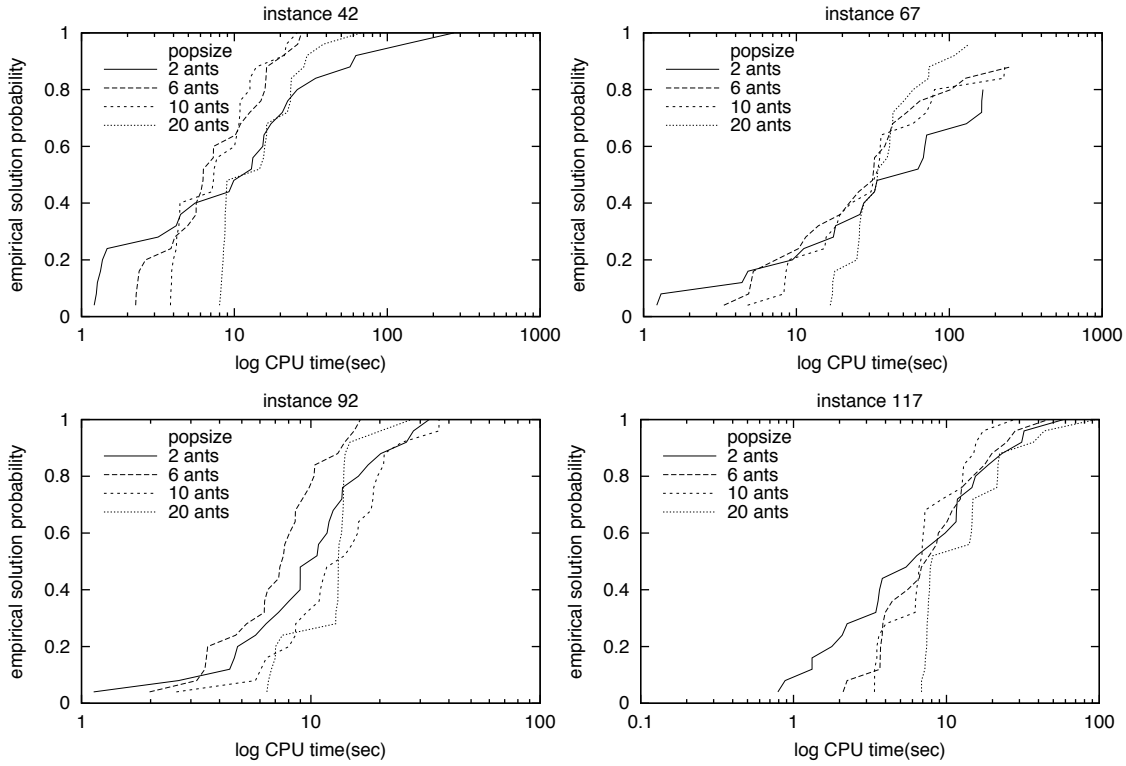


Figure 3.7: Run time distributions for a heterogeneous population of ants that uses the *half/half* local search described on the page before. 25 tests with a 5 minute bound are done on each instance with colonies containing 2, 6, 10 or 20 ants.

can sufficiently explain the run time distributions on instance 117 where independent of the size of the colony 50 % of the runs have terminated successfully within 7 seconds. The second effect, stagnation, occurs on instance 67 where the 20 ant colony has the highest probability to find the optimal solution within 100 seconds. 'Learning', however, is less easy to observe, yet that does not mean that it is not existent.

Since, in general, the best behavior is obtained with 6 or 10 ants, in the following the colony will have 10 ants.

Directing the population

In Section 3.5.1, I proposed some ways to improve the solution costruction by the ants. First, I added candidate lists. Based on some preliminary tests, the length of the candidate list was set to 20. With a candidate list of size 20 on many instances the performance improved. Next, I tested the modifications to the pheromone trail update mechanism (see Equation 3.3 on page 56) and the action choice rule (see Equation 3.4 on page 56). These modification yield no improvement. An explanation for these results is that the candidate list makes the ACO algorithm greedier, whereas the other mechanism support further exploration. It seems that no exploration is needed here.

Figure 3.8 on the next page depicts the run time distributions on two instances for an ant colony system of ten ants where on five of the ants' solutions an interchange local search succeeded by an insert local search is applied and on the other five an insert local search succeeded by an interchange local search. It clearly demonstrates the devastating effect the alternative updating rule or the alternative action choice rule can have on the run time of an algorithm. In addition, it illustrates the performance improvement that can be obtained by using a candidate list.

Problem representation

Another thing that could make a difference is the interpretation of the pheromone trail definition which also influences the way the problem is represented in the graph. The absolute, position based, representation performs clearly better then the relative, adjacency based, representation (see Figure 3.9 on page 63). Most of the time, though, it is even better to have this representation than not to use pheromone at all. On instance 19 adjacency based pheromone yields the best performance, on instance 38 the worst. So, the job's position in the sequence is generally more important than whether it is near an other job, why it is not always the case will be investigated in Chapter 4.

Heuristic information

Finally, it could be that heuristic information other than AU has a better effect on the algorithm's performance. In Table 3.1 on page 40, the effectiveness of earliest due date (EDD), modified due date (MDD) and apparent urgency (AU) was estimated by measuring the quality of solutions constructed by these heuristics. With AU the smallest deviation to the globally optimal solution was attained, yet this was at a larger computational cost relative to the times needed by MDD and EDD.

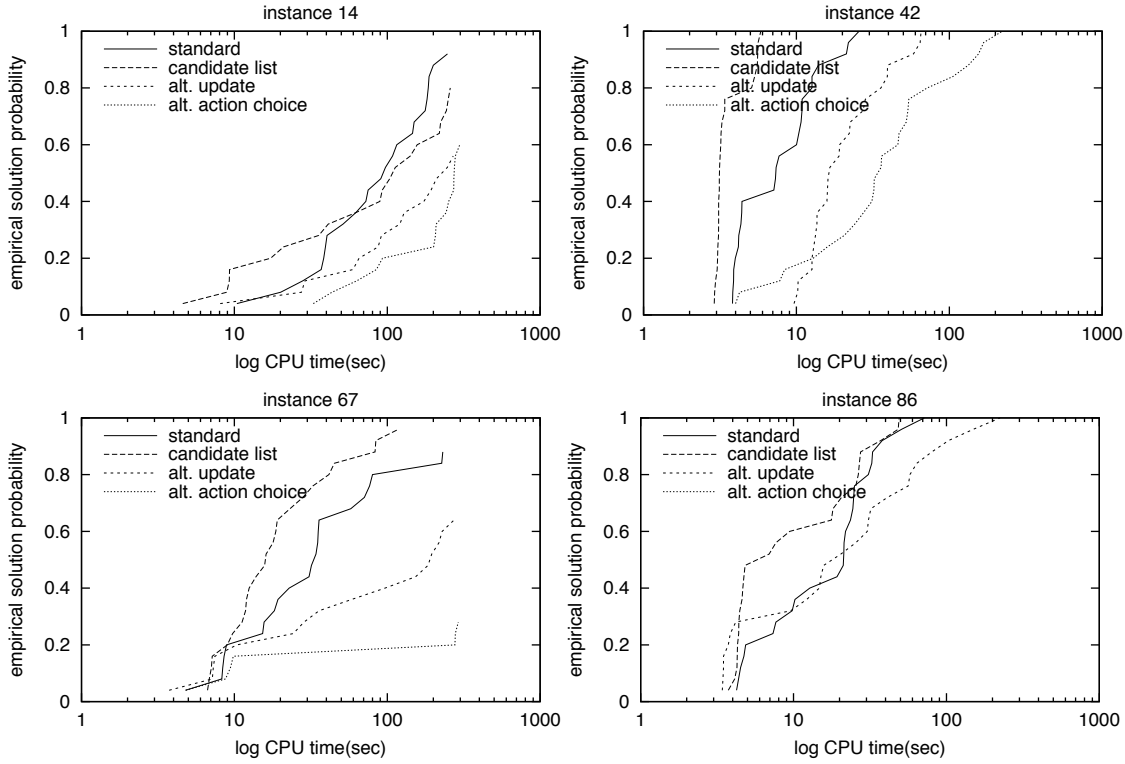


Figure 3.8: Run time distributions for an ACS of 10 ants with *half/half* local search (described on page 59). Compared are the ACS as presented initially (*standard*), an ACS with a candidate list of length 20 (*candidate list*), an ACS with candidate list and alternative pheromone trail update (*alt. update*), and an ACS with candidate list and alternative action choice rule (*alt. action choice*).

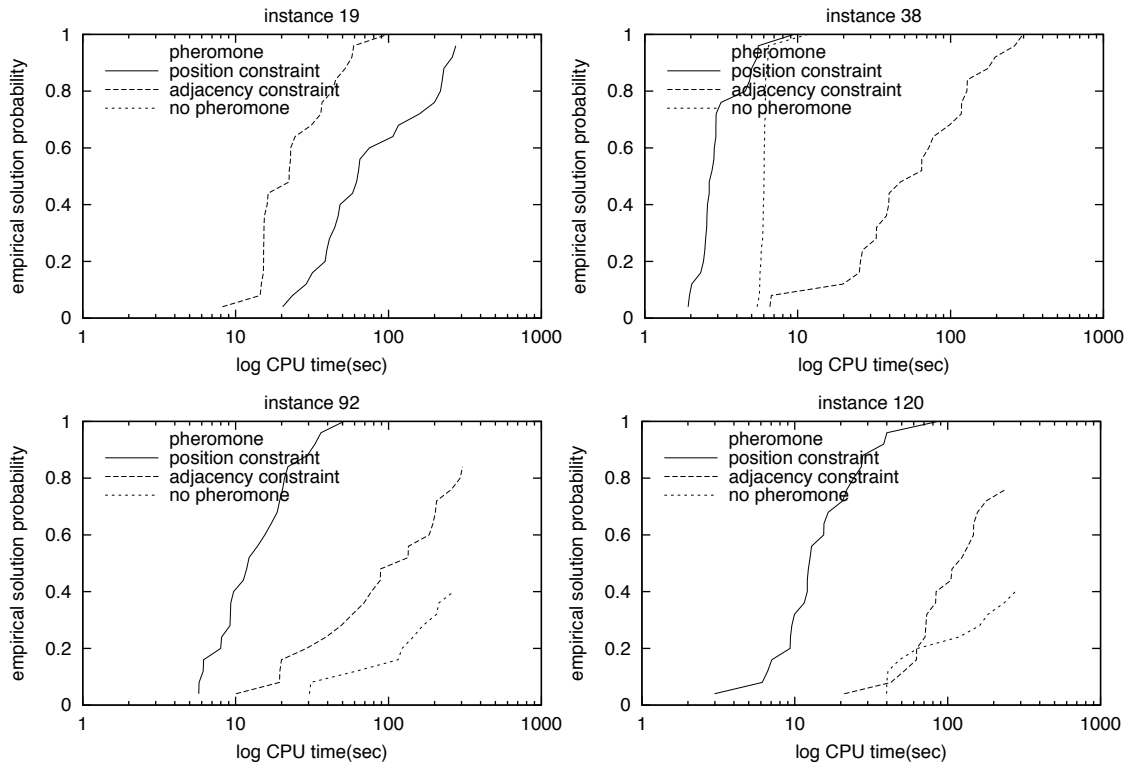


Figure 3.9: Comparison of run time distributions of an ACS of 10 ants with *half/half* local search and a candidate list of length 20 where pheromone is associated to putting a job on a position (*position constraint*), with an ACS that associates pheromone to putting a job after another one (*adjacency constraint*) and an ACS that does not use pheromone at all (*no pheromone*) Except for instance 19, the AU rule provides the heuristic information. For the 19*th* instance the run time distribution of ACS runs that made use of the MDD rule is given, because best known solutions could not be found at all when AU was used.

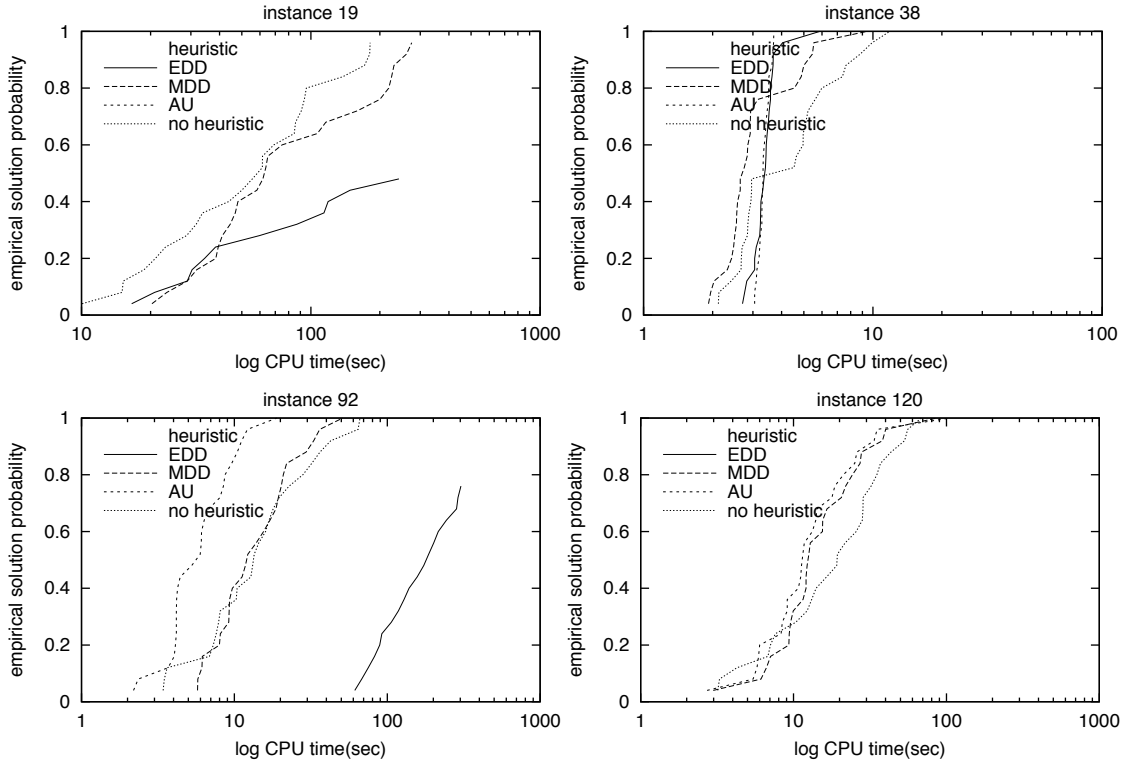


Figure 3.10: Comparison of run time distributions of 10 ant *half/half* ACSs with a candidate list of length 20 where pheromone reflects the 'position constraint' that use earliest due date (EDD), modified due date (MDD), apparent urgency (AU), or no heuristic information at all.

For ant colony optimization the high computational cost of AU might be particularly disadvantageous as the information is requested for each step each ant makes. Another characteristic of AU, which makes things even worse, is that it is a dynamic heuristic (see page 5). An important difference between static and dynamic heuristic information is that static information does not depend on the partial solution constructed by the ant while dynamic information does. When using static information, like in the EDD-based heuristic information, the values of $\tau_{ij}(t) \cdot {\eta_{ij}}^{\beta}$ can be precomputed and need only to be updated occasionally. With dynamic heuristic information, this is typically not possible and in this case the solution construction incurs a higher computational cost. Yet, this may be made up for by a higher accurateness of dynamic heuristic information. In Figure 3.10 I give the run time distributions for four instances using the three different heuristics and not using heuristic information at all. The EDD seems to provide false heuristic information, since it often yields worse performance would be obtained if not heuristic was used at all. However, the fact that on instance 19 the best performance is achieved when no heuristic is used does not imply that AU and MDD give bad estimates for the desirable sequence of jobs to solve this instance. On the contrary, it seems that AU is so strong that it holds the ants so near to the old local optimum that local search

can not escape from it to find a better solution.

In general, AU gives the best performance, so this dispatching rule should be used in the final ACS configuration. However, on instance 14 and especially on instance 19 AU hinders ACS in finding the best known solution. These instances differ from other instances in that they have a small range of due dates (RDD). Therefore a rather ad hoc rule will be used that states that ACS uses MDD for RDD ≤ 0.3 and AU otherwise.

3.5.3 Future work

An observation that can be made from the computational results discussed in Section 3.5.2 is that there exists no single optimal ACO configuration for the entire set of SMWTP instances, a configuration that works very well on some instances may work very badly on other instances.

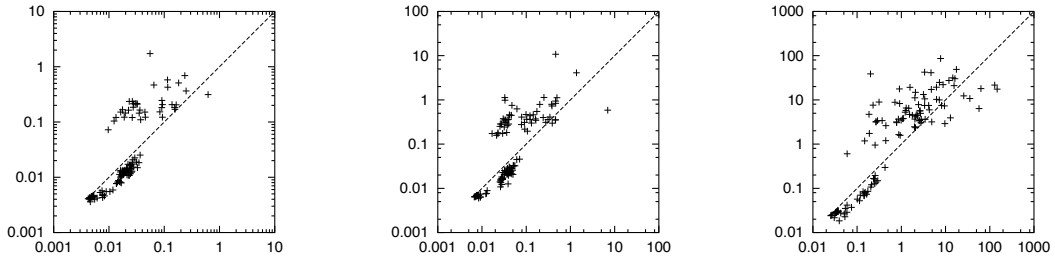
Three directions can be followed to obtain a configuration that works well on all instances. First of all, one can try to identify properties of an instance that make it suited for a certain configuration and on the basis of this knowledge construct an algorithm that classifies the instance and chooses a configuration on the basis of this classification. The identification of such properties is one of the topics in Chapter 4. One could also seek refuge in heterogenous ant colonies like is done in the *half/half* configuration where one type of local search is applied to 50 % of the ants' solutions and another type of local search to the other 50 %. This idea can be extended to using more than two types of local search or making the ants differ in other aspects like their greediness and the heuristic information they employ. Finally, one could make the algorithm self–tuning by using search feedback to adjust the configuration. For example, one might want to adjust to colony size and the greediness of the ants to provoke more exploitation or exploration of the search space dependent on the feedback.

Actually, there might be still another way to obtain a configuration that works well on all instances: apply a dynasearch local search to the ants' solutions. After all, it could be that the dynasearch local search used by iterated dynasearch is so powerful that the ACS becomes more robust with respect to other configuration issues, because a bad configuration of the ants might be 'repaired' by the dynasearch local search and dynasearch might outperform other local search procedures on all instances, making *half/half*–like ideas superfluous. Still, in Section 3.3.3 and Section 3.4.3 I have argued that the performance of iterated dynasearch might not be that exceptional at all. Yet, to find out whether this is true, re-implementation will be required.

3.6 Iterated local search versus ant colony optimization

For comparison of iterated local search and ant colony optimization, I have run the final algorithms of Section 3.4 and Section 3.5 on the benchmark set available in ORLIB. In addition, I have generated some SMWTP problem instances of 200 jobs and tested the algorithms on these instances.

To recapitulate: the ILS algorithm uses a *interchange+insert* local search, a local search on the interchange neighborhood succeeded by a local search on the insert neigh-



(a) 40 **(b)** 50 **(c)** 100

Figure 3.11: Ant colony optimization versus iterated local search for the SMWTP instance of ORLIB. Each point has two components. Its x value is the average computation needed by the ILS algorithm, the y value is the average run time of the ACO algorithm for that instance. Both algorithms were tested 100 times on each instance except for the ACO tests on the 100 jobs instances which were repeated only 25 times. The CPU bound was set to 10 minutes on 40– and 50–job instances and 20 minutes for 100–job instances.

borhood. Two solution are compared at the initialization, one is the result of a local search application to a sequence of jobs in non–decreasing MDD order, the other of an application to a sequence in non–decreasing AU order. The best of these solutions is modified and local search is applied on the modified solution. The solution is modified by random left insert kicks. The number of kicks is varied within the range $[4, 20]$ in the way described on page 46. A solution found by the local search is accepted for modification is it has a lower weighted total tardiness associated to it than the current best known solution. The ACO algorithm has a colony of 10 ants. An *interchange+insert* local search is applied to 5 solutions constructed by the ants. *Insert+interchange* local search is applied to the remaining 5 solutions. A candidate list of length 20 is used to improve the ants' solution construction. On instances that have a small range of due dates (RDD ≤ 0.3), the MDD dispatching rule provides the heuristic information, AU is used on the other instances. The remaining parameters settings can be found on page 58.

Figure 3.11 summarizes the results for the ORLIB benchmark. It compares the average run times of the ACO algorithm (y–axis) and the ILS algorithm (x–axis) for the each of the 125 instances in the set of 40, 50, and 100 jobs. Both algorithms were tested 100 times on each instance except for the ACO tests on the 100 jobs instances which were repeated only 25 times. The CPU bound was set to 10 minutes on 40– and 50–job instances and 20 minutes for 100–job instances. The dotted line indicates equal CPU time for ILS and ACO and therefore divides the instances into those solved faster by ACO and those solved faster by ILS — on average. Instance to the right side of the line are solved faster by ACO, on the left faster by ILS.

The resulting plots are remarkably similar independent of instance size. With increasing number of jobs the required computational effort grows, yet this has little effect

on the balance between ACO and ILS. Generally ILS outperforms ACO. An exception to this rule is formed by clusters of instances near the origin of the plot. These instances are solved after only one local search application. The reason that ACO is faster on these instances is that its initialization involves only one local search, while the initialization mechanism chosen for ILS needs two local searches. So, the advantage of ACO on small instances is just an artifact which would not have occurred if one would check the termination criterion right after the first local search application. However, also the advantage of ILS on instances that require on average less than 10 or perhaps even 20 or 30 local search applications can be attributed to the artifact that the ILS algorithm checks the solution every local search application while the ACO algorithm with a colony of ten ants checks only once every 10 applications.

A closer examination of the algorithms' performance can be made for the 18 instances for which Figure 3.12 on the following page gives the run time distributions for the ACO algorithm (dotted line) and the ILS algorithm (solid line). The instances are either among the ten instances on which the ACO algorithm required the most CPU time or among the eleven instances on which the ILS algorithm required the most CPU time. The ILS algorithm outperforms ant colony optimization on two third of these instances. In many of the instances on which ILS is slower, ACO requires only few iterations. For example, the ACO algorithm can solve the $98th$ instance (column two, row four in the figure) within one iteration in 50% of the tests, over 90 % is solved within two iterations and in the worst case it requires 3 iterations. Yet, the ILS algorithm requires an average of 114 iterations to solve the same instance. An extreme case is instance 9. This instance is solved right at the initialization of the ILS algorithm, whereas the ACO requires on average 14 iterations. A case of 'lucky' initialization.

Though Figure 3.11 on the preceding page does not point in that direction, it could be that the ant colony optimization would be more competitive with iterated local search on larger problems. To test out, I have generated nine 200–job instances. Five of the instances are characterized by a due date range RDD = 0.2 and a tardiness factor TF = 0.6, the remaining have RDD = 0.2 and TF = 0.8. The instances are names according to the scheme used for the ORLIB instances: instance 11–15 have TF = 0.6 and instance 16–19 TF = 0.8. Because these values of TF and RDD have proved to yield hard single machine total tardiness problems in Chapter 2, they were chosen for these instances. In order to obtain 'best known' solutions, I ran ILS ten times for one hour on the instances and compared the total weighted tardiness of the solutions found. Because the values of the ten runs were in complete agreement with each other, it seems safe to take them as 'best known', so that's what I did.

ILS and ACO were run 25 times with a CPU bound of 3600 seconds. Figure 3.13 on page 69 gives the run time distributions for these instances. Unfortunately for ACO supporters, on these instances the ILS algorithm requires far less run time to find the 'best known' solution than the ACO algorithm. However, the picture given in the figure might deformed by the artifact of checking for termination only once per iteration. For example, on the $14th$ instance the ACO algorithm has found the 'best known' solution with a probability of almost 80 % within one iteration. Yet, the ILS algorithm has ended all 25 runs successfully before the ACO algorithm can finish its first iteration. Still, it

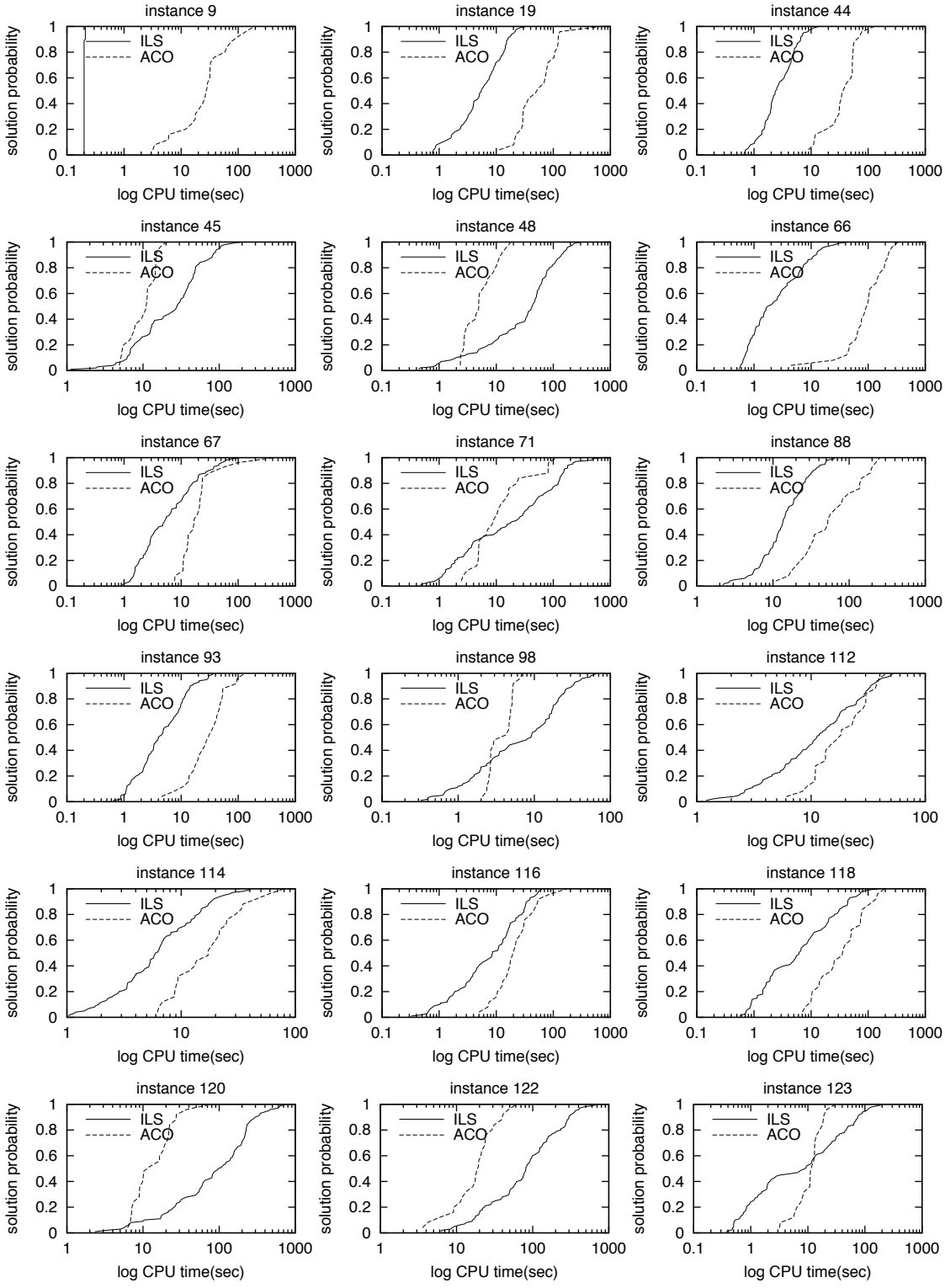


Figure 3.12: Run time distributions for the toughest 100–job instances.

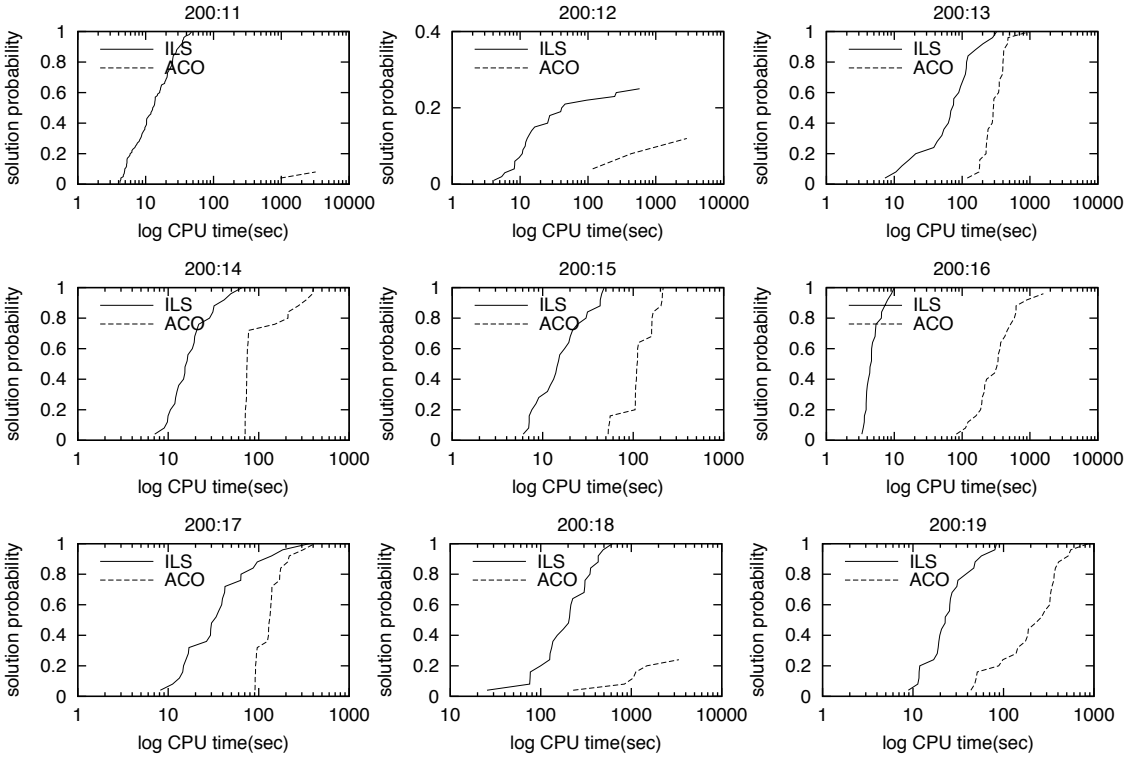


Figure 3.13: Run time distributions of ILS and ACO on 200–jobs SMWTP instances.

seems justified to state that iterated local search is more suited for the single machine total weighted tardiness scheduling problem than ant colony optimization.

3.7 Summary

This chapter has presented a study into the application of ant colony optimization (ACO) to the single machine total weighted tardiness scheduling problem (SMWTP). An important component in ACO algorithms for the SMWTP is local search and consequently a considerable part of the chapter has been devoted to the configuration of local search. In addition, an algorithm for the SMWTP was described which is based on the iterated local search (ILS) meta-heuristic. Since ILS is a much simpler way to embed local search than ACO, its performance forms an nice reference point for the ACO algorithm. Finally, issues in the design of an ACO algorithm for the SMWTP were discussed.

This chapter has shown that it is possible to obtain good performance by applying ACO to the SMWTP. However, even better performance is obtained when the ILS meta-heuristic is used. So, the ILS approach is more suited. Still, it should be noted that the most important factor for successful application is not whether one uses kick–moves or ants, but which type of local search is embedded in the algorithm. The local search which is used by ILS and ACO achieves good performance by combination of the insert and the interchange neighborhood and a few application of this local search suffice to find many best known solutions of instances in the ORLIB benchmark set.

The algorithms were not directly compared with iterated dynasearch which is currently the best algorithm for the SMWTP. This might be done in the near future. Considering that the ACO and ILS algorithm could find, for all known benchmark instances available in ORLIB, the optimal or best known solution within reasonable computation times, whereas iterated dynasearch found on average only 123.2 times the best known solutions (Congram et al. 1998, Table 6) and considering the doubts about the optimality of design of iterated dynasearch expressed in Section 3.4.3, there is a chance that the ILS algorithm presented in this chapter will leave iterated dynasearch behind in performance.

The computational experience in this chapter indicates that there exist enormous differences between individual SMWTP instances with respect to the computational effort required to find the best known or optimal solution. In addition, instances that are hard to solve with one algorithm might be straightforward for a slightly different algorithm. What exactly causes an instance to be hard is the subject of the next chapter.

4 Analysis

4.1 Introduction

The experiments described in the previous chapter may give indications which kind of algorithm design yields the best performance, yet they provide little information why certain design choices are preferable or why the single machine total weighted tardiness scheduling problem (SMWTP) is relatively easily solved. The question that should be addressed is what makes a problem type or instance hard for a search algorithm.

Two directions are followed in this chapter to investigate this aspect. First, the relationship between job data, structure of the global optima, and the algorithm's effectiveness is studied. Second, it is investigated what fitness landscapes are induced by the search operators for the SMWTP.

An empirical analysis of these questions is performed on the 125 100–job SMWTP instances which are available in ORLIB. The principal tool for the analysis is the correlation coefficient $\rho(x, y)$ which is a measure for the coherence of stochastic variables x and y. It is given by $\rho(x, y) = \mathbf{C}(x, y)/(\sigma(x) \cdot \sigma(y))$ where $\mathbf{C}(x, y)$ is the covariance of x and y and $\sigma(x)$ is the variance of x. It will for example be used to estimate how strongly certain problem characteristics and the algorithm's run time are related.

4.2 Problem characteristics

Two statistics, the tardiness factor (TF) and the range of due dates (RDD), are repeatedly used to characterize SMWTP instances. TF and RDD serve as parameters that direct the generation of instances. For example, the ORLIB benchmark set for the SMWTP that is used here contains five instances for each pair of TF and RDD in the set $\{0.2, 0.4, 0.6, 0.8, 1.0\}$. TF and RDD can also be measured on a given instance to adjust an algorithm like the apparent urgency dispatching rule whose scaling parameter k is set dependent on the TF–value of an instance (see page 34). Apparently, these statistics also may help discriminate easy instances from hard ones, but what values of TF and RDD characterize hard instances? Is there possibly a relation between these statistics and properties of the optimal solutions? In the following, it is investigated how tardiness factor and range of due dates relate to the number of late jobs and runs in the globally optimal solutions and it is determined to what extend these statistics can predict the effort needed by iterated local search and ant colony optimization to find the optimal solutions.

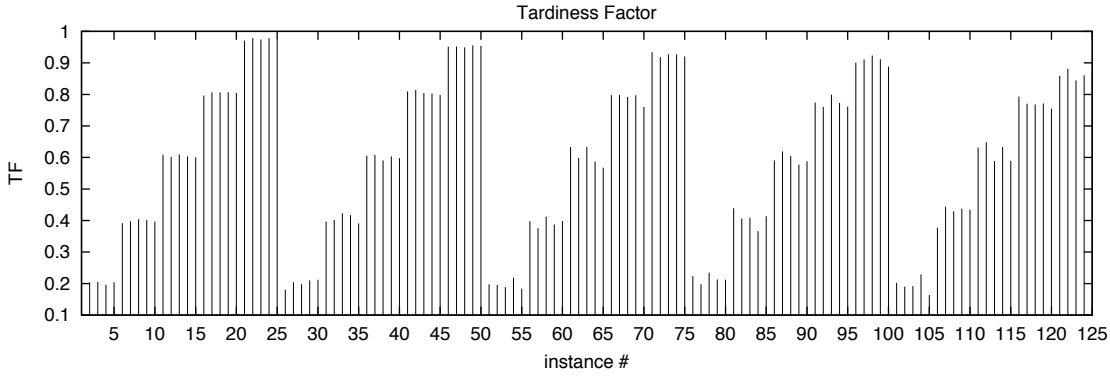


Figure 4.1: Tardiness factors of the 125 100–job instances in ORLIB.

4.2.1 Tardiness factor and range of due dates

The due date tightness factor or tardiness factor (TF) is obtained by computing

$$\mathrm{TF} = 1 - \frac{d_{\mathrm{avg}}}{C_{\max}} = 1 - \frac{\sum d_j}{n \sum p_j} \tag{4.1}$$

It indicates how far the average due date differs from the largest possible completion time of any job. A low value of TF means that the jobs' due dates will tend to be loose and jobs can relatively easy be processed before they are due. The opposite is the case if TF has a high value.

The due date range factor or range of due dates (RDD) is defined as

$$\mathrm{RDD} = \frac{d_{\max} - d_{\min}}{C_{\max}} \tag{4.2}$$

For instances with a small RDD–value, when the jobs have a similar due date, the tightness of the due date will have a greater impact than for instances with a high RDD–value.

Figure 4.1 and Figure 4.2 on the facing page show the values of TF and RDD that are associated with the instances of the 100–job benchmark for the SMWTP. The due dates for these instances are said to be randomly drawn from the uniform distribution $[(1 - \mathrm{TF} - \frac{\mathrm{RDD}}{2}) \cdot C_{\max}, (1 - \mathrm{TF} + \frac{\mathrm{RDD}}{2}) \cdot C_{\max}]$, with five instances for each pair of TF and RDD from the set {0.2, 0.4, 0.6, 0.8, 1.0}. This description does not correspond to the empirically measured values of TF and RDD depicted in these figures, however. The due dates of instances 21–25, 46–50, 71–75, 96–100, and 121–125 are spread only half as much as they should be (Figure 4.2). Consequently the instances do not have the promised tardiness factor (Figure 4.1). Apparently, the lower bounds for uniform distribution of the due dates was specified as $\max\{(1 - \mathrm{TF} - \frac{\mathrm{RDD}}{2}), 0\} \cdot C_{\max}$ instead of $(1 - \mathrm{TF} - \frac{\mathrm{RDD}}{2}) \cdot C_{\max}$, preventing negative due dates to occur. If due dates are not allowed to be negative, then for the average of the due dates to approximate zero they must have a small range. This explains that TF = 1 is not reached for large values of

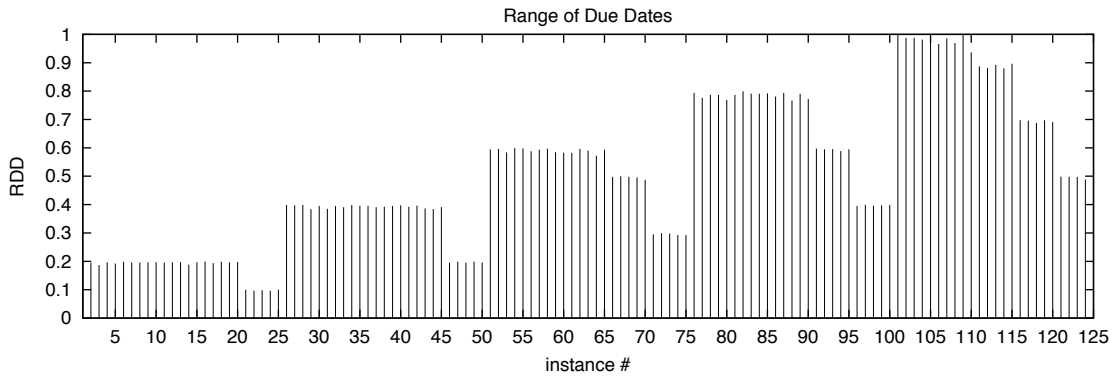


Figure 4.2: Due date ranges of the 125 100–job instances in ORLIB.

	ILS	log ILS	ACO	log ACO
TF	0.264	0.736	0.422	0.740
RDD	-0.001	-0.249	-0.032	-0.228

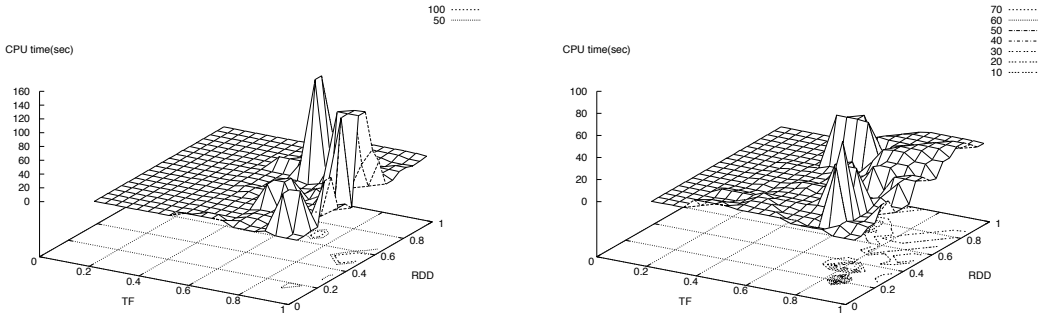
Table 4.1: Correlation between job statistics and algorithm efficiency.

TF. Accordingly, the range of due dates is effectively cut in half when TF should be one but jobs cannot be committed before the system's starting time. So, the lower bound of zero results in instances that do not have the appropriate TF and RDD. In addition, the distribution of the due dates in these instances will not be as uniform as in "regular" instances since a large share of the due dates will have the common value of zero. In the following I will consider the empirically measured TF and RDD values of the instances, but this neglects the differences of the due date distribution though they could be an important factor for the instance hardness.

4.2.2 Hardness in relation to TF and RDD

In Figure 4.3 on the next page the average time needed to find an optimal solution with iterated local search (ILS) and ant colony optimization (ACO) for an instance is plotted against the tardiness factor (TF) and the range of due dates (RDD) of the instances. For most combinations of TF and RDD the instances are easily solved for both ILS and ACO. Hard instances have $TF > 0.5$ and for ACO and $TF > 0.5 \wedge 0.4 < RDD < 0.7$ in the case of ILS. Note that nothing can be said about the hardness of instances that have $TF > 0.8 \wedge RDD > 0.7$, as such instances do not occur in ORLIB's benchmark for the SMWTP.

The correlation coefficients between the algorithms' efficiency measured in average CPU–time and the instance–characteristics TF and RDD are given in Table 4.1. Clearly, the job–statistics alone are not sufficient to predict the effort needed by iterated local search or ant colony optimization to find the optimal solution for a given problem. A stronger correlation is obtained between logarithmically scaled run times and TF and RDD. This scaling accounts for the fact that run times increase exponentially, not



(a) iterated local search

(b) ant colony optimization

Figure 4.3: Study of run times of ant colony optimization and iterated local search in dependence to tardiness factor (TF) and range of due dates (RDD). A description of the experimental setting in which the run times were measured can be found in Section 3.6.

linearly, as the instances become harder. A reason for the low correlation with RDD may be that only linear correlation is measured by the correlation coefficient, but perhaps a non–linear regression is needed. A better correlation may also be obtained if RDD and TF are considered both.

4.2.3 Properties of the global optima determined by TF and RDD

We have seen that hard SMWTP instances all fall within a small range of TF and RDD values. Perhaps these statistics reflect a particular structure of the problem that makes it hard to solve.

To investigate this issue, I have run the ILS algorithm starting from random initial solutions to generate 10 global optimal solutions for each of the 125 100–job instances. For each solution the number of late jobs was counted and the number of runs was determined. Recall that a job is *late* if it is completed after the committed date and a *run* is a sequence of jobs that share the same property of being late or non–late. These statistics are actually characteristics which all global optima for an instance have in common: the difference between the maximum and minimum number of late jobs in the set of global optima for an instance was at most two jobs. For runs likewise, the largest difference was 2 runs. In addition, a closer look at the partition of global optima for an instance into run reveals that the optima consist almost without exception of the same consecution of runs. Different ordering of jobs within the runs is the only cause for having multiple global optima.

The tardiness factor of an instance turns out to be a precise predictor for the number of late jobs in the global best solutions. The correlation coefficient of these statistics is $\rho(\mathrm{TF}, late) = 0.965$. This coefficient reflects the observation that jobs will be early if

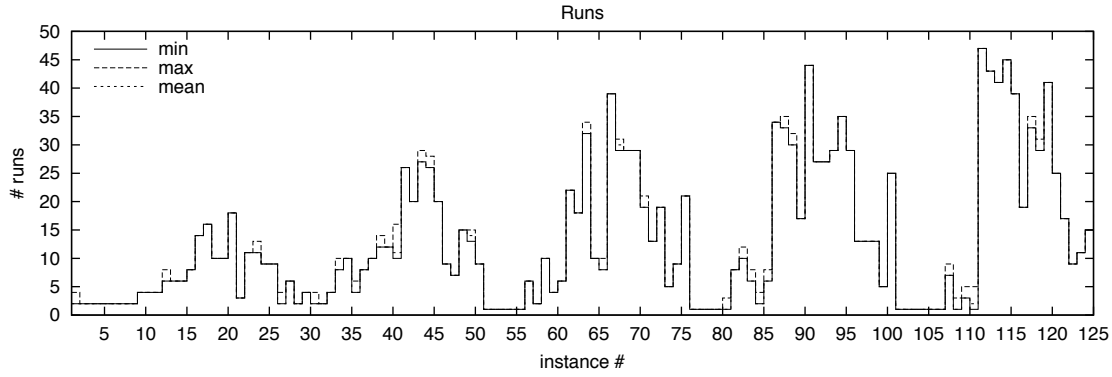


Figure 4.4: The number of runs of jobs in global optima partitioned into runs of late jobs and runs of non–late jobs.

	TF	RDD	*log* ILS	*log* ACO
run	0.519	0.201	0.685	0.638

Table 4.2: Correlation between number of runs in best known solutions, tardiness factor (TF), range of due dates (RDD) and logarithmic run times for iterated local search (ILS) and ant colony optimization (ACO).

due dates are near the worst occurring completion time and will be late if due dates are near the release date of the jobs.

The relationship between TF and RDD and the number of runs in global optima is somewhat more complex. Figure 4.4 plots the number of runs per instance. Clearly, high ranges of due dates and modest tardiness factors cause a high number of runs. The reason why this is interesting is that this could partially explain why some instances take more time to solve than others. The speedups used in the previous chapter make heavy use of the run data-structure (see Algorithm 3.3 on page 38) which allows to evaluate the effect a local search move has for all jobs of a run at the same time and avoids having to calculate the effect of a move for each single job of a run. The highest gain in evaluation–speed can be achieved when there are only few runs in the permutation. Perhaps the number of runs is a better indicator for the size of the search space than the number of jobs. At least, Table 4.2 indicates that the number of runs has a significant influence on the instance–hardness. Yet, one has to keep in mind the run times were measured on algorithms that used a local search that depends on the run data-structure for the evaluation of its moves. It remains an open question if runs are that important an indicator of then instance hardness when they are not taken advantage of in the algorithm itself.

4.3 Fitness landscapes

Supplementary to the analysis just made, one could try to determine in what way the fitness landscapes on which the search algorithms operate differ among the instances. This is the aim of the fitness landscape analysis.

4.3.1 The concept

Formally, the fitness landscape is defined by

1. the set of all possible solutions $\mathcal{S}$;
2. an objective function that assigns to every $s \in \mathcal{S}$ a fitness value $f(s)$;
3. a neighborhood structure $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$.

This captures the intuitive idea of navigating search in a mountainous area. The local search algorithm can be imagined as a wanderer that performs a biased walk in this landscape. Sticking to this picture, it is obvious that the task for the wanderer strongly depends on the ruggedness of the landscape, the distribution of the valleys and craters and the local optima in the search space and the overall number of the local minima. Being such an intuitive idea, the metaphor of fitness landscapes is commonly used to aid the understanding of heuristic search methods for solving a combinatorial optimization problem.

Essentially, there are two interesting elements in the analysis of fitness landscapes. The first is which of the possible neighborhoods is the most appropriate. A fitness landscape is induced by the particular operator which is used to define the neighborhood. An operator that causes a smooth fitness landscape that is easy to search is of course preferable over operators that cause rugged fitness landscapes. Measuring the ruggedness of fitness landscapes is the subject of Section 4.3.2.

Secondly, more detailed knowledge on what the landscape 'looks like', that is, on the gross relationship of the position of local optima relative to the global optima and on the number of local optima, is needed to determine which search algorithm is most suited for it. This is the second interesting element of fitness landscape analysis. In particular, it would be nice if the topology of the fitness landscapes would explain the good performance of iterated local search or ant colony optimization. Section 4.3.3 is concerned with that subject.

4.3.2 Landscape ruggedness

The ruggedness of a fitness landscape indicates how hard it will be for a local search algorithm to search that fitness landscape. If neighboring points in the fitness landscape have more or less similar objective function values, the fitness landscape is smooth. On the other hand, a fitness landscape is rugged if small moves lead to large changes in solution quality.

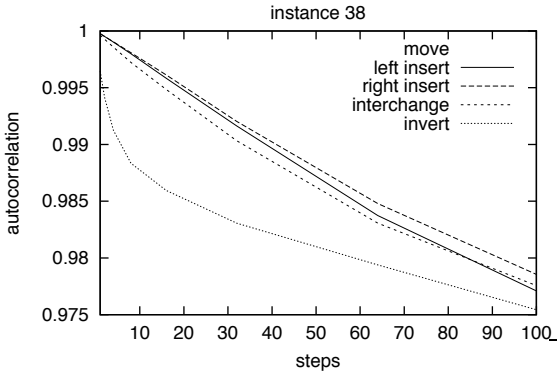


Figure 4.5: Plot of the random walk correlation function $r(s)$ for $s = 1, \ldots, 100$. The random walk consists of 10 000 right insert, left insert, interchange, or invert moves.

	left ins	*right ins*	*inter*	*invert*
14	5524.36	5813.45	3936.51	458.85
19	7041.75	7462.19	6134.47	1055.47
38	4183.6	4218.91	2716.89	283.51
42	7041.75	6896.05	5617.48	1059.95
67	7298.77	7245.88	5916.66	1075.93
71	8064.02	7999.5	6622.02	1999.5
86	3905.75	4443.94	2550.52	276.51
92	6802.22	6896.05	5813.45	1059.95
117	6622.02	6134.47	4830.42	731.03
120	7751.44	8196.22	5318.65	1202.87

Table 4.3: Correlation lengths for random walks of 10000 right insert, left insert, interchange, or invert moves.

Basically, the ruggedness reflects the correlation structure of neighbored solutions in the fitness landscape. To measure the ruggedness of a fitness fitness landscape, Weinberger (1990) suggested to use random walks. The *random walk correlation function*

$$r(s) = \frac{\sum_{t=1}^{m-s}(f(x_t) - \overline{f}) \cdot (f(x_{t+s}) - \overline{f})}{\sigma^2(f)} \tag{4.3}$$

of a time series $\{f(x_t)\}$ defines the correlation of two points s steps away along a random walk through the fitness landscape. Figure 4.5 plots the correlation of two points against the number of steps between the points for a number of operators. The figure will be discussed in more detail later on, it is enough to note that one can discern good and bad operators based on the curves in this plot. That is to say, if the correlation remains high for points that are separated by many steps, the fitness landscape is really smooth.

The correlation length l of the fitness landscape is a summary statistic for curves like in Figure 4.5. It is defined as

$$l = -\frac{1}{\ln(|r(1)|)} \tag{4.4}$$

where $r(1) \neq 0$. It assumes that the correlation between neighboring points will decay exponentially with the number of steps these points are separated. In that case, the correlation length is the number of steps after which there is no correlation left between the points.

Table 4.3 gives the correlation lengths for a time series obtained by a random walk for the 10 SMWTP instances that on which the tuning has concentrated in Section 3.4.2 and Section 3.5.2. The walk consists of 10 000 random applications of either the interchange ($\cdots \pi_i \cdots \pi_j \cdots$), right insert ($\cdots \pi_i \cdots \pi_j \cdots$), left insert ($\cdots \pi_i \cdots \pi_j \cdots$) or the invert move. The *invert* move is a non–standard operator for sequencing problems. Given positions i and j in permutation π, the invert move produces a permutation π' that has the jobs between positions i and j in the inverse order. The correlation lengths

in Table 4.3 on the page before are much lower for this operator than for the others which suggests that an invert neighborhood on the SMWTP should lead to worse performance.

What is really striking, however, is that the correlation lengths in Table 4.3 on the preceding page are so large and that the autocorrelation in Figure 4.5 on the page before remains so high. These values are much larger than what is observed on other problems (Merz and Freisleben 1999). This might indicate that SMWTP instances are relatively easy to solve. Furthermore, the huge difference in performance between algorithms which use the right insert neighborhood and algorithms which search the left insert neighborhood (see Section 3.3.2 and Section 3.4.2) is not reflected at all in the correlation lengths for random walks based on these operators. Landscape ruggedness is too coarse a measure to provide such detailed information.

4.3.3 Distribution of local optima

Another approach to the analysis of fitness landscapes is to generate local optima and study how these local optima are distributed over the fitness landscape and what's their relationship to global optima. Such an approach is, for instance, followed by Boese, Kahng and Muddu (1994). These authors suggest that, in the cases of the traveling salesman problem and graph bisection, local optima tend to be relatively close to each other and to the global optimum. Thus, for at least some problem fitness landscapes, there is a *big valley* structure, where a global optimum is surrounded by local optima of progressively increasing average quality as they approach that global optimum. On such fitness landscapes iterated local search and ant colony optimization are almost certain to outperform multi–start local search. The average distance between locally optimal solutions is relatively small, compared to the size of the search space. Because the interesting part is contained in a relatively small region of the total search space, it is advantageous if one can focus on search near the current local optimum, which is already situated in that area. This is what ant colony optimization and iterated local search do by embedding a local search method into a global guiding mechanism which provides new nearby starting solutions.

Distance Metrics

Mapping fitness landscapes requires knowledge on both altitude and amplitude of points in the fitness landscapes. Altitude is defined by the objective function of the problem. In principle, the distance between two solutions π and π' on a fitness landscape should be measured by the minimal number of applications of the operator which would convert π into π'. However, in general no polynomial algorithm is known for calculating this number (Reeves 1997).

Yet, there are a number of surrogate distance metrics that could be used in such cases to give a rough approximation of the true distance. Reeves (1997) proposes the following three measures to estimate how many local search moves are at least needed to transform one sequence into another.

The adjacency metric. This metric counts the number of times n_{adj} a pair i, j of jobs is adjacent in both π and π'. The distance $d_{adj}(\pi, \pi')$ is then measured by subtracting this amount from the maximum number of jobs that can be adjacent in a sequence, that is, $d_{adj}(\pi, \pi') = n - 1 - n_{adj}$.

The precedence metric. This metric tries to be a little more sophisticated. Rather than simply looking at adjacencies, the number of times job j is preceded by job i in both π and π', n_{prec}, is counted. To get a 'distance', this quantity is subtracted from $n(n-1)/2$, which corresponds to the maximum closeness of two solutions. $d_{prec}(\pi, \pi') = n(n-1)/2 - n_{prec}$.

The position–based metric. This metric compares the actual positions of the jobs in the permutations π and π'. For a sequence π we can define the 'inverse' permutation σ where the position of job π_i is given by $\sigma_{\pi_i} = i$. The position–based distance is then $d_{pos}(\pi, \pi') = \sum_{j=1}^{n} |\sigma_j - \sigma'_j|$.

In addition to these three metrics, I consider a variant that counts only the distance for jobs that are late in the objective solution π'. Such a modification is motivated by the conjecture that often the ordering of non–late jobs is not very important for the weighted tardiness objective. The tardiness of non–late jobs is zero. If these jobs are shifted from their current position to a position more in front of the queue, their contribution to the SMWTP objective will remain zero. If the due dates are not too tight, many jobs can even be shifted backwards without effecting the quality of the permutation. In this way one can obtain many local optima which are effectively one and the same if the non–late jobs are ignored. By just considering late jobs the 'noise' generated by these redundant solutions is filtered out.

Search operators

Distance metrics could help identify the constraints that are important for solving a problem. A high correlation between the distances given by a metric and costs given by the objective function suggests the use of search operators that are best approximated by the metric. Still, the identification of constraints makes no sense if these constraints cannot be taken into account in the algorithm's design. Section 3.5.1 already illustrated how the graph in ant colony optimization can be used to concentrate on the important properties of the problem. To a less extend the same can be achieved with local search operators. If the relative position of jobs is important, the transpose operator $(\cdots \pi_i \pi_{i+1} \cdots)$ will do well. Interchange $(\cdots \pi_i \cdots \pi_j \cdots)$ is suited for incorporation of the more general precedence constraint, insert $(\cdots \pi_i \cdots \pi_j \cdots)$ will be useful is the absolute position of jobs is what matters. Since insert and interchange subsume transpose, these operators can also be used when adjacency is the dominating factor. But, because the neighborhood they operate on is larger, search with these operators is likely to consume more time.

Distance correlations

The fitness distance relationship between metric and objective function can be examined for any particular case by giving a plot of the distance versus fitness of local optima relative to the closest global optimum. However, it is clearly impossible to plot such a graph for every operator, metric and problem instance. Therefore the correlation coefficients between distances and differences in objective function of points in the landscape are computed.

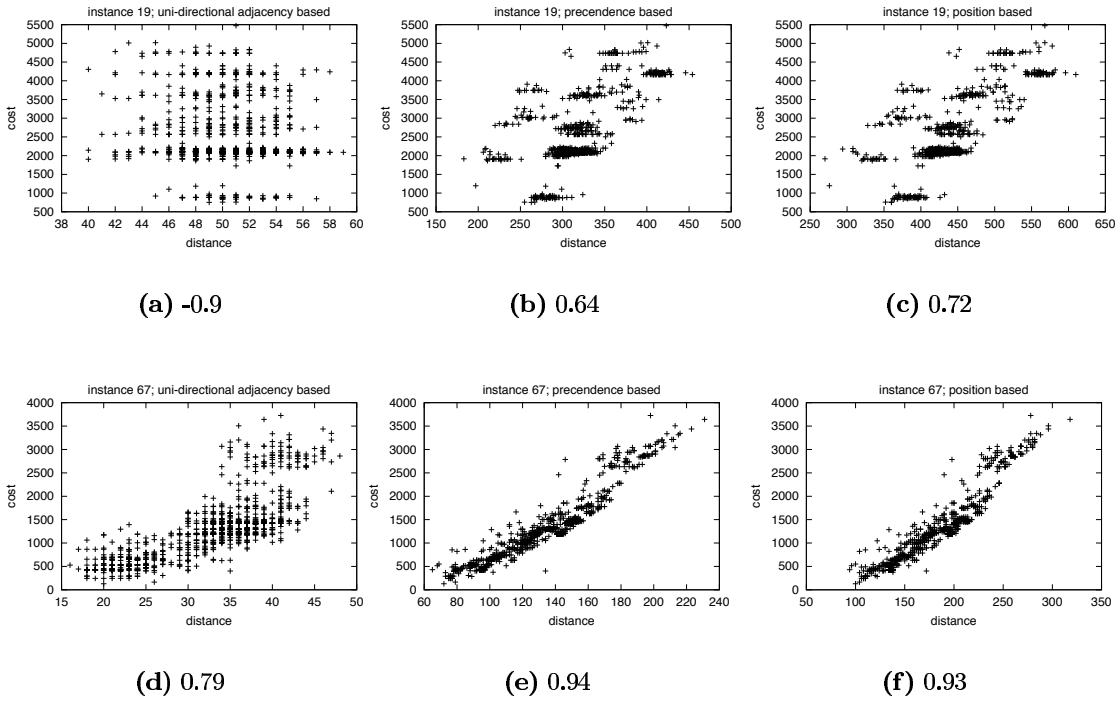
Observations

Along with the SMWTP benchmark, ORLIB provides a list with the best known total weighted tardiness for each instance. The sequences of jobs that constitute a solution which is assumed to be globally optimal are obtained by running the ILS algorithm until it reaches this best known total weighted tardiness value. For each instance 10 'global optima' are obtained this way. The local optima are obtained by running a local search 1000 times per instance. For each local optimum found, the distance to the nearest global optimum is computed together with the difference between the objective function values of the local and global optimum. None of the local optima for the instances considered in this section is also a global optimum.

The scatter plots in Figure 4.6 on the facing page show the dispersion of local optima over the search space of instance 19 and 67 for the three basic distance metrics. A distribution of local optima like in Figure 4.6(e) almost ideal, it seems to indicate a landscape topology resembling to a big valley (introduced on page 78). Hopping from one local optimum to the next is likely to bring the algorithm near the global optimum if the next optimum has a higher fitness. Correspondingly, the distances and the fitnesses of the local optima in the plot are highly correlated.

In Figure 4.6(b) and Figure 4.6(c) there seem to be clusters of local optima that are close in distance and cost. One could interpret such a distribution as reflecting a landscape that consists of a number of craters. An effective algorithm should be able to escape from these craters in order to arrive at a point nearer to the global optimum. This is considerably harder than searching through a big valley. In Section 3.5.2 instance 19 has already shown to be a peculiar instance where it proved best not to use heuristic information and the pheromone should best reflect the relative order of the heuristic. Probably, using heuristic information the ant are too much attracted towards a cluster and therefore the algorithm gets stuck at one of the clusters depicted in Figure 4.6(b). Even though the correlation of Figure 4.6(a) is worse than the correlation of the position based metric (Figure 4.6(c)), a pheromone representation reflecting the constraints of the adjacency based metric allowed for better performance (Figure 3.9 on page 63). From the point of view of the adjacency based metric the local optima are scattered around without forming clusters. Apparently, ants who adopt this point of view are better able to lead the local search out of the cluster. Having escaped the cluster the local search can then move on.

Table 4.4 on the next page gives the correlations between the distance defined by



(a) -0.9 **(b)** 0.64 **(c)** 0.72

(d) 0.79 **(e)** 0.94 **(f)** 0.93

Figure 4.6: Fitness distance correlations for local optima of instance 67 and 19 generated by local search on the interchange neighborhood. The correlation coefficients the correspond to the plots are given in the caption of the sub-figures.

	interchange l.opts				insert l.opts				inter+ins. l.opts			
	adj	prec	pos	Lpos	adj	prec	pos	Lpos	adj	prec	pos	Lpos
14	0.51	0.44	0.45	0.43	0.34	0.52	0.48	0.50	0.43	0.40	0.38	0.56
19	-0.09	0.64	0.72	0.57	0.43	0.68	0.67	0.68	0.63	0.80	0.82	0.77
38	0.73	0.77	0.77	0.80	0.62	0.52	0.53	0.29	0.71	0.62	0.60	0.80
42	0.27	0.78	0.84	0.62	0.32	0.61	0.60	0.56	0.23	0.52	0.58	0.22
67	0.79	0.94	0.93	0.92	0.46	0.85	0.84	0.83	0.80	0.94	0.93	0.92
71	0.29	0.83	0.88	0.86	0.38	0.78	0.79	0.79	0.60	0.86	0.87	0.86
86	0.57	0.87	0.92	0.46	0.49	0.79	0.78	0.71	0.66	0.89	0.91	0.57
92	0.73	0.95	0.97	0.96	0.41	0.82	0.81	0.82	0.66	0.97	0.98	0.97
117	0.42	0.95	0.96	0.95	0.26	0.83	0.80	0.80	0.50	0.98	0.99	0.98
120	0.65	0.95	0.96	0.96	0.48	0.80	0.78	0.78	0.75	0.94	0.95	0.94

Table 4.4: Fitness distance correlations for local optima generated by runs of local search starting at a random solution. Coefficients are computed for several metrics: *adj* denotes the adjacency based metric, *prec* the precedence based metric and *pos* the position based metric. *Lpos* is a position based metric that only counts the late jobs in the global best solution.

	left ins	*right ins*	*inter*	*invert*
TF	0.859	0.862	0.871	0.830
ILS	0.621	0.628	0.597	0.519
ACO	0.624	0.654	0.593	0.465

(a) ruggedness (correlation lengths) & problem characteristics

	adj	prec	pos	Lpos
TF	0.354	0.765	0.825	0.677
ILS	0.278	0.624	0.658	0.523
ACO	0.286	0.593	0.624	0.460

(b) fitness distance correlation & problem characteristics

Table 4.5: Correlations between fitness landscapes and problem characteristics. ILS is the log CPU time the ILS algorithm needs to solve the instance averaged over 100 tries, ACO is the same for the ACO algorithm for 25 tries.

some metric and the quality of the solution for the ten instances also investigated in Section 4.3.2. Results are given for local optima generated by local search on the interchange neighborhood, for local optima of insert local search, and for optima generated by putting these search procedures after each other. The differences in correlation between the sets of local optima are only possible because the sets contain different local optima. In addition to the random generation of initial solutions, differences of neighborhood cause different local optima to be found. The set generated by the interchange+insert local search will for example contain local optima that are closer to the global optimum than the sets of insert. In general the fitness distance correlation seems to be the weakest for the set generated by the insert local search. The local optima being relatively remote might perhaps be the cause.

When measuring the distance using the adjacency metric, in general a lower fitness distance correlation is obtained. This confirms the finding already done in Section 3.5.2 that the absolute, position based, representation is a better way to represent the problem than the relative representation. On the other hand, it remains vague whether one should have the local search operators that are approximated by the position based metric or the ones approximated by the precedence metric. Like the auto-correlation, the fitness distance correlation is first of all high, so no very sophisticated means to search the fitness landscape seem to be needed. However, one wonders how it can be the case that fitness distance correlations of 0.95 are measured for the local optima on instance 120, while the iterated local search algorithm requires on average the longest run time on just this instance (see Table A.4 on page 100).

4.4 Fitness landscapes & problem characteristics

In this section, I investigate if differences in landscape ruggedness and fitness distance correlations can explain the run times of the algorithms. In addition I try to identify the problem characteristics that lead to the emergence of certain landscapes.

Some correlations between fitness landscape properties and problem characteristic are given in Table 4.5. The fitness landscapes properties highly depend on the tardiness factor (TF), but bear less relationship with the effort needed by the search algorithms

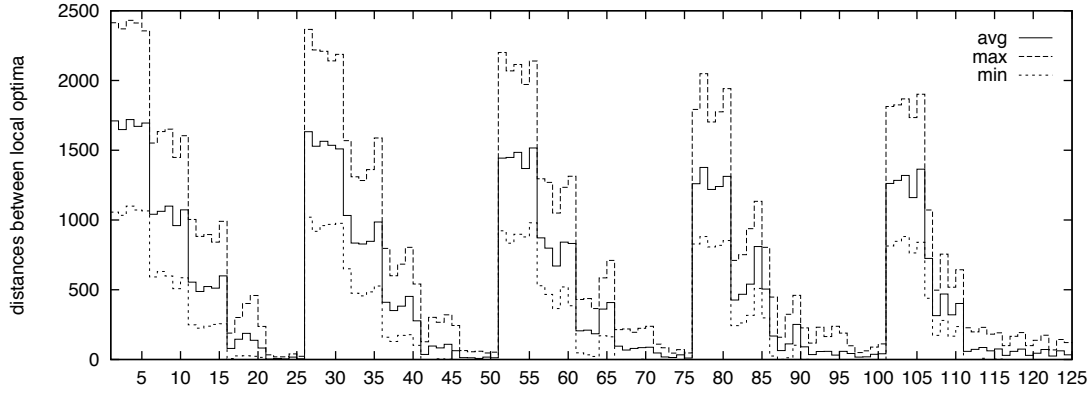


Figure 4.7: Precedence based distances between 1000 local optima generated by interchange+insert local search on random initial solutions for instance of the 100–job ORLIB benchmark set.

to solve the problem. The correlation between the fitness landscape properties and computational effort seems to be mainly due to the fact that effort is related to tardiness factor. In particular, the correlation in Table 4.5(a) would imply that smooth fitness landscapes with long correlation lengths are hard to solve for iterated local search or ant colony optimization. This is not the kind of relationship one would expect. Similarly Table 4.5(b) suggests that high correlations between distance and fitness cause the instance to be hard.

Apparently something goes wrong in the analysis. An explanation for the low autocorrelation for random walks through the search space of instances with a low tardiness factor might be that such instances have local optima scattered all over the search space, whereas instances with a high tardiness factor (TF) do not. This hypothesis seems to hold: $\rho(\mathrm{TF}, d^{avg}_{prec}) = -0.936$, indicating there is a strong negative correlation between TF and the average distance between local optima in the 125 100-job SMWTP instances of ORLIB, and $\rho(d^{avg}_{prec}, l_{interchange}) = -0.801$, which testifies that a large average distance coincides with a low correlation length for a random walk consisting of interchange moves.

It is quite obvious that landscape ruggedness and the probability to encounter local optima, which depends on the spread of the optima over the landscape, are connected. A landscape is rugged if there are sudden and or big differences between neighboring points in a landscape. If there is no local optimum nearby the landscape will probably resemble a plateau or a not so steep hill, no sudden changes are expected here. On the other hand, a local optimum is typically a place with sub–optimal points nearby. A move from the local optimum to the neighboring sub–optimum can be a sudden change.

Figure 4.7 gives the actual minimum, maximum and average distances for the 1000 local optima generated by application of interchange+insert local search on each of the instances. Considering that the maximum distance between two sequences is $n(n-1)/2 = 4950$ for the 100–job problems, on many instances the local optima seem to be all around. Moreover the figure confirms the relationship between the average distance

between local optima and the distribution of the due date. In instances where the due dates are near the latest possible completion time, most of the jobs will be non–late. Since the ordering of non–late jobs has no influence on the total weighted tardiness as long as the jobs complete before they are due, these instances will have many solutions which have the same value. When, in addition, the differences between the due dates are small, this effect will be even stronger because there are less due dates far before the average date which lies near the latest completion time. Considering the minimum distances in Figure 4.7 on the page before these solutions will be perceived as local optima scattered over the landscape defined by the precedence metric[1].

Likewise, the fact that global optima are scattered around on instances with a low TF value might disturb the predictive power of fitness distance correlations. The fitness–distance analysis of the distribution of the local optima assumes there is one global optimum which serves as reference point. The occurrence of multiple global optima is no problem if they are very close to the reference point. Unfortunately, the distances between the global optima are considerable for many instances, just like the distances between local optima. Actually, the average distance between the 10 global optima and the average distance between the 1000 local optima are highly correlated. Correlation coefficients like $\rho(d_{pos}^{avg}(gb), pos) = -0.882$, for the average distances between the ten global optima and the fitness distance correlation of the local optima when the position–based metric is used, support the hypothesis that the existence of multiple global optima makes fitness distance correlation useless as general indicator for the hardness of SMWTP instances.

Yet, the average distance between the global optima and the average distance between the local optima are themselves good indicators of the hardness of an instance: $\rho(d_{pos}^{avg}(gb), \log \text{ILS}) = -0.814$ and $\rho(d_{pos}^{avg}(gb), \log \text{ACO}) = -0.804$, where $d_{pos}^{avg}(gb)$ is the average position–based distance between the global optima, $\log$ ILS is the average logarithmic run time of the ILS algorithm and $\log$ ACO is the average logarithmic run time of the ACO algorithm. It seems the global optima being scattered over the landscape leads to a shattering of the search space into small regions with each a global optimum. The run time will often be reduced to searching this region, which explains why instances with scattered global optima are easy to solve.

4.5 Summary

The aim of this chapter has been to identify the factors that bring about hard instances of the single machine total weighted tardiness scheduling problem. Based on the analysis performed in the foregoing, the following explanation seems plausible.

The distribution of the due dates of the jobs is an important factor for the hardness of an instance. More in particular, the tardiness factor and the range of due dates together determine how many jobs will be late in near optimal solutions. The number of late jobs in turn has an effect on the number of local and global optima the instance

[1] For the position–based metric similar correlations between distance between local optima, ruggedness and TF have been perceived.

will contain and the extend in which these optima will be scattered over the landscape. The scattering leads to shattering, that is a partitioning of the search space into small regions. Random initial solutions will have a high likelihood to be near a global optimum and the search regions are faster searched than the entire search space. Therefore these instances will be easy.

In addition to the dispersion of the optima other factors contribute to the hardness/easiness of an instance. For example, instances differ in the number of runs near optimal sequences have when the are partitioned in arrays of subsequent late and non–late jobs. The local search algorithm presented in Chapter 3 could achieve enormous speedups for the evaluation of sequences that contain a few lengthy runs and therefore the algorithm's run–time is highly dependent on the number of runs.

Plots of the distribution of local optima over the search space can help explain the algorithms' performance on particular instances. However, as a general tool fitness distance correlation and landscape ruggedness do not seem to be particularly useful, these measures are too coarse to provide detailed information to guide the algorithm design and too specific for information on the entire set of instances.

5 Conclusion

This last chapter brings the report to completion. It summarizes the contributions, critically reflects the achievements of this thesis and points out related research issues that are worth further investigation.

5.1 Contributions

In this report, I have been concerned with the question whether ant colony optimization (ACO) forms a suitable approach to the single machine total weighted tardiness scheduling problem (SMWTP). I have dealt with this question in the following way. I started with the ACO algorithm that Bauer (1998) had developed for the single machine total tardiness problem (SMTTP). Some aspects of the algorithm design seemed peculiar, so I checked them and subsequently assessed the algorithm's quality (Chapter 2). I then extended the algorithm and tuned it for the SMWTP (Chapter 3). The SMWTP is significantly harder to solve to optimality with exact algorithms than the SMTTP. This fact makes this problem much more interesting for meta-heuristic approaches like ant colony optimization. Finally, in Chapter 4, I performed an analysis trying to relate properties of problem instances to the performance of the algorithms.

Early in the course of the investigation it turned out that the only challenging task for the SMWTP is finding the optimal solutions or best–known solutions on the instances for which the optimality has not yet been proven (the conjecture in this thesis is that these best–known solutions are optimal).

That is why I took the run times an algorithm needs to find the optimal solution as a measure for its performance. These run times were collected for a large number of publicly available problem instances. To put performance into perspective, the ACO algorithm was compared against an algorithm that is based on the iterated local search (ILS) meta-heuristic.

The main contributions of this thesis can be summarized as follows.

The single machine total tardiness problem (Chapter 2)

- I have come up with an improved ACO algorithm for the SMTTP which performs significantly better than the ACO approach of Bauer et al. (1999a) on which it is based.
- I have applied iterated local search to the SMTTP as well. This algorithm is among the best approximation algorithms for the SMTTP known to date.

- I have compared the approximation algorithms based on iterated local search and ant colony optimization against the state–of–the–art branch–and–bound algorithm of Swarc et al. (1998) for problems up to 400 jobs. On problems up to 300 jobs the exact algorithm is the most effective, for problems with more jobs the results indicate that the approximation algorithms might form an alternative for the exact algorithms as they are better capable to cope with the exponential growth of the search space.

The single machine total weighted tardiness scheduling problem (Chapter 3)

- I have developed some very powerful local search algorithms for the SMWTP by incorporating ideas of the variable neighborhood search meta-heuristic.
- I have extended the algorithms of Chapter 2 to the SMWTP and obtained algorithm that are competitive with state–of–the–art algorithms for the SMWTP.
- I have provided some new ideas for tuning of the algorithms and evaluated their success. More in particular, the idea to have a heterogeneous population of ants with each half of the ants applying a different local search has proven to be very useful.

Analysis (Chapter 4)

- I have investigated how the tardiness factor and the range of due dates of an SMWTP instance relate to the run times of the algorithms on these instances.
- I have revealed how the algorithms' run time on an instance is related to the structure of the optimal solution of that instance.
- I have mapped fitness landscapes induced by operators on instances of the SMWTP and linked the landscape topology to properties of the problem. Two elements of the landscape were investigated in detail: (i) the fitness distance correlation of a set of local optima relative to a global optimum, that is, the correlation between the difference of total weighted tardiness and the distance measured as the least number of local search moves needed to cover the locally optimal solution into the global optimum, and, (ii) the landscape ruggedness, that is, the degree in which the fitness of neighboring points in the landscape differ.

5.2 Discussion

In this thesis research, I have studied the application of ant colony optimization to single machine problems by means of *competitive testing*, that is, I tried to show that ant colony optimization 'beats' other meta-heuristics, iterated local search in particular, for a series of problems instances when the run times to find the optimal solution are measured. Competitive testing is widely used in the algorithmic research community.

Yet, criticism is rising. In his article "Testing heuristics: We have it all wrong", Hooker (1996) exposes the vices of competitive testing: it is hard to make fair comparisons between algorithms and to assemble realistic test problems. Competitive testing tells us which algorithm is faster but not why. Because it requires polished code, it consumes time and energy that could be spend doing more scientific experiments.

Objection 1 (Unfair competition) Competition between iterated local search and ant colony optimization is not fair since they differ in tuning effort invested. Adjusting parameters may make an algorithm more effective on a given set of problems. The difficulty is that is it not clear how much tuning is legitimate and how much tuning of adversary code is commensurate with the tuning applied to the proponent code. Therefore tuning presents a serious impediment to the fairness of the match.

Reply 1 Tuning is really the only test environment factor that might lead to unfair competition. The ILS and ACO were tested on the same computer and made use of the same operating system, language, compiler and compiler settings, even largely the same data-structures. I was not suddenly a much more skilled programmer while implementing one of the algorithms, so this latter aspect shouldn't pose any problem either.

A fair approach to the tuning dilemma would be not to tune at all or, on the other end of the spectrum, to exhaust the entire tuning potential of the meta-heuristic. Unfortunately, neither of these approaches is feasible in practice. Not to tune would require that there exist general–purpose parameter settings, but no developer will see any rationale for deliberately picking parameter settings that results in poor performance. In addition, it would require that design choices of how exactly to apply the meta-heuristic to a specific problem would be cooked up beforehand, yet these choices can not possibly be general purpose, since the only reason to believe an algorithm will do better than a random search is that its operators are correlated to the features of the problem (Wolpert and Macready 1995).

Exhaustive tuning is no option either, considering the sheer infinite number of valid modifications that can be made to the generic algorithmic scheme. The approach I followed is situated somewhere in between these extremes and documented in Chapter 3. As far as it can be determined, about the same effort was spend on iterated local search and ant colony optimization. Most time is spend on tuning the local search. This favors none of the algorithms in particular as both algorithms could take advantage of the local search algorithms. So, it is by extensive sharing of data-structures that the competition is kept fair.

Objection 2 (Unrealistic test problems) There are basically two problems with the randomly generated benchmark problems that were used in this thesis. First of all, they are randomly generated and therefore they are not likely to represent the real problem. Secondly, they are benchmark problems. Often benchmarks are biased and limited in scope. Competing algorithms which co-evolve around these problems might well loose the real problem out of sight as the are only evaluated on the benchmark instances.

Reply 2 To begin with the questions of bias and scope, these are less impending if one allows for random generation. If one generates the problem, one can influence the problem characteristics provided one knows which factors cause the nature of the problem. With tardiness factor (TF) and range of due dates (RDD) one can for instance influence the hardness of a problem instance to a great extend. In addition, one can randomly generate instances which resemble 'real problems'.

The TF and RDD values of this benchmark set range over all values that could possibly be of interest. The TF and RDD which may be encountered in real world SMWTP cases, are certain to fall within this range. The choice to generate five instances for each pair of TF and RDD in the set {0.2, 0.4, 0.6, 0.8, 1.0} results in a large majority of the instances being easy to solve. This distribution of problem hardness does probably not reflect that of real world cases. Yet, it is better than not having problems to solve at all.

Objection 3 (No insight) When algorithms compete, they are packed with the cleverest devices their authors can concoct and therefore differ in many respects. It is usually impossible to discern which of these devises are responsible for the differences in performance.

Reply 3 This problem is largely alleviated by the extensive documentation. Since the implementation decisions are documented in detail, the relative influence of the algorithms components can be determined.

What's more, I did spend a whole chapter trying to obtain insight (Chapter 4). This proved not a trivial exercise and still can be extended, yet yielded some valuable insight.

Objection 4 (Wrong allocation of time and resources) Competitive testing diverts time and energy from more productive experimentation.

Reply 4 Objection 4 assumes that no insight is obtained (which is Objection 3). Not that much time was spend of efficient coding. Most modifications to the algorithm did not require much time to implement. Moreover, this process of hacking around yielded new insights into the nature of the problem as well as deeper understanding of the algorithms' behavior. So, in that sense at least, the experimentation described in this report has been productive. Besides, to repeat Reply 3, time and resources, indeed a whole chapter, has been allocated to 'productive experimentation'.

Scientific testing Hooker (1996) proposes 'scientific testing' as an alternative to competitive testing. In an earlier publication, Hooker (1994) explains this methodology thus:

> In its early stages, an empirical study of an algorithm might well involve nothing more than running a few tests to see what happens. But after a while one develops an informed hunch about what is likely to affect performance. This is a *hypothesis*. The hypothesis is then tested empirically

> using time-honored techniques of experimental design and statistical analysis. Eventually one may put together a unifying picture that appears to explain why certain factors are important. This is a *theory*. From the theory one can deduce consequences that can be put to the test.

On first sight this description corresponds perfectly to what I have been doing in this thesis. The "early stages" are covered in Chapter 3 and the subsequent hypothesis testing is done in Chapter 4. On page 84, I even tried to put together a unifying picture that might explain why certain factors are important. On the other hand, Objection 4 on the facing page is not totally unjustified in that by far the most attention has been paid to the "early stages". Still, the hypothesis testing has been very productive because I could take advantage of the insights obtained by competitive testing and tuning.

5.3 Future work

New questions have risen from the observations made in this thesis and the critical reflection on the approach in the previous section. Possible extensions of this work are outlined below.

The single machine total tardiness problem

- Without doubt, the algorithms for the SMTTP can be improved. It would be interesting to test these algorithms on SMTTP instances of 300 jobs and more and identify under what circumstances state–of–the–art exact algorithms for the SMTTP can be outperformed.

The single machine total weighted tardiness scheduling problem

- In addition to what has already been tested in Chapter 3, there is still a number of things that is worth trying.
  - local search
    - ▷ Incorporate speedups like the candidate list. Instead of checking the whole neighborhood at each local search step, one could only consider the top n moves that are the most promising according to the heuristic information. If local search is combined with an ACO algorithm, one could use the ants' graph to provide the heuristic information.
    - ▷ Explore other ways to combine the neighborhoods. The best performance was in Chapter 3 achieved by preceding insert descent with interchange descent. Other combinations of the interchange and insert operators could possibly yield even better performance.
    - ▷ Try dynasearch. Good performance on the SMWTP has been achieved by iterated dynasearch (Congram et al. 1998). Dynasearch could be used in stead of the variable neighborhood search which is now used as local

search in the ILS and ACO algorithm. An open question for future research is how use of dynasearch would affect the performance of ILS and ACO.

  - iterated local search
    - ▷ One could use a more problem specific kick mechanism. For example, apply the kick only to the late jobs. It is likely that a kick–move would then tend to be more disruptive, so less kick–moves would be needed in such a case.
  - ant colony optimization
    - ▷ The ACO algorithm was particularly successful when one type of local search was applied to half of the colony's solutions and another type of local search was applied to the other half. One could push this idea of an heterogeneous colony further by letting more local search variants be at the colony's disposition or by letting ants use different heuristic information.
    - ▷ Similarly, a self–tuning ACO algorithm which uses search feedback to adjust its configuration to the particular instance currently being solved may be interesting.

- To get a better impression of the quality of the ACO and ILS algorithms presented in this thesis, they should be tested on larger SMWTP instances and extended to a wider range of single machine scheduling problems.

- Finally, configuration ideas that have proved to be useful on the SMWTP, like that of a heterogeneous colony, might also result in improvements for ACO algorithms in other applications where a strong instance dependence of the best algorithm configuration has been noted.

Analysis

- An analysis of the SMTTP like the one done for the SMWTP would possibly clarify the relationship between these problem–types.

- The occurrence of hardest instance at specific values of the tardiness factor and range of due dates suggest there might exist of a kind of phase transition at a critical point of the parameters used in the instance generation in the following sense: When approaching the critical point from one side, the hardness increases, reaching a maximum at the critical point and if this critical point is passed, the instances become easier to solve again. Hogg, Huberman and Williams (1996) observed this phenomenon in many other combinatorial optimization problems.

- Factors that determine the hardness of an instance in addition to tardiness factor and range of due dates should be identified. For example, one could consider whether the range of weights would have an impact on the harness of the solution.

In general, I think that Hooker is right in saying that we should shift our attention towards more 'scientific testing'. Still, prior to testing hypotheses and building theories, the collection of observations about the behavior of algorithms will continue to be needed. Competitive testing of meta-heuristics fulfills this need, therefore it will stay an important part in future research and the ongoing project to study the application of ant colony optimization and iterated local search to even more combinatorial optimization problems should/will continue.

Appendix

A. Summary results of ILS for the SMWTP

In this appendix summary results are given for the tests described in Section 3.4.2. The aim of these tests is to configure iterated local search (ILS) for the single machine total weighted tardiness scheduling problem. The tests focus on 10 100–job instances out of

	14	*19*	*38*	*42*	*67*	*71*	*86*	*92*	*117*	*120*
tardiness factor	0.6	0.8	0.6	0.8	0.8	0.9	0.6	0.8	0.75	0.75
range of due dates	0.2	0.2	0.4	0.4	0.5	0.3	0.8	0.6	0.7	0.5

Table A.1.: Characterization of the SMWTP instances

the 125 which are available in ORLIB. The instances are the $14th$, the $19th$, the $38th$, the $42th$, the $67th$, the $71th$, the $86th$, the $92th$, the $117th$ and the $120th$ instance. Table A.1 gives some statistics to characterize these instances.

Table A.2 compares local search algorithms embedded in ILS. in Table A.3 on the next page results are given for ILS variant that differ in the number of kicks that is applied to modify the local search solution. Table A.4 on page 100 gives for the summary results for the tests described in Section 3.6.

Table A.2.: Efficiency of local search neighborhoods in the context of ILS which modifies the local search solutions by means of 6 random left insert moves. All local search procedures use the best improvement pivoting rule. 25 tests with a computation bound of 300 seconds are performed on each instance. Given are, out of 25 tests, the shortest run time (t_{min}), the longest run time (t_{max}) and the average run time (t_{avg}) to solve the instance. In addition the run times' standard deviation (σ_t) and the number of tests that resulted in a globally optimal solution (n_{opt}) are given.

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
			INTERCHANGE						INTERCHANGE		
14	6.82	265.61	139.71	98.88	12	*19*	0.39	27.04	9.52	7.61	25
38	0.14	0.66	0.35	0.16	25	*42*	0.54	188.10	36.39	41.74	25
67	0.87	159.56	29.95	48.91	25	*71*	0.53	215.24	17.28	46.40	25
86	1.11	38.80	7.82	8.97	25	*92*	0.22	3.28	1.21	0.87	25
117	0.23	55.81	11.11	14.99	25	*120*	3.36	232.98	106.13	77.30	19
			RIGHT INSERT						RIGHT INSERT		
14	1.58	14.11	7.43	6.31	3	*19*	1.98	268.39	36.73	59.74	23
38	0.78	151.84	19.27	37.94	25	*42*	151.26	151.26	151.26	0.00	1
67	32.52	235.24	91.23	65.09	16	*71*	9.32	184.90	44.06	55.72	9
.....						.....					

Table A.2.: (continued)

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
86	25.23	267.78	130.06	86.16	16	*92*	6.77	297.28	75.75	77.43	25
117	4.92	78.75	22.26	19.11	25	*120*	25.72	158.71	80.79	42.84	20
		INTERCHANGE+INSERT						INTERCHANGE+INSERT			
14	35.04	215.50	128.63	79.43	6	*19*	0.47	61.45	11.84	12.30	25
38	0.41	2.25	0.91	0.49	25	*42*	0.72	28.49	3.82	5.59	25
67	0.89	256.12	21.87	52.03	25	*71*	0.72	204.05	22.21	50.91	24
86	0.31	8.05	2.68	2.24	25	*92*	0.30	0.32	0.31	0.01	25
117	0.35	32.62	3.45	6.64	25	*120*	1.94	149.63	53.21	43.88	25
		INSERT+INTERCHANGE						INSERT+INTERCHANGE			
14	0.00	0.00	0.00	0.00	0	*19*	2.08	267.74	88.46	82.07	25
38	1.44	22.40	3.75	4.39	24	*42*	225.57	225.57	225.57	0.00	1
67	2.18	252.32	126.93	85.69	9	*71*	8.12	182.02	59.91	50.54	25
86	16.16	191.88	87.15	51.42	21	*92*	2.25	56.41	16.53	13.31	25
117	3.66	40.79	14.13	9.62	25	*120*	13.51	270.02	93.07	74.70	25
		$\kappa = 50$ INTERCHANGE+INSERT						$\kappa = 50$ INTERCHANGE+INSERT			
14	195.19	296.63	245.91	71.73	2	*19*	1.45	125.68	39.65	32.80	25
38	109.41	135.03	122.22	18.12	2	*42*	0.69	6.77	2.23	1.59	25
67	0.85	104.93	10.01	21.49	25	*71*	0.49	185.67	15.06	37.94	25
86	0.51	12.22	2.93	3.19	25	*92*	0.28	0.29	0.29	0.01	25
117	0.36	24.52	2.19	5.10	25	*120*	2.32	202.40	44.66	52.57	25

Table A.3.: Number of kicks for ILS with a left insert kick and interchange+insert local search. The kick strength is given in ROMAN. ILS is run 25 times with a CPU bound of 600 seconds. See Table A.2 on the preceding page for a description of the entries.

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
			II						II		
14	8.00	8.00	8.00	0.00	1	*19*	1.57	264.44	53.13	64.54	25
38	0.47	4.31	1.22	0.99	25	*42*	0.86	79.03	7.65	15.77	25
67	0.95	237.21	34.19	67.21	25	*71*	0.50	149.20	13.73	32.46	23
86	0.33	18.66	3.99	5.19	25	*92*	0.31	0.33	0.31	0.01	25
117	0.29	17.01	1.23	3.30	25	*120*	6.61	133.97	54.20	36.43	25
			IV						IV		
14	158.53	236.19	197.36	54.91	2	*19*	0.88	34.92	11.97	9.58	25
38	0.41	3.06	0.91	0.69	25	*42*	0.84	17.42	5.39	4.44	25
67	1.01	157.68	24.78	40.02	25	*71*	0.57	59.47	8.59	13.35	23
86	0.53	15.21	3.93	4.08	25	*92*	0.31	0.32	0.31	0.00	25
117	0.31	2.53	0.75	0.58	25	*120*	2.52	171.02	36.29	36.80	25
			VI						VI		
14	35.04	215.50	128.63	79.43	6	*19*	0.47	61.45	11.84	12.30	25
38	0.41	2.25	0.91	0.49	25	*42*	0.72	28.49	3.82	5.59	25
67	0.89	256.12	21.87	52.03	25	*71*	0.72	204.05	22.21	50.91	24
.....						.....					

Table A.3.: (continued)

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
86	0.31	8.05	2.68	2.24	25	*92*	0.30	0.32	0.31	0.01	25
117	0.35	32.62	3.45	6.64	25	*120*	1.94	149.63	53.21	43.88	25
			VIII						VIII		
14	1.29	225.11	87.19	75.61	23	*19*	0.59	29.12	8.08	7.12	25
38	0.44	1.82	0.87	0.38	25	*42*	0.76	11.28	4.08	2.84	25
67	0.89	46.59	8.53	12.37	25	*71*	1.06	120.85	19.36	36.02	24
86	0.34	4.78	1.80	1.27	25	*92*	0.31	0.32	0.31	0.00	25
117	0.33	13.65	2.30	3.06	25	*120*	4.20	197.96	42.11	52.13	25
			X						X		
14	1.00	172.40	54.79	53.00	25	*19*	0.76	25.65	8.20	6.90	25
38	0.42	1.64	0.70	0.32	25	*42*	0.38	11.02	2.56	2.34	25
67	1.00	69.45	18.21	22.04	25	*71*	0.40	166.76	21.17	38.64	23
86	0.38	6.62	1.89	1.66	25	*92*	0.31	0.32	0.31	0.00	25
117	0.33	16.77	3.85	4.56	25	*120*	4.53	236.80	56.34	66.62	25
			XII						XII		
14	1.64	124.67	26.55	29.19	25	*19*	0.60	17.41	5.60	4.04	25
38	0.43	1.63	0.77	0.34	25	*42*	0.54	10.19	3.97	2.99	25
67	1.45	103.68	16.43	22.48	25	*71*	0.39	202.80	25.99	49.49	24
86	0.40	7.41	1.92	1.68	25	*92*	0.31	0.32	0.31	0.01	25
117	0.33	17.49	3.82	4.86	25	*120*	5.05	251.40	69.42	67.29	25
			XIV						XIV		
14	1.02	88.19	16.66	18.67	25	*19*	0.40	43.29	5.79	8.59	25
38	0.43	1.85	0.75	0.32	25	*42*	0.41	6.56	3.06	1.90	25
67	1.13	122.48	24.32	28.75	25	*71*	1.00	271.57	66.25	79.89	24
86	0.37	6.23	1.85	1.44	25	*92*	0.31	0.32	0.31	0.00	25
117	0.32	36.06	4.56	7.83	25	*120*	1.82	287.61	90.51	77.11	25
			XVI						XVI		
14	0.60	28.18	8.20	7.40	25	*19*	0.55	21.81	7.21	6.39	25
38	0.42	1.79	0.86	0.44	25	*42*	0.54	11.04	3.69	3.06	25
67	0.83	145.84	26.37	33.57	25	*71*	1.47	271.48	51.82	72.15	25
86	0.40	4.67	1.53	0.98	25	*92*	0.31	0.32	0.31	0.00	25
117	0.36	31.76	7.33	8.42	25	*120*	2.38	285.63	85.85	71.87	24
			XVIII						XVIII		
14	0.78	17.64	6.51	4.90	25	*19*	0.55	55.43	11.52	11.37	25
38	0.44	1.77	0.89	0.41	25	*42*	0.67	8.62	2.98	2.27	25
67	3.20	276.99	48.02	66.32	25	*71*	1.80	233.25	69.73	64.72	24
86	0.40	7.84	1.71	1.70	25	*92*	0.31	0.32	0.31	0.00	25
117	0.33	34.93	5.05	8.07	25	*120*	7.02	278.50	101.10	72.78	24
			XX						XX		
14	0.80	28.86	5.80	6.69	25	*19*	0.80	45.28	11.15	10.91	25
38	0.45	1.75	0.87	0.39	25	*42*	0.84	16.59	4.59	3.52	25
67	1.86	256.91	55.18	57.04	25	*71*	5.75	283.49	77.39	84.76	19
86	0.45	5.08	2.40	1.51	25	*92*	0.31	0.37	0.32	0.01	25

Table A.3.: (continued)

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
117	0.90	29.19	7.28	8.40	25	*120*	8.55	291.12	125.49	94.45	20

Table A.4.: Summary results on performance of final ILS algorithm on 100–job and 200–job SMWTP instances (see Section 3.6). The algorithm is run 100 times per instance with a bound of 600 seconds on the 100–job instances, 25 times with 3600 seconds on the 200–job problems. Given are the shortest run time (t_{min}), the longest run time (t_{max}), the average run time (t_{avg}) to solve the instance,and the run times' standard deviation (σ_t). In addition these statistics are given for the number of iterations, that is, i_{min} for the shortest number of iteration, i_{max} for the maximum, i_{avg} the average, and σ_t the standard deviation. Finally, the number of tests that resulted in a globally optimal solution (n_{opt}) is given.

	t_{min}	t_{max}	t_{avg}	σ_t	i_{min}	i_{max}	i_{avg}	σ_i	n_{opt}
100:9	0.20	0.21	0.20	0.00	0	0	0.00	0.00	100
100:19	0.75	33.51	7.73	6.51	5	224	59.67	45.43	100
100:44	0.66	13.93	3.37	2.46	1	75	18.92	13.82	100
100:45	1.11	195.19	35.54	33.99	11	1493	278.67	258.71	100
100:48	0.44	309.31	57.08	59.21	2	1625	301.80	310.87	100
100:66	0.54	43.45	4.70	6.71	1	230	27.48	37.58	100
100:67	0.93	105.79	11.92	17.28	4	639	75.25	105.22	100
100:71	0.47	698.72	63.30	106.99	3	5245	484.69	803.68	100
100:88	2.18	64.22	17.62	13.06	17	380	111.87	75.73	100
100:93	0.64	36.96	7.28	7.80	2	213	46.15	45.33	100
100:98	0.32	65.01	12.93	14.19	1	548	114.45	120.02	100
100:112	1.20	53.38	15.55	12.65	9	285	88.39	65.44	100
100:114	1.03	39.95	8.81	7.97	7	265	61.96	50.69	100
100:116	0.32	63.51	14.61	15.33	1	395	96.41	95.79	100
100:118	0.56	151.11	16.32	24.47	1	908	100.23	149.04	100
100:120	2.33	809.21	146.95	150.84	17	4463	812.80	827.65	100
100:122	6.19	718.47	129.55	133.74	78	7421	1349.01	1379.40	100
100:123	0.37	208.73	25.87	39.02	1	1261	159.81	236.88	100
200:11	4.19	48.03	15.95	10.62	1	56	18.34	13.98	25
200:12	4.11	576.89	61.63	126.81	2	464	52.64	103.40	25
200:13	7.28	323.04	92.71	79.12	4	278	78.32	67.38	25
200:14	7.08	65.83	20.89	13.94	2	64	18.60	13.88	25
200:15	6.09	47.77	19.63	13.12	3	57	21.48	16.49	25
200:16	3.34	10.03	5.13	1.79	2	12	4.60	3.00	25
200:17	8.17	363.35	58.24	76.19	5	312	51.40	66.31	25
200:18	25.46	621.21	234.34	148.40	29	568	232.60	136.86	25
200:19	8.86	83.64	29.39	19.92	11	94	33.40	22.79	25

B. Summary results of ACO for the SMWTP

This appendix gives the summary results for the tests described in Section 3.5.2 for the configuration of an ant colony optimization (ACO) algorithm for the single machine total weighted tardiness scheduling problem. The tests focus on ten 100–job instances (described on 97). To test a configuration the algorithm is run 25 times with a CPU bound of 300 seconds on the instance.

The initial ACO algorithm for the SMWTP has the following parameter settings: $\alpha = 1$, $\beta = 2$, $\rho = 0.1$, $q_0 = 0.9$. The colony contains 20 ants. Local search is applied to all solutions these ants construct and the AU dispatching rule provides the heuristic information. The initial solution is formed by a sequence sorted in non–decreasing AU order on which a local search is applied. Further details on the implementation are given in Section 2.4.2.

Table B.1.: Results for local search variants embedded in the ACO algorithm. See Table A.2 on page 97 for a description of the entries.

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
			INSERT						INSERT		
14	58.07	242.42	127.99	76.52	5	*19*	0.00	0.00	0.00	0.00	0
38	60.63	186.38	109.82	39.77	17	*42*	20.46	304.63	130.31	93.75	20
67	27.59	195.40	72.12	53.55	10	*71*	131.86	309.58	206.15	76.40	7
86	14.01	138.50	59.83	32.59	25	*92*	24.27	171.01	91.21	48.52	25
117	77.04	304.91	185.49	61.47	21	*120*	93.11	261.23	170.77	52.76	13
			INSERT+INTERCHANGE						INSERT+INTERCHANGE		
14	13.21	292.30	115.70	87.42	21	*19*	0.00	0.00	0.00	0.00	0
38	22.69	176.31	65.94	38.56	25	*42*	10.43	20.70	14.63	4.46	25
67	8.73	41.65	21.75	8.79	25	*71*	16.51	287.50	104.36	65.64	21
86	13.39	128.12	33.11	24.61	25	*92*	13.33	32.74	18.04	5.54	25
117	16.67	81.26	30.63	13.40	25	*120*	14.15	157.25	44.03	37.51	25
			INTERCHANGE+INSERT						INTERCHANGE+INSERT		
14	24.93	245.83	137.49	85.44	13	*19*	0.00	0.00	0.00	0.00	0
38	9.64	148.22	55.30	44.53	24	*42*	3.29	4.49	3.92	0.27	25
67	1.92	238.17	24.42	49.52	23	*71*	3.81	208.94	61.91	61.02	22
86	3.99	4.88	4.47	0.23	25	*92*	3.05	3.61	3.35	0.15	25
117	36.83	250.43	163.88	74.93	9	*120*	3.36	90.75	21.55	18.48	25
			HALF/HALF						HALF/HALF		

Table B.1.: (continued)

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
14	62.24	292.47	130.89	65.12	15	*19*	0.00	0.00	0.00	0.00	0
38	7.70	8.76	8.12	0.27	25	*42*	7.99	67.44	17.48	13.40	25
67	16.66	134.38	42.89	29.04	24	*71*	6.32	171.64	37.61	50.39	25
86	8.90	55.99	20.16	13.63	25	*92*	6.43	27.45	12.77	4.54	25
117	6.87	94.86	17.32	18.79	25	*120*	8.43	58.48	25.05	14.15	25

Table B.2.: Investigation into the ideal colony size for an ACO algorithm with *half/half* local search. The colony size is given in ROMAN. See Table A.2 on page 97 for a description of the entries.

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
			II						II		
14	6.37	251.72	89.64	72.96	22	*19*	0.00	0.00	0.00	0.00	0
38	0.74	3.22	1.31	0.70	25	*42*	1.21	273.35	29.76	57.98	25
67	1.22	166.07	53.94	57.09	20	*71*	1.38	88.36	18.27	24.98	22
86	1.03	105.27	24.01	25.47	25	*92*	1.13	32.83	11.92	7.96	25
117	0.79	58.34	11.33	13.31	25	*120*	2.93	152.58	26.49	33.67	25
			VI						VI		
14	8.84	286.54	133.16	100.33	13	*19*	0.00	0.00	0.00	0.00	0
38	2.15	4.68	2.59	0.63	25	*42*	2.26	27.48	9.61	7.37	25
67	3.35	251.11	42.88	56.21	22	*71*	1.72	47.39	8.68	10.54	22
86	2.57	50.11	20.95	15.27	25	*92*	1.98	16.45	7.94	3.84	25
117	2.11	46.44	10.93	10.17	25	*120*	4.53	129.98	24.79	32.54	25
			X						X		
14	10.39	247.52	102.69	68.78	23	*19*	0.00	0.00	0.00	0.00	0
38	3.61	8.01	4.61	1.46	25	*42*	3.82	25.70	9.25	6.15	25
67	4.79	230.87	48.92	62.39	22	*71*	3.15	128.21	31.56	40.29	24
86	4.24	71.67	21.20	16.08	25	*92*	2.59	36.22	15.05	8.52	25
117	3.38	27.83	8.64	5.80	25	*120*	4.15	110.05	21.56	22.74	25
			XX						XX		
14	62.24	292.47	130.89	65.12	15	*19*	0.00	0.00	0.00	0.00	0
38	7.70	8.76	8.12	0.27	25	*42*	7.99	67.44	17.48	13.40	25
67	16.66	134.38	42.89	29.04	24	*71*	6.32	171.64	37.61	50.39	25
86	8.90	55.99	20.16	13.63	25	*92*	6.43	27.45	12.77	4.54	25
117	6.87	94.86	17.32	18.79	25	*120*	8.43	58.48	25.05	14.15	25

Table B.3.: (continued)

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}

Table B.3.: Summary results for additional tests on which are presented in Figure 3.8 on page 62, Figure 3.9 on page 63 and Figure 3.10 on page 64. A description of the variants can be found in the caption of these figures. See Table A.2 on page 97 for a description of the entries.

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
			CANDIDATE LIST						CANDIDATE LIST		
14	4.59	258.10	105.03	91.33	20	*19*	0.00	0.00	0.00	0.00	0
38	3.05	3.70	3.36	0.19	25	*42*	2.90	5.85	3.69	1.05	25
67	6.68	118.53	27.16	28.88	24	*71*	1.83	12.14	3.92	2.28	25
86	3.74	49.08	14.97	14.23	25	*92*	2.19	18.93	6.63	3.72	25
117	2.53	21.30	8.68	6.60	25	*120*	2.72	105.61	17.68	20.20	25
			ALT. UPDATE						ALT. UPDATE		
14	8.08	269.92	122.24	84.53	14	*19*	0.00	0.00	0.00	0.00	0
38	2.97	3.73	3.29	0.19	25	*42*	9.64	65.35	24.58	16.78	25
67	3.74	273.45	95.80	96.28	16	*71*	2.43	15.65	8.11	4.09	19
86	3.42	224.29	40.37	52.64	25	*92*	5.62	12.49	7.79	2.25	25
117	2.98	108.56	24.98	24.41	25	*120*	6.67	75.78	20.93	15.39	25
			ALT. ACTION CHOICE						ALT. ACTION CHOICE		
14	32.85	297.57	188.32	96.50	15	*19*	0.00	0.00	0.00	0.00	0
38	4.36	5.11	4.58	0.19	25	*42*	4.00	222.30	56.06	56.98	25
67	4.80	291.40	125.70	146.62	7						
			ADJACENCY CONSTRAINT						ADJACENCY CONSTRAINT		
14	7.79	277.83	94.91	76.85	21	*19*	55.39	252.49	149.89	74.48	5
38	6.30	7.48	7.10	0.32	25	*42*	151.55	294.15	214.66	61.91	4
67	0.00	0.00	0.00	0.00	0	*71*	6.54	291.70	98.95	91.96	16
86	3.87	23.02	9.23	6.34	25	*92*	17.22	62.97	37.11	20.54	5
117	9.42	254.59	107.60	67.63	23	*120*	0.00	0.00	0.00	0.00	0
			NO PHEROMONE						NO PHEROMONE		
14	7.08	77.66	31.91	23.37	25	*19*	0.00	0.00	0.00	0.00	0
38	5.43	11.64	6.21	1.16	25	*42*	8.48	229.17	88.03	122.56	3
67	296.58	296.58	296.58	0.00	1	*71*	8.40	290.67	83.15	81.28	24
86	0.00	0.00	0.00	0.00	0	*92*	30.40	267.11	136.01	79.90	10
117	7.73	260.95	112.67	82.17	22	*120*	39.91	276.95	120.75	88.75	10
			NO HEURISTIC						NO HEURISTIC		
14	2.05	26.02	9.23	5.62	25	*19*	10.05	181.40	69.20	52.93	24
38	2.12	11.85	4.75	2.61	25	*42*	2.72	169.11	28.67	41.82	25
67	6.28	145.22	36.20	39.91	23	*71*	2.20	230.52	35.35	61.26	25
86	14.88	135.20	54.23	28.89	25	*92*	3.42	66.31	19.44	17.19	25
117	3.27	98.50	31.18	27.68	25	*120*	3.19	68.42	23.79	17.55	25
			EDD						EDD		

Table B.3.: (continued)

	t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}		t_{min}	t_{max}	t_{avg}	σ_t	n_{opt}
14	3.15	35.49	14.08	8.42	25	*19*	16.54	242.22	78.32	67.77	12
38	2.70	5.87	3.43	0.59	25	*42*	38.91	272.41	125.00	60.57	25
67	22.05	82.08	44.66	28.73	4	*71*	12.63	261.53	118.18	76.40	21
86	4.55	37.32	11.86	7.31	25	*92*	61.34	303.43	158.72	79.13	19
117	0.00	0.00	0.00	0.00	0	*120*	0.00	0.00	0.00	0.00	0
			MDD						MDD		
14	2.79	20.87	9.40	5.23	25	*19*	20.29	275.44	103.32	84.67	24
38	1.92	9.42	3.35	1.66	25	*42*	3.34	22.47	12.76	5.77	25
67	7.95	198.29	32.45	42.34	25	*71*	3.00	114.79	14.43	21.88	25
86	6.02	140.12	26.67	26.90	25	*92*	5.74	51.04	16.39	11.11	25
117	2.57	62.23	9.77	13.37	25	*120*	2.98	86.21	18.73	16.95	25

Table B.4.: Summary results on the performance of the final ACO algorithm on 100–job and 200–job SMWTP instances (see Section 3.6). The algorithm is run 25 times with a CPU bound of 600 seconds for the 100–job instances and 3600 seconds for the 200–job instances. See Table A.4 on page 100 for a description of the entries.

	t_{min}	t_{max}	t_{avg}	σ_t	i_{min}	i_{max}	i_{avg}	σ_i	n_{opt}
100:9	3.19	189.89	40.96	43.85	1	65	14.04	15.10	25
100:19	11.32	689.75	86.26	131.25	3	202	25.04	38.49	25
100:44	9.01	128.21	42.34	25.77	2	43	13.72	8.89	25
100:45	4.87	19.96	10.77	4.20	2	9	4.64	2.06	25
100:46	2.35	18.06	6.36	4.35	1	9	2.76	2.17	25
100:66	4.41	335.24	118.63	81.10	1	99	34.92	24.17	25
100:67	7.70	422.22	37.49	82.13	2	135	12.00	26.40	25
100:71	2.44	115.28	22.25	31.23	1	59	10.96	16.08	25
100:88	10.89	253.10	82.41	71.26	3	84	27.44	24.69	25
100:93	4.51	132.94	38.65	30.53	1	44	12.16	10.12	25
100:98	2.09	6.94	3.94	1.43	1	3	1.60	0.65	25
100:112	6.09	46.15	21.00	11.18	2	19	8.04	4.55	25
100:114	6.21	80.43	22.08	17.74	2	30	8.00	6.74	25
100:116	4.98	174.71	31.57	35.25	2	90	15.76	18.27	25
100:118	7.03	193.79	52.17	49.06	2	68	17.64	17.18	25
100:120	6.27	78.11	17.54	15.02	2	26	5.64	5.10	25
100:122	3.57	61.42	21.61	13.93	2	37	12.36	8.36	25
100:123	3.21	30.78	12.32	6.17	1	13	4.64	2.56	25
200:11	974.24	3204.20	2089.22	1576.82	23	81	52.00	41.01	2
200:12	117.32	2867.68	1139.69	1504.84	2	53	21.00	27.87	3
200:13	119.18	1007.02	330.61	170.17	2	17	5.68	2.88	25
200:14	70.82	402.46	132.03	107.53	1	6	1.88	1.64	25
200:15	48.72	232.31	117.93	40.34	1	5	2.44	0.87	25
200:16	81.95	1578.21	394.84	320.69	2	42	10.58	8.59	24
200:17	90.89	413.30	149.74	74.59	2	9	3.36	1.68	25

Table B.4.: (continued)

	t_{min}	t_{max}	t_{avg}	σ_t	i_{min}	i_{max}	i_{avg}	σ_i	n_{opt}
200:18	228.13	3364.63	1348.15	1071.42	6	86	34.50	27.54	6
200:19	43.61	869.60	270.36	191.69	1	19	5.84	4.17	25

Bibliography

Aarts, E. H. L. and Lenstra, J. K. (eds): 1997, *Local Search in Combinatorial Optimization*, John Wiley & Sons, Chichester.

Abdul-Razaq, T. S., Potts, C. N. and Van Wassenhove, L. N.: 1990, A survey of algorithms for the single machine total weighted tardiness scheduling problem, *Discrete Applied Mathematics* **26**, 235–253.

Alidaee, B. and Gopalan, S.: 1997, A note on the equivalence of two heuristics to minimize total tardiness, *European Journal of Operational Research* pp. 514–517.

Anderson, E. J., Glass, C. A. and Potts, C. N.: 1997, Machine scheduling, *in* E. H. L. Aarts and J. K. Lenstra (eds), *Local Search in Combinatorial Optimization*, John Wiley & Sons, chapter 11, pp. 361–414.

Barr, R. S., Golden, B. L., Kelly, J. P., Resende, M. G. C. and Steward, W. R.: 1995, Designing and reporting on computational experiments with heuristic methods, *Journal of Heuristics* **1**(1), 9–32.

Bauer, A.: 1998, *Ant colony optimization für das single machine total tardiness Problem*, Master's thesis, Department of Management Science, University of Vienna, Vienna, Austria.

Bauer, A., Bullnheimer, B., Hartl, R. F. and Strauss, C.: 1999a, An ant colony optimization approach for the single machine total tardiness problem, *Proceedings of the 1999 Congress on Evolutionary Computation (CEC'99)*, IEEE Press, Piscataway, NJ, pp. 1445–1450.

Bauer, A., Bullnheimer, B., Hartl, R. F. and Strauss, C.: 1999b, Minimizing total tardiness on a single machine using ant colony optimization, *POM Working Paper 5/99*, Department of Management Science, University of Vienna, Vienna, Austria.

Baum, E. B.: 1986, Iterated descent: A better algorithm for local search in combinatorial optimization problems, Unpublished manuscript.

Beasley, J. E.: 1990, OR library: Distributing test problems by electronic mail, *Journal of the Operational Research Society* **41**, 1069–1072.

Ben-Daya, M. and Al-Fawzan, M.: 1996, A simulated annealing approach for the one–machine mean tardiness scheduling problem, *European Journal of Operational Research* **93**, 61–67.

Bentley, J. L.: 1992, Fast algorithms for geometric traveling salesman problems, *ORSA Journal on Computing* **4**(4), 387–411.

Boese, K. D., Kahng, A. B. and Muddu, S.: 1994, A new adaptive multi-start technique for combintorial global optimizations, *Operations Research Letters* **16**, 101–113.

Brooks, R. A.: 1990, Elephants don't play chess, *in* P. Maes (ed.), *Designing autonomous agents*, MIT press, Cambridge, Mass.

Bullnheimer, B., Hartl, R. F. and Strauss, C.: 1999a, An improved ant system algorithm for the vehicle routing problem, *Annals of Operations Research* **89**, 319–328.

Bullnheimer, B., Hartl, R. F. and Strauss, C.: 1999b, A new rank-based version of the Ant System: A computational study, *Central European Journal for Operations Research and Economics* **7**(1), 25–38.

Congram, R. K., Potts, C. N. and Van de Velde, S. L.: 1998, An iterated dynasearch algorithm for the single–machine total weighted tardiness scheduling problem, *Technical report*, Faculty of Mathematical Studies, University of Southampton, Southampton, UK.

Crauwels, H. A. J., Potts, C. N. and Van Wassenhove, L. N.: 1998, Local search heuristics for the single machine total weighted tardiness scheduling problem, *INFORMS Journal on Computing* **10**(3), 341–350.

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

Di Caro, G. and Dorigo, M.: 1998, AntNet: Distributed stigmergetic control for communications networks, *Journal of Artificial Intelligence Research* **9**, 317–365.

Dorigo, M.: 1992, *Optimization, Learning and Natural Algorithms* (in Italian), PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Italy.

Dorigo, M. and Di Caro, G.: 1999, The Ant Colony Optimization meta-heuristic, *in* D. Corne, M. Dorigo and F. Glover (eds), *New Ideas in Optimization*, McGraw Hill, London, UK, pp. 11–32.

Dorigo, M. and Gambardella, L. M.: 1997a, Ant colonies for the traveling salesman problem, *BioSystems* **43**, 73–81.

Dorigo, M. and Gambardella, L. M.: 1997b, Ant Colony System: A cooperative learning approach to the traveling salesman problem, *IEEE Transactions on Evolutionary Computation* **1**(1), 53–66.

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

Dorigo, M., Maniezzo, V. and Colorni, A.: 1991, Positive feedback as a search strategy, *Technical Report 91-016*, Dipartimento di Elettronica, Politecnico di Milano, Italy.

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

Du, J. and Leung, J. Y.-T.: 1990, Minimizing total tardiness on one machine is NP–hard, *Mathematics of Operations Research* **15**, 483–495.

Emmons, H.: 1969, One machine sequencing to minimize certain functions of job tardiness, *Operations Research* **17**, 701–715.

Gambardella, L. M. and Dorigo, M.: 1995, Ant-Q: A reinforcement learning approach to the traveling salesman problem, *Proceedings of the Twelfth International Conference on Machine Learning (ML-95)*, Morgan Kaufmann Publishers, Palo Alto, California, pp. 252–260.

Gambardella, L. M. and Dorigo, M.: 1996, Solving symmetric and asymmetric TSPs by ant colonies, *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation (ICEC'96)*, IEEE Press, Piscataway, NJ, pp. 622–627.

Gambardella, L. M. and Dorigo, M.: 1997, HAS-SOP: An hybrid Ant System for the sequential ordering problem, *Technical Report IDSIA-11-97*, IDSIA, Lugano, Switzerland.

Gambardella, L. M., Taillard, E. and Agazzi, G.: 1999, MACS-VRPTW: A multiple ant colony system for vehicle routing problems with time windows, *in* D. Corne, M. Dorigo and F. Glover (eds), *New Ideas in Optimization*, McGraw Hill, London, UK, pp. 63–76.

Garey, M. R. and Johnson, D. S.: 1979, *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness*, Freeman, San Francisco, CA.

Glover, F. and Laguna, M.: 1997, *Tabu Search*, Kluwer Academic Publishers, Norwell, MA.

Goldberg, D. E.: 1989, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Massachusetts.

Goss, S., Aron, S., Deneubourg, J. L. and Pasteels, J. M.: 1989, Self-organized shortcuts in the Argentine ant, *Naturwissenschaften* **76**, 579–581.

Graham, R. L., Lawler, E. L., Lenstra, J. K. and Rinnooy Kan, A. H. G.: 1979, Optimization and approximation in deterministic sequencing and scheduling, a survey, *in* P. L. Hammer, E. L. Johnson and B. H. Korte (eds), *Discrete Optimization*, Vol. 5 of *Annals of Discrete Mathematics*, North-Holland, Amsterdam, NL, pp. 287–326.

Hansen, P. and Mladenović, N.: 1997, Variable Neighbourhood Search for the p-Median, *Technical Report Les Cahiers du GERAD G-97-39*, GERAD and École des Hautes Études Commerciales.

Hansen, P. and Mladenović, N.: 1999, An introduction to variable neighborhood search, *in* S. Voss, S. Martello, I. H. Osman and C. Roucairol (eds), *Meta-Heuristics — Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 433–458.

Haupt, R.: 1989, A Survey of Priority Rule-Based Scheduling, *OR Spektrum* **11**, 3–16.

Hogg, T., Huberman, B. A. and Williams, C. P.: 1996, Special volume on frontiers in problem solving: Phase transitions and complexity, *Artificial Intelligence*.

Holsenback, J. E. and Russell, R. M.: 1992, A heuristic algorithm for sequencing on one machine to minimize total tardiness, *Journal of the Operational Research Society* **43**, 53–62.

Hooker, J. N.: 1994, Needed: An empirical science of algorithms, *Operations Research* **42**(2), 201–212.

Hooker, J. N.: 1996, Testing heuristics: We have it all wrong, *Journal of Heuristics* **1**, 33–42.

Hoos, H. H. and Stützle, T.: 1998, Evaluating Las Vegas algorithms — pitfalls and remedies, *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, Morgan Kaufmann Publishers, San Francisco, CA 94104, pp. 238–245.

Hopfield, J. J. and Tank, D. W.: 1985, "Neural" computation of decisions in optimization problems, *Biological Cybernetics* **52**, 141–152.

Jackson, J. R.: 1955, Scheduling a production line to minimize maximum tardiness, *Research Report 43*, Management Science Research Project, UC LA, Los Angeles, CA.

Johnson, D. S.: 1990, Local optimization and the traveling salesman problem, *in* M. S. Paterson (ed.), *Automata, Languages and Programming*, Vol. 443 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, Germany, pp. 446–461.

Johnson, D. S. and McGeoch, L. A.: 1997, The travelling salesman problem: A case study in local optimization, *in* E. H. L. Aarts and J. K. Lenstra (eds), *Local Search in Combinatorial Optimization*, John Wiley & Sons, Chichester, pp. 215–310.

Kara, B. Y.: 1998, Benchmark instances for the single machine total tardiness problem, available on http://www.bilkent.edu.tr/~bkara/start.html.

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

Koulamas, C. P.: 1994, The total tardiness problem: Review and extensions, *Operations Research* **42**, 1025–1041.

Kumal, V., Nau, D. S. and Kanal, L. N.: 1988, A general branch-and-bound formulation for AND/OR graph and game tree search, *in* L. N. Kanal and V. Kumar (eds), *Search in Artificial Intelligence*, Springer Verlag, Berlin, chapter 3, pp. 91–130.

Lawler, E. L.: 1977, A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness, *Annals of Discrete Mathematics* **1**, 331–342.

Lawler, E. L. and Wood, D. E.: 1966, Branch-and-bound methods: A survey, *Operations Research* **14**(4), 699–719.

Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G. and Shmoys, D. B.: 1993, Sequencing and scheduling algorithms and complexity, *in* A. R. K. S.C. Graves and P. Zipkin (eds), *Logistics of Production and Inventory*, Vol. 4 of *Handbooks in Operations Research and Management Science*, North-Holland, Amsterdam, NL, pp. 445–522.

Lenstra, J. K., Rinnooy Kan, A. H. G. and Bruckner, P.: 1977, Complexity of machine scheduling problem, *in* P. L. Hammer, E. L. Johnson, B. H. Korte and G. L. Nemhauser (eds), *Studies in Integer Programming*, Vol. 1 of *Annals of Discrete Mathematics*, North-Holland, Amsterdam, NL, pp. 343–362.

Lin, S.: 1965, Computer solutions for the traveling salesman problem, *Bell Systems Technology Journal* **44**, 2245–2269.

Maniezzo, V. and Colorni, A.: 1999, The Ant System applied to the quadratic assignment problem, *IEEE Transactions on Data and Knowledge Engineering*.

Martin, O. and Otto, S. W.: 1996, Combining simulated annealing with local search heuristics, *Annals of Operations Research* **63**, 57–75.

Martin, O., Otto, S. W. and Felten, E. W.: 1991, Large-step Markov chains for the traveling salesman problem, *Complex Systems* **5**(3), 299–326.

Matsuo, H., Suh, C. J. and Sullivan, R. S.: 1987, A controlled search simulated annealing method for the single machine weighted tardiness problem, *Working paper 87-12-2*, Department of Management, University of Texas at Austin, TX.

McCulloch, W. S. and Pitts, W.: 1943, A logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics* **5**, 115–137.

Merz, P. and Freisleben, B.: 1999, Fitness landscapes and memetic algorithm design, *in* D. Corne, M. Dorigo and F. Glover (eds), *New Ideas in Optimization*, McGraw Hill, London, UK, chapter 3.

Mladenović, N. and Hansen, P.: 1997, Variable Neighborhood Search, *Computers & Operations Research* **24**, 1097–1100.

Morton, T. E. and Pentico, D. W.: 1993, *Heuristic Scheduling Systems with Applications to Production Systems and Projects Management*, John Wiley & Sons, Chichester, UK.

Morton, T. E., Rachamadugu, R. M. and Vepsalainen, A.: 1984, Accurate myopic heuristics for tardiness scheduling, *GSIA Working Paper 36-83-84*, Carnegie–Mellon University, PA.

Osman, I. and Kelly, J. P. (eds): 1996, *Meta-Heuristics: Theory & Applications*, Kluwer Academic Publishers, Norwell, Massachusetts.

Panwalkar, S. S., Smith, M. L. and Koulamas, C. P.: 1993, A heuristic for the single machine tardiness problem, *European Journal of Operational Research* **70**, 304–310.

Pinedo, M.: 1995, *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, Englewood Cliffs, New Jersey 07632.

Potts, C. N. and Van Wassenhove, L. N.: 1982, A decomposition algorithm for the single machine tardiness problem, *Operations Research Letters* **32**, 177–181.

Potts, C. N. and Van Wassenhove, L. N.: 1985, A branch and bound algorithm for the total weighted tardiness problem, *Operations Research* **33**, 363–377.

Potts, C. N. and Van Wassenhove, L. N.: 1991, Single machine tardiness sequencing heuristics, *IIE Transactions* **23**, 346–354.

Ramalhinho Lourenço, H.: 1995, Job-shop scheduling: Computational study of local search and Large-step optimization methods, *European Journal of Operational Research* **83**, 347–364.

Ramalhinho Lourenço, H. and Serra, D.: 1998, Adaptive approach heuristics for the generalized assignment problem, *Technical Report Technical Report Economic Working Papers Series No.304*, Universitat Pompeu Fabra, Dept. of Economics and Management, Barcelona, Spain.

Reeves, C. R.: 1997, Landscapes, operators and heuristic search, *Technical report*, School of Mathematical and Information Sciences, Coventry University, UK.

Reinelt, G.: 1994, *The Traveling Salesman: Computational Solutions for TSP Applications*, Vol. 840 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin.

Russell, R. M. and Holsenback, J. E.: 1997, Evaluation of leading heuristics for the single machine tardiness problem, *European Journal of Operational Research* **96**, 538–545.

Russell, S. J. and Norvig, P.: 1995, *Artificial Intelligence: a modern approach*, Prentice Hall, Englewood Cliffs, New Jersey 07632.

Schoonderwoerd, R., Holland, O., Bruten, J. and Rothkrantz, L.: 1996, Ant-based load balancing in telecommunications networks, *Adaptive Behavior* **5**(2), 169–207.

Schrage, L. E. and Baker, K. R.: 1978, Dynamic programming solution of sequencing problems with precedence constraints, *Operations Research* **26**, 444–449.

Stützle, T.: 1997, $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System for the quadratic assignment problem, *Technical Report AIDA–97–4*, FG Intellektik, FB Informatik, TU Darmstadt.

Stützle, T.: 1998a, An ant approach to the flow shop problem, *Proceedings of the 6th European Congress on Intelligent Techniques & Soft Computing (EUFIT'98)*, Vol. 3, Verlag Mainz, Aachen, pp. 1560–1564.

Stützle, T.: 1998b, *Local Search Algorithms for Combinatorial Problems: Analysis, Improvements, and New Applications*, PhD thesis, FG Intellektik, FB Informatik, TU Darmstadt.

Stützle, T.: 1999, Iterated local search for the flow shop problem, submitted to *European Journal of Operations Research*.

Stützle, T. and Dorigo, M.: 1999a, ACO algorithms for the quadratic assignment problem, *in* D. Corne, M. Dorigo and F. Glover (eds), *New Ideas in Optimization*, McGraw Hill, London, England, pp. 33–50.

Stützle, T. and Dorigo, M.: 1999b, ACO algorithms for the traveling salesman problem, *in* K. Miettinen, P. Neittaanmäki, M. Mäkelä and J. Périaux (eds), *Evolutionary Algorithms in Engineering and Computer Science: Recent Advances in Genetic Algorithms, Evolution Strategies, Evolutionary Programming, Genetic Programming and Industrial Applications*, John Wiley & Sons, Chichester, pp. 163–183.

Stützle, T. and Hoos, H. H.: 2000, $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System, *Future Genereration of Computer Systems*. To appear.

Swarc, W., Della Croce, F. and Grosso, A.: 1998, Solution of the single machine total tardiness problem, *Management Research Center Paper 98-01*, School of Business Administration, University of Wisconsin–Milwaukee, Milwaukee, WI 53201.

Weinberger, E. D.: 1990, Correlated and uncorrelated fitness landscapes and how to tell the difference, *Biological Cybernetics* **63**, 325–336.

Wilkerson, L. J. and Irwin, J. D.: 1971, An improved algorithm for scheduling independent tasks, *AIIE Transactions* **3**, 239–245.

Wolpert, D. H. and Macready, W. G.: 1995, No free lunch theorems for search, *Technical Report SFI-TR-95-02-010*, Santa Fe Institute.

List of Acronyms

ACO ..... ANT COLONY OPTIMIZATION A field that takes the foraging behavior of real ant colonies as inspiration for the design of algorithms for tackling combinatorial optimization problems. The unifying framework is called ACO meta-heuristic, the algorithms are called ACO algorithms. (Dorigo and Di Caro 1999) *Page 8*

ACS ..... ANT COLONY SYSTEM One of the ACO algorithms developed as improvement over AS. It features an aggressive action choice rule combined with a global update restricted to the pheromone trail of the global–best solution and an additional local update rule. (Gambardella and Dorigo 1996, Dorigo and Gambardella 1997b) *Page 10*

ACS-SMTTP′ .. IMPROVED ANT COLONY SYSTEM FOR THE SMTTP Variant of ACS-SMTTP where local search is applied to all solutions built by the ants. *Page 24*

ACS-SMTTP ... ANT COLONY SYSTEM FOR THE SMTTP An ACS algorithm for the SMTTP developed by Bauer (1998). Forms the basis for the ACO algorithm for the SMWTP described in this thesis. (Bauer et al. 1999a, Bauer et al. 1999b) *Page 19*

AI ..... ARTIFICIAL INTELLIGENCE The study and construction of rational agents. (Russell and Norvig 1995) *Page v*

AS ..... ANT SYSTEM The seminal ACO–algorithm. It gave encouraging results when applied to the TSP. (Dorigo et al. 1991, Dorigo 1992, Dorigo et al. 1996) *Page 11*

AU ..... APPARENT URGENCY A dispatching rule that puts jobs in non–decreasing order of apparent urgency, given by $AU_j = (w_j/p_j) \cdot \exp(-\max\{d_j - C_j, 0\})/k\overline{p})$, for a job j with processing time p_j, due date d_j, weight w_j and completion time C_j, where k is a scaling parameter and $\overline{p}$ is the average processing time of the remaining jobs. (Morton et al. 1984, Morton and Pentico 1993) *Page 34*

EDD ..... EARLIEST DUE DATE Dispatching rule that sorts and schedules jobs according to ascending due dates. (Jackson 1955) *Page 18*

ILS..... ITERATED LOCAL SEARCH A simple and powerful meta-heuristic which consists in repeatedly applying a local search algorithm to solutions obtained by small modifications to one of the previously visited locally optimal solutions. (Martin et al. 1991, Stützle 1998b) *Page 12*

IRIDIA..... INSTITUT DE RECHERCHES INTERDISCIPLINAIRES ET DE DÉVELOPPEMENTS EN INTELLIGENCE ARTIFICIELLE AI–laboratory at the Université Libre de Bruxelles. *Page v*

LS..... LOCAL SEARCH A general approach for finding high quality solutions to hard combinatorial optimization problems in reasonable time based on the iterative exploration of neighborhoods of solutions trying to improve the current solution by local changes. (Aarts and Lenstra 1997) *Page 5*

MDD..... MODIFIED DUE DATE Dispatching rule for the SMTTP based on the findings of Emmons (1969) that puts jobs in non–decreasing order of modified due date, given by $MDD_j = \max\{C_j + p_j, d_j\}$, where C_j is the completion time of job j with processing time p_j and due date d_j. *Page 18*

$\mathcal{MM}$AS..... $\mathcal{MAX}$–$\mathcal{MIN}$ ANT SYSTEM An ACO–algorithm derived from AS. It features a simple mechanism for limiting the strengths of the pheromone trails in order to avoid premature convergence of the search. (Stützle and Hoos 2000) *Page 11*

MS..... MINIMUM SLACK Dispatching rule that puts jobs in non–decreasing order of minimum slack, given by $MS_j = \max\{d_j - C_j, 0\}$, where C_j is the completion time of a job j that has due date d_j. *Page 34*

NBR..... NET BENEFIT OF RELOCATION Construction heuristic for the SMTTP based on the theorems of Emmons (1969). (Holsenback and Russell 1992) *Page 18*

$\mathcal{NP}$..... NON–DETERMINISTIC POLYNOMIAL A class of decision problems that have polynomial time solutions but on a non–deterministic Turing machine. A problem is $\mathcal{NP}$–easy if there is a polynomial time algorithm, if, on the other hand, the best possible algorithm will be exponential it is $\mathcal{NP}$–hard. (Garey and Johnson 1979) *Page 4*

ORLIB..... OPERATIONS RESEARCH LIBRARY A benchmark library available on http://mscmga.ms.ic.ac.uk with benchmark instances for a wide variety of combinatorial optimization problems. (Beasley 1990) *Page 34*

PSK..... PANWALKAR SMITH KOULAMAS A construction heuristic for the SMTTP that uses the MDD dispatching rule. (Panwalkar et al. 1993, Alidaee and Gopalan 1997) *Page 18*

RDD..... RANGE OF DUE DATES A statistic that is used in combination with TF to characterize instances of the SMTTP and SMWTP. *Page 72*

RTD..... RUN TIME DISTRIBUTION The distribution of the run–time required for an algorithm to achieve a specific goal — like finding an optimal solution to a given problem instance. (Hoos and Stützle 1998) *Page 14*

SMTTP..... SINGLE MACHINE TOTAL TARDINESS PROBLEM A special case of the SMWTP that assumes the jobs have equal non–zero weights and whose objective it is to minimize the total tardiness of jobs that are processed without preemption on a single machine. Treated in Chapter 2. (Pinedo 1995) *Page 18*

SMWTP..... SINGLE MACHINE TOTAL WEIGHTED TARDINESS SCHEDULING PROBLEM The problem treated in this thesis. Its objective is to minimize the total weighted tardiness of jobs that are processed without preemption on a single machine. (Pinedo 1995) *Page 33*

SPT..... SHORTEST PROCESSING TIME Dispatching rule that puts the jobs in non–decreasing order of processing times. *Page 18*

TF..... TARDINESS FACTOR A statistic that is used in combination with RDD to characterize instances of the SMTTP and SMWTP. *Page 72*

TSP..... TRAVELING SALESMAN PROBLEM The problem of a salesman who wants to find, starting from his home town, a shortest possible trip through a given set of customer cities and to return to its home town. *Page 9*

WI..... WILKERSON IRWIN Bauer (1998, p. 20) Heuristic for the SMTTP that resembles a descent algorithm that applies the transpose neighborhood to the EDD sequence. (Wilkerson and Irwin 1971) *Page 18*

WSPT..... WEIGHTED SHORTEST PROCESSING TIME Dispatching rule that puts jobs in non–decreasing order of the weighted processing times given by w_j/p_j for weight w_j and processing time p_j. *Page 34*

Index