Local Search Algorithms for Combinatorial Problems

—

Analysis, Improvements, and New Applications

Am Fachbereich Informatik
der Technischen Universität Darmstadt
vorgelegte

Dissertationsschrift

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

von

Dipl.-Wirtschaftsing. Thomas G. Stützle

aus Donauwörth

Referent: Prof. Dr. Wolfgang Bibel
Korreferent: Dr. Marco Dorigo

Tag der Einreichung: 28. Oktober 1998
Tag der mündlichen Prüfung: 9. Dezember 1998

Darmstadt 1998
Hochschulkennziffer D17

Thomas G. Stützle

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Intellektik
Alexanderstr. 10
64283 Darmstadt

Email: stuetzle@informatik.tu-darmstadt.de
URL: http://www.intellektik.informatik.tu-darmstadt.de/~tom/

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation zur Erlangungen des akademischen Grades eines Doktors der Naturwissenschaft (Dr. rer. nat)

Referent: Prof. Dr. Wolfgang Bibel
Korreferent: Dr. Marco Dorigo

Tag der Einreichung: 28. Oktober 1998
Tag der Disputation: 9. Dezember 1998

Hochschulkennziffer D17

Acknowledgments

There are several important periods in our life. The transition between these periods sometimes is smooth and softly; sometimes the end of a period marked is by some event. In the scientific life one such event is the defense of the PhD thesis. This book represents the outcome of a three years period of research efforts and is the summary of my contributions as an individual researcher ... or is it really?

When taking a glance back, the outcome of this research has strongly been influenced by many people who accompanied me during the whole or part of the work in my private and academic life. Therefore, is this also the occasion to very warmly thank these people and to acknowledge their support.

In the first place I thank my parents with all my heart. They have given me continuous support during all the years, have guided me through the early years of my life, and always have provided a stable environment. In the same breath I deeply thank my dearest partner Josefina Santangelo for sharing the recent years of my life with me. Living together with her has been a great pleasure and she has influenced me very positively all over these years.

For the research work in the recent years the personal and scientific interaction with Holger Hoos has been most stimulating, for what I'm very indebted to him. We had a marvelous time discussing common research issues, working together on several projects, and writing joint papers. Research would have been less fun without him.

I very warmly thank my thesis adviser Prof. Wolfgang Bibel for the support during and after this work, for providing the necessary scientific freedom to carry out this research, and for his patience during this time. I am also very grateful to Hesham Khalil who has been of invaluable help in the later stages of writing this thesis. This thesis would not be the way it is without his support. I also like to thank very much the current and the past members of the Intellectics Group for many interesting discussions and their help during this time; in particular I like to thank Maria Tiedemann, Michael Thielscher, Christoph Herrmann, Olaf Steinmann, Antje Strohmaier, Ulrich Scholz, and Thomas Rath.

I am also indepted to Marco Dorigo for the comments on my first papers, interesting discussions on ant colony optimization, and especially for the excellent time and the very fruitful co-operation during my stay at IRIDIA after the end of this PhD thesis.

I'm also very grateful to Olivier Martin for the interesting discussions at various conferences and for providing the source code of his Lin-Kernighan implementation for the traveling salesman problem, which has been used in parts of this thesis.

During these years and the months after the PhD examination I had many interesting discussions with many people at conferences and visiting professors at our lab. These discussions have

sharpened my understanding of different research issues, provided new points of view and directions for my research or simply made research more enjoyable. I specifically remember discussions with Helena Ramalhinho Lorençou, Peter Merz, Bernd Freisleben, Bernd Bullnheimer, Gianni Di Caro, Nenad Mladenovic, David Woodruff, Maria Christina Riff-Rojas, Kung-Kiu Lau, Pratim P. Chakrabarti, Sujoy Ghose, David McAllester, Bart Selman, Tim Walters, Robert Preis, Peter Hahn, Javier Yañez, Jose Santesmases and many others. I also especially like to thank Prof. Georg Bol from TU Karlsruhe for guiding my first real research attempts.

Finally, I acknowledge the support of the German Research Foundation (DFG) via a grant awarded by the Graduiertenkolleg “Intelligente Systeme für die Informations- und Automatisierungstechnik.”

Contents

Chapter 1

Introduction

1.1 Motivation

Many problems of practical and theoretical importance within the fields of Artificial Intelligence and Operations Research are of a combinatorial nature. Combinatorial problems involve finding values for discrete variables such that certain conditions are satisfied. They can be classified either as optimization or satisfaction problems. In *optimization problems*, the goal is to find an optimal arrangement, grouping, ordering, or selection of discrete objects usually finite in number [150]. The probably most widely known example is the traveling salesman problem (TSP) [151] in which a shortest closed tour through a set of cities has to be found. Other examples are assignment, scheduling, facility layout, and vehicle routing problems. In *satisfaction problems*, a solution satisfying given constraints has to be found. A prototypical example is the constraint satisfaction problem (CSP) [162] in which one has to decide whether an assignment of values to variables can be found such that a given set of constraints is satisfied. An important special case of the CSP is the well-known satisfiability problem in propositional logic, but CSPs are also at the core of frequency assignment, graph coloring, temporal and spatial reasoning, as well as resource allocation. For the purpose of this thesis we will not differentiate rigorously between optimization and satisfaction problems because both types are closely related.

Combinatorial problems are intriguing because they are easy to state but often very difficult to solve. For example, no algorithm exists for finding the optimal solution to a TSP within polynomial time. Similarly, no algorithm can be guaranteed to decide in polynomial time whether a given CSP instance is satisfiable or not. This phenomenon has been encountered on a wide variety of combinatorial problems and led to the development of complexity theory [86, 198] and, in particular, to the theory of $\mathcal{NP}$-completeness. It aims at classifying problems depending on how difficult they are to solve. The class of $\mathcal{NP}$-complete problems has the important property that no polynomial time algorithm for any of its members exists to date and in case a polynomial time algorithm for one $\mathcal{NP}$-complete problem were found, all could be solved in polynomial time. They are therefore considered as *inherently intractable* from a computational point of view. Thus, in the worst case any algorithm that tries to solve an $\mathcal{NP}$-complete problem requires exponential run time. In particular, the TSP and the CSP belong to this class and, thus, are among the hardest combinatorial problems.

Combinatorial problems occur in many practical applications and there is an urgent need for algorithms which solve them efficiently. The algorithmic approaches to combinatorial optimiza-

tion problems can be classified as either *exact* or *approximate*. Exact algorithms are guaranteed to find an (optimal) solution in finite time by systematically searching the solution space. Yet, due to the $\mathcal{NP}$-completeness of many combinatorial optimization problems, the time needed to solve them may grow exponentially in the worst case. To practically solve these problems, one often has to be satisfied with finding good, approximately optimal solutions in reasonable, that is, polynomial time. This is the goal of approximate algorithms such as local search or solution construction algorithms. Approximate algorithms cannot guarantee optimality of the solutions they return, but empirically they have often been shown to return good solutions in short computation time.

The most widely and successfully applied approximate algorithms are local search algorithms. Local search algorithms start from some given solution and try to find a better solution in an appropriately defined *neighborhood* of the current solution. In case a better solution is found it replaces the current solution and the local search is continued from there. The most basic local search algorithm, called *iterative improvement*, repeatedly applies these steps until no better solution can be found in the neighborhood of the current solution and stops in a *local optimum*. A disadvantage of this algorithm is that it may stop at poor quality local minima. Thus, possibilities have to be devised to improve its performance. One would be to increase the size of the neighborhood used in the local search algorithm. Obviously, there is a higher chance to find an improved solution, but it also takes longer time to evaluate the neighboring solutions, making this approach infeasible for larger neighborhoods. Another possibility is to restart the algorithm from a new, randomly generated solution. Yet, the search space typically contains a huge number of local optima and this approach becomes increasingly inefficient on large instances.

To avoid these disadvantages of iterative improvement algorithms, many generally applicable extensions of local search algorithms have been proposed. They improve the local search algorithms either by accepting worse solutions, thus, allowing the local search to escape from local optima, or by generating good starting solutions for local search algorithm guiding them towards better solutions. In the latter case, often the experience accumulated during the run of the algorithm is used to guide the search in subsequent iterations. These general schemes to improve local search algorithms are now called *metaheuristics* [195, 194]. The computational results obtained by applications of these metaheuristics are often quite impressive. They can be used to find solutions to certain classes of nontrivial TSP instances with as many as 10^5 cities within one percent of the optimal solution value [129]. Similarly, recent progress in local search algorithms for the satisfiability problem [222, 172] has shown that these techniques are also applicable to large and hard to solve satisfaction problems and outperform systematic algorithms on a wide variety of problem classes.

The recent success of metaheuristics is also clearly recognizable in an increasing number of books [3, 207], collections of articles [204, 194], special issues of journals [97, 202, 149] and a new conference series, called Metaheuristics International Conference, that concentrates solely on this subject. Work on local search, in general, and metaheuristics, in particular, is not only widely applied in the Operations Research community but also receives increased attention in the Artificial Intelligence community. This is witnessed by the fact that in all major AI conferences like the International Joint Conference on Artificial Intelligence, the National Conference on Artificial Intelligence and its European counterpart, local search algorithms are applied to important AI problems and many successful applications are reported.

1.2 Research Motives and Contributions

In this thesis we present improvements of existing metaheuristics, provide tools for their analysis and the characterization of their run-time behavior, and explore new applications of the algorithms. More specifically, the main contributions of this thesis belong to the following research topics.

Ant Colony Optimization

Recent progress in local search algorithms has often been inspired by analogies to naturally occurring phenomena like the physical annealing of metals or biological evolution. These phenomena led to strongly improved algorithmic approaches which are nowadays known as simulated annealing [139] or genetic algorithms [98, 115]. The most recent of these nature-inspired algorithms is ant colony optimization (ACO) [67, 48, 63, 65, 68], inspired by the foraging behavior of real ant colonies. The heuristic is based on a colony of artificial ants which construct solutions to combinatorial optimization problems and communicate indirectly via pheromone trails. The search process is guided by positive feedback taking into account the solution quality of the constructed solutions and the experience of earlier iterations of the algorithm. ACO is currently one of the fastest evolving metaheuristics as shown by the special session on ACO at the INFORMS98 meeting, the first international workshop on ant colony optimization held in Brussels in October 1998 and the fact that it is one of the main topics of two forthcoming books [29, 52].

Since ACO is a very young field, still much work has to be invested in improving the performance of the algorithmic approaches. Our main goal will be to improve ant system [67, 68], the seminal work on ACO algorithms, proposing extensions that show significantly better performance. Our improved algorithm is called $\mathcal{MAX}$–$\mathcal{MIN}$ ant system [235, 237]. Since its first implementations [235] $\mathcal{MAX}$–$\mathcal{MIN}$ ant system is used as a hybrid algorithm, combining solution construction by ants with local search algorithms. In fact, today the most efficient ACO algorithms for combinatorial optimization problems use local search algorithms to improve their performance [66, 165, 236].

We show that with $\mathcal{MAX}$–$\mathcal{MIN}$ ant system very high quality solutions can be obtained in the application to the TSP, finding in many cases optimal solutions for instances with several hundreds of cities. The application of our algorithm to the quadratic assignment problem (QAP) shows that it is currently among the best known heuristics to tackle structured QAP instances. This conclusion is underlined by the fact that we were able to find a new best known solution to the largest QAP instance contained in QAPLIB [39], a well known benchmark set for this problem. Additionally, we present a new application of an ACO algorithm to the permutation flow shop problem (FSP) and show that this algorithm compares favorably with many other heuristic algorithms proposed for this problem.

Iterated Local Search

Suppose a locally optimal solution has been found by a local search algorithm. Rather than restarting each local search from a completely new generated solution, the idea of iterated local search (ILS) is to apply local search repeatedly to initial solutions obtained by perturbations of

a previously visited locally optimal solutions [20, 169, 180, 233]. In the current standard form of applying ILS algorithms, these perturbations are typically made from the best locally optimal solution found [168, 129]. A major advantage of ILS algorithms is that (i) they are conceptually rather simple, (ii) they are very easy to implement, and (iii) are very efficient when applied to many combinatorial optimization problems [129, 180, 168].

Based on a detailed analysis of the run-time behavior of ILS algorithms, we propose extensions to these algorithms which considerably improve their performance. The proposed extensions concern ILS algorithms operating only on single solutions as well as the introduction of population-based ILS algorithms. The experimental results show that despite their simplicity, ILS algorithms are among the best approaches for the combinatorial optimization problems attacked in this thesis. In particular, improved solutions could be found on some of the hardest benchmark instances of the permutation flow shop problem. We also extensively compare the performance of the ACO algorithms and of the ILS algorithms and give a detailed account of the relative merits of both approaches.

Tabu Search

A further contribution of this thesis is the improvement of the *min conflicts* heuristic for CSPs by the application of tabu search techniques. We show that the proposed extensions of the *min conflicts* heuristic compare favorably to earlier extensions. A special focus in that chapter is put on the characterization of the run-time behavior of local search algorithms for CSPs, based on joint work with Holger Hoos [122, 119]. In particular, we show that the run-time distributions of the best performing variants of the *min conflicts* heuristic on hard instances can closely be approximated by exponential distributions. As will be shown later, this result implies that the simplest form of parallelization, which consists in executing multiple independent runs of an algorithm in parallel, will be very effective.

Run-time Distributions

The analysis of the algorithms applied in this thesis, especially the iterated local search algorithms and the tabu search application to CSPs, uses empirically measured run-time distributions to characterize the run-time behavior of these algorithms. This characterization can be used to yield important conclusions concerning the speed-up of parallel independent runs of an algorithm. It also facilitates the comparison of algorithms and the run-time distributions give valuable hints on possible improvements of the algorithms. This later use is exemplified using the application of ILS algorithms to the TSP, the QAP, and the FSP.

1.3 Outline of the Thesis

This thesis is organized as follows.

Chapter 2 outlines the main concepts and the background for the work presented in this thesis. We briefly introduce combinatorial optimization problems and present two example problems, the traveling salesman problem and the constraint satisfaction problem. Both are used to discuss aspects of local search and are tackled by the algorithms presented in the subsequent

chapters. Next, we summarize important results of the theory of $\mathcal{NP}$-completeness and justify the need for approximate algorithms. The main part of the chapter comprises an introduction to local search algorithms and the local search extensions used in this thesis. We conclude with a discussion of the commonalities and differences among metaheuristic approaches and a review of work on the search space analysis for combinatorial optimization problems.

Local search algorithms need to be evaluated empirically. In Chapter 3 we present our new empirical methodology, developed in joint work with Holger Hoos. This methodology is based on measuring run-time distributions on single problem instances. Initially, run-time distributions have been applied to characterize the behavior of local search applied to satisfaction problems [118, 119]. Here we extend the methodology to optimization problems. Our empirical methodology will be applied in Chapter 5 to improve iterated local search algorithms and in Chapter 6 to characterize the run-time behavior of local search algorithms applied to CSPs.

In Chapter 4 we introduce $\mathcal{MAX}$–$\mathcal{MIN}$ ant system and analyze its performance on three combinatorial optimization problems, the TSP, the QAP, and the FSP. The chapter starts with a review of the early work on ant system and outlines other proposed algorithmic improvements of ant system. Then, we present the modifications that led to $\mathcal{MAX}$–$\mathcal{MIN}$ ant system and give a detailed analysis of their benefits. Extensive computational results on the TSP confirm the effectiveness of our algorithm; in particular, it is currently the best performing ACO algorithm for symmetric TSPs. The application of our algorithm to the quadratic assignment problem shows that it is currently among the best known heuristics to tackle structured QAP instances. Next, a new ACO application to the permutation flow shop problem is presented. Additionally, we propose parallelization possibilities of ACO algorithms and computational results show the significant improvements with respect to computation times as well as solution quality we achieved.

In Chapter 5 we analyze the run-time behavior of iterated local search, develop improved ILS algorithms for the TSP, and present new ILS applications to the QAP and the FSP. We discuss extensions of iterated local search approaches and propose a unified framework for these algorithms. The extensions we propose concern the execution of a single algorithm run as well as population-based extensions. We show that our improved ILS applications belong for each of the problems attacked to the best available algorithms and we compare their performance to those obtained with $\mathcal{MAX}$–$\mathcal{MIN}$ ant system.

The application of tabu search to the CSP is presented in Chapter 6. We use tabu search to improve the *min conflicts* heuristic applied to binary CSPs and large graph coloring instances and show that our proposed extensions compare favorably to previously developed algorithms. Additionally, we functionally approximate the run-time behavior of the local search algorithms applied to binary CSPs.

Finally, in Chapter 7, we summarize the main contributions of this thesis and outline directions for future research.

Chapter 2

Combinatorial Optimization, Local Search and Metaheuristics

Local search has become a widely accepted technique for the solution of hard combinatorial optimization problems [3]. This chapter serves mainly as an introduction into this topic of the thesis.

We start with the discussion of combinatorial optimization problems and introduce as examples the traveling salesman problem and the general class of constraint satisfaction problems. Then, we give an overview of concepts of the theory of $\mathcal{NP}$-completeness and we discuss general solution methods for combinatorial optimization. Subsequently, we introduce general issues concerning the application of local search to combinatorial optimization problems and present the recent development of a complexity theory for local search algorithms. To apply local search algorithms some initial solution has to be given, which often is generated by greedy construction heuristics. Thus, we have inserted a concise outline of general principles of construction heuristics in Section 2.6.

In the largest part of this chapter we present some of the most widely applied local search variants, concentrating on those which are used in some part of this thesis. These local search variants are in fact algorithmic frameworks that, in principle, can be applied to many different optimization problems without major modifications. The term *metaheuristics* is getting widely accepted when referring to these algorithmic frameworks. A metaheuristic is defined to be a general heuristic method which is used to guide an underlying local search algorithm towards promising regions of the search space containing high quality solutions [195].

We will concisely present the main features of the metaheuristics studied in this thesis; these are ant colony optimization [67, 63, 68], iterated local search [169, 168, 234], and tabu search [90, 91, 92, 107]. More detailed descriptions of these algorithms can then be found in the following chapters. Additionally, we briefly review simulated annealing [139, 44], genetic algorithms [115, 98], and greedy randomized adaptive search procedures [72, 73] because these are used in the discussion of commonalities and differences between the various metaheuristics or for comparisons of our newly developed metaheuristic implementations to previously proposed algorithms.

It is widely agreed that the performance of metaheuristics and local search is strongly tied to characteristics of the search space. Therefore, we end this chapter by an outline of aspects

on search space characteristics for combinatorial optimization problems. In the following chapters we will use some insights into search space characteristics to explain the performance of metaheuristic algorithms or to propose improved metaheuristic variants.

2.1 Combinatorial Optimization Problems

In this thesis we are concerned with combinatorial optimization problems. We now give a short outline of some important notions concerning combinatorial optimization problems, more detailed descriptions are found in [189, 199]. To concentrate on optimization problems is not a limitation, because, without restriction of generality, satisfaction problems can be formulated as optimization problems, as will be pointed out below.

A *combinatorial optimization problem* is either a maximization problem or a minimization problem with an associated set of *instances*. Without loss of generality we will restrict ourselves in this thesis to minimization problems, because every maximization problem can easily be converted into a minimization problem.

Definition 2.1.1 *An* instance *of a combinatorial optimization problem is a pair $(\mathcal{S}, f)$, where $\mathcal{S}$ is the finite set of candidate solutions and $f : \mathcal{S} \mapsto \mathbb{R}$ is a function which assigns to every $s \in \mathcal{S}$ a value $f(s)$. $f(s)$ is also called* objective function value*. The goal of a combinatorial optimization problem is to find a solution $s \in \mathcal{S}$ with minimal objective function value, that is, $f(s_{opt}) \leq f(s') \; \forall \; s' \in \mathcal{S}$. s_{opt} is called a* globally optimal solution *of $(\mathcal{S}, f)$, the set $\mathcal{S}_{opt}$ is the set of all globally optimal solutions.*

Thus, the term *problem* refers to the general question to be answered, usually having several parameters or variables with unspecified values. The term *instance* refers to a problem with specified values for all parameters. In the case of minimization problems, the objective function is also often called *cost function* and the objective function value is called *cost value*. We assume, without loss of generality, that the cost function assumes only nonnegative values, that is, $f(s) \geq 0$ for all $s \in \mathcal{S}$.

A combinatorial optimization problem can actually be attacked in three different versions which are:

> *Search version*: Given an instance $(\mathcal{S}, f)$, find an optimal solution, that is, an element $s_{opt} \in \mathcal{S}_{opt}$.
>
> *Evaluation version*: Given an instance $(\mathcal{S}, f)$, find the optimal objective function value $f(s_{opt})$.
>
> *Decision version*: Given an instance $(\mathcal{S}, f)$ and a bound L, decide whether there is a feasible solution $s \in \mathcal{S}$ with $f(s) \leq L$.

Clearly, the search version is the most general of these as with the knowledge of an optimal solution, the evaluation version and the decision version are trivially solved. $\mathcal{S}$ will be called the search space. The finiteness of $\mathcal{S}$ suggests that any given instance $(\mathcal{S}, f)$ can be solved by

enumerating the whole set of solutions and picking the one with minimal cost. Yet, this approach is infeasible for many problems as the size of the search space, denoted in the following by $|\mathcal{S}|$, grows superpolynomially with instance size.

In the following we introduce two example problems which will be used throughout this thesis to illustrate specific issues concerning the solution of combinatorial optimization problems using local search algorithms; both are tackled in this thesis with approximate algorithms. One of the two problems is the well known Traveling Salesman Problem, the other is the general constraint satisfaction problem and its special case, the satisfiability problem in propositional logic. With this second example we also illustrate how satisfaction problems can be converted into optimization problems when solving them with local search methods. Addressing all these problems as optimization problems also gives a coherent point of view on the problems used in this thesis.

2.1.1 Example 1: The Traveling Salesman Problem

The traveling salesman problem (TSP) is probably the most widely studied combinatorial optimization problem and has attracted a large number of researchers for a long time. Intuitively, it is the problem faced by 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. More formally, a TSP can be represented by a complete weighted directed graph $G = (\mathcal{V}, \mathcal{A}, d)$ with $\mathcal{V}$ being the set of nodes (the cities), $\mathcal{A}$ being the set of arcs and $d : \mathcal{A} \mapsto \mathbb{N}$ a weight function associating a positive, integer weight $d(c_i, c_j)$ to every arc (c_i, c_j), corresponding to the distance between cities c_i and c_j. The goal is to find a shortest closed path that visits every city exactly once, that is, a Hamiltonian path (often simply called *tour*). For symmetric TSPs, the distances between the cities are independent of the direction of traversing the cities, that is, $d(c_i, c_j) = d(c_j, c_i)$ for every pair of nodes. In the more general asymmetric TSP (ATSP) at least for one pair of nodes we have $d(c_i, c_j) \neq d(c_j, c_i)$.

In the TSP we are asked to find the shortest tour, that is, a permutation π of the cities $\{c_1, c_2, \ldots, c_n\}$ such that $f(\pi)$ is minimal, where $f(\pi)$ is given as:

$$f(\pi) = \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \; + \; d(c_{\pi(n)}, c_{\pi(1)}) \tag{2.1}$$

In the case of symmetric TSPs, we will use *Euclidean* TSP instances where the cities are given points in the Euclidean space and the inter-city distances are calculated using the Euclidean norm. An example of two TSP instances is given in Figure 2.1, taken from the TSPLIB benchmark library [210], accessible at `http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html`, which contains a large collection of instances for which computational results have been published previously.

The TSP is extensively studied in literature [151, 211] and has served as a standard benchmark problem for new algorithmic ideas. The high attractiveness of the TSP is due to the fact that it is (i) very easy to formulate and understand, (ii) it is the simplest case of an ordering problems like the flow shop problem and the job shop problem which are relevant to industry, and (iii) it is hard to solve exactly. The TSP also has influenced the emergence of many important fields like local search algorithms [55, 155, 156], polyhedral theory [103], and the development

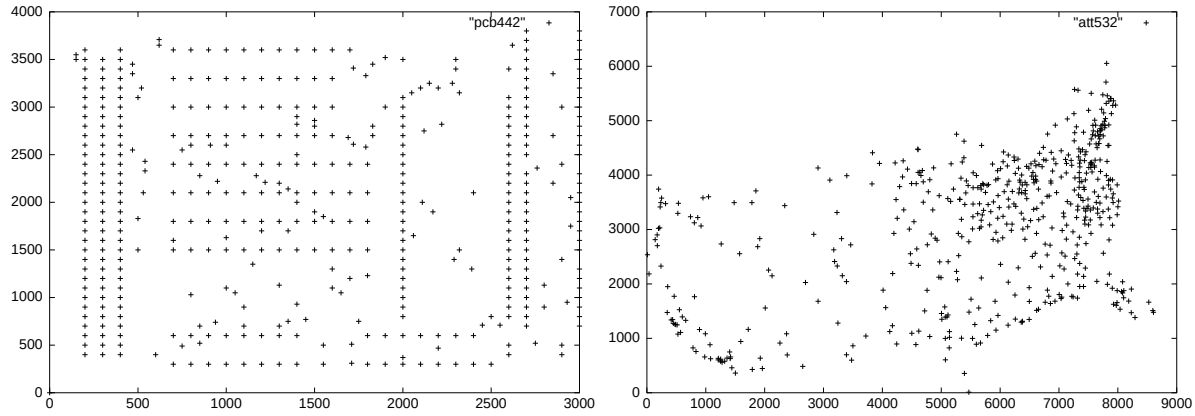

Figure 2.1: TSP instances `pcb442` (left side) and `att532` (right side) from TSPLIB. The instance `pcb442` stems from a drilling problem in a printed circuit board application, the instance `att532` comprises 532 cities in the USA. Both problems are well known and have been used as benchmark problems in numerous articles.

of complexity theory [86]. A discussion of the various local search approaches to TSPs and an assessment of their relative merits will be postponed until Section 4.3.

2.1.2 Example 2: The Constraint Satisfaction Problem

Constraint satisfaction problems (CSPs) are an important class of combinatorial problems in Artificial Intelligence. They offer a general framework for reasoning about constraint problems and many practical problems like spatial and temporal reasoning [60, 105], graph coloring [111, 31], frequency assignment problems, and scheduling [216, 217] can be formulated as constraint satisfaction problems. In a constraint satisfaction problem one is given a finite set of variables each with a finite domain and a set of constraints that restrict the possible values the variables may take simultaneously. More formally, a CSP can be defined as follows.

Definition 2.1.2 *A* finite Constraint Satisfaction Problem $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is defined by a set of variables* $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$*, every variable* x_i *having associated a discrete, finite domain* D_i*, and a set of constraints* $\mathcal{C}$*. We denote* $\mathcal{D} = \cup_{i=1}^{n} D_i$*. A constraint* $C_k \in \mathcal{C}$ *restricts the set of allowed value combinations for variables, that is,* $C_k(x_{k_1}, \ldots, x_{k_m}) \subset D_{k_1} \times \cdots \times D_{k_m}$*,* m *is called the arity of a constraint.*

We call the assignment of a value $d_i \in D_i$ to a variable x_i a variable instantiation. A variable instantiation for all variables will shortly be called *candidate solution* s. By $s(x_i) = d_i$ we denote the value that variable x_i takes under candidate solution s. If the tuple $(d_{k_1}, \ldots, d_{k_m})$ is not in $C_k(x_{k_1}, \ldots, x_{k_m})$ the constraint is said to be *violated* and the variables participating in the constraint C_k are said to be in *conflict*. The task in a CSP is to find a candidate solution such that all constraints are satisfied or to report that no such solution exists. A CSP is called *satisfiable* if such a solution can be found, otherwise it is called *unsatisfiable*.

CSPs can easily be formulated as combinatorial optimization problems. The search space $\mathcal{S}$ is given by the set of all candidate solutions, $\mathcal{S} = \{(x_i, s(x_i)) \mid x_i \in \mathcal{X} \text{ and } s(x_i) \in D_i\}$. The size of the search space is the product of the domain sizes, that is, $|\mathcal{S}| = \Pi_{i=1}^{n}|D_i|$. A straightforward objective function $f(s)$ for CSPs is the number of violated constraints. If $f(s)$ is defined in this way (alternatively one could maximize the number of satisfied constraints), then a CSP is satisfiable if and only if $\exists s \in \mathcal{S} : f(s) = 0$. Such an instantiation will also be called shortly solution.

An important special case of CSPs in theory and practice is the satisfiability problem, well known from propositional logic (SAT). SAT may be interpreted as a special case of CSP in which the variables $\{x_1, \ldots, x_n\}$ only take two values 1 and 0. x_i is interpreted as "*true*", if it is assigned the value 1 (*true*) and as "*false*", if it is assigned the value 0 (*false*). A literal l_i over $\mathcal{X}$ is a propositional variable x_i or its negation $\neg x_i$. In the first case the literal is called *positive* and *negative* in the later. The literal l_i is interpreted as "true" if it is positive and the variable is assigned the value 1 or if it is negative and the variable is assigned value 0. We assume that the constraints are given as a set of clauses C_i, each clause consisting of literals that are connected by logical *or* ($\vee$), that is, $C_i = \bigvee_{j=1}^{m_i} l_j$. A clause is satisfied if at least one literal is interpreted as "*true*" under the current assignment. A formula Φ is given as a set of clauses that have to be satisfied simultaneously; it is given in *conjunctive normal form* (CNF).

$$\Phi = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{m_j} l_{ij}$$

Formula Φ is satisfied if all clauses can be satisfied simultaneously. SAT is a particular type of a constraint satisfaction problem as variables are restricted to have binary domains and the constraints are given in a special form, in general as clauses.

2.2 Computational Complexity

In general, one is interested in solving combinatorial problems as efficiently as possible where efficient usually means as fast as possible. Hence, an important criterion for the classification of problems is the time the best known algorithms need to find a solution for the given problem. This issue is addressed by the theory of computational complexity and, in particular, by the theory of $\mathcal{NP}$-completeness. Its main aim is to classify problems according to their difficulty to be solved by any known algorithm. For the classification of problems it has been shown to be useful to address the question regarding problem complexity as a worst-case measure, that is, the complexity of a problem is determined by the hardest conceivable instance.

The time-complexity of an algorithm is measured by a time-complexity function that gives, depending on the instance size, the maximal run-time for the algorithm to solve an instance. The size of a problem instance reflects the amount of data to encode an instance in a *compact* form. Often it suffices to have an intuitive understanding of the size of an instance; for example, the size of a TSP instance can be measured by the number of cities in the graph, in the case of SAT the size can be measured by the number of boolean variables. The time-complexity is typically given, in terms of the number of elementary operations like value assignments or comparisons; it is formalized by the $\mathcal{O}(\cdot)$ notation. Let f and g be two functions from $\mathbb{N} \mapsto \mathbb{N}$, then we write $f(n) = \mathcal{O}(g(n))$ if there are positive integers c and n_0 such that for all $n > n_0$, $f(n) \leq c \cdot g(n)$.

An algorithm runs in polynomial time, if the run-time is bounded by a polynomial; if the run-time cannot be bounded by some polynomial, the algorithm is said to be an exponential time algorithm.

In complexity theory, a basic difference is made between efficiently solvable problems (*easy* problems) and inherently intractable ones (*hard* problems). Usually, a problem is considered efficiently solvable if a solution can be found in a number of steps bounded by a polynomial of the input size. If the number of steps needed to solve an instance grows super-polynomially, we say that a problem is inherently intractable.

2.2.1 $\mathcal{NP}$–completeness

The theory of $\mathcal{NP}$-completeness formalizes the distinction between easy and hard problems. In general, the theory of $\mathcal{NP}$-completeness is concerned with the decision version of combinatorial problems. The generality of the conclusions drawn is not very limited by this fact because it is obvious that the optimization version of a problem is not easier to solve than the decision version and if the optimization version of a problem can be solved efficiently, then the same is true for the decision version. Optimization problems typically have an associated decision problem; for example, in the TSP case, the associated decision version asks whether a tour with cost bound $f(\pi) < L$ exists. The evaluation version of an optimization problem can be solved as a series of decision problems using binary search on the bound L.

The theory of $\mathcal{NP}$–completeness distinguishes between two basic classes of problems. One is the class $\mathcal{P}$ of tractable problems.

Definition 2.2.1 *The class $\mathcal{P}$ is the class of decision problems that can be solved by a polynomial time algorithm.*

The class $\mathcal{NP}$ can be defined informally in terms of a *nondeterministic algorithm*. Such an algorithm can be conceived as being composed of a *guessing stage* and a *checking stage*. If we are given some instance I, in the first stage some solution is *guessed*. This solution is verified by a deterministic polynomial algorithm in the second stage. The class $\mathcal{NP}$ is the class of problems that can be solved by such a nondeterministic algorithm. For the class $\mathcal{NP}$ this *polynomial-time verifiability* of the property for some given solution s is essential. The polynomial time verifiability also implies that the guessed solution is of polynomial size.

Definition 2.2.2 *The class $\mathcal{NP}$ consists of those problems that can be solved by a nondeterministic polynomial-time algorithm.*

Any decision problem that can be solved by a deterministic polynomial-time algorithm also can be solved by a nondeterministic polynomial-time algorithm, that is $\mathcal{P} \subseteq \mathcal{NP}$. Yet, polynomial-time nondeterministic algorithms appear more powerful than polynomial-time deterministic algorithms. In fact, a relation between them can be established by the following theorem; for the proof we refer to [86].

Theorem 2.2.1 *If $\Pi \in \mathcal{NP}$, then there exists a polynomial p such that Π can be solved by a deterministic algorithm having time complexity $\mathcal{O}(2^{p(n)})$.*

Probably the most important open question in theoretical computer science today is whether $\mathcal{P} = \mathcal{NP}$? It is widely believed nowadays that $\mathcal{P} \neq \mathcal{NP}$, yet no proof of this conjecture has been found so far.

A problem usually is considered intractable if it is in $\mathcal{NP}\backslash\mathcal{P}$. As one cannot show that $\mathcal{NP}\backslash\mathcal{P}$ is not empty, unless a proof for $\mathcal{P}$ not equal $\mathcal{NP}$ is found, the theory of $\mathcal{NP}$-completeness focuses on proving results of the weaker form if $\mathcal{P} \neq \mathcal{NP}$, then $\Pi \in \mathcal{NP}\backslash\mathcal{P}$. One of the key ideas needed for this approach is the notion of polynomial-time reducibility among problems.

Definition 2.2.3 *A problem* Π *is* polynomially-reducible *to a problem* Π'*, if a polynomial-time algorithm exists that maps each instance of* Π *onto an instance of* Π' *and that for each instance of* Π *"yes" is output iff for the corresponding instance of* Π' *the output of the decision procedure is "yes".*

Informally this definition says that if Π can be polynomially reduced to Π', then problem Π' is at least as difficult to solve as problem Π. Using the notion of polynomial reducibility we can proceed to define the class of $\mathcal{NP}$-complete problems.

Definition 2.2.4 *A problem* Π *is* $\mathcal{NP}$*-complete iff (i)* $\Pi \in \mathcal{NP}$ *and for all* $\Pi' \in \mathcal{NP}$ *holds that* Π' *is polynomially reducible to* Π.

The class of $\mathcal{NP}$-complete problems is in some sense the class of the *hardest problems in* $\mathcal{NP}$. If a $\mathcal{NP}$-complete problem can be solved by a polynomial time algorithm, then all problems in $\mathcal{NP}$ can be solved in polynomial time. Yet, so far for no $\mathcal{NP}$-complete problem a polynomial time algorithm could be found. Thus, if one can prove that a problem Π is $\mathcal{NP}$-complete common belief suggests that no deterministic polynomial-time algorithm exists and the problem cannot be solved efficiently. The first problem that was shown to be $\mathcal{NP}$-complete is the satisfiability problem of propositional logic.

Theorem 2.2.2 *The satisfiability problem of propositional logic is* $\mathcal{NP}$*-complete.*

Until today a huge bunch of problems have been proved to be $\mathcal{NP}$-complete; among those are the traveling salesman problem, the flow shop problem, the quadratic assignment problem, and many others. We will encounter some of these problems in this thesis, where efficient algorithms to find near-optimal solutions to these problems are discussed.

In this thesis we are concerned with optimization problems and address the search version of the problem, that is, we want to find optimal solutions. Clearly, the search version is not easier than the associated decision problem. Thus, proving that the decision version of a problem is $\mathcal{NP}$-complete implies that also the search version is hard to solve. Problems which are at least as hard as $\mathcal{NP}$-complete problems but not necessarily element of $\mathcal{NP}$ are called $\mathcal{NP}$-hard.

Definition 2.2.5 *A problem* Π *is* $\mathcal{NP}$*-hard iff for all* $\Pi' \in \mathcal{NP}$ *holds that* Π' is polynomially reducible to Π.

Therefore, any $\mathcal{NP}$-complete problem is also $\mathcal{NP}$-hard. On the other side, if the decision version of an optimization problems is $\mathcal{NP}$-complete, the optimization problem is $\mathcal{NP}$-hard.

2.2.2 Approximation Algorithms

An important conclusion from the $\mathcal{NP}$-hardness of many combinatorial optimization problems is that we cannot expect an exact algorithm to solve a given instance to optimality in polynomially bounded computation time. Thus, it is often necessary to relax the requirement of finding optimal solutions and instead to try to find *good* solutions in polynomial time. Algorithms that try to find good solutions to optimization problems are also called *approximate* or *heuristic algorithms*. A natural question arising for approximate algorithms is how close, in the worst case, is the returned solution to the optimum. To indicate the quality of the returned solution the *relative error* is defined as follows.

Definition 2.2.6 *The* relative error *of a feasible solution y with respect to an instance x of an optimization problem Π is defined as*

$$E(x, y) = \frac{|opt(x) - f(y)|}{\max\{opt(x), f(y)\}} \tag{2.2}$$

By using the maximum of opt(x) and $f(x)$, the relative error is bounded to take values between 0 and 1. The relative error is close to 1 if the feasible solution is far from the optimal solution; it is close to 0 if the feasible solution is close to the optimum. Using the relative error we can define the notion of an ϵ-approximation algorithm.

Definition 2.2.7 *An algorithm A is an ϵ-approximation algorithm if it returns for every instance x of an optimization problem π a solution with $E(x, y = A(x)) \leq \epsilon$.*

Concerning the problem side, we are interested in the smallest ϵ such that there exists a polynomial-time ϵ-approximation algorithm. This ϵ is called the approximation threshold of a problem π.

Definition 2.2.8 *The* approximation threshold *of a problem π is the greatest lower bound of all $\epsilon > 0$ such that there is a polynomial-time ϵ-approximation algorithm.*

Now, problems can be classified according to the relative error obtained in the worst case. Usually, the following classes are defined [10].

Definition 2.2.9 *An optimization problem Π belongs to the class $\mathcal{APX}$ if it admits an ϵ-approximate polynomial-time algorithm A for some ϵ with $0 < \epsilon < 1$.*

Thus, $\mathcal{APX}$ is the class of problems that can be approximated up to a constant factor. For many problems it can even been shown that no polynomial-time approximation algorithm can exist that provides a relative error lower than some given ϵ, unless $\mathcal{P} = \mathcal{NP}$. Such a problem is, for example, the graph coloring problem; it is well known that guaranteeing a relative error of $\epsilon < 1/4$ is $\mathcal{NP}$-complete. Some problems even cannot be approximated within a constant factor, that is their approximation threshold is one. Hence, for such a problem finding a near-optimal solution is as hard as solving the problem itself. An example for such a problem is the TSP. Yet, for special cases of the TSP performance guarantees can be given. For example, if the distances obey the triangle inequality (the triangle inequality says that for all cities i, j, k holds $d(c_i, c_j) \leq d(c_i, c_k) + d(c_k, c_j)$; informally speaking, the triangle inequality says that

detours do not pay off), a performance guarantee of $1/3$ can be obtained using the Christofides algorithm [45].

For some problems an *approximation scheme* exists, that is, an algorithm A that for any given instance $(\mathcal{S}, f)$ of Π and any given $\epsilon \in (0, 1)$ returns a feasible solution with relative error at most ϵ.

Definition 2.2.10 *An optimization problem Π belongs to the class PTAS if it admits a polynomial-time approximation scheme, that is, an approximation scheme whose time complexity is bounded by a polynomial, depending on ϵ, of the instance size.*

Recall that the time complexity of an approximation scheme may be exponential in ϵ; it only has to be polynomial in the instance size. Therefore, to yield a small relative error ϵ may be infeasible. Interestingly, a problem which has been shown to have a PTAS is the *Euclidean* TSP [9]. (Note that Euclidean TSP instances obey the triangle inequality, yet TSP instances obeying the triangle inequality need not be Euclidean and for non-Euclidean instances which obey the triangle inequality no PTAS is known.) Yet, there may even exist problems that allow an approximation scheme that is even a polynomial in ϵ; in this case the approximation scheme is called *fully* polynomial.

2.3 Solution Methods for Combinatorial Problems

Due to the practical importance of combinatorial (optimization) problems, many algorithms for their solution have been devised. These algorithms can be classified as either *exact* or *approximate* algorithms. Exact algorithms are guaranteed to solve every finite size instance of a combinatorial optimization problem within an instance-dependent run-time. Yet, due to the inherent complexity of combinatorial optimization problems, many of them are $\mathcal{NP}$-hard, exact methods need exponential run-time in the worst case and one would have, informally speaking, wait for years to get an answer. Therefore, one has to resort to more ad-hoc methods and typically to sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in polynomial-time using approximate algorithms.

2.3.1 Exact Algorithms

For finite size problems a straightforward exact algorithm is to simply enumerate the full solution space. Yet, such an algorithm is infeasible due to the exponential size of the solution space. To increase efficiency, all modern exact methods use pruning rules to discard parts of the search space in which the (optimal) solution cannot be found. These approaches are doing an *implicit enumeration* of the search space. For optimization problems the best known examples are branch & bound algorithms, also known as A^* in the AI community [152, 201], and dynamic programming [24]. For satisfaction problems most algorithms are improvements over simple backtracking-style algorithms. Examples are algorithms for the solution of CSPs [147, 162, 249] or the Davis-Logemann-Loveland procedure [59] for the satisfiability problem and its modern variants [54, 76, 153]. Clearly, an advantage of exact methods for satisfaction problems is that they are able to show that instances cannot have solutions. But still, an exact algorithm may not find solutions on satisfiable instances in reasonable time.

For some specific problems, exact algorithms have been improved significantly in recent years and yield impressive results. This is the case, for example, for the *Euclidean* traveling salesman problem. Using modern branch & cut algorithms [196, 8, 104], the largest non-trivial instance that has been solved optimally comprises 13509 cities. But, the time needed to find the optimal solution and prove its optimality may take very long time. For example, it is reported in [8] that it took several CPU-years on a network of modern Workstations (Sun Sparc-2) to solve an instance with 7397 nodes. Furthermore, the efficiency of the optimization codes depends very strongly on the characteristics of the instances. There are still much smaller instances of TSPLIB exist for which the optimality could not be proved yet. Despite these successes, on many combinatorial optimization problems the performance of exact algorithms is much less formidable. In [129] it is suggested that "the TSP is not a *typical* combinatorial optimization problem, since most such problems seem significantly harder to solve to optimality". Such an example is the later introduced quadratic assignment problem, for which instances of dimension $n > 25$ are currently beyond the capabilities of state-of-the-art branch & bound algorithms.

2.3.2 Approximate Algorithms

General Remarks

Approximate algorithms differ essentially from exact ones as they cannot guarantee to find optimal solutions in finite time or prove that no solutions exist in the case of satisfaction problems. But, for optimization problems, they often find high quality solutions much faster than exact algorithms and are able to successfully attack large instances. Approximate algorithms can be classified as either constructive or local search algorithms. Additionally, approximate methods may also be obtained by stopping exact methods before completion, for example, after some given time bound. Yet, this type of approximate algorithms will not further be discussed here.

Constructive algorithms generate solutions from scratch by adding to an initially empty solution solution components in some order until a solution is complete. They are typically the fastest approximate methods, yet they often return solutions of inferior quality when compared to local search algorithms.

The most effective approximate algorithms today are *local search* algorithms. *Local search* algorithms start from some initial solution and iteratively 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 iterative improvement which only replaces the current solution with a better one and stops as soon as no better, neighbored solutions can be found anymore. For a discussion of this algorithm and specific issues for local search, we refer to the next section. Often, constructive algorithms are used to generate good initial solutions for the subsequent application of a local search algorithm. For many problems this has been shown to be a promising approach to provide better solutions than when starting the local search from randomly generated solutions.

Local search is not a particularly new method for attacking $\mathcal{NP}$-hard problems. First applications of local search have already been described in the late fifties and the early sixties [55, 155]. Yet, the initial interest in local search algorithms decreased because of the lack of new conceptual work and its success has only been based on its practical usefulness. Addition-

ally, if high solution qualities are required and large problems are to be solved, high computing power is needed which was not available in the early years of computer science [1].

It is only in the last ten to fifteen years that local search algorithms have become very popular and that they are very successfully applied to many problems. The renewed interest in local search algorithms has several reasons [1]. An important aspect is that local search algorithms are intuitively understandable, flexible, generally easier to implement than exact algorithms, and in practice have shown to be very valuable when trying to solve large instances. The solution of large instances has been made feasible by the development of more sophisticated data structures, for example, to search more efficiently the neighborhood of solutions, and the enormous increase in computer speed and memory availability. Also on the theoretical side considerable progress has been made. One such progress is the treatment of local search from a complexity point of view [130] that lead to a renewed interest of theoreticians in local search algorithms. Additionally, some of the recently developed local search algorithms can be analyzed mathematically; an example are the convergence proofs for simulated annealing [87, 106].

In fact, simulated annealing is a general search scheme that can be applied to many different problems to improve local search performance. Several other such general search schemes have been developed in recent years and many of them are inspired by naturally occurring processes. The inspiring processes have a strong appeal to design new local search paradigms and have led to considerably improved applications of local search. Among the naturally inspired metaheuristics are simulated annealing [139, 44], evolutionary algorithms [12] (with the main representatives being genetic algorithms [115, 98], evolution strategies [220], and evolutionary programming [75]), neural networks [123], and ant colony optimization [63, 48, 68]. These general search schemes are nowadays called *metaheuristics* [195, 194]. In general, metaheuristics are defined to be an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts of search space exploration and exploitation in order to find efficiently near-optimal solutions [195]. Metaheuristics are designed to be *general-purpose* algorithms that can be applied without major modifications to many problems. Apart from the nature-inspired metaheuristics, several others have been devised to intelligently solve hard combinatorial optimization problems without recurring to analogies with natural phenomena. The best known of these are tabu search [90, 91, 92, 107], iterated local search algorithms [170, 126, 109] and GRASP [72, 73].

In this thesis, we focus on applications of the three metaheuristics ant colony optimization, iterated local search, and tabu search. The first of these approaches has been rather unexplored, at the time when we started this research, and we investigate its potential for the efficient solution of combinatorial optimization problems. Iterated local search is chosen because it is one of the most simple metaheuristics. On the other side, we apply tabu search because in some preliminary tests it appeared to be the best performing extension of local search algorithms for constraint satisfaction problems (see also Chapter 6). For a short description of these algorithms we refer to Section 2.7.

2.4 General Issues of Local Search

Local search is the most successful general approach for finding high quality solutions to hard combinatorial optimization problems in reasonable time. It is based on the iterative exploration

of neighborhoods of solutions trying to improve the current solution by local changes. The type of local changes that may be applied to a solution is defined by a neighborhood structure.

Definition 2.4.1 *A* neighborhood structure *is a function* $\mathcal{N} : \mathcal{S} \mapsto 2^{\mathcal{S}}$ *that assigns to every* $s \in \mathcal{S}$ *a set of neighbors* $\mathcal{N}(s) \subseteq \mathcal{S}$. $\mathcal{N}(s)$ *is also called the* neighborhood *of* s.

The choice of an appropriate neighborhood structure is crucial for the performance of the local search algorithm and often has to be done in a problem specific way. It defines the set of solutions that can be reached from s in one single step of a local search algorithm. Typically, a neighborhood structure is not defined by explicitly enumerating the set of possible neighbors, but rather implicitly by defining the possible local changes that may be applied to a solution. The neighborhood structure can also be conceived as a graph, called the *neighborhood graph*. The nodes of the graph represent the solutions and two solutions are connected by an edge (in case the neighborhood structure is symmetric) if they are neighbors. Thus, the execution of a local search algorithm corresponds to a walk on the neighborhood graph.

In general, the solution found by a local search algorithm will not be a globally optimal solution, it may only be guaranteed to be optimal with respect to local changes.

Definition 2.4.2 *A* local minimum *is a solution* s *such that* $\forall s' \in \mathcal{N}(s) : \ f(s) \leq f(s')$. *We call a local minimum a* strict local minimum *if we have* $\forall s' \in \mathcal{N}(s) : \ f(s) < f(s')$.

When solving a minimization problem we are searching for a distinguished local minimum, the *global* minimum, that is, a solution s_{opt} such that $\forall s' \in \mathcal{S} : \ f(s_{opt}) \leq f(s')$. The quality of a solution is given by the objective function value $f(s)$, that usually is the one associated with the combinatorial optimization problem. Yet, for some problems other objective functions than the one given in the problem definition are sometimes used for local search. An example is non-oblivious local search for MAX-SAT [138]. In this case, the reason for using another objective function is the better theoretical approximation guarantee.

Certainly, the solution quality obtained by a local search algorithm increases with larger neighborhood size. A neighborhood which guarantees that every local optimal solution is also a global optimum is called *exact*. Unfortunately, such neighborhoods are typically of exponential size and searching an improved neighboring solution may take exponential time in the worst case. For practical reasons it is required that each step of local search can be done in polynomial time. This requires that the neighborhood can be searched in polynomial time and that the cost difference $\Delta f = f(s) - f(s')$ can be calculated in polynomial time. Note that for many problems neighborhoods exist such that Δf can be calculated fast without the need to compute $f(s')$ from scratch. Indeed, the empirical success of local search is often due to the fact that the neighborhood can be examined fast and that the empirical number of changes applied to find a local minimum is a low-order polynomial, often linear in problem size.

The most basic local search algorithm is *iterative improvement*. Iterative improvement typically starts with an initial solution generated randomly or by some constructive heuristic and tries to repeatedly improve the current solution by moves to better neighboring solutions. If in the neighborhood of the current solution a better solution s' is found, it replaces the current solution and the search is continued from s'; if no better solution is found, the algorithm terminates in a local minimum. In Figure 2.2 is given an outline of the iterative improvement algorithm.

```
procedure IterativeImprovement (s ∈ S)
    while Improve(s) ≠ no do
        s = Improve(s)
    end
    return s
end IterativeImprovement
```

Figure 2.2: Algorithmic skeleton of iterative improvement.

```
procedure FirstImprovement(s ∈ S)
    w = ∅
    while (N(s)/w ≠ ∅) do
        s' = GenerateSolution(s)
        if (f(s') < f(s))
            return s'
        else
            w = w ∪ s'
    end
end FirstImprovement
```

Figure 2.3: Algorithmic skeleton of a first improvement pivoting rule

The procedure Improve returns a better neighboring solution if one exists, otherwise it returns *no*. As in this thesis we are only attacking minimization problems, we will also use the term *iterated descent* for this most basic local search algorithm.

In a local search algorithm the *pivoting rule* determines which neighboring solution replaces the current one [262]. The most often used ones are the *first-improvement* and the *best-improvement* rule. In first-improvement the neighborhood is scanned and the first found lower cost solution replaces the current solution. A first-improvement procedure, called FirstImprovement is given in Figure 2.3. GenerateSolution repeatedly generates neighboring solutions of s and the first lower cost solution is returned. In a local optimum the whole neighborhood is checked. The time needed to check the whole neighborhood is also called the *check-out time*. The best-improvement pivoting rule examines in each step the whole neighborhood and returns the best neighboring solution. The first-improvement pivoting rule has the advantage that local minima are typically found faster, yet generally more moves have to be performed than when using best-improvement. Sometimes also a best-improvement pivoting rule may yield solutions of slightly higher quality. An advantage of using first-improvement is that if the neighborhood is scanned in a random order, from the same starting solution different final solutions can be reached. We refer to [199] for a more detailed discussion of these issues.

Example To illustrate the application of local search algorithms, we give examples of neighborhood structures for the TSP and the CSP. For the symmetric TSP the neighborhood can be

defined by removing a fixed number of k arcs and replacing them with k other arcs, leading to a `k-opt` algorithm. For $k = 2$ and $k = 3$, we obtain the well known `2-opt` [55] and the `3-opt` [155] iterative improvement algorithms for the TSP. These are also the most often used ones because for k larger than three the run-time increases fast, because the number of tours in the k-neighborhood is of order $\mathcal{O}(n^k)$ and an examination of the neighborhood is computationally too expensive compared to the gain in solution quality. A straightforward first-improvement algorithm for the TSP searches the neighborhood consisting of all possible `k-opt` moves until an improving move is found, replaces the current tour by the new one and continues. If the tour cannot be improved any more, the whole neighborhood has to be examined to establish local optimality. Hence, the check-out time for a `k-opt` algorithm is $\mathcal{O}(n^k)$.

For a CSP the neighborhood can be defined as consisting of the set of solutions which differ in at most one variable instantiation. A straightforward iterative improvement algorithm then starts with some candidate solution and tries to reduce the number of violated constraints iteratively by assigning a different value to some variable in each step. In Chapter 6 we will discuss local search algorithms for CSPs in more detail.

2.5 Computational Complexity of Local Search

In general, one cannot guarantee a bound on the solution quality of local optima for general combinatorial optimization problems. Guarantees may only be given for *exact* neighborhoods, that is, if every local minimum is guaranteed to be also a global minimum. Yet, exact neighborhoods are of exponential size and therefore searching it is infeasible. Furthermore, for some problems the number of steps needed by an iterative improvement algorithm to find a local optimum cannot be bounded by a polynomial. This is contrasted by the fact that empirically for all problems local minima can be found in reasonably low amounts of time. To address the question of the worst-case complexity of local search algorithms, in [130] the class $\mathcal{PLS}$ (polynomial-time local search) is introduced. Intuitively, $\mathcal{PLS}$ is the class of local search algorithms for which each local search step requires polynomial time. This means that local optimality can be verified in polynomial time or, in case a solution is not local optimal, a neighboring solution of better cost can be generated in polynomial time.

Definition 2.5.1 *A local search problem* Π *is in the class* $\mathcal{PLS}$ *of* polynomial-time local search *problems if the following three polynomial time algorithms exist.*

(i) Algorithm A which applied to instance x *computes a candidate solution* $s \in \mathcal{S}$.

(ii) An algorithm B which computes $f(s)$.

(iii) An algorithm C which either determines that s *is locally optimal or finds a better solution* $s' \in \mathcal{N}(s)$.

Of main importance for the class $\mathcal{PLS}$ is the number of steps needed to find a local optimum. To be able to define $\mathcal{PLS}$-complete problems, the notion of reducibility, here $\mathcal{PLS}$-reducibility, has to be introduced.

Definition 2.5.2 *A problem $\Pi \in \mathcal{PLS}$ is* $\mathcal{PLS}$-reducible *to another problem Π' if there exist polynomially computable functions g and h such that*

(i) g maps instance x of Π to an instance $g(x)$ of Π';

(ii) h maps (solution of $g(x), x$) pairs to a solution of x;

(iii) for all instances x of Π, if s is a local optimum for instance $g(x)$ of Π', then $h(s, x)$ is a local optimum for x.

Like the reductions used for proving $\mathcal{NP}$-completeness, $\mathcal{PLS}$ reductions can be composed, that is, they are transitive, and if we can find a locally optimal solution to problem Π' in polynomial time, we can also do so for problem Π. We can now give the definition of $\mathcal{PLS}$-completeness.

Definition 2.5.3 *A problem Π in $\mathcal{PLS}$ is* $\mathcal{PLS}$-complete *if every problem in $\mathcal{PLS}$ can be reduced to it.*

Like for $\mathcal{NP}$-complete problems, $\mathcal{PLS}$-complete problems contain the hardest problems in $\mathcal{PLS}$. If for any of these problems a local optimum can be found in polynomial time, so can for all problems in $\mathcal{PLS}$. Yet, it is conjectured that the class of polynomially solvable search problems is a strict subset of $\mathcal{PLS}$. Thus, in the worst case super-polynomial run-time may be required by any algorithm to find a local minimum of a $\mathcal{PLS}$-complete problem. The first well-known combinatorial optimization problem that was shown to be $\mathcal{PLS}$-complete is graph-partitioning of weighted graphs under the Kernighan-Lin neighborhood [137]. In [197] $\mathcal{PLS}$-completeness has also been shown for the most efficient local search algorithm for the TSP, the Lin-Kernighan heuristic [156]. Also for `k-opt` with sufficiently large k $\mathcal{PLS}$-completeness has been shown [146]; the question of $\mathcal{PLS}$-completeness of `2-opt`and `3-opt` remains open [129].

2.6 Construction Heuristics

Local search algorithms may start from a randomly generated solution. Yet, usually a better idea is to invest some effort to construct, guided by the objective function, a reasonably good solution and to start the local search from such a solution. This often has the advantage that the local search finds better quality local minima and needs less steps to reach a local minimum. Construction heuristics are also of interest on their own, because they are typically the fastest methods to get reasonably good solution. Often, construction heuristics generate a solution in a greedy manner adding one solution component at a time until a complete solution is obtained. The myopic benefit of adding a component can be measured by the contribution to the objective function value and the component which causes the least cost increase is chosen. A generic greedy construction heuristic is outlined in Figure 2.4. The function GreedyComponent returns a solution component e that least increases the objective function value of the partial solution s. The addition of component e to s is denoted by the operator $\otimes$. These steps are then repeated until a solution is completed. Yet, a problem which often occurs with greedy construction heuristics is that, due to the myopic initial decision, in the later construction phase

```
procedure Greedy Construction Heuristic
    s = empty solution
    while s no complete solution do
        e = GreedyComponent(s)
        s = s ⊗ e
    end
    return s
end Greedy Construction Heuristic
```

Figure 2.4: Algorithmic skeleton of a greedy construction heuristic

poor decisions have to be made. Therefore, the constructed solutions need not be locally optimal with respect to some simple neighborhood and the poor decisions made in the solution construction can often be repaired by a local search.

An example for a very commonly used construction heuristic for the TSP is the *nearest neighbor heuristic* (NN). It is probably the most intuitive construction heuristic for the TSP and it mimics the simple rule of thumb to always choose at the current city the nearest city next. Initially, in the NN heuristic a starting city $c_{\pi(1)}$ is chosen randomly and then iteratively from the current city the nearest not visited city is taken as the next city $c_{\pi(i+1)}$ (that is, a city which minimizes $d(c_{\pi(i)}, c_k)$ with $k \neq c_{\pi(j)}, 1 \leq j \leq i$). Ties are broken randomly. A disadvantage of the NN heuristic is that at the end of the construction process long arcs have to be added to the tour. Typically, nearest neighbor tours contain a few *bad* mistakes that can easily be undone in a subsequent local search phase [211]. For CSPs a greedy construction heuristic could work as follows. First a variable is assigned some value of its domain. Then in each step one more variable involved in a constraint with an instantiated variable is assigned a value. This value assignment can be done such that it minimizes the number of constraints that will become definitely violated by the current partial assignment.

Greedy construction heuristics typically construct reasonably good starting points for a local search algorithm. Yet, a disadvantage is that the possible number of different solutions is strongly limited; for example, there are at most n different NN tours (excluding ties) for the TSP. Thus, if local search should be applied very often, many different starting solutions are needed. To achieve this goal, randomized construction heuristics can be used in which the components to be added to a partial solution are chosen according to a probability distribution.

2.7 Metaheuristics for Local Search

A major drawback of iterative improvement local search is that it may stop at very poor quality local minima. A straightforward possibility is to restart the local search from a new, randomly or greedily generated solution until some stopping criterion, like a maximally allowed CPU-time or a maximal number of restarts, is met. The best local minimum found during this *multiple descent* approach is then returned as the final solution. While multiple descent may give good results for small instances, an often faced problem with such an approach is the curse of di-

mensionality: For increasing problem size — and therefore increasing dimensionality of the search space — the number of local minima may increase exponentially and more effective ways to improve solution quality have to be devised. Additionally, with restarts from random initial solutions possible search space structures (see Section 2.9 for their discussion) are not exploited.

In recent years a significant amount of research effort has been spent in the design of general heuristic methods which are applicable to a wide range of different combinatorial optimization problems. For these *general-purpose* methods the term *metaheuristics* has been coined. Metaheuristics are typically high-level strategies which guide an underlying, more problem specific heuristics, to increase their performance. The main goal is to avoid the disadvantages of iterative improvement and, in particular, multiple descent by allowing the local search to escape from local optima. This is achieved by either allowing worsening moves or generating new starting solutions for the local search in a more "intelligent" way than just providing random initial solutions. Many of the methods can be interpreted as introducing a bias such that high quality solutions are produced quickly. This bias can be of various forms and can be cast as descent bias (based on the objective function), memory bias (based on previously made decisions) or experience bias (based on prior performance). Many of the metaheuristic approaches rely on probabilistic decisions made during the search. But, the main difference to pure random search is that in these algorithms randomness is not used blindly but in an intelligent, biased form.

Many of the recently developed metaheuristics are inspired by natural phenomena. Examples are ant colony optimization, simulated annealing, and genetic algorithms. The terms used in the context of these algorithms are strongly influenced by the inspiring phenomenon itself. When applied to combinatorial optimization problems, the main aim of these algorithms is not to give a model of the natural occurring phenomenon, but to provide efficient solution techniques to yield high quality solutions in reasonable time and the algorithms will differ rather strongly from the natural analogy, when applied to specific problems. Yet, the inspiring phenomenon still may propose specific ideas of how to further improve the algorithms for specific applications.

In this section we present ant colony optimization, iterated local search, and tabu search which are applied and discussed in more detail in Chapters 4-6, respectively. Additionally, the main features of simulated annealing, genetic algorithms, and GRASP are outlined. These latter three algorithms are also used in the next section to identify commonalities and differences between these methods. We detail the most basic concepts and present for each metaheuristic an algorithmic scheme in a pseudo code notation. For more general introductions to local search and specific metaheuristics we refer to the literature [3, 96, 207].

2.7.1 Ant Colony Optimization

Ant colony optimization (ACO) is a new population-based search metaphor inspired by the foraging behavior of real ants [63, 67, 68, 65]. (Recent information on ACO can be found at `http://iridia. ulb.ac.be/~dorigo/ACO/ACO.html`.) Ants are simple insects that live together in colonies and show astonishing capabilities through their cooperative behavior like finding shortest paths from a food source to their home colony. The ants exchange information via pheromones; pheromones are chemical substances which the ants may lay down

in varying quantities to mark, for example, a path. While isolated ants move essentially at random, an ant encountering a previously laid pheromone trail can detect it and may follow the pheromone trail. The ants' probability to follow the pheromone trail is the larger the higher the pheromone intensity is. At the same time the ants following the pheromone trail may lay down additional pheromone and a positive feedback loop results: the more ants previously have chosen the pheromone trail, the more ants will follow it in the future. One of the basic ideas of ant colony optimization is to use an algorithmic counterpart of the pheromone trail as a medium for cooperation and communication among a colony of artificial ants which is guided by *positive feedback*.

The seminal work on ACO is ant system (AS) [67, 63, 68]. AS is a population oriented, cooperative search algorithm. The ants are simple agents which are used to construct solutions to combinatorial optimization problems guided by *(artificial) pheromone trails* and heuristic information. The pheromone trails are associated with *solution components* (in the application of ant system to the TSP, these are the arcs connecting the cities). Solutions are constructed probabilistically preferring to use solution components with a high pheromone trail and a promising heuristic information. In fact, the ants implement a randomized construction heuristics. Randomized construction heuristics differ from greedy heuristics in the fact that probabilistically a component is added to the partial solution instead of making a deterministic choice. AS then consists of two phases. In a first phase all ants construct a solution and in a second phase the pheromone trail information is updated. This is done by first reducing the pheromone trails by a constant factor to avoid unlimited accumulation. Then, the ants reinforce the components of their solutions by depositing an amount of pheromone which is the higher the better the solution quality is.

Ant system has first been applied to the TSP. Although it was able to find very good solutions for some small instances, the solution quality when applied to larger instances is not satisfying and the computation times become very high. Therefore, in recent years several extensions of the basic algorithm have been proposed. Among these extensions are ant-Q [82], ant colony system (ACS) [83, 66], $\mathcal{MAX}$–$\mathcal{MIN}$ ant system [235, 237] and the rank-based version of ant system [36]. All these extensions are in some sense "greedier" than ant system, but differ significantly in important aspects of the search control. Recently, the ACO metaheuristic [65, 64] has been proposed which gives a framework for recent algorithmic developments in ant colony optimization. We will use the term ACO algorithm in the following to indicate that a specific ant-based algorithm fits into the framwork of the ACO metaheuristic.

The most important part in ACO algorithms, in general, is how the pheromone trails are used to generate better solutions in future iterations of the algorithm. The idea is that by combining solution components that in previous iterations have shown to be part of good solutions, even better solutions may be generated. Thus, ACO algorithms can be seen as adaptive sampling algorithms – adaptive in the sense that they consider past experience to influence future iterations.

The performance of ACO algorithms can be significantly improved by adding a local search phase [165, 66, 235, 236], in which some or all ants are allowed to improve their solutions by a local search algorithm. Hence, the most efficient ACO algorithms are in fact hybrid algorithms combining a probabilistic solution construction by a colony of ants with a subsequent local search phase. These locally optimal solutions are then used to provide feedback. Thus, the best performing ACO algorithms identify components of the best locally optimal solutions and by

procedure *Ant Colony Optimization*
  Initialize pheromone trails, calculate heuristic information
  while(termination condition not met) **do**
    $p =$ ConstructSolutions(*pheromone trails, heuristic information*)
    $p =$ LocalSearch(p) *% optional*
    GlobalUpdateTrails(p)
  end
end *Ant Colony Optimization*

Figure 2.5: Algorithmic skeleton of ant colony optimization algorithms for static combinatorial optimization problems.

combining these features in an appropriate construction process they direct the sampling of new starting solutions for the local search towards promising regions of the search space.

In Figure 2.5 we give an algorithmic skeleton into which fit the ACO algorithm applications for static combinatorial optimization problems (static combinatorial optimization problems are those in which all relevant problem data are available before the start of the algorithm and do not change during the algorithm's run); for an outline of the more general *ACO metaheuristic* we refer to [65]. In the main loop of the algorithm, first solutions are generated for all ants of the colony (the colony is indicated by p) by a function ConstructSolutions. The solution construction typically uses the pheromone information and a problem specific local heuristic information. The solutions are then improved by a local search phase (LocalSearch). This local search phase is optional; in fact, it is not used in all applications of ACO algorithms to combinatorial optimization problems. Finally, the solutions are used to update the pheromone trails in a function GlobalUpdateTrails.

For a detailed description of the algorithms falling into the ant colony optimization paradigm, we refer to Chapter 4. There, we present in detail $\mathcal{MAX}$–$\mathcal{MIN}$ ant system. The term $\mathcal{MAX} - \mathcal{MIN}$ is derived from the fact that in $\mathcal{MAX}$–$\mathcal{MIN}$ ant system explicit pheromone trail limits are introduced to increase the solution exploration of the algorithm. We present applications of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system to the traveling salesman problem, the quadratic assignment problem, and the flow shop problem, showing that $\mathcal{MAX}$–$\mathcal{MIN}$ ant system is currently the best performing ACO algorithm for these problems and is very competitive with other approaches.

2.7.2 Iterated Local Search

A major problem for local search algorithms is that they may get trapped in local minima in the search space. Exactly in such a situation, an action should take place that allows the local search to leave local minima and to continue the search for possibly better solutions. One straightforward possibility is to modify the current locally optimal solution s using a modification larger than those used in the local search algorithm. The application of such a move yields some intermediate solution s' beyond the neighborhood searched by the local search algorithm and allows to leave local minima. The local search is then continued from s'. Iterated local search

(ILS) [168, 169, 180, 234] systematically uses this idea to solve combinatorial optimization problems. In ILS a local search algorithm is applied repeatedly from initial solutions obtained by modifications to one of the previously visited locally optimal solutions.

ILS is a simple, yet powerful metaheuristic to improve the performance of local search algorithms. The simplicity stems from the underlying principle and the fact that only few lines of code have to be added to an already existing local search procedure to implement an ILS algorithm. ILS also can be expected to perform better than to restart local search from a new, randomly generated solution. This is emphasized by the fact that ILS algorithms are currently among the best performing approximation methods for many combinatorial optimization problems like the traveling salesman problem [129, 168].

To apply an ILS algorithm to a given problem, three "ingredients" have to be defined. One is a procedure Modify, that perturbs the current solution s (usually a local optimum) leading to some intermediate solution s'. We will refer to the perturbation also as *kick-move* in the following. Next, LocalSearch is applied taking s' to a local minimum s''. Finally, one has to decide which solution should be chosen for the next modification step. This decision is made according to an AcceptanceCriterion that takes into account the previous solution s, the new candidate solution s'' and possibly the search *history*.

The perturbation introduced by the procedure Modify is used with the objective to leave the current locally optimal solution. Often it is a randomly chosen move from a higher-order neighborhood than the one used in the local search algorithm or a move which the local search cannot reverse in one single step. The particular kick-move applied may depend also on the search history. One possibility is to choose the strength of the "kick-move" or the "direction" of the kick-move based on *history*. For the local search, in principle, any local search algorithm can be applied. Yet, it is obvious that ILS performance with respect to solution quality and computation speed strongly depends on the local search algorithm. The acceptance criterion is used to decide which solution is perturbed next. The choice of the acceptance criterion is important, because it controls the balance between exploitation and exploration. Exploitation is achieved, for example, by accepting only better local optima for Modify. In fact, such an acceptance criterion is applied in most ILS implementations. Such a choice may be preferable if near the current local optimum even better solutions may be found. Exploration may be achieved, in the extreme case, by accepting every new local optimum s'', similar to a random walk over the local optima. Yet, also the search *history* may be used to decide whether some earlier found locally optimal solution should be chosen or it may influence the decision between s and s''.

An algorithmic outline for ILS algorithms is given in Figure 2.6. It should be mentioned here that different algorithms, which perfectly fit into this framework, are addressed using different terms in the literature. Examples are *large step Markov chains* [169], *chained local optimization* [168], *simple variable neighborhood search* [109], or algorithm specific names like iterated Lin-Kernighan [129] etc. All earlier proposed ILS algorithms fit into this framework and different implementations differ mainly in just one of the choices for LocalSearch, Modify, and AcceptanceCriterion. Therefore, this algorithmic frame can be seen as a unifying framework for algorithms which are based on the same idea, namely that of repeatedly applying local search to perturbations of previously found local optima.

procedure *Iterated Local Search*
generate initial solution s
$s = \mathsf{LocalSearch}(s)$
$s_{best} = s$
while(termination condition not met) **do**
$s' = \mathsf{Modify}(s, history)$
$s'' = \mathsf{LocalSearch}(s')$
if $(f(s'') < f(s_{best}))$
$s_{best} = s''$
$s = \mathsf{AcceptanceCriterion}(s, s'', history)$
end
return s_{best}
end *Iterated Local Search*

Figure 2.6: Algorithmic skeleton of an iterated local search procedure (ILS)

ILS will be discussed in more detail in Chapter 5. In particular, we investigate the role of the acceptance criterion for earlier proposed ILS algorithms applied to the traveling salesman problem and new applications to the asymmetric traveling salesman problem, the quadratic assignment problem, and the flow shop problem.

2.7.3 Tabu Search

Tabu search (TS) is an iterative local search metaheuristic [90, 91, 92, 107]. The most distinctive feature of tabu search compared to other metaheuristics is the systematic use of a *memory* to guide the search process. For general overviews on tabu search and detailed discussions of its features, we refer to [91, 92, 97] and the recently published book by Glover and Laguna [96].

The most widely applied feature of tabu search is the use of a short term memory to escape from local minima. This version is denoted as *simple tabu search* in [91]. TS typically uses an aggressive local search that in each step tries to make the best possible move from s to a neighboring solution s' even if that move worsens the objective function value. To prevent the local search to immediately return to a previously visited solution and to avoid cycling, in TS moves to recently visited solutions are forbidden. This can be implemented by explicitly memorizing previously visited solutions and forbidding moving to those. More commonly, reversing recent moves is forbidden by disallowing the introduction of move attributes to a solution. In particular, reverse moves are forbidden for tl iterations; the parameter tl is called the *tabu tenure*. Forbidding possible moves has the same effect as restricting dynamically the neighborhood $\mathcal{N}(s)$ of the current solution s to a subset $\mathcal{A}$ of admissible solutions. Thus, tabu search can also be considered as a dynamic neighborhood search technique [112]. Yet, the tabu conditions may be too restrictive and they may forbid moves to attractive, unvisited solutions. *Aspiration criteria* are used to override the tabu status of certain moves and to avoid such situations. Most commonly, the aspiration criterion drops the tabu status of moves leading to a better solution than the best one visited so far.

```
procedure SimpleTabuSearch
    Initialize memory structures, generate initial solution s, s_best = s
    while termination condition not met
        A = GenerateAdmissibleSolutions(s)
        s = SelectBestSolution(A)
        UpdateMemoryStructures
        if (f(s) < f(s_best))
            s_best = s
    end
    return s_best
end SimpleTabuSearch
```

Figure 2.7: Algorithmic skeleton of a "Simple Tabu Search" algorithm

In Figure 2.7, we give a general algorithmic outline of a simple tabu search algorithm. The function GenerateAdmissibleSolutions is used to determine the subset of neighboring solutions which are not tabu or are tabu but satisfy the aspiration criterion. Since tabu search is an aggressive search strategy, the best admissible move is returned by the function SelectBestSolution and the tabu list is updated by function UpdateMemoryStructures. The best found solution is stored in s_{best}; $f(s_{best})$ is also used to determine the aspiration criterion.

To increase the efficiency of tabu search, techniques exploiting the *long-term memory* of the search process are used. These methods are used to achieve intensification or diversification of the search process. Intensification strategies correspond to efforts of revisiting promising regions of the search space either by recovering elite solutions (that is, the best solutions obtained so far) or attributes of these solutions. Diversification refers to exploring new search space regions corresponding to the introduction of new attribute combinations. Many long term memory strategies in the context of TS are based on a frequency memory on the occurrence of solution attributes. For a detailed discussion of particular techniques exploiting long-term memory we refer to [92, 95, 96].

To date, TS appears to be one of the most successful metaheuristics. For many problems, TS implementations are among the algorithms giving the best tradeoff between solution quality and the computation-time required [191, 253]. Yet, this efficiency is often due to a very significant fine-tuning effort of an apparently large collection of parameters and implementation choices [112]. Apart from the importance of a careful choice of the neighborhood structure, several other aspects are important for the empirical success of tabu search. Regarding the computation speed, it is important to use an efficient neighborhood evaluation and in many cases advanced data structures are used to speed up the computation. Another possibility is, instead of computing the exact cost of a neighboring solution, to use an estimate for the objective function value. This is important, in case the calculation of the exact cost is computationally expensive or itself would involve the solution of an ($\mathcal{NP}$-hard) optimization problem.

A crucial parameter for the performance of tabu search is the tabu tenure tl. If it is chosen too small, cycling may occur; if it is too large, the search path is too restricted and high quality solutions may be missed. A good parameter setting for tl can only be found empirically and requires considerable fine tuning. Therefore, several approaches to make the particular settings of tl more robust or to adjust tl dynamically during the run of the algorithm have been intro-

duced. More robustness regarding the choice of a particular value for tl is achieved in robust tabu search [241] by choosing the value of tl randomly from an interval $[tl_{min}, tl_{max}]$. Reactive tabu search [18] uses the search history to adjust the tabu tenure tl dynamically during the run of the algorithm. In particular, if solutions are repeated this is interpreted as evidence that cycling occurs and the tabu tenure is increased. If, on the contrary, no repetitions are found during a sufficiently long period, the tabu tenure is decreased gradually. Additionally, an escape mechanism based on a series of random changes is used to avoid getting trapped in a specific region of the search space.

2.7.4 Other Methods

Simulated Annealing

Simulated annealing (SA), independently proposed as an optimization algorithm in [139] and [44], is an iterative local search method motivated by an analogy between the physical annealing of solids (crystals) and combinatorial optimization problems. Physical annealing is the process of initially melting a substance and then lowering the temperature very slowly, spending a long time at low temperatures. The aim of the physical annealing process is to grow solids with a perfect structure; such a state corresponds to a state of minimum energy and the solid is said to be in a ground state. If the cooling is done too fast, the resulting crystal will have a metastable structure with irregularities and defects. Such an undesirable situation may be avoided by a careful annealing in which the temperature descends slowly through several temperature levels and each temperature is held long enough to allow the solid to reach thermal equilibrium. SA tries to solve combinatorial optimization problems by a process analogous to the physical annealing. The analogy associates the set of solutions of the problem with the states of the physical system, the objective function corresponds to the physical energy of the solid, and the ground state corresponds to a globally optimal solution.

When applying SA, in each step a tentative solution s' is generated. If s' improves the objective function value, it is accepted; if s' is worse than the current solution, then it gets accepted with a probability which depends on the objective function difference $f(s) - f(s')$ of the current solution s and s' and a parameter T, called temperature. This parameter T is lowered (as it is also done in the physical annealing process) during the run of the algorithm reducing the probability to accept worsening moves. The probability p_{accept} to accept worse solutions is often defined according to the Metropolis distribution.

$$p_{accept}(T, s, s') = \begin{cases} 1 & \text{if } f(s') < f(s) \\ \exp(\frac{f(s)-f(s')}{T}) & \text{otherwise} \end{cases} \tag{2.3}$$

In Figure 2.8 we give a general algorithmic outline for SA. SA starts from some initial solution s and generates in each step a new solution s'. This new solution s' is accepted or rejected according to an acceptance criterion like given in Equation 2.3.

To implement a simulated annealing algorithm, some parameters and functions have to be specified. Typically a random element of the neighborhood is returned by a function GenerateRandomSolution and accepted by a function AcceptSolution according to Equation 2.3.

```
procedure Simulated Annealing
    generate initial solution s, s_best = s, initial value for T_0, n = 0,
    while outer-loop criterion not satisfied do
        while inner-loop criterion not satisfied do
            s' = GenerateRandomSolution(s)
            s = AcceptSolution(T_n, s, s')
            if (f(s) < f(s_best))
                s_best = s
        end
        T_{n+1} = UpdateTemp(T_n), n = n + 1
    end
    return s_best
end Simulated Annealing
```

Figure 2.8: Outline of a simulated annealing algorithm

For an SA algorithm also an *annealing schedule* (often also called *cooling schedule*) has to be given. It is defined by an initial temperature T_0, a scheme saying how the new temperature is obtained from the previous one (UpdateTemp in Figure 2.8), the number of iterations to be performed at each temperature (inner loop criterion in Figure 2.8) and a termination condition (outer loop criterion in Figure 2.8).

Simulated annealing is of special appeal to mathematicians due to the fact that under certain conditions the convergence of the algorithm to an optimal solution can be proved [87, 106, 161, 215]. Mathematically, simulated annealing can be modeled using the theory of Markov chains. For a fixed temperature T a transition matrix P is defined giving for each solution s the probability to accept any $s' \in N(s)$ as the next state, independent of the iteration number. Under the condition that the probability to get from a solution s to any other solution s' is non-zero, the (homogeneous) Markov chain has a unique stationary distribution. For $T \mapsto 0$ the limiting distribution is a uniform distribution over the set of optimal solutions, that is, the simulated annealing algorithm converges asymptotically to the optimal solution. Yet, stationarity of the Markov chain is required and infinitely many steps are theoretically needed to reach it. Other convergence proofs relax the requirement of homogeneous Markov chains and allow inhomogeneous Markov chains in which the transition probabilities are not independent of the number of iterations [106].

Despite the attractiveness of such proofs, the practical consequences of these results are rather limited. The most severe limitation is that theoretically an infinite number of states has to be visited by the algorithm to guarantee convergence. By simply enumerating the search space one could guarantee to find the optimal solution even in finite time! Thus, a more important question would refer to the convergence speed of simulated annealing; yet, results addressing this question are rare. Also, practical annealing schedules decrease the temperature much faster than is implied by the theoretical results and therefore the convergence proofs do not apply in this case. In practical applications of simulated annealing one keeps track of the best solution found during the search process. Therefore, it would suffice to prove that the optimal solution

will be seen during the search process. Obviously, having proved the convergence of simulated annealing directly implies that the optimal solution will be found perhaps much earlier during the search process. Yet, such a feature alone also guarantees that the globally optimal solution is found even by pure *random search*, which is known to be a bad algorithm for combinatorial optimization problems.

Genetic Algorithms

Genetic algorithms (GAs) [115, 98] are a specific type of *evolutionary algorithms* [12]. Evolutionary algorithms are population-based, adaptive search algorithms designed to attack optimization problems. They are inspired by models of natural evolution of species and use the principle of natural selection which favors individuals that are more adapted to a specific environment for survival and further evolution. Each individual in an evolutionary algorithm typically represents a solution with an associated fitness value. The three main operators used are *selection*, *mutation*, and *recombination*. Selection prefers fitter individuals to be chosen for the next generation and for the application of the mutation and recombination operator. Mutation is a unary operator that introduces random modifications to an individual. Recombination combines the genetic material of two individuals, also called *parents*, by means of a *crossover* operator to generate new individuals, called *offsprings*.

The three main algorithmic developments within the field of evolutionary algorithms are genetic algorithms, evolution strategies [205, 220] and evolutionary programming [75]. These algorithms have been developed independently and, although these algorithms initially have been proposed in the sixties and seventies, only in the beginning of the nineties the researchers became aware of the common underlying principles of these approaches [12]. (For a detailed discussion of similarities and differences between these approaches we refer to [12].) Here we focus on genetic algorithms since they appear to be the best suited evolutionary algorithms for combinatorial optimization problems, which are the target of this thesis.

In the first GA applications, individuals were represented by bit strings of fixed length [115]. Yet, this type of representation proved to be insufficient to efficiently attack certain types of combinatorial problems [176], like permutation problems (permutation problems are problems in which a solution may be represented by a permutation of the numbers $1, \ldots, n$), that are naturally encoded in other ways. Therefore, for such problems usually more general, problem specific encodings are applied. The *crossover* operator is usually understood as the main operator driving the search in genetic algorithms [115, 98]. The idea of crossover is to exchange useful information between two individuals and in this way to generate a hopefully better offspring. Mutation is understood as a background operator which introduces small, random modifications to an individual. Yet, recent results suggest that the role of mutation has been underestimated [12]. The selection operator is used to keep the population at a constant size, choosing preferably individuals with higher fitness (survival of the fittest). The complete cycle of recombination, mutation and selection is called *generation*. Evolution strategies and evolutionary programming differ from genetic algorithms by representing solutions directly as real valued parameters (in case of genetic algorithm applications to continuous parameter optimization problems the numbers are coded in binary form) and the much stronger reliance on mutation as a primary search operator. Indeed, in evolutionary programming only mutation is used for modifying solutions.

procedure *Genetic Local Search*
  Generate initial population p, calculate fitness values, initialize parameters
  $p = \mathsf{LocalSearch}(p)$
  while(termination condition met) **do**
    $p' = \mathsf{Recombination}(p)$
    $p'' = \mathsf{Mutation}(p)$
    $p''' = \mathsf{LocalSearch}(p', p'')$
    $p = \mathsf{Selection}(p, p''')$
  end
end *Genetic Local Search*

Figure 2.9: Algorithmic skeleton for genetic algorithms

Many applications of genetic algorithms are concerned with combinatorial optimization problems. For that application, the use of a population is a convenient way to increase the exploration of the search space. Genetic algorithms try to evolve the population to locate promising regions of the search space in which high quality solutions may be found. Yet, often they lack a certain degree of exploitation of these regions and fine-tuning abilities are missing. Therefore, for many combinatorial optimization problems, individuals are improved by a local search [30, 239, 186, 251, 174, 175] or the crossover operator incorporates some form of local search [187]. These hybrid algorithms are referred to in the literature as *genetic local search* (GLS) [251, 142, 77, 174] or, more generally, as *Memetic Algorithms* [183, 184]. Thus, many of the most successful genetic algorithms in combinatorial optimization actually evolve a population of locally optimal solutions.

In Figure 2.9 we give an algorithmic skeleton for the application of genetic local search algorithms. Here, p denotes the population. A set of new individuals p' is generated in the function Recombination applying some crossover operator. After some individuals of the population are mutated, local search is applied to the newly generated solutions represented in p' and p''. In the last step the new population is determined by the Selection function.

GRASP

Greedy randomized adaptive search procedures (GRASP) [72, 73] are another example of a metaheuristic which allows to escape from local minima by generating new starting solutions. Each GRASP iteration consists of two phases, a construction phase and a local search phase. In the construction phase a solution is constructed from scratch, adding one solution component at a time. At each construction iteration the components to be added are contained in a *restricted candidate list* which is defined according to a *greedy function*. In contrast to the greedy construction heuristics discussed in Section 2.6, not necessarily the best component is added. Instead, in each solution construction step one of the components of the restricted candidate list is chosen at random according to a uniform distribution. The algorithm is called adaptive because the greedy function value for each component is updated reflecting the changes due to the previously added component. The constructed solutions are not guaranteed to be locally

```
procedure GRASP
    Initialize parameters
    while (termination condition not met) do
        s = ConstructGreedyRandomizedSolution()
        s' = LocalSearch(s)
        if f(s') < f(s_best)
            s_best = s'
    end
    return s_best
end GRASP
```

Figure 2.10: A generic algorithmic skeleton for GRASP.

optimal with respect to some simple neighborhood definition. Hence, in the second phase local search is applied to improve solutions.

The goal of using a randomized construction heuristic is to generate a large number of different, reasonably good starting solutions for the local search algorithm. Randomization is used to avoid the disadvantage of deterministic construction heuristics which can only generate a very limited number of solutions. Relatively good solutions are generated in GRASP due to the use of the greedy heuristic in the choice of the candidate set of solution components. Another important aspect is that by using a greedy construction heuristic the subsequently applied local search generally needs much fewer iterations to reach a local optimum compared to local search starting from randomly generated solutions. Hence, the descent local search terminates much faster and in the same computation time more often a local search can be applied.

In Figure 2.10 we outline an algorithmic skeleton for GRASP. Note that for GRASP the notion *adaptive* is used in another sense than in the case of ant colony optimization or evolutionary algorithms. In GRASP adaptive concerns the update of the greedy function of the components depending on the previously chosen component. No kind of adaption is typically used between the single iterations of the algorithm.

2.8 Synopsis: Characteristics of Metaheuristics

A metaheuristic will be successful on a given optimization problem if it can provide a balance between the exploitation of the accumulated search experience and the exploration of the search space to identify regions with high quality solutions in a problem specific, near optimal way. The main difference between the existing metaheuristics concerns the particular way in which they try to achieve this balance. The different metaheuristic approaches can be characterized by different aspects concerning the search path they follow or how memory is exploited. In this section, we discuss these aspects according to some general criteria which may be used to classify the presented algorithms. For a more formal classification of local search algorithms based on an abstract algorithmic skeleton we refer to [252].

Trajectory methods vs. discontinuous methods An important distinction between different metaheuristics is whether they follow one single search trajectory corresponding to a closed walk on the neighborhood graph or whether larger jumps in the neighborhood graph are allowed. Of the presented metaheuristics, simulated annealing and tabu search are typical examples of trajectory methods. These methods usually allow moves to worse solutions to be able to escape from local minima. Also local search algorithms which perform more complex transitions which are composed of simpler moves may be interpreted as trajectory methods. Such algorithms are, for example, variable depth search algorithms like the Lin-Kernighan heuristic for the TSP [156] and algorithms based on ejection chains [94]. In ant colony optimization, iterated local search, genetic algorithms, and GRASP starting points for a subsequent local search are generated. This is done by constructing solutions with ants, modifications to previously visited locally optimal solutions, applications of genetic operators, and randomized greedy construction heuristics, respectively. The generation of starting solutions corresponds to *jumps* in the search space; these algorithms, in general, follow a discontinuous walk with respect to the neighborhood graph used in the local search.

Population-based vs. single-point search Related to the distinction between trajectory methods and discontinuous walk methods is the use of a population of search points or the use of one single search point. In the latter case only one single solution is manipulated at each iteration of the algorithm. Tabu search, simulated annealing, iterated local search, and GRASP are such single-point search methods. On the contrary, in ACO algorithms and genetic algorithms, a population of ants or individuals, respectively, is used. (Note that population-based methods are typically discontinuous.) In ant colony optimization a colony of ants is used to construct solutions guided by the pheromone trails and a heuristic function and in genetic algorithms the population is modified using the genetic operators. Using a population-based algorithm provides a convenient way for the exploration of the search space. Yet, the final performance depends strongly on the way the population is manipulated.

Memory usage vs. memoryless methods Another possible characteristic of metaheuristics is the use of the search experience (memory, in the widest sense) to influence the future search direction. Memory is *explicitly* used in tabu search. Short term memory is used to forbid revisiting recently found solutions and to avoid cycling, while long term memory is used for diversification and intensification features. In ant colony optimization an indirect kind of *adaptive* memory of previously visited solutions is kept via the pheromone trail matrix which is used to influence the construction of new solutions. Also, the population of the genetic algorithm could be interpreted as a kind of memory of the recent search experience. Recently, the term *adaptive memory programming* [245] has been coined to refer to algorithms that use some kind of memory and to identify common features among them. Also iterated local search, in a widest sense could be classified as an adaptive memory programming algorithm, although only very poor use of the recent search experience is made in ILS algorithms like choosing the best solution found so far for the modification step. On the contrary, simulated annealing and GRASP do not use memory functions to influence the future search direction and therefore are memoryless algorithms.

One vs. various neighborhood structures Most local search algorithms are based on one single neighborhood structure which defines the type of allowed moves. This is the case for simulated annealing and tabu search. Iterated local search algorithms typically use at least two different neighborhood structures $\mathcal{N}$ and $\mathcal{N}'$. The local search starts with neighborhood $\mathcal{N}$ until a local optimum is reached and in such a situation a kick-move is applied to catapult the search to another point. In fact, the kick-moves can be interpreted as moves in a secondary neighborhood $\mathcal{N}'$. For the subsequent local search again the primary neighborhood $\mathcal{N}$ is used. Often, an appropriate strength of the kick-move is not known or may depend on the search space region. Therefore, it may be advantageous to choose several neighborhoods $\mathcal{N}_1, \ldots, \mathcal{N}_k$ of different size for the kick-moves. Simple variable neighborhood search (VNS) introduces this idea by systematically changing the neighborhood [180]. The mutation operator in genetic algorithms has the same effect as the kick-move in ILS and therefore may also be interpreted as a change in the neighborhood during the local search. Applications of the crossover operator have been interpreted as moves in hyper-neighborhoods [252], in which a cluster of solutions – in genetic algorithms these clusters are of size two – is used to generate new solutions. On the other side, the solution construction process in ant colony optimization and GRASP is not based on a specific neighborhood structure. Nevertheless, one could interpret the construction process used in ACO and GRASP as a kind of local search, but this interpretation does not reflect the basic algorithmic idea of these approaches.

Dynamic vs. static objective function Some algorithms modify the evaluation of the single search states during the run of the algorithm. One particular example, which has not been discussed before, is the breakout method [182] proposed for the solution of satisfiability and graph coloring problems. The basic idea is to introduce penalties for the inclusion of certain solution attributes which modify the objective function. Based on the breakout method, *guided local search* [254] has been proposed and applied to combinatorial optimization problems like the TSP. Also tabu search may be interpreted as using a dynamic objective function, as some points in the search space are forbidden, corresponding to infinitely high objective function values. Yet, all the other algorithms introduced so far use a static objective function.

Nature-inspired vs. non-nature inspiration A minor point for the classification of metaheuristics is to take into account their original source of inspiration. Many methods are actually inspired by naturally occurring phenomena. The algorithmic approaches try to take advantage of these phenomena for the efficient solution of combinatorial optimization problems. Among the presented methods, ant colony optimization, simulated annealing, and genetic algorithms belong to these nature-inspired algorithms. The others, tabu search, iterated local search, and GRASP have been inspired more by considerations on the efficient solution of combinatorial problems.

In Table 2.1 we summarize the discussion of the various methods according to these criteria. We do not claim that *all* implementations of these algorithms correspond to this classification, but it rather gives an indication of the particular characteristics of these methods in their "standard" use.

Table 2.1: Summary of the characteristics discussed in this section. $\surd$ means that the feature is present, $\exists$ that this feature is partially present and $\neg$ that the feature does not appear.

Feature	SA	TS	GA	ACO	ILS	GRASP
Trajectory	$\surd$	$\surd$	$\neg$	$\neg$	$\neg$	$\neg$
Population	$\neg$	$\neg$	$\surd$	$\surd$	$\neg$	$\neg$
Memory	$\neg$	$\surd$	$\exists$	$\surd$	$\exists$	$\neg$
Multiple neighborhoods	$\neg$	$\neg$	$\exists$	$\neg$	$\surd$	$\neg$
Dynamic $f(x)$	$\neg$	$\exists$	$\neg$	$\neg$	$\neg$	$\neg$
Nature-inspired	$\surd$	$\neg$	$\surd$	$\surd$	$\neg$	$\neg$

2.9 Search Space Analysis

It is widely agreed that the performance of metaheuristics depends strongly on the characteristics of the underlying search space. Therefore, several recent researches addressed the analysis of search spaces to either explain the performance of metaheuristics or to devise specific algorithmic variants to better exploit certain known search space characteristics of combinatorial optimization problems [7, 27, 78]. Central to the search space analysis of combinatorial optimization problems is the notion of *fitness landscape* [229, 259]. (In the context of simulated annealing the term energy landscape is often used.) Intuitively, the fitness landscape can be imagined as a mountainous region with hills, craters, and valleys. The local search algorithm can be pictured as a wanderer that performs a biased walk in this landscape. Its goal is to find the lowest point (in the case of minimization problems) 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 minima in the search space, and the overall number of the local minima. 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 $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$.

The fitness landscape determines the shape of the search space as encountered by a local search algorithm. The neighborhood structure induces a distance measure between solutions: let the distance $d(s, s')$ between solutions s and s' be the minimum number of moves that have to be performed to transform one solution into the other.

An important criterion for the effectiveness of local search algorithms is the *ruggedness* of the fitness landscape. Consider first trajectory based local search algorithms like simulated annealing: if the average cost difference between neighboring solutions is on average small, the landscape will be well suited for a local search algorithm; if the average cost difference between neighboring solutions is high, the landscape is rather rugged and may contain many local minima and be badly suited for a local search algorithm [7].

To measure the ruggedness of fitness landscapes the autocorrelation function has been proposed [259]. It is given by

$$\rho(d) = \frac{\mathbf{E}[(f(s) - f(s'))^2]_{d(s,s')=d}}{\mathbf{E}[f^2] - (\mathbf{E}[f])^2} \tag{2.4}$$

$\mathbf{E}[X]$ denotes the expectation of a random variable (note that here the objective function value of a particular solution is considered to be a random variable) and $\rho(d)$ is the correlation coefficient between two solutions that are at distance d. Hence, $\rho(1)$ gives the average correlation between two neighboring solutions. If this correlation is very high, the average cost difference between two neighboring solutions is relatively small. Thus, the landscape is only little rugged and a local search algorithm should show good performance on such a problem. Yet, if the correlation is low, the solution quality of neighboring solutions may differ very strongly. Note that in such a case the solution quality of the current solution gives only a very limited indication of the quality of neighboring solutions (in fact, if the correlation is zero, the current solution gives no information on the quality of neighboring solutions). The correlation can be calculated exactly for some problems if the probability distribution of $f(s)$ can be derived from the underlying instance generating process [6]. Yet, for a given particular instance often the correlation coefficient has to be estimated. To do so, in [259] it is proposed to perform a random walk over the fitness landscape, to interpret the trajectory as a time series and to calculate the empirical autocorrelation function $\hat{\rho}(d)$.

To summarize the information of the autocorrelation function, several measures have been proposed. One of these is the *correlation length* l [229] which is

$$l = -\frac{1}{\ln \hat{\rho}(1)} \tag{2.5}$$

The correlation length gives information on how far from the current solution – on average – one has to move such that there is not any more a significant correlation between the cost values of solutions. Clearly, smoother landscapes will have a longer correlation length. A similar measure is the *autocorrelation coefficient* [7] defined as $\xi = 1/(1 - \rho(1))$. In [7] it has been shown that ξ gives a good indication of the hardness of instances of various combinatorial optimization problems for trajectory based local search algorithms like simulated annealing. This proposed classification appears to be in considerable concordance with computational experiences reported on these problems by other researchers.

Yet, some metaheuristics like ACO and ILS follow discontinuous trajectories, embedding a local search method into a global guiding mechanism which provides new starting solutions. Regarding the performance of the underlying local search algorithm, the previously introduced measures give an indication on how hard a problem is. Yet, these measures are not enough to give a detailed indication of the performance of discontinuous walk methods. For these, global characteristics of the fitness landscape topology are also important. One such characteristic is the average distance between locally optimal solutions. On some problems, it has been shown that the average distance between locally optimal solutions is relatively small, compared to the size of the search space. If a problem (an example is the TSP) has such a characteristic, it may be easier for the local search because the interesting part is contained in a relatively small part of the total search space.

For the investigation of the suitability of a fitness landscape for adaptive multi-start algorithms the analysis of the correlation between solution costs and the distance between solutions has been proven to be a useful tool [27, 132, 135, 186]. The idea is to measure the correlation

between the cost and the distance of solutions of a combinatorial optimization problem. In particular two different "versions" of this method have been used in the literature.

(1) Study the correlation between cost and the average distance to the other local minima [186, 27].

(2) Study the correlation between cost and the distance to the best local minimum or the known global optimum [25, 27, 132, 135].

In the same spirit the fitness-distance correlation (FDC) has been proposed in [132]. It was first used to analyze the hardness of a problem for a genetic algorithm, but it is also very useful to give hints on the effectiveness of adaptive algorithms using discontinuous trajectories. It measures the correlation of distances and cost to the nearest global optimum. The fitness-distance correlation is defined as:

$$\rho(f,d) = \frac{\mathbf{E}[fd] - \mathbf{E}[d]\mathbf{E}[f]}{\sqrt{\mathbf{E}[f^2] - (\mathbf{E}[f])^2} \cdot \sqrt{\mathbf{E}[d^2] - (\mathbf{E}[d])^2}} \tag{2.6}$$

Why is the FDC important for the interpretation of search performance of discontinuous metaheuristics? Clearly, the notion of *search space region* is tightly coupled to the notion of distance between solutions. The task of adaptive restart algorithms like iterated local search, genetic algorithms, and ant colony optimization is to guide the search towards regions of the search space containing high quality solutions and, possibly, the global optimum. Recall that the most important guiding mechanism of these methods is the objective function value of solutions. This guidance mechanism relies on the intuition that the better a solution the more likely it is to find even better solutions close to it. In particular, the fitness-distance correlation describes the relation between the fitness (cost) of solutions and their distance to very good solutions or the global optimum. Consider minimization problems: a high, positive correlation between the solution cost and the distance to the global optimum indicates that the better the solutions, the closer the algorithm gets, on average, to the global optimum. If no such correlation exists or it is very weak, the fitness gives only little guidance towards better solutions. Even worse, if cost and distance are negatively correlated, guiding the search by objective function values may be even misleading because, on average, the better the solution quality, the farther away the algorithm gets from the global optimum or the best solutions. This last phenomenon has been observed, for example, in deceptive problems for genetic algorithms [132] or in heuristic search [143].

One of the most widely studied problems with respect to search space analysis is the symmetric TSP. Several studies of the search space characteristics have been made for the TSP. In these studies, the distance measure used is the number of different arcs between two tours. A first configuration space analysis for the symmetric TSP with a randomly generated distance matrix has been done in [141] finding that good tours tend to have more arcs in common. In [186] plots of the cost versus the average distance between local minima of a Euclidean TSP show a strong correlation. This study is extended in [185] were additionally it is shown that better local optima tend to have more arcs in common and the conjecture is made that the number of different arcs occurring in `2-opt` local minima of Euclidean TSP instances is bound by approximately $4.3 \cdot n$. This result implies that the local minima, in general, are contained in a rather small part of the search space. These earlier studies are extended in [27] by plotting

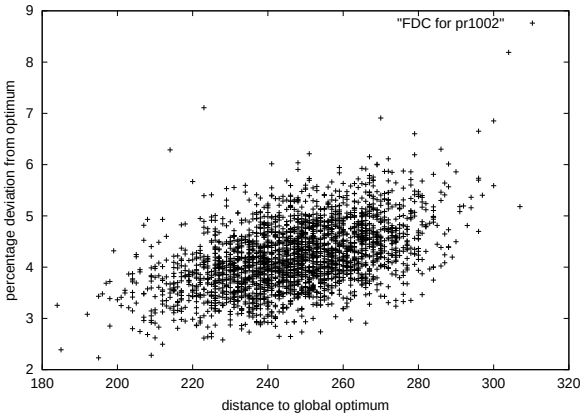

Figure 2.11: Fitness distance plot for TSP instance `pr1002` with 1002 cities. Each point gives the distance from the global optimum (x-axis) and the solution quality as the percentage deviation from the optimum (y-axis) of a `3-opt` tours. The plot is based on 2500 `3-opt` tours.

and calculating fitness-distance correlations for Euclidean TSPs and graph bi-partitioning. They consider two kinds of plots. In one graphic they plot the correlation between cost and average distance from other local minima, in another the cost versus distance from the best found local optimum. They show that for both problems a strong positive correlation between cost and distance exists. This implies that the better the solutions are, the closer they are to the best solution (or to the global optimum if it is known).

We illustrate some of the findings made in earlier research regarding the fitness distance analysis on the TSP in Figure 2.11. The plot is based on 2500 local minima found by a `3-opt` local search algorithm for TSPLIB instance `pr1002`. We plotted the cost versus the distance from the optimal solution. The plot shows a strong correlation between the solution quality and the distance to the global optimum which can be seen by the fact that better local optima tend to be closer to optimal solution. The correlation coefficient in this case is $\rho(f, d) = 0.55$.

2.10 Summary

In this chapter we have introduced some aspects of combinatorial optimization problems. Due to their practical importance, many efforts have been made to efficiently find optimal solutions. Yet, due to the $\mathcal{NP}$-hardness of most of these problems, such approaches quickly become infeasible and one rather has to be satisfied with finding very high quality solutions, which are not necessarily optimal, in reasonable run-time. Recent research has shown that the best performing algorithms for achieving this goal are based on local search. Many of these algorithms are algorithmic frameworks which, in principle, are applicable to a wide variety of optimization problems with only minor problem specific modifications. These algorithmic frameworks are often called metaheuristics. We have presented the main features of some of the most widely applied metaheuristics and discussed their application principles. We characterized the common features and the differences between these metaheuristics and have introduced search space features which may influence the performance of these algorithms.

Chapter 3

Evaluating Local Search Algorithms by Run-time Distributions

A priori it is often not obvious how specific metaheuristics perform when applied to specific combinatorial problems. Hence, they have to be evaluated and compared empirically. In this chapter we first discuss some general issues concerning the evaluation of metaheuristics. In the second section we present a novel type of empirical evaluation methodology. It is based on measuring and characterizing run-time distributions arising from the application of local search algorithms to optimization and decision problems. This methodology has been developed in joint work with Holger Hoos on satisfaction problems [120, 122, 119], here we present an extension to optimization problems. We use this methodology mainly in Chapter 5 to analyze and improve iterated local search algorithms and in Chapter 6 to analyze the performance of local search algorithms for constraint satisfaction problems.

3.1 Evaluation of Metaheuristics

Analytical evaluation

With analytical methods some intriguing results on the worst-case or the average-case complexity of algorithms have been obtained [9, 197]. Results on the approximability of problems have led to bounds on the worst-case performance of approximate algorithms. Yet, results on worst case performance rely on certain pathological instances which may not occur in practical applications and average-case results suppose a probability distribution over randomly generated problems, but these randomly generated problems may not correspond to those found in real-life instances. Additionally, results on the worst or average-case behavior often are based on rather simplistic algorithms. For the metaheuristics discussed here, often the best that can be done is just to apply the results obtained for the rather simplistic algorithms, if these simpler algorithms are subsumed by the metaheuristic (this is the case if the simpler algorithms construct the initial solution used by the metaheuristic). But, metaheuristics are often rather complex and combine several methods in one algorithm. These algorithms typically show much better performance than would be suggested by analytical methods. The analytical analysis is not sufficient to give a significant discrimination between the performance of the different metaheuristic algorithms and to give hints on which metaheuristic should be applied to a particular problem [116].

For some metaheuristics it can be proven that the algorithm converges asymptotically towards the optimal solution. This type of results has been obtained, for example, for simulated annealing [87, 106, 161, 215], probabilistic tabu search [71], and genetic algorithms [110]. Yet, for practical purposes these theoretical results are not sufficient. In particular, they give no hint on which metaheuristic should be chosen for a specific application. Furthermore, metaheuristics for which no formal convergence proofs are available, often show better empirical performance on many problems. This is the case with tabu search and iterated local search when compared, for example, with simulated annealing [19, 129, 233].

Empirical Evaluation

Because of the difficulties of a theoretical analysis, one has to resort to empirically evaluate the performance of metaheuristics. The empirical evaluation of metaheuristics performance should take into account three main aspects [14].

- *Solution quality* of the metaheuristic produced. Here the primary concern is how close the solutions are to the optimal solution.
- *Computational effort*, measured as the speed of computation.
- *Robustness* of the algorithm, understood as the ability of the heuristic to perform well over a wide range of test problems.

The first two measures are often also referred to as the *effectiveness* and the *efficiency* of a heuristic algorithm.

With respect to the solution quality it is important to report at least the average solution quality produced by the applied metaheuristics, because for increasing problem size this seems to be the most important discriminating factor concerning the solution quality produced by approximate algorithms [219]. Additional information can be important; examples are the standard deviation of the solution quality obtained and the best and the worst solution found in a number of runs.

The efficiency of the algorithms is often measured in CPU-time. When attacking different combinatorial optimization problems by ant colony optimization (ACO) or iterated local search (ILS) in this thesis, we always use the same implementation with respect to data structures and the local search algorithm embedded in these two metaheuristics. In fact, the ILS algorithm (see Section 2.7.2 and Chapter 5) is in all cases directly derived from the corresponding implementation of the ACO algorithm (see Section 2.7.1 and Chapter 4). In this case, the comparison based on CPU-time is fair. To abstract from the particular implementation, we give additionally the number of algorithm iterations to obtain the presented results. Instead of describing the algorithm performance based on the explicit run-time, often it may be preferable to describe the computational effort by operation counts [4]. Using operation counts with an appropriate cost model has the advantage of abstracting from a particular implementation and the programming skills of the programmer.

Concerning the robustness of the metaheuristics, we indicate, where possible, the influence of the problem characteristics on the performance of the applied algorithms. When using hybrid methods consisting of a solution construction mechanism and subsequent local search to

improve solutions, differences in performance may be due to just one or both of the two parts. Yet, in general, to identify the ultimate reasons for the behavior of an algorithm on given set of instances is a rather difficult task.

Another issue for the meaningful comparison of algorithms is the selection of the test problems. Test problems should be challenging and widely available to be able to compare the results to other algorithms. In this way a better view of the ultimate value and the limitations of the heuristic approach can be obtained [127]. The most straightforward possibility is to test the algorithms on benchmark instances which are well known in the literature and which have been used in earlier studies. An advantage of using standard benchmark problems is that one can compare an algorithm's performance to that of existing techniques and in this way assess the merits of the approach. Additionally, for some benchmark problems the optimal solutions may be known or at least the best found solutions are available from literature. Thus, one can assess the quality of the solutions found by the heuristic algorithm or compare it to published results.

In this thesis we will test the ACO algorithms and the ILS algorithms on three well-known combinatorial optimization problems. For all three problems benchmark libraries containing a wide variety of instances are available. The probably most widely used test problem is the traveling salesman problem (TSP), a problem that has inspired a substantial amount of research [151, 211]. For the TSP, the instances are all taken from TSPLIB [210], a benchmark library for the TSP. The other two problems are the quadratic assignment problem (QAP) using instances contained in QAPLIB [39] (available via `http://serv1.imm.dtu.dk/~sk/qaplib/`) and the flow shop problem (FSP) using instances from ORLIB (available via `http://mscmga.ms.ic.ac.uk/`). Whereas large TSP instances of several hundreds of cities can now regularly be solved to optimality, the QAP is considered as one of the hardest optimization problems and general instances of size $n \geq 25$ cannot be solved exactly today. Thus, one has to use heuristic algorithms to find good approximate solutions in reasonable time. The FSP is a prototypical scheduling problem and has received significant attention in the literature [69].

In AI literature, algorithms for constraint satisfaction problems (CSPs) are often tested on binary CSPs. Yet, only few benchmark instances are available. Therefore, most local search algorithms for CSPs are tested on randomly generated binary CSP instances. These instances are specified by four parameters $\langle n, k, p, q \rangle$, where n is the number of variables, k is the uniform domain size, p is the probability that a constraint between two variables exists, and q is the conditional probability that a value pair satisfies a constraint. It has been shown that with specific parameter settings of the instance generator – on average – particularly hard instances for complete as well as incomplete algorithms are generated; for other parameter settings the generated instances are rather easily solved or easily shown to be unsolvable by complete algorithms [226, 203]. The randomly generated instances show a *phase transition phenomenon* like observed also in physical systems [248, 140]. Other benchmark instances used in the context of algorithms for CSPs typically stem from the graph coloring domain or from frequency assignment problems. In the experimental comparison of local search heuristics for CSPs in Chapter 6 we will use randomly generated CSP instances and graph coloring instances from the DIMACS benchmark suite where some benchmark instances can be found. For the satisfiability problem in propositional logic (SAT) recently a comprehensive benchmark collection, called SATLIB, has been created by Holger Hoos and the present author [121].

3.2 Run-time Distributions for Evaluating Local Search

3.2.1 Run-time Distributions

Given a specific instance of a combinatorial optimization problem, two runs of the same metaheuristic will generally produce a different output, because metaheuristics are *probabilistic algorithms*. This randomness stems from random decisions made by the algorithm like random initial solutions, random tie breaking rules, probabilistic acceptance criteria of solutions, etc. Hence, the performance of these heuristics with respect to computation time and solution quality can be described by a two-dimensional random variable (C, T), where C is the random variable describing the solution cost and T is the random variable describing the run-time. Note that the common distribution of these random variables depends on the particular instance which has to be solved and the algorithm applied. Knowledge of this random variable would provide all the information needed to analyze the algorithm's behavior on that particular instance. Yet, this knowledge is not generally available and the shape of the distribution function of this two-dimensional random variable has to be estimated. In fact, when an algorithm is run on a given instance, one is measuring empirical realizations of this random variable.

Instead of the combined distribution, often one focuses on the marginal distributions of (C, T), which also provide important information on the behavior of an algorithm. One of these two marginal distributions is the *run-time distribution*, which describes the run-time needed by an algorithm to reach a given solution quality c. The second is the *solution quality distribution* which describes the solution quality obtained after executing a given algorithm for time t.

To measure run-time distributions, the optimization problem is addressed as a decision problem and one is measuring the run-time needed to give an answer. That is, we ask the question: how long do we have to run the metaheuristic to reach a solution quality of c? This required solution quality may be specified as some fixed objective function value. Alternatively, if the optimal solution is known, the required solution quality may be specified as the percentage deviation from the optimum which is $q = (f(s) - f(s_{opt}))/f(s_{opt}) \cdot 100\%$. Clearly, the shape of the run-time distribution will depend on the required solution quality. The algorithm dependent run-time to reach a solution quality of c is described by a random variable T_c with associated distribution function $G_c(t)$, whereby the subscript indicates the dependence on the solution cost c. To measure solution quality distributions, the algorithm is given some instance dependent run-time t and the question posed is, what is the distribution of the solution quality obtained after running the algorithm for time t? In this case the solution quality distribution can be described by a random variable C_t with associated distribution function $F_t(c)$. Obviously, the solution quality distribution is a function of the run-time allocated for the algorithm.

The empirical run-time distribution and the solution quality distribution provide valuable information to empirically evaluate, compare, and find improvement possibilities for local search algorithms [120]. To get knowledge of their functional form one can run a given algorithm many times and collect data to empirically estimate the distributions. In each run it suffices to report its solution quality each time a new best solution is found, the computation time needed to obtain it and possibly some other statistic data like the number of iterations done which can be useful in an analysis of the algorithm's performance. Then, a posteriori the empirical distributions can be estimated. Let k be the number of times the algorithm is run, c be the solution quality that should be achieved, $f(s_j)$ be the best solution found so far, and $rt(j)$ be the run-time

of the j-th algorithm run. Then, the estimated run-time distribution to reach a solution quality bound c is computed as

$$\widehat{G}_c(t) = \frac{\mid \{j \mid rt(j) \leq t \ \wedge \ f(s_j) \leq c\} \mid}{k} \tag{3.1}$$

For some problems, optimal solutions or tight lower bounds on the optimal solution value may be known. In such a case it may be preferable, as also indicated before, to require the algorithm to get within a certain percentage of the optimal solution value instead of explicitly fixing some value.

Practically, an algorithm may fail to reach a required solution quality within reasonable computing time, because it is unrealistic to expect that an instance is solved to optimality in every trial. Thus, additionally an upper CPU-time limit has to be imposed. In this case we actually measure truncated run-time distributions. Still, these give valuable information like the probability to reach a specific solution quality bound. If an indication of asymptotic behavior of the algorithm is desired, one should run the algorithm long enough such that further progress in solution quality is rather low.

A special case of run-time distributions occurs for satisfaction problems, where a candidate solution satisfying *all* constraints has to be obtained. Thus, for decision problems one is mainly interested in a very specific run-time distribution — that corresponding to find the optimal solution in the case of optimization problems. Yet, also in case of satisfaction problems solution quality distributions could be of interest. Recall that any local search algorithm — applied to satisfaction or optimization problems — performs a biased walk on the neighborhood graph, guided by the objective function value. In satisfaction problems, the objective function is typically given by the number of violated constraints and implicitly the assumption is made that better objective function values indicate that we are closer to a solution. If an algorithm finds candidate solutions with lower objective function values than others, this may give an indication that this algorithm also is better suited to solve satisfaction problems. The development of the solution quality may also be particularly useful to compare these algorithms on large and very difficult satisfaction problems. Furthermore, local search algorithms for satisfaction problems can be applied also directly to the optimization versions of these problems like MAX-CSP [79, 81, 232, 256] and MAX-SAT [15, 16, 107], for which solution quality distributions are again very meaningful.

3.2.2 Examples of Run-time and Solution Quality Distributions

To give an example for the run-time behavior of a particular metaheuristic we present in Figure 3.1 run-time distributions and solution quality distributions for iterated `3-opt` applied to TSP instance `att532` of TSPLIB. Iterated `3-opt` is a ILS algorithm using an `3-opt` iterative improvement algorithm for the TSP. The details of the algorithm are presented in Chapter 5, they are not important for the following discussion. For this instance the optimal solution is known and we require the algorithm to reach a solution within a give percentage excess over the optimal solution when measuring run-time distributions. The empirical distributions are based on 100 independent runs of the algorithms and we limited the maximal run-time of the algorithm to 1 CPU hour on a 167MHz Sun UltraSparc I processor.

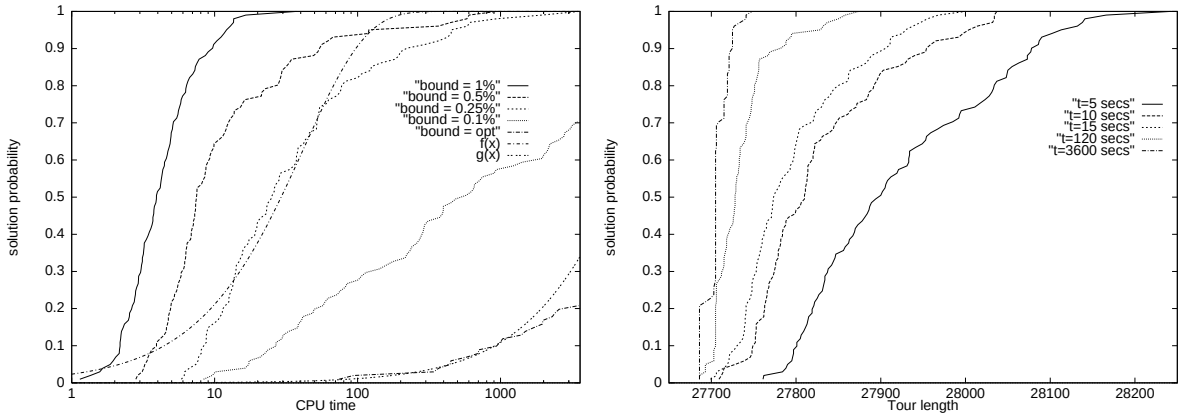

Figure 3.1: Run-time distribution (left) and solution quality distribution (right) for iterated `3-opt` on TSP instance `att532` (see Chapter 5 for a description of the algorithm). For the run-time distribution (right side) the x-axis is the CPU-time and the y-axis represents the cumulative distribution of finding the required solution quality. For the solution quality distribution the$x$-axis is the tour length and the y-axis gives the cumulative distribution of the tour length reached after a given CPU-time bound. The distributions are based on 100 trials of the algorithm.

Several observations can be made from these plots. Consider first the run-time distributions (on the left side of Figure 3.1) obtained by imposing bounds of $q \in \{1\%, 0.5\%, 0.25\%, 0.1\%, 0.0\%\}$. The x-axis represents the CPU-time used and the y-axis gives the empirical probability to find a solution within the required bound. In this figure additionally two functions are plotted, we discuss the reason for their inclusion in the next section. The run-time distributions show that it is rather easy for the algorithm to reach a reasonably good solution within, say, 1% of the optimum. Such a solution is reached in all trials after roughly 30 seconds. If a higher solution quality is desired, the required run-times grow considerably (recall that a log-scale is used on the x-axis). For example, if we require solution quality within 0.5% of optimal, the run-time distribution flattens off considerably. After 15 seconds, 74% of the runs were able to find such a solution, but in one run it took 935 seconds to do so! To obtain even better solutions with a high probability much longer run-times need to be allowed. At the run-time distributions it can also be observed that the algorithm requires some initialization time t_{init} before finding any optimal solution. For example, the ILS algorithm used here finds the first optimal solution after only 74 seconds. Theoretically, an optimal solution may be found just at the start of the algorithm, but the probability of doing so is vanishingly small if one considers the size of the search space. This initialization time clearly depends on the hardness and the size of the instance. Thus, there is an obvious tradeoff between the solution quality to be obtained and the required run-times; for higher quality solutions considerably more run-time has to be invested.

The solution quality distributions on the right side of Figure 3.1 were obtained after executing the algorithm for $t_{max} \in \{5, 10, 30, 120, 3600\}$ seconds. Here the x-axis represents the absolute solution quality and the y-axis the probability to find a solution better than a certain value. A first observation is that with increasing CPU time the solution quality distributions get steeper and more shifted to the left, towards higher quality solutions. The progress in so-

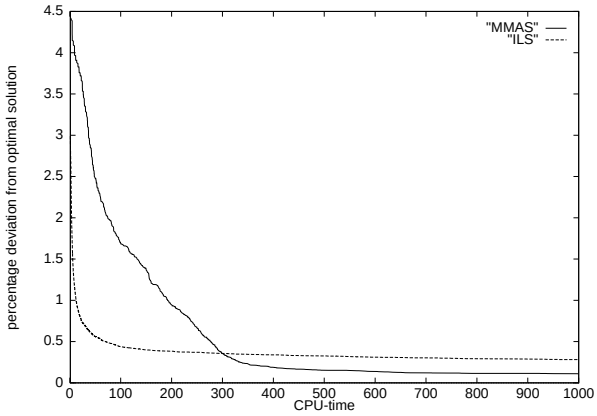

Figure 3.2: Solution quality trade-off with iterated `3-opt` and $\mathcal{MAX}$–$\mathcal{MIN}$ ant system with `3-opt` local search applied to TSP instance `rat783`. On the x-axis is given the CPU-time, on the y-axes is given the percentage deviation from the optimal solution averaged over 100 trials of each algorithm.

lution quality is fast in the beginning and slows down considerably for longer run-times. Yet, as observed before, very high quality solutions are only obtained at the cost of rather long run times. A further interesting observation is that in our example, the distribution at $t_{max} = 3600$ becomes multi-modal which can be seen, in this representation based on distribution functions, at the sharp transition from the best obtained solution, which, in fact, is the optimum, to a solution 0.068% away from the optimum. (The empirical probability to obtain the optimal solution is 0.21 for our algorithm and the probability to reach the other solution of length 27705 is 0.36.) We have observed this phenomenon also on other TSP instances and other problems. Often, such instances appear to have "hard cliffs" for the local search algorithm, corresponding to deep local minima, which are difficult to pass.

Another popular possibility to illustrate the tradeoff between run-time and solution quality is to plot the development of the average solution quality found versus the run-time. An example of such a plot is given in Figure 3.2 for iterated `3-opt` and $\mathcal{MAX}$–$\mathcal{MIN}$ ant system when applied to TSPLIB instance `rat783`. (The details on the $\mathcal{MM}$AS algorithm are presented in Section 4.3.) It can be observed that for short run times, the ILS algorithm produces better quality solutions than $\mathcal{MM}$AS. Yet, the graphs of the solution quality development of both heuristics cross at roughly $t = 300$ seconds and from then on $\mathcal{MM}$AS produces – on average – better solutions. Thus, there is a tradeoff concerning the application of these algorithms. If near optimal solutions should be obtained very fast and only little computation time is available, it is preferable to use the ILS based algorithm while if one is willing to spent some more time, the other algorithm is preferable because it produces higher quality solutions. Additionally note that, although the improvement in solution quality for the ILS algorithm and $\mathcal{MM}$AS slows down with longer run times, the solution quality still improves further, as can better be seen in the run-time distributions for the same algorithms, which are presented in the next section.

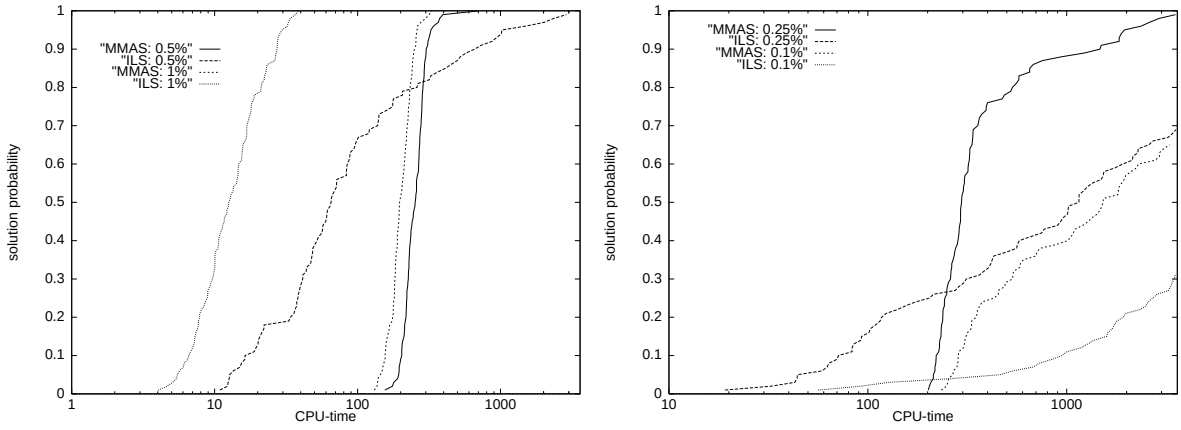

Figure 3.3: We compare $\mathcal{MM}$AS with `3-opt` and iterated `3-opt` applied to TSP instance `rat783` using run-time distributions (based on the same data used in Figure 3.2). On the left side we plotted the run-time distributions to reach a solution within 1% and within 0.5%, on the right side those to reach a solution within 0.25% and within 0.1% of the optimal solution search. Run-time distributions are sampled over 100 independent trials (see the text for more details).

3.2.3 On the Usefulness of Run-time Distributions

Descriptive Advantages of Run-time distributions

The empirical evaluation of run-time distributions may be useful to describe an algorithms behavior. First, run-time distribution may simply serve to visualize the behavior of an algorithm, as illustrated in Figure 3.1 on the left side. Such a visual information may often be more helpful than the mere presentation of statistical data. Standard statistical data like the average solution quality and the standard deviation of the average can also be obtained from the same data used for the run-time distributions.

Second, run-time distributions can also be very useful for the comparison of algorithms. In Figure 3.3 we give the run-time distributions for various bounds on the solution quality for ILS and $\mathcal{MM}$AS already compared in Figure 3.2 on the basis of the run-time versus solution quality tradeoff. On the left side the run-time distributions to reach a solution within 1% and within 0.5% of the optimal are plotted, on the right side those to reach a solution not worse than 0.25% and 0.1%, respectively, above the optimal solution value. Both algorithms reach a solution quality of 1% or 0.5% in all trials, but the ILS algorithm reaches such a solution much faster than $\mathcal{MM}$AS. If one is satisfied with such a solution, the ILS algorithm is preferable over the $\mathcal{MM}$AS. Yet, if higher solution quality is required, the run-time distributions cross over (see those for 0.5% and higher solution quality on the right side of Figure 3.3). The required solution quality is reached faster by ILS with a small probability (in this case the ILS run-time distribution is above the one of $\mathcal{MM}$AS before they cross), while $\mathcal{MM}$AS seems to be the more reliable algorithm if longer run-times are allowed, because from the crossing point on the probability to reach a high quality solution is significantly larger than with the ILS algorithm. Actually, if we require the algorithms to find the optimal solution, the run-time distribution of $\mathcal{MM}$AS lies completely above that of the ILS algorithm (not shown here).

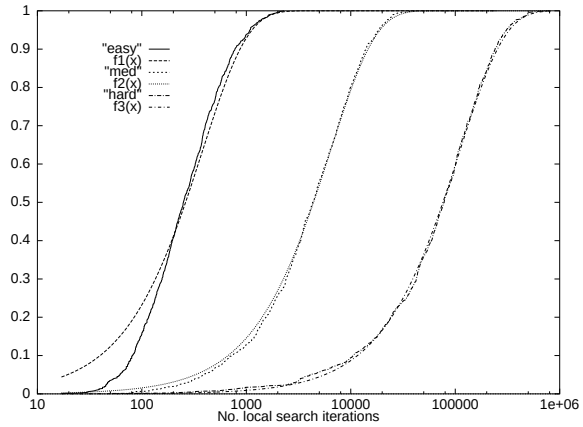

Figure 3.4: Run length distributions for WMCH with approximately optimal noise parameter setting measured on the easiest, the medium and the hardest instance of 250 randomly generated binary CSP instances. $f1$ to $f3$ represent approximations of the empirically measured distributions via exponential distributions. See the text for more details.

As can be seen from this example, algorithms can be easily compared using run-time distributions. At the run-time distributions it can additionally be observed that the probability of finding the required solution for the two algorithms increases with additional CPU-time.

Third, Another advantage of run-time distributions is that they may be approximated with probability distributions known from statistical theory and this approximation may be used to characterize an algorithms performance. The goodness of fit of the approximation can then be tested by statistical tests like the χ^2 test [213]. An example is given in Figure 3.4, where we plotted the run-*length* distribution of `wmch`, a local search algorithm for the solution of CSPs, measured on three randomly generated instances. These three instances are generated according to the model presented in Section 3.1 using the parameters $\langle n = 20, k = 10, p = 0.5, q = 0.62\rangle$. With these parameter settings the instances are drawn from the phase transition region of binary CSPs and are therefore – on average – the hardest to solve for exact and approximate algorithms [114, 226, 46]. But still, the variability of the hardness among the instances differs as much as several orders of magnitude concerning the average search effort for finding a solution. The three instances are taken from a set of 250 randomly generated instances and correspond to the easiest, the median and the hardest one with respect to average number of steps required to find a solution. We refer to these instances as *easy*, *med*, and *hard*.

In this example, the computational effort is measured as the number of local search steps, which is an appropriate operation count in this case. The run-length distribution, despite being based on a discrete measure, is approximated by a continuous probability distribution. The main reason is that continuous distributions are easier to handle mathematically. For all three instances the run length distribution is well approximated by an exponential distribution, which is a particular statistical distribution. Only for the *easy* and *med* instance, the empirical distributions show a small deviation for short runs. This deviation in the lower part of the curve are also the reason that for the easiest instance the χ^2-test rejected the approximation at the $\alpha = 0.05$ level, whereas for the two harder instances the approximations passed the test.

Observing run-time distributions and approximating them by statistical distributions need not be restricted to single instances. In fact, hypothesis on the run-time behavior of algorithms on a specific class of instances can be formulated and tested empirically. Based on such a methodology an experimental approach of testing local search algorithms along the lines proposed by Hooker [117] can be undertaken. Hooker [116, 117] criticizes that the empirical analysis of algorithms usually remains at the stage of simply collecting data and argues that, analogous to empirical methodology used in other sciences, one should furthermore attempt to formulate hypotheses based on this data which, in turn, can be experimentally refuted or validated. Based on our methodology we can meet this demand. For example, one such hypothesis is that for optimal parameter settings for `wmch`, the run-length distribution for the solution of a single, hard instance from the crossover region of random, binary CSPs can be well described by exponential distributions. This hypothesis can be tested by running a series of experiments on a large set of such instances. Doing so, our empirical evidence confirms this conjecture (see Chapter 6 for a more detailed analysis). Similar results have been established for other local search algorithms applied to Random-3-SAT, CSP encoded graph coloring, and SAT encoded problems from other domains like AI planning [120, 122].

Improving Algorithms by Exploiting Run-time Behavior

Run-time distributions also can give valuable hints on situations in which algorithms can be improved. In particular, algorithms may show some stagnation behavior like getting trapped in certain regions of the search space without finding improved solutions. Such a behavior may be noted by the fact that high quality solutions are found in some runs very fast, while otherwise the algorithm may need a long time to find such solutions. Thus, for higher run-times the rate by which the probability of finding a solution of satisfying quality increases may slow down considerably. To motivate our main point we start with an example.

Recall the example run-time distribution given in Figure 3.1 (page 46). Suppose one is satisfied with a solution 0.5% above the optimal solution value and such a solution should be obtained as fast as possible with the available algorithm. In the worst case more than 900 seconds were needed, but with a probability of $\widehat{G_{0.5}}(15) = 0.74$ such a solution empirically could already be obtained after only 15 seconds. Thus, naturally the question arises whether restarting the algorithm may lead, in the long run, to an increase of the probability of finding a solution. Suppose, for example, we have 30 seconds of computation time available and the algorithm is restarted after 15 seconds. Then, we can execute two runs of the algorithm, and the empirical probability of finding the required solution quality can be calculated as 0.93. If instead one long run of 30 seconds is executed, we have $\widehat{G_{0.5}}(30) = 0.84 < 0.93$. If the algorithm is restarted every 15 seconds, after 900 seconds one would expect to find a solution with the required quality with a probability of $1 - 7.91 \cdot 10^{-36} \approx 1$. Hence, restarting the algorithm results in a significantly increased probability of finding the required solution.

To illustrate the significantly improved worst case behavior with respect to the time needed to find the required solution quality, we ran the algorithm with a fixed cutoff of $t_{max} = 15$ seconds. If we did not reach the specified solution quality, the algorithm is restarted from a new initial solution. The run-time distribution of the restart algorithm is given in Figure 3.5 together with the version of the algorithm without restarts. Additionally, an exponential distribution is plotted, its significance is discussed below.

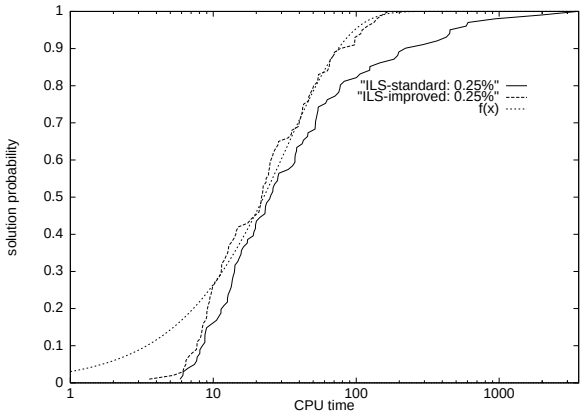

Figure 3.5: Improved ILS algorithm by restarts using 3-opt local search on TSP instance `att532`. The cutoff value is chosen in such a way that a solution quality within 0.25% of the optimal solution is obtained fast. $f(x)$ is an exponential distribution. See the text for more details.

The following discussion is based on some properties of the exponential distribution, which has the distribution function $G(t) = 1 - e^{-\lambda t}$, where $1/\lambda$ is the expectation value of the distribution. Additionally, we need to introduce two more terms. The *failure probability* is given by $\overline{G_c}(t) = 1 - G_c(t)$, that is, the probability not to find the required solution quality within computation time t. The *success rate* is defined as $r(t) = g_c(t)/\overline{G_c}(t)$, where $g_c(t)$ is the density function of T_c.[1] Intuitively, one can interpret $r(t) \cdot \Delta t$ as the probability that a solution is found in the next Δt time units, given that one has not found a solution until time t. For an exponentially distributed run-time the following theorem holds [213].

Theorem 3.2.1 *Let T_c be the random variable with distribution function $G_c(t)$ describing the algorithm's run time to reach a solution quality c. If T_c is exponentially distributed we have $\overline{G}_c(t) = (\overline{G_c}(t/p))^p$.*

The theorem says that for an exponentially distributed run-time, the probability not to find the required solution quality is the same whether we execute one long run of the algorithm with run-time t or whether we execute p short runs of length t/p. The exponential distribution is the only statistical distribution with a constant success rate, which is $r(t) = \lambda$. We will discuss in the following important conclusions for the parameterization and the design of metaheuristics.

Clearly, whether restart may improve the algorithm's performance strongly depends on the steepness of the run-time distribution. If the empirical run-time distribution is in fact an exponential distribution, restarting the algorithm has basically no effect on its performance (see Theorem 3.2.1). In other words, the success rate of the algorithm is constant. If the run-time distribution is steeper than an exponential distribution, the probability of finding the required solution quality increases relatively stronger when running the algorithm for a longer time; the

[1] In reliability theory , $r(t)$ is commonly used to characterize the aging of components [13]. In that context $r(t)$ is called *failure rate*, since it describes the tendency of a component to *fail*. Yet, in our context $r(t)$ describes the tendency of an algorithm to find a solution.

success rate of the algorithm increases. In such a case, restart of the algorithm would be detrimental and would reduce the obtainable success probability by the algorithm. If the empirical run-time distribution is less steep than the exponential distribution, on the long run the algorithm will profit from occasional restarts – its empirical solution probability increases stronger by using restart than by continuing to run it for a longer time.

One specific importance of the exponential distribution is to determine appropriate cutoff values for the restart of an algorithm. If the run-time distribution is known or has been determined empirically, appropriate cutoff values can be determined a posteriori. In the following we are interested in maximizing the success probability in the long run. An optimal cutoff value is obtained by first finding the maximal value λ^* for which the exponential distribution and the empirical run-time distribution have at least one common point,[2] more formally

$$\lambda^* = \sup\{\lambda \mid \exists t > 0 : (1 - exp\{-\lambda t\}) - \widehat{G_c}(t) = 0\} \tag{3.2}$$

If the condition for λ^* is satisfied for more than one single point, an optimal cutoff time t_{opt} can be determined as

$$t_{opt} = \inf\{t \mid (1 - exp\{-\lambda^* t\}) - \widehat{G_c}(t) = 0\} \tag{3.3}$$

An important special case occurs, if $\lambda^* = \infty$. This, intuitively, means that the exponential distribution should be arbitrarily steep, since the larger λ the steeper is the exponential distribution.

Consider again the plots of the run-time distributions in Figure 3.1. The two additionally plotted curves correspond to exponential distributions which loosely approximate the empirical run-time distributions in the central part. The exponential distributions indicate that the performance of the studied algorithms can be increased by restarts. This has been shown in Figure 3.5, where again an exponential distribution is plotted. This exponential distribution, as expected, fits well the upper part of the algorithm with restarts. Note, that the deviation of the empirical distribution from the exponential distribution in the lower part is a systematic deviation due to the initialization time of the algorithm. This initialization time could be modeled by distributions which are shifted by a fixed offset Δt to the right. In case of a shifted exponential distribution, the distribution function would be given as $G(t) = \max\{0, 1 - e^{-\lambda(t-\Delta t)}\}$. This initialization time has direct consequences on the usefulness of restarts. In case the algorithm is restarted too early, that is, $t_{max} < t_{init}$, the solution quality achieved will be very low.

Interestingly, the leftmost run-time distribution in Figure 3.1, corresponding to a bound of 1%, is of such a shape that the optimal cutoff value would be $\lambda^* = \infty$; the performance of the algorithm cannot be improved by restarts. The main reason is that the required solution quality can easily be obtained by iterated `3-opt`. This example also shows that the usefulness of restart depends crucially on the required solution quality. Often it is the case that for a low solution quality the best is never to use restart, while, as we will see, for a high solution quality restart often improves performance. Hence, the behavior of the algorithms differs strongly based on the required solution quality. It should be noted that in this thesis we will be mainly interested in run-time distributions corresponding to high quality solutions or even the optimal solution.

[2] The following formulas assume that the empirical run-time distribution does not solve the instance in all trials, that is, the run-time distribution does not reach one. The case were the run-time distribution reaches an empirical probability of one can be treated similarly.

The steepness of the run-time distribution also has important consequences for parallel processing by *multiple independent* runs of an algorithm. This is the most straightforward and most simple method of parallelizing randomized algorithms. Recall, that the speed-up in case of parallel processing is defined as $s = \textit{sequential time}/\textit{parallel time}$. As stated in Theorem 3.2.1, for exponential run-time distributions using p processors with time t/p each we get the same success probability as for the sequential algorithm with time t. Thus an optimal speedup is obtained, since $t/(t/p) = p$. If the run-time distributions are steeper than an exponential distribution, some efficiency is lost with parallel processing, that is, the resulting speed-up will be suboptimal. On the other hand, if the run-time distribution is less steep than an exponential distribution, parallel processing would yield even a super-optimal speed-up when compared to the sequential version of the algorithm without restarts.

Usually, run-time distributions are not known a priori when trying to solve previously unseen instances. Yet, one can measure run-time distributions on some single instances. Then, one can empirically formulate and test hypotheses about the run-time behavior on the problem classes of interest. If the same type of run-time distributions occurs for all the single instances, one may assume that this behavior is representative for the entire problem class. We will give examples of such hypotheses in Chapter 5, applying iterated local search to the TSP, the QAP, and the FSP. In particular we will show that for the standard form of applying ILS which consists in applying the kick-move always to the best solution found so far, on most instances the run-time distributions with respect to high quality solutions are less steep than an exponential distribution. Thus, these algorithms could in the simplest case be improved by occasional restarts.

If restarts of an algorithm are used, cutoff values have to be chosen which determine when to stop the algorithm and to start over. Yet, when using *hard* cutoff times these may be not the optimal ones when solving an instance not seen before. When trying to solve such an instance, a better possibility often may be to use *soft* restart criteria which are related to the algorithm's search progress. If, for example, an ILS algorithm does not find an improved solution for a fixed number of iterations, one may assume that the algorithm is stuck and restart.

One interpretation of the fact that algorithms can be improved by restart is that they may get trapped in deep local minima or in certain regions of the search space. Iterative improvement algorithms, for example, stop at the first locally optimal solution found, and restarts from random initial solutions is the most simple strategy to improve their performance. Empirically, it has been shown that with metaheuristics like the ones applied in this thesis significantly better performance than with random restarts are obtained. As we will see, some of the most powerful metaheuristics again show a run-time behavior which implies that they can be improved by occasional restarts. Hence, one interpretation of this fact is that also these metaheuristics may have a certain lack of *explorative power* and they may still get trapped in search space regions. Hence, if such "stagnation behavior" is encountered the explorative power of the algorithm has to be increased. In the simplest case this is done by a random restart, but we will present also other possibilities of achieving this goal in this thesis, mainly in the context of iterated local search algorithms in Chapter 5.

Given this simple possibility to improve the ILS algorithm, in the following we will use the comparison with an exponential distribution as our criterion for the efficiency of metaheuristic algorithms. The main reason is that algorithms with exponential run-time distributions do not profit from random restarts and cannot be improved in this simple way. Yet, having an exponentially distributed run-time distribution cannot be taken alone as a good criterion for

an algorithm's efficiency. Consider, for example, multiple descent, which consists in restarting iterative improvement algorithms repeatedly from random initial solutions. The run-time distribution of multiple descent can be well approximated by an exponential distribution. Yet, multiple descent is outperformed on many problems by metaheuristic algorithms. In terms of run-time distributions this corresponds to the fact that the metaheuristic's run-time distributions are shifted to the left with respect to the one of multiple descent. Thus, *the goal of algorithm design should be to obtain metaheuristics which reach very high quality solutions as fast as possible and which have an associated run-time distribution at least as steep as an exponential distribution.*

3.3 Related Work

Run-time distributions have been occasionally measured in the Operations Research and the Artificial Intelligence literature. Taillard [241] observes exponential run-time distributions for a tabu search application to the QAP and concludes that the algorithm does not profit from restart. Similar results are reported by the same author for a tabu search application to the job shop scheduling problem [243]. Additionally, conditions are given under which restarting the algorithm may improve its performance. Exponential run-time distributions also have been observed by Battiti and Tecchiolli [17] for the application of a tabu search algorithm to the QAP; they conclude that high speed-up can be obtained by parallel processing based on multiple independent runs. In [247], run-time distributions are used to estimate the speed-up of parallel independent runs of a tabu search algorithm for the job shop scheduling problem. As these examples show, run-time distributions have been observed occasionally; however they are apparently not used systematically in the Operations Research literature for the empirical analysis of algorithms.

Most work in Artificial Intelligence on the empirical analysis of algorithms for SAT and CSPs concentrates on analyzing run-length of *exact* search algorithms. Kwan shows that the run-time for modern complete search algorithms for randomly generated binary CSPs is not normally distributed [148]. In [80], the cost distribution for randomly generated Random-3-SAT and binary CSPs from the phase transition region is approximated by continuous probability distributions. In [113], the run-length distribution of a backtracking algorithm applied to graph coloring problems is investigated. The authors make conjectures on the obtainable speed-up for parallel processing and find that the speed-up is strongly dependent on the cost distribution. Especially for multi-modal cost distributions they report high potential speed-ups. A similar approach is taken in the work of Gomes and Selman [99]. They intend to design algorithm portfolios using backtracking algorithms for a special kind of CSP. Based on the run-time distributions they found that one algorithm was strictly better than all the others and therefore the portfolio would only consist of one single algorithm. In a later work [100] they showed that the run-time distributions are "heavy-tailed" and conclude that rapid random restarts of the algorithm are an effective way for improving its performance.

Our work on the empirical evaluation of metaheuristics with run-time distributions is in part motivated by the theoretical work on Las Vegas algorithms [5, 160]. A Las Vegas algorithm [11] is a randomized algorithm that always produces a correct answer when it terminates but whose run-time is a random variable. In [5], the authors discuss general policies for reducing

the tail probability of Las Vegas algorithms. In [160], Luby, Sinclair and Zuckerman discuss policies for selecting cutoff times for known and unknown run-time distributions. For known run-time distributions, they show that an optimal strategy consists in restarting the algorithm after a *fixed* cutoff time. The more challenging situation arises when one is confronted with an unknown instance and for this case they present an optimal strategy for selecting cutoff values. Yet, for specific problem classes one can obtain knowledge about the general type of run-time distributions by conducting experiments on single instances [120]. Information gained in such a way can be used to devise improved strategies for the class of interest, as illustrated in Chapter 5 in the context of iterated local search.

3.4 Summary

In this chapter we have introduced a new empirical methodology for the analysis of metaheuristics and, more in general, for the analysis of randomized algorithms of Las Vegas type. The methodology is based on systematically measuring run-time distributions. The advantages of this methodology are (i) the visualization of an algorithm's behavior, (ii) the facilitation of the comparison of different algorithms, (iii) the possible characterization of an algorithm's behavior on a large class of problem instances, and (iv) the indication of situations in which algorithms can be improved. In this thesis we will strongly make use of this methodology for the analysis of ILS behavior and the design of improved ILS variants (see Chapter 5), as well as for the characterization of the local search behavior on hard binary CSP instances in Chapter 6. A discussion of the properties of run-time distributions has highlighted the special role of the exponential distribution, a specific distribution well known from statistical literature, and we could formulate the goal of metaheuristic design as to obtain metaheuristics which reach very high quality solutions as fast as possible and which have an associated run-time distribution at least as steep as an exponential distribution.

Chapter 4

$\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

The seminal work on ant colony optimization (ACO) is ant system (AS) [67, 63, 68]. Like many other metaheuristic algorithms, AS was first applied to the TSP, showing the viability of the approach on some rather small TSP instances. Yet, the performance on larger instances was not very satisfactory with respect to solution quality and computation time. In this chapter we present $\mathcal{MAX}$–$\mathcal{MIN}$ ant system ($\mathcal{MM}$AS) [235, 237, 236] which significantly improves the performance of AS. Additionally, the research on $\mathcal{MM}$AS has been one of the first foundations of the fact that, when applied to a wide variety of combinatorial optimization problems, the performance of ant algorithms is best when used in combination with local search algorithms that improve the solutions constructed by the ants [165, 66, 235, 236]. Currently, $\mathcal{MM}$AS is one of the best performing ACO algorithms for combinatorial optimization problems like the traveling salesman problem (TSP), the quadratic assignment problem (QAP), and the flow shop problem (FSP). The research in the field of ant colony optimization has recently experienced a rapid growth of interest as shown by an increased number of successful applications [65]. By providing an improved algorithm and a new application, we contribute to this new emergent field.

The chapter is structured as follows. In the first section we present AS using the TSP as an example application and give an overview of algorithmic improvements over AS. The following section contains a detailed presentation of $\mathcal{MM}$AS and presents computational results that show the effectiveness of the modifications to AS which have been introduced by $\mathcal{MM}$AS. While this first application uses $\mathcal{MM}$AS as a randomized construction heuristic, in the remainder of this chapter we use $\mathcal{MM}$AS as a hybrid algorithm in which local search algorithms are used to improve the ants' solutions. In Section 4.3 we first apply the so extended $\mathcal{MM}$AS to the TSP followed by applications to the QAP and the FSP in Sections 4.4 and 4.5, respectively. With these three different applications of $\mathcal{MM}$AS we also examine important issues in the design of hybrid algorithms. In particular, in the application to the TSP we investigate the issue which ants should improve their solution by a local search to achieve a good compromise between solution quality and computation speed. As a result we show that a gradual increase of the number of ants which apply local search is the best alternative. In the application to the QAP we examine which local search algorithm should best be combined with $\mathcal{MM}$AS and show that for the QAP the best choice is strongly dependent on the instance type. In the FSP application we show that for very short computation times the use of a single ant is enough to find high quality solutions. The computational results obtained for all the three example applications show that $\mathcal{MM}$AS is a very effective ACO algorithm and that $\mathcal{MM}$AS is among the

best algorithms for the attacked problems. Additionally, in Section 4.6, we discuss parallelization strategies for ACO algorithms and some indicative results with $\mathcal{MM}$AS show that parallel implementations may achieve a significant speed-up and gain in overall performance. Finally, we review other applications of ACO algorithms to combinatorial optimization problems and summarize our results.

4.1 Introduction

Ant colony optimization is a new metaheuristic approach which is inspired by the foraging behavior of real ant colonies. Ant system (AS) [67, 63, 68] is the seminal work on ant colony optimization, laying the basis for the following activities in this field. It has been first presented by Dorigo, Maniezzo, and Colorni in [67] and also as part of Marco Dorigo's PhD Thesis [63]. Ant system was first applied to the TSP to which the search metaphor of ant colony optimization is easily adapted.

In AS, and more in general in ACO algorithms, ants are simple agents which iteratively construct candidate solution to a given combinatorial optimization problem. The ants' solution construction is guided by (artificial) pheromone trails (which imitate the chemical pheromone trails real ants use as an indirect form of communication) and problem-dependent heuristic information. In principle, ACO algorithms can be applied to any combinatorial optimization problem by defining *solution components* which the ants use to iteratively construct candidate solutions and on which they may deposit pheromone (see [64, 65] for more details). An individual ant constructs candidate solutions by starting with an empty solution and then iteratively adding solution components until a complete candidate solution is generated. After the solution construction is completed, the ants give indirect feedback on the solutions they have constructed by depositing pheromone on solution components which they have used in their solution. Typically, solution components which are part of better solutions or are used by many ants will receive a higher amount of pheromone and, hence, are more likely to be used by the ants in future iterations of the algorithm. To model the evaporation of biological pheromone trails, before the pheromone trails get reinforced, all pheromone trails are decreased by a factor ρ.

Recent research in ant colony optimization has shown that for applications to combinatorial optimization problems, best performance is obtained if the ants are enhanced by additional capabilities. For example, in many of the most efficient implementations of ACO algorithms, the ants may apply local search algorithms [165, 66, 235, 236] to improve the solutions they have constructed. Another possible improvement is the use of a lookahead function which is used during the solution construction. The latter approach has shown benefit in the application of ant system to the shortest supersequence problem [177].

In the following we present in detail the application of AS to the TSP and then discuss the modifications to AS which have been introduced by proposed algorithmic improvements, while $\mathcal{MM}$AS will be presented in detail in the next section.

4.1.1 Ant System

For the application of AS and the later introduced ACO algorithms to the TSP, a pheromone trail $\tau_{ij}(t)$ is associated to each arc (i, j), where $\tau_{ij}(t)$ is a real number which is modified during

the algorithm's run. The iteration counter t is used to indicate that the pheromone trail of the solution components changes during the algorithm's run. The pheromone trails are represented in a pheromone trail matrix which has n^2 entries, one for each arc. When ant system is applied to symmetric TSP instances, we always have $\tau_{ij}(t) = \tau_{ji}(t)$; for the application to asymmetric TSPs (ATSPs) we may have $\tau_{ij}(t) \neq \tau_{ji}(t)$. In the first case, we will also talk of (undirected) edges instead of (directed) arcs when referring to ATSPs. When constructing a tour, an ant probabilistically prefers to choose a short arc with high pheromone trail leading to a city it has not visited. We will denote the set of still unvisited cities by ant k as $U(k)$.

Initially, three different versions of AS have been proposed [63, 48, 68]. These are called *ant-density*, *ant-quantity*, and *ant-cycle*. In ant-density and ant-quantity ants only deposit pheromone directly after crossing an arc. In ant-quantity the amount of pheromone is inversely proportional to the length of the arc crossed, whereas in ant-density a constant amount of pheromone per unit distance is put down. In ant-cycle the ants are allowed to lay down pheromone only when they have completed a tour, that is, a *global* pheromone trail updating rule is applied as opposed to a *local* pheromone trail updating rule in the ant-density and ant-quantity model. Since the ant-cycle model has shown to perform significantly better than the other two versions, we only present this later version in the following and refer to it generally as ant system.

Tour Construction

In AS, each of the m ants in the colony constructs a solution to the TSP. To do so an ant uses only local information available at each node. This information consists of a heuristic function η_{ij} and a pheromone trail $\tau_{ij}(t)$. Additionally, an ant uses a list to keep track of the cities it has visited and to store the partial tour constructed so far. This list is also used to prevent transitions to already visited cities and to force the ant to construct legal tours. The solution construction by the artificial ants can then be imagined as a walk over a weighted, fully connected graph where the nodes represent the cities and the arcs the connections between the cities. For the tour construction, each ant k is initially set on some randomly chosen city. Then, starting from that city, an ant moves to a still unvisited city until a tour is completed. An ant probabilistically prefers to move to cities which are close and which are connected by an arc with a high pheromone value. In particular, an ant which is at city i chooses the next city j according to the following probability distribution.

$$p_{ij} = \begin{cases} \dfrac{\tau_{ij}^{\alpha}(t) \cdot \eta_{ij}^{\beta}}{\sum_{l \in U(k)} \tau_{il}^{\alpha}(t) \cdot \eta_{il}^{\beta}} & \text{if city } j \text{ is not yet visited} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

In AS the heuristic function chosen is $\eta_{ij} = 1/d_{ij}$. It can be interpreted as the greedy desire of choosing city j next; the smaller the distance d_{ij}, the higher is the probability of choosing next city j. The probability is also positively correlated to the pheromone trail $\tau_{ij}(t)$. The parameters α and β determine the relative influence of the pheromone trail and the heuristic information. If $\alpha = 0$ the construction algorithm corresponds to a randomized greedy tour construction. In fact, the higher β the closer the constructed tours will be to those returned by the nearest neighbor heuristic (see Section 2.6 on page 21). The ants repeat these steps until a tour is completed and then they calculate the length of their tour.

Update of Pheromone Trails

After all ants have constructed their tours, the pheromone trails are updated. This is done by first lowering the pheromone trail of *all* arcs by a constant factor and then allowing each ant to add pheromone on the arcs it has used. The update is done according to

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) \tag{4.2}$$

where $0 < \rho < 1$ is the persistence of the pheromone trail, thus $1 - \rho$ represents the pheromone evaporation. The parameter ρ is used to avoid unlimited accumulation of the pheromone trails and it enables the algorithm to forget bad choices done previously. If an arc is not chosen by the ants, the associated pheromone trail decreases exponentially. $\Delta\tau_{ij}^{k}(t)$ is the amount of pheromone ant k puts on the arcs it has visited; it is defined as follows.

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} Q/f(s_k) & \text{if arc } (i,j) \text{ is used by ant } k \\ \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

Q is the total amount of pheromone deposited by an ant and $f(s_k)$ is the length of the kth ant's tour. By Equation 4.3 the more pheromone an arc receives, the better the ant's tour is. In general, arcs which are used by many ants and which are contained in shorter tours will receive more pheromone and therefore are also more likely to be chosen in future iterations of the algorithm.

The pheromone trail $\tau_{ij}(t)$ can be interpreted as a form of long term memory and, in some sense, ACO algorithms learn which arcs are contained in good solutions. In this sense, the algorithm is used to identify solution components that should preferably be chosen to construct good solutions.

Search Stagnation and the Branching Factor

The major concern in AS is the treatment of pheromone trail intensities. If the pheromone trail intensities are very high on one or very few arcs incident from a city and very low on the others, the ants are very likely to choose the next arc among those with a very high trail intensity (due to Equation 4.1). The long term effect of the pheromone trails is to reduce the size of the effective search space successively by concentrating the search on a relatively small number of arcs. To characterize the amount of exploration, the λ-branching factor, $0 < \lambda < 1$, was introduced in [82]. Its definition is based on the following notion: If for a given city i the concentration of pheromone trail on almost all the arcs exiting from it becomes very small, the freedom of choice for extending partial solutions from that city is very limited. Consequently, if this situation arises simultaneously for all nodes of the graph, the part of the search space that is effectively searched by the ants becomes very small.

The λ-branching factor for a city i is defined as follows: If τ_i^{max} is the maximal and τ_i^{min} the minimal trail intensity on arcs exiting from node i, the λ-branching factor is given by the number of arcs exiting from i which have a trail intensity $\tau_{ij} > \tau_i^{min} + \lambda \cdot (\tau_i^{max} - \tau_i^{min})$. The average λ-branching factor $\bar{\lambda}$ is the average of the λ-branching factors of all nodes and gives an indication of the size of the search space explored by the ants. If, for example, $\bar{\lambda}$ is very close

to 1, on average only one arc incident from a node has a high probability of being chosen.[1] In such a situation the ants are very likely to follow the same path over and over again and the phenomenon of search *stagnation* occurs [68]. Such situations have been observed for ant system for high values of the parameter α [68].

Some Observations on the Run-Time of Ant System

One disadvantage of AS is its high run time which makes its application to large TSPs infeasible. In AS at each iteration m ants construct a tour and each tour construction has complexity $O(n^2)$. Since typically the number of ants in the population is as high as the number of cities, that is, $m = n$, this leads to an overall complexity of $O(n^3)$ for each AS iteration [68]. Therefore, the run time grows very fast with increasing problem size. The high run time is mainly due to the sequential implementation of an inherently parallel algorithm. The tour construction steps for the individual ants can be done independently and this provides an obvious hint of how to parallelize the algorithm (we refer to Section 4.6 for a discussion of parallelization possibilities for ant algorithms).

An important part of the run-time of the algorithm is due to the complexity of the solution construction. Therefore, some effort should be invested in reducing the complexity of the construction steps. One such possibility is to use *candidate lists* for each city [211] which, in the context of ACO, was first proposed with the application of ant colony system [83] to the TSP. Using candidate lists is a widely used technique to speed up local search for the TSP [129, 169, 211]. Typically, a candidate list comprises a fixed number of nearest neighbors for each city but also other possibilities like including additionally the adjacent cities of the Delauney graph have been proposed [211]. The reasoning behind this method is that in good tours most cities are connected to one of their nearest neighbors. In many TSP instances the optimal tour can be found within a surprisingly low number of nearest neighbors, for example, an optimal solution is found for instance `pcb442` (a 442 city instance) within a subgraph of the 6 nearest neighbors and for instance `pr2392` (a 2392 city instance) within a subgraph of the 8 nearest ones [211].

When using candidate lists for the tour construction, the next city is chosen among those of the candidate list if possible. Only if all the members of the candidate list of a city are already included in the partial tour, other cities are considered. Using candidate lists has two main positive effects. First, it reduces drastically the run-time of the algorithm. Now, a tour can be constructed in $O(n)$, reducing the complexity of one iteration of the algorithm to $O(n^2)$, although with a high constant hidden in the $O(\cdot)$–notation. Second, by restricting the set of possible cities to the most promising ones, it can be expected that, especially for large TSPs, the ants are able to concentrate their search faster on good solutions.

4.1.2 Improvements on Ant System

The basic form of AS performs rather poorly on larger TSP instances and therefore several algorithmic improvements have been proposed to improve its performance. All these improvements have in common that they introduce a form of elitism which is used to direct the search

[1] The minimal $\bar{\lambda}$ for symmetric TSPs is 2, but throughout the thesis we normalize $\bar{\lambda}$ to 1.

more strongly towards the best tours. The first improvement was the *elitist* strategy for ant system (AS$_e$) [63, 68], others are Ant-Q [82] and its successor ant colony system (ACS) [83, 66], $\mathcal{MAX}$–$\mathcal{MIN}$ ant system ($\mathcal{MM}$AS) [235, 237, 236], and the rank-based version of ant system (AS$_{rank}$) [36]. While AS$_e$, $\mathcal{MM}$AS, and AS$_{rank}$ are direct modifications of AS, Ant-Q and ACS introduce major modifications and change strongly the search behavior compared to AS. We will give a short description of all these extensions except for $\mathcal{MM}$AS, which is described in detail in Section 4.2.

Elitist Strategy for Ant System

AS$_e$, introduced in [63, 68], is the first attempt to improve AS's performance. The idea is to give a strong additional reinforcement to the arcs of the best tour found since the start of the algorithm; this tour is denoted as s_{gb} (global-best) in the following. This is achieved by adding to the arcs of s_{gb} a quantity $e \cdot Q/f(s_{gb})$ whenever the pheromones are updated. Thus, Equation 4.2 is modified to

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) + e \cdot \Delta\tau_{ij}^{gb}(t) \tag{4.4}$$

where e is the weight of tour s_{gb}. $\Delta\tau_{ij}^{k}(t)$ is defined as in Equation 4.3 and $\Delta\tau_{ij}^{gb}(t)$ is defined analogously to $\Delta\tau_{ij}^{k}(t)$ using the global-best tour. Some limited results presented in [63, 68] suggest, that using the elitist strategy with an appropriate number of elitist ants better tours than with ant system are found and these are discovered earlier in the run. Yet, for too large weight e the search concentrates early around suboptimal solutions leading to an early *stagnation* of the search.

Ant-Q and Ant Colony System

Ant-Q has been introduced in [82], recognizing the connections between reinforcement learning algorithms [133] and ant colony optimization. Ant-Q exploits the connections with Q-learning [258], a specific type of reinforcement learning algorithm. Ant colony system (ACS) [83, 66] is a direct successor of Ant-Q, proposed by the same authors. ACS and Ant-Q differ only in one specific aspect, which is described below. We focus on ACS because it is conceptually simpler than Ant-Q and it is preferred by its authors.

ACS differs in three main aspects from ant system. First, ACS uses a more aggressive action choice rule. Second, pheromone is added only to arcs belonging to the global-best solution. Third, each ant applies a local pheromone update rule to the arc it has just crossed and removes some pheromone on this arc.

The action choice rule in ACS, called *pseudo-random proportional rule*, is given as follows.

$$j = \begin{cases} \arg\max_{l \in U(k)}\{\tau_{il} \cdot \eta_{il}^{\beta}\} & \text{if } p \leq p_0 \\ \\ S & \text{otherwise} \end{cases} \tag{4.5}$$

where p is a random number uniformly distributed in $[0, 1]$. Thus, the best possible move, as indicated by the pheromone trail and the heuristic information, is made with probability

$0 \leq p_0 < 1$ (exploitation); with probability $1 - p_0$ a move is made based on the random variable S with distribution given by Equation 4.1 (biased exploration).

In ACS only the global-best ant is allowed to add pheromone after each iteration. Thus, the update according to Equation 4.2 is modified to

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \rho \cdot \Delta\tau_{ij}^{gb}(t) \tag{4.6}$$

where $\Delta\tau_{ij}^{gb}(t) = 1/f(s_{gb})$ is used in ACS and Ant-Q and the parameter ρ represents the pheromone decay parameter. It is important to note that this pheromone update rule is only applied to the arcs (i, j) of the global-best tour, not to all arcs like in AS.[2] This choice, together with the pseudo-random-proportional rule is made to make the search more directed [66].

Additionally to the global updating rule, in ACS the ants use a local update rule that they apply immediately after crossing an arc during the tour construction.

$$\tau_{ij} = (1-\xi) \cdot \tau_{ij} + \xi \cdot \Delta\tau_{ij} \tag{4.7}$$

with $0 < \xi < 1$. As explained in [66], the effect of the local updating rule is to make an already chosen arc less desirable for a following ant and in this way generate more diverse tours.

The only difference between ACS and Ant-Q is the definition of the term $\Delta\tau_{ij}$. Ant-Q uses a formula for $\Delta\tau_{ij}$ which was inspired by Q-learning. In Ant-Q the term $\Delta\tau_{ij}$ corresponds to the discounted evaluation of the next state and is set to $\gamma \cdot \max_{U(k)}\{\tau_{il}\}$ [82]. In ACS, $\Delta\tau_{ij}$ is set to a small constant value $\tau_0 = (n \cdot f(s_{nn}))^{-1}$, where $f(s_{nn})$ is the tour length of a nearest neighbor tour. It could be shown that this choice led to approximately the same performance for Ant-Q and ACS, but it has the advantage to require much less computation.

Rank-Based Version of Ant System

Another improvement over ant system is the *rank-based version of ant system* (AS_{rank}) [36]. In AS_{rank}, always the global-best tour is used to update the pheromone trails, similar to AS_e. Additionally, a number of the best ants of the current iteration is allowed to add pheromone. To this aim the ants are sorted by tour length ($f(s_1) \leq f(s_2) \leq \ldots \leq f(s_m)$), and the quantity of pheromone an ant may deposit is weighted according to the rank r of the ant. Only the $w-1$ best ants of each iteration are allowed to deposit pheromone and additionally the global-best solution gives the strongest feedback of weight w. The rth best ant of the current iteration is allowed to deposit $\max\{0, w-r\} \cdot Q$ units of pheromone on the arcs used in its tour. Thus the modified update rule is

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \sum_{r=1}^{w-1} \Delta\tau_{ij}^{r}(t) + w \cdot \Delta\tau_{ij}^{gb}(t), \tag{4.8}$$

[2] Initially, also using the iteration-best solution was considered for the pheromone update. In [66] it is claimed that *the difference between the two schemes is minimal*. Yet, this claim is only true for very small TSP instances. We tested ACS with the iteration-best solution on some medium sized TSPLIB instances and the performance degraded strongly with increasing problem size.

where

$$\Delta\tau_{ij}^{r}(t) = \begin{cases} \max\{0, w-r\} \cdot Q/f(s_r) & \text{if arc } (i,j) \text{ is used in the tour of the } r\text{th best ant} \\ 0 & \text{otherwise} \end{cases} \tag{4.9}$$

In [36], AS$_{rank}$ has been compared to AS$_e$, to AS, to a genetic algorithm and to simulated annealing. For the larger TSP instances (the largest instance with 132 cities) the approaches based on AS showed to be superior to the genetic algorithm and the simulated annealing procedure. Among the AS-based algorithms, both, AS$_{rank}$ and AS$_e$ performed significantly better than AS, with AS$_{rank}$ showing slightly better results than AS$_e$.

4.2 $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

4.2.1 Motivation for $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

In initial experiments with AS, we found that adding more greediness significantly improves performance and good solutions are found much more quickly. AS without some greedy component gives rather poor results especially when applied to larger problem instances (see also experimental results in Section 4.2.7). Initially, we studied an extension of AS which we called *generalized elitist strategy*. In this extension only the ants within a factor ϕ of the best solution were allowed to update the pheromone trails. This variation was further extended by an adaptive choice of ϕ depending on the λ-branching factor. In another extension, ants were only probabilistically allowed to reinforce their tours, better ants having a higher probability to be chosen for a pheromone update. These variations led to improved performance compared to AS, but a major problem was the early stagnation of the search as indicated by the average branching factor $\bar{\lambda}$. In such situations, we have $\bar{\lambda} \approx 1$. Yet, interestingly, the best solutions were found at rather low values for $\bar{\lambda}$, when the search was *nearly* stagnated. We interpreted this fact as follows: the best performance is achieved if the search has concentrated around the best solutions found during the run of the algorithm but early stagnation of the search may strongly compromise performance.

Thus, to achieve significantly better performance than AS on TSPs one has to include some form of exploitation of the best solutions found during search *and* a mechanism to avoid too early stagnation of the search. This led us to the development of the $\mathcal{MAX}$–$\mathcal{MIN}$ ant system ($\mathcal{MM}$AS) , which differs in three main aspects from AS.

(i) To exploit the best solutions found during an iteration or during the run of the algorithm, after each iteration only one single ant is used for pheromone update. This ant may be the *iteration-best* or the *global-best* ant.

(ii) To limit the stagnation of the search, a more direct influence on the pheromone trail limits is exerted by limiting the allowed range of possible pheromone trails to an interval determined by $[\tau_{min}, \tau_{max}]$, where τ_{min} is the minimal pheromone trail and τ_{max} the maximal pheromone trail on any arc.

(iii) Additionally, we initialize the pheromone trails to τ_{max}, leading to a higher exploration of tours at the start of the algorithm.

At the first glance, adding limits on the pheromone trail may seem quite unnatural. But as already mentioned, ant system is only rather loosely coupled to the original behavior of the ants and its main goal is to provide an effective tool for the solution of combinatorial optimization problems. In the next sections we discuss more thoroughly the differences between $\mathcal{MM}$AS and AS and justify the effectiveness of the three main modifications by experiments.

4.2.2 Pheromone Trail Update

As mentioned before, in $\mathcal{MM}$AS only one ant is used to update the pheromones after each iteration. Two possibilities for the choice of the ant which updates the pheromone trails have been considered. We either allow the ant with the best tour in the current iteration, the *iteration-best* ant or the *global-best* ant to update the pheromone trails. $\mathcal{MM}$AS uses the following update rule.

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}^{best}, \tag{4.10}$$

where τ_{ij}^{best} is $1/f(s_{best})$, and s_{best} may be either the ant with the iteration-best tour or the one with the global-best tour. The term $\Delta\tau_{ij}^{best}$ is defined analogous to $\Delta\tau_{ij}^{k}(t)$ in Equation 4.3 but using either the iteration-best or the global-best tour.

$\mathcal{MM}$AS and ant colony system (ACS) exploit more strongly the best solutions by using only one single ant in the pheromone trail update. Yet, an important difference between $\mathcal{MM}$AS and ACS is the different interpretation of exploitation and exploration. Exploitation in ACS is mainly interpreted as choosing a high parameter value of p_0 (see Equation 4.5). In this way the accumulated knowledge on the problem is exploited by constructing a solution that can be interpreted as a slight modification of the best solution found so far. Exploration in ACS is obtained by using a biased random move according to 4.1 with a probability of $1-p_0$ [66]. In $\mathcal{MM}$AS, exploitation is mainly interpreted as choosing one single ant, either the iteration-best or the global-best ant, for the pheromone update. Together with a rather high parameter value for ρ this will slowly shift the probability distribution given by Equation 4.1 towards solution components (arcs) which have been shown to be contained in the best solutions. Still, a judicious choice between these two alternatives for the pheromone updating ant influences the strength of the exploitation of the search space. When using only the global-best ant, the search may concentrate too fast around this global-best solution and the exploration of possibly better tours is limited. Hence, the danger of getting trapped in poor quality solutions is higher. Using the iteration-best ant favors the exploration of possibly better tours since especially in the starting phase of the algorithm the iteration-best tours will differ considerably and early mistakes are more easily avoided. Also intermediate approaches can be applied like, choosing by default the iteration-best ant to update the pheromone trails and using the global-best tour only every fixed number of iterations.

In the worst case, independent of the choice of the iteration-best ant and the global-best ant for the trail update, certain solution components may almost always be chosen by Equation 4.1. To avoid such phenomena and to increase the exploration of tours, pheromone trail limits are introduced in $\mathcal{MM}$AS, which are discussed next.

4.2.3 Pheromone Trail Limits

If some form of elitist strategy is used in combination with AS, the early stagnation of the search is a major problem. Stagnation of the search occurs, for example, in the following situation. If the pheromone trail on only one arc incident from a city is very high compared to the others, this arc will be chosen almost always in Equation 4.1. If such a situation arises at all nodes, the tour corresponding to the best one found so far will be constructed by most ants and the search for better solutions stagnates.

To avoid such a situation one may influence the selection probabilities, which are determined by the pheromone trails and the heuristic information, for the next city. Hence, one possibility is to limit the influence of the pheromone trail and avoid that these grow too high or become too low during the run of the algorithm. To achieve this goal, we introduce explicit limits on the pheromone trail allowed by using two parameters τ_{min} and τ_{max} and we require for the pheromone trail on all arcs that $\tau_{min} \leq \tau_{ij}(t) \leq \tau_{max}$. By limiting the range of values for the pheromone trail, the probability distribution given by Equation 4.1 can be indirectly influenced. In particular, if $\tau_{min} > 0$, the probability of choosing an arc is comprised in an interval $[p_{min}, p_{max}]$, where $0 < p_{min} \leq p_{max} \leq 1$. If we assume that the maximal intercity distance in the TSP instance is bounded, $p_{min} > 0$ holds. If only one arc can be chosen to generate a feasible solution, we have $p_{min} = p_{max} = 1$. For the application to the TSP, provided no candidate sets are used, this situation can occur only if the ant has to return to its start city to generate a closed tour, otherwise we will always have $p_{max} < 1$. Hence, by having $p_{min} > 0$ in $\mathcal{MM}$AS, any tour and, thus, also the optimal one can be constructed with probability larger than zero. But this probability may be vanishingly small. With this property it can also be easily proved that $\mathcal{MM}$AS will find the optimal solution asymptotically. On the other side, the best solutions during the search are found when $\bar{\lambda}$ is already very low or, in other words, shortly before stagnation occurs. Thus the values for the upper and lower trail limit should take into account both issues, avoiding stagnation of the search and not inhibiting a concentration of the search on a small subset of the arcs, where good solutions may preferably be found.

For the following discussion we introduce informally the notion of *convergence* for $\mathcal{MAX}$–$\mathcal{MIN}$ ant system. We say that $\mathcal{MM}$AS has *converged* if for each city, the pheromone trail on one incident arc from it has a pheromone trail of level τ_{max}, and all other arcs have a pheromone trail which is at the minimum level. The tour that would be built by choosing always the arc with maximal pheromone trail will typically correspond to the best tour found by the algorithm. The concept of convergence of $\mathcal{MM}$AS differs in one slight but important aspect from the concept of search stagnation. While stagnation describes the situation where all ants follow the same path, in convergence situations of $\mathcal{MM}$AS this is not necessarily the case due to the use of the trail limits.

In some preliminary experiments, we found that especially the lower trail limits are very important. We propose to choose specific values for τ_{max} and τ_{min} according to the particular instance. We propose to set the maximal pheromone trail to an estimation of its analytically maximal value which is given in the following proposition.

Proposition 4.2.1 *τ_{max}, the maximal possible pheromone trail referring to a solution component is given by*

$$\tau_{max} = \frac{1}{1-\rho} \cdot \frac{1}{f(s_{opt})} \tag{4.11}$$

Proof: By Equation 4.10 the largest possible amount of pheromone added after any iteration corresponds to $1/f(s_{opt})$, where $f(s_{opt})$ is the optimal solution value for a specific problem. After each iteration at most this amount of pheromone is added. Thus, the discounted pheromone trail up to iteration t corresponds to

$$\sum_{i=1}^{t} \rho^{t-i} \cdot \frac{1}{f(s_{opt})}$$

Asymptotically, because $\rho < 1$, this sum converges to

$$\frac{1}{1-\rho} \cdot \frac{1}{f(s_{opt})}$$

For $t \mapsto \infty$ the influence of the initial pheromone trail $\tau(0)$ is negligible.

In our implementation we use $f(s_{gb})$, the tour length of the global-best solution to determine an appropriate value for τ_{max} by Equation 4.11. Clearly, the optimal solution value is not known before the run and $f(s_{gb})$ is used as an estimate of that value. Thus, the upper trail limit is adapted during the run of the algorithm. The initial pheromone trail $\tau(0)$ could be set to any value, because after the first iteration the pheromone trail will be forced to fall within $[\tau_{min}, \tau_{max}]$. Yet, we propose to set $\tau(0)$ to some very high value, such that after the first iteration the trails correspond to τ_{max} (see also the next section for a discussion of the trail initialization).

Still, reasonable values for τ_{min} have to be determined. We choose these values according to the following assumptions.

(i) The best tours are found shortly before the stagnation of the search occurs. In such a case the probability that the best tour found is constructed in one iteration is significantly higher than zero. Better tours may be found close to the best tour found.

(ii) The main influence on the tour construction is determined by the relative difference between upper and lower trail limits.

Note that the validity of the first assumption depends strongly on the search space characteristics of the problem. It says implicitly that around good solutions there is a reasonable chance to find even better ones. This assumption is certainly the case for the TSP for which a significant fitness-distance correlation exists (see Section 2.9 for a short discussion of this aspect). The second assumption is used in the following to specify one possible way to determine heuristically reasonably values for the lower trail limit. The rationale of the following derivation of the τ_{min} settings is based on the intuition that the trail limits should be wide enough to allow a focus of the search around the best solutions found (this is the case in convergence situations of $\mathcal{MM}$AS) and at the same time to allow for a sufficient large exploration (that is, by making the trail limits tight enough) of possibly better tours.

Based on some further simplifications we propose a systematic way for determining appropriate lower trail limits. The first simplification consists in neglecting the influence of the heuristic information on the probability distribution in Equation 4.1. This is possible if the influence of the heuristic information is low, which, as is typically done in $\mathcal{MM}$AS, is the case

if the parameter β is chosen rather low or if no heuristic information is used at all. Given these assumptions, good values for τ_{min} can be found by relating the convergence of the algorithm to the minimum trail limit. When $\mathcal{MM}$AS has *converged*, the best solution found is constructed with a probability p_{best} which is significantly higher than 0.[3] In this situation, an ant constructs the best solution found if it makes at each city the "right" decision and chooses an arc with maximum pheromone trail τ_{max}. In fact, the probability p_{dec} of choosing the "right" city directly depends on the distribution of the pheromone trails and these values are indirectly determined by τ_{max} and τ_{min}. For the sake of simplicity, let us assume that p_{dec} is constant at all cities during the tour construction. Then, an ant has to make n times the "right" decision and, hence, it will construct the best solution with a probability of p_{dec}^n. By setting

$$p_{dec}^{(n)} = p_{best}, \tag{4.12}$$

we can determine p_{dec} as

$$p_{dec} = \sqrt[n]{p_{best}}. \tag{4.13}$$

So, given a value for p_{best}, we can now determine appropriate settings for τ_{min}. On average, an ant has to choose among $n/2$ cities if no candidate lists are used or among $cl/2$ cities if candidate lists of size cl are used. Thus — on average — an ant has to choose among *avg* cities. (Recall that for the determination of p_{dec} we also made the assumption that it remains constant, which also justifies the use of *avg*.) Then, the probability to make the right decision according to Equation 4.1 can be calculated as[4]

$$\frac{\tau_{max}}{\tau_{max} + avg \cdot \tau_{min}} = p_{dec} \tag{4.14}$$

Solving this equation for τ_{min} yields

$$\tau_{min} = \frac{\tau_{max} \cdot (1 - p_{dec})}{avg \cdot p_{dec}} = \frac{\tau_{max} \cdot (1 - \sqrt[n]{p_{best}})}{avg \cdot \sqrt[n]{p_{best}}} \tag{4.15}$$

Based on this equation, we can now determine τ_{min}, given a value for p_{best}. Choosing values for p_{best} is directly related to the amount of exploration done by $\mathcal{MAX}$–$\mathcal{MIN}$ ant system when it has converged. Thus, p_{best} provides a good way of investigating the effect of the lower trail limits on the performance of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system.

In Section 4.2.5 we will investigate the proposed settings of τ_{min}. In particular we show the usefulness of the lower trail limits by comparing the solution quality obtained by $\mathcal{MM}$AS with and without using these limits.

4.2.4 Pheromone Trail Initialization

In $\mathcal{MAX}$–$\mathcal{MIN}$ ant system we initialize the pheromone trails is such a way that after the first iteration all pheromone trails correspond to τ_{max}. This can easily be achieved by setting $\tau(0)$ to some arbitrarily high value. After the first iteration of $\mathcal{MM}$AS, the trails will be forced to

[3] Intuitively, we require p_{best} to be relatively large and later give numeric examples of reasonable values for p_{best} for the the $\mathcal{MM}$AS application to the TSP.

[4] Equation 4.14 is obtained by requiring that the arc with pheromone trail τ_{max} is chosen and we have *avg* other arcs with associated pheromone trail τ_{min}.

take values within the imposed bounds, in particular, they will be set to $\tau_{max}(1)$. This type of trail initialization is chosen to increase the exploration of solutions during the first iterations of the algorithm. To illustrate this fact, consider the following example: Due to the trail evaporation (determined by parameter ρ), after the first iteration the relative difference between the pheromone trails on solution components will differ by a factor of at most ρ, after the second by ρ^2, etc. If, on the contrary, the trails would be initialized to their lower limits τ_{min}, the relative differences between the pheromone trail would increase much more strongly; in particular, in this latter case, the factor between τ_{min} and the amount of pheromone deposited on a solution element is $(1-\rho) \cdot (avg \cdot p_{dec})/(1-p_{dec})$. With the empirically chosen parameter settings, this factor is significantly higher than the relative difference among the pheromone trail when initializing the trails to τ_{max}. For example, with the parameter settings chosen for the experimental investigation in the next section, in the first case this factor would amount to 6.44, while when initializing the trails to τ_{max} this factor corresponds to 1.02. Thus, the selection probabilities according to Equation 4.1 evolve more slowly when initializing the trails to τ_{max} and, hence, the exploration of solutions is favored. The experimental results presented in Section 4.2.5 confirm the conjecture that the higher exploration of the search space due the trail initialization in $\mathcal{MM}$AS improves performance.

A specific interpretation of the search progress in $\mathcal{MM}$AS is implied, by the particular way in which the pheromone trail are initialized. After each iteration of the algorithm the pheromone trail will be reduced due to evaporation to $\tau_{ij}(t) = \rho \cdot \tau_{ij}(t-1)$. Since only the best ant is allowed to update the pheromone, only the pheromone trails of arcs participating in the best tours increase their pheromone trail or maintain them at the upper trail limit. Arcs which do not receive any or very rare reinforcement will continuously lower their pheromone trail and be selected more rarely by the ants. In this sense, $\mathcal{MAX}$–$\mathcal{MIN}$ ant system tries to avoid the errors made in the past. An error is associated with choosing arcs that lead to rather bad tours; call them *bad* arcs. So, the pheromone trail on bad arcs sinks down slowly and only *good* arcs can maintain a higher level of pheromone trail. These good arcs are then combined by the probabilistic tour construction to generate better tours.

In fact, the parameter ρ can be interpreted as a learning rate. In some sense, with higher values for ρ the exploration of other solutions in the initial phase of the algorithm is increased. Higher values for ρ indicate a slower learning of the "good" arcs, since the trail level on the other arcs sinks down more slowly and it is easier to correct early mistakes made by reinforcing arcs contained in worse tours. Yet, ρ should not be chosen too high to ensure convergence of the algorithm. Clearly, the convergence speed will depend on the particular setting of ρ. Our understanding of ρ as a learning rate parameter will be underlined in the following sections.

4.2.5 Experiments with $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

In this section we study the effectiveness of the three main modifications introduced by $\mathcal{MM}$AS and the influence of specific parameter settings on $\mathcal{MM}$AS performance using its application to the TSP. Since we are mainly interested in the effectiveness of the introduced modifications to ant system, the experimental study is based on the number of tour constructions and we do not use local search to improve solutions. For the performance of $\mathcal{MM}$AS on the TSP with local search we refer to Section 4.3.

As far as not indicated otherwise, we use the following parameter settings. The amount of pheromone an ant deposits on an arc is always $Q = 1$, because the particular value of Q has not a significant influence on the final performance [68]. The parameters α and β are set to 1 and 2, respectively. ρ is set to 0.98, corresponding to a rather slow "learning". This is done because we are more interested in the performance of longer runs; with lower values of ρ on short runs better tours can be found (see also next section). For the pheromone update we use only the iteration-best ant. The trail limits were chosen as proposed in Section 4.2.3, using $p_{best} = 0.05$. The number m of ants is set to the number n of cities . The ants are initially put on a randomly chosen city and the candidate list size *cl* is set to 20.

The TSP benchmark instances are all taken from TSPLIB; for all instances the optimal solution value is known. We will refer to the benchmark instances by the identifier used in TSPLIB which also indicates the number of cities (instance `eil51` has 51 cities, etc.). The only exception is instance `kro124p` which has 100 cities.

All the experiments were performed with a *ceteris paribus* assumption. In each experiment only one single parameter is varied and therefore the performance differences can only be attributed to the variation of this single parameter.

Parameter Values for ρ

To examine the influence of different values of the pheromone trail evaporation rate ρ, which determines the convergence speed of $\mathcal{MM}$AS towards good solutions, we present curves for the tradeoff between the average solution quality versus the number of tour constructions for the two symmetric TSP instances `kroA100` and `d198` and the two ATSP instances `kro124p` and `ftv170` using different settings of ρ. The maximum number of tour constructions is $2500 \cdot n$.

In Figure 4.1, it can be observed that for a low number of tour constructions, better tours are found when using lower values of ρ. This is due to the fact that for lower ρ the pheromone trails on arcs which are not reinforced decrease faster and, hence, the search concentrates earlier around the best tours seen so far. If ρ is high, too few iterations are performed to reach marked differences between the pheromone trails on arcs of high quality tours and arcs which are not used in the best tours. For a larger number of tour constructions, however, it pays off using higher ρ values, leading to a higher exploration of the search space in the beginning of the algorithm's run. Additionally, it is interesting to note that with more tour constructions the average performance increases generally for all values of ρ. This is mainly due to the effect of the lower trail limits (see also next section).

An indication of the performance of $\mathcal{MM}$AS can be given by the development of the average branching factor during the algorithm's run. In general, the best solutions are found when the average branching factor (see Section 4.1.1 on page 60) is rather low, that is, when the algorithm has almost converged. The exact average branching factor depends on the parameter λ. To eliminate the influence of different settings for λ we use a fixed value $\lambda = 0.05$. If only few iterations are performed and ρ is rather high, the branching factor is still much higher than 1 and the quality of the tours is still rather poor. In such a situation, more iterations are needed to achieve convergence of the algorithm. In fact, the average branching factor at which the best tours are found also depends on the problem size. For example, on instances `eil51`, `kroA100`, `lin318`, and `att532` the best solutions are found at values of $\bar{\lambda}$ corresponding to 1.31, 1.07, 1.04, and 1.004, respectively. Hence, the larger the problem size, the lower the

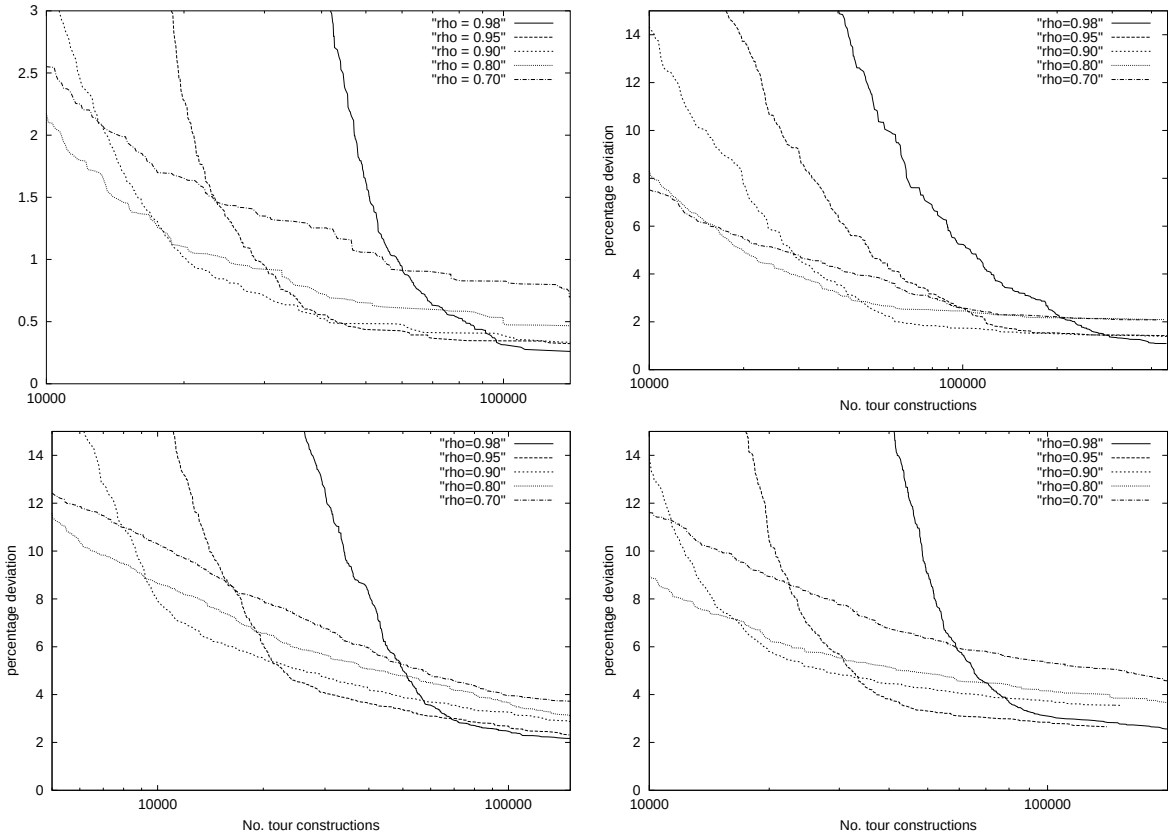

Figure 4.1: Influence of parameter ρ on the tradeoff between solution quality (given on y-axis) and the number of tour constructions (given on x-axis) on symmetric TSP instances `kroA100` (upper left side), `d198` (upper right side), and ATSP instances `kro124p` (lower left side), and `ftv170` (lower right side). Note the log-scale on the x-axis; the upper and leftmost parts of the curves were cut off to focus on the important details.

value of $\bar{\lambda}$ at which the best solutions are found. Actually, this observation can be interpreted as follows. The number of nodes at which the trails still have not converged, may be estimated by $(\bar{\lambda} - 1) \cdot n$. Hence, the best solutions are found if the algorithm has almost converged and the exploration of tours has concentrated around the best found tour.

Lower Pheromone Trail Limits

In this section we investigate the effectiveness of the lower trail limits. We compare experimental results obtained by systematically varying p_{best} (as proposed in Section 4.2.3) and without using lower pheromone trail limits ($\tau_{min} = 0$). As before, we allow maximally $2500 \cdot n$ tour constructions, which is sufficient to achieve convergence of $\mathcal{MM}$AS on every instance.

The average solution qualities obtained on four symmetric and four asymmetric TSP instances are given in Table 4.1. For *all* instances the average solution quality is always better if lower trail limits are used. It should be noted the relative difference between the average solution quality obtained with or without lower pheromone trail limits appears to increase with

Table 4.1: Computational results for systematically varying p_{best} and without lower pheromone trail limits ($\tau_{min} = 0$). Given are the average tour length, averaged over 25 trials (10 for `d198` and `lin318`), and in parenthesis the percentage deviation from the minimal tour length. Note that the smaller p_{best} the tighter are the trail limits. Results on symmetric TSP instances are given on the upper part, results on ATSP instances on the lower part of the table.

instance	$p_{best} = 0.0005$	$p_{best} = 0.005$	$p_{best} = 0.05$	$p_{best} = 0.5$	$\tau_{min} = 0$
`eil51`	428.5 (0.59%)	427.9 (0.46%)	427.8 (0.43%)	**427.7** (0.39%)	427.8 (0.43%)
`kroA100`	21344.8 (0.29%)	21352.8 (0.33%)	**21336.9** (0.26%)	21353.9 (0.34%)	21373.2 (0.43%)
`d198`	16024.9 (1.55%)	15973.2 (1.22%)	**15952.3** (1.09%)	16002.3 (1.41%)	16047.6 (1.70%)
`lin318`	42363.4 (0.80%)	**42295.7** (0.64%)	42346.6 (0.75%)	42423.0 (0.94%)	42631.8 (1.43%)
`ry48p`	14703.9 (1.96%)	14612.9 (1.32%)	14557.2 (0.94%)	**14555.4** (0.93%)	14556.5 (0.93%)
`ft70`	39332.8 (1.71%)	39222.2 (1.42%)	39191.8 (1.34%)	**39190.1** (1.34%)	39207.5 (1.38%)
`kroA124p`	37369.1 (3.14%)	37117.9 (2.45%)	**37003.2** (2.13%)	37155.9 (2.56%)	37159.4 (2.56%)
`ftv170`	**2824.0** (2.51%)	2828.9 (2.68%)	2825.3 (2.55%)	2834.1 (2.87%)	2840.0 (3.08%)

increasing instance size, which is most notably on the symmetric instances. Also, $\mathcal{MM}$AS's performance seems to be quite robust with respect to the particular value chosen for the pheromone trail limits (via p_{best}). Even when no lower trail limits are used, the results are quite good (compared, for example, with the results given for longer runs in Section 4.2.7 for other AS variants). Hence, facilitating a slow convergence by using high values for ρ, and introducing elitism by choosing the iteration-best ant for the pheromone update seems to be effective in practice. But as our results show, using additionally lower trail limits, the performance can be significantly improved. Hence, the use of the lower trail limits in $\mathcal{MM}$AS is definitely advantageous.

Pheromone Trail Initialization

As discussed before, in $\mathcal{MM}$AS the pheromone trails are initialized to their upper trail limit. To show the usefulness of the proposed trail initialization we compare it to a trail initialization at the lower pheromone trail limits; the computational results are given in Table 4.2. We find that with the proposed trail initialization for all instances, except for the smallest symmetric TSP instance, a better solution quality could be obtained; again the differences appear to increase with increasing instance size. Hence, the higher exploration of the search space achieved in this way seems to be important to achieve a better solution quality.

Global versus Iteration-Best Update

As argued before, updating the pheromone trails with the iteration-best solution s^{ib} may give advantages over using the global-best solution s^{gb}. Here, we compare these two choices by running the same experiments as previously, but always using s^{gb} for the pheromone trail update. Additionally, we investigated the influence of the lower pheromone trail limits by running each of the experiments with and without imposing lower pheromone trail limits.

The results are given in Table 4.3. The average performance when using the iteration-best ants for pheromone update are significantly better than using only the global-best ant. For example, on the symmetric TSP instances the worst solution obtained with the standard settings

Table 4.2: Computational results for pheromone initialization to the upper trail limit ($\tau(0) = \tau_{max}$) and to the lower trail limit ($\tau(0) = \tau_{min}$). Given are the average tour length, averaged over 25 trials (10 for `d198` and `lin318`), and in parenthesis the percentage deviation from the minimal tour length. The results for setting $\tau(0) = \tau_{max}$ are reproduced from the previous section. Results on symmetric TSP instances are given on the upper part, results on ATSP instances on the lower part of the table.

instance	$\tau(0) = \tau_{max}$	$\tau(0) = \tau_{min}$
`eil51`	427.8 (0.43%)	427.7 (0.39%)
`kroA100`	21336.9 (0.26%)	21362.3 (0.37%)
`d198`	15952.3 (1.09%)	16051.8 (1.72%)
`lin318`	42346.6 (0.75%)	42737.6 (1.68%)
`ry48p`	14557.2 (0.94%)	14625.9 (1.41%)
`ft70`	39191.8 (1.34%)	39221.23 (1.42%)
`kro124p`	37003.2 (2.13%)	37322.00 (3.01%)
`ftv170`	2824.9 (2.54%)	2868.8 (4.13%)

Table 4.3: Computational results for comparison of global-best update (s^{gb}) versus iteration-best update (s^{ib}) with and without using lower pheromone trail limits (indicated by either $+$limits or $-$no-limits). Given are the average tour length, averaged over 25 trials (10 for `d198` and `lin318`), and in parenthesis the percentage deviation from the minimal tour length. Results on symmetric TSP instances are given on the upper part, results on ATSP instances on the lower part of the table.

instance	s^{ib}+limits	s^{gb}+limits	s^{ib}−no-limits	s^{gb}−no-limits
`eil51`	427.8 (0.43%)	429.2 (0.75%)	427.8 (0.43%)	434.1 (1.89%)
`kroA100`	21336.9 (0.26%)	21417.1 (0.64%)	21373.2 (0.43%)	21814.7 (2.50%)
`d198`	15952.3 (1.09%)	16136.1 (2.26%)	16047.6 (1.70%)	16473.7 (4.40%)
`lin318`	42346.6 (0.75%)	42901.0 (2.08%)	42631.8 (1.43%)	44558.5 (6.02%)
`ry48p`	14557.2 (0.94%)	14615.4 (1.34%)	14556.5 (0.93%)	15046.1 (4.33%)
`ft70`	39191.8 (1.34%)	39384.3 (1.84%)	39207.5 (1.38%)	39895.3 (3.16%)
`kro124p`	37003.2 (2.13%)	37338.4 (3.06%)	37159.4 (2.57%)	39173.0 (8.12%)
`ftv170`	2824.9 (2.54%)	2869.9 (4.17%)	2840.00 (3.08%)	2892.8 (5.00%)

for $\mathcal{MM}$AS is better than the average solution quality when using s^{gb} with pheromone trail limits for all instances. In general, using exclusively s^{gb} for the pheromone trail update seems not to be a very good idea for $\mathcal{MM}$AS, yet, the lower pheromone trail limits help to significantly improve the performance when using s^{gb}.[5] Nevertheless, mixed strategies which sometimes use s^{gb} may be helpful to achieve better exploitation of the search results. Experiments on larger instances have shown that using such mixed strategies with a frequency of using s^{gb} increasing over time may yield a faster convergence of the algorithm and produce improved results.

[5] Additional experiments have shown that the results with s^{gb} update can be improved by using tighter pheromone limits, that is, by setting p_{best} to a lower value. Also, by using higher values for ρ improved results may be obtained. Yet, using only s^{gb} update never reached the solution quality obtained when using the iteration-best ants.

Elitism versus Socialism

Another question is whether in $\mathcal{MM}$AS only the best ant or all ants (something like socialism — all ants have the same right to update pheromone, let's call it the *socialist–$\mathcal{MAX}$–$\mathcal{MIN}$ ant system*) should be allowed to update the pheromones. Also intermediate solutions like allowing a number of the best ants to update the pheromones (like an oligarchy) are conceivable, but we did not consider this possibility here. The experimental results for socialist–$\mathcal{MAX}$–$\mathcal{MIN}$ ant system have shown that the parameter ρ should be chosen much smaller. The best performance was obtained with values of ρ around 0.35. Note, that socialist–$\mathcal{MAX}$–$\mathcal{MIN}$ ant system corresponds to ant system with additional upper and lower limits on the pheromone trail. Yet, for socialist–$\mathcal{MAX}$–$\mathcal{MIN}$ ant system the results (not presented here) are rather poor. They are roughly comparable to those of the basic version of ant system, which, obviously, is not astonishing. Thus, elitism is necessary to obtain good performance in the TSP domain.

4.2.6 Pheromone Trail Smoothing

So far we have shown that the pheromone trail limits are helpful for increase the solution quality obtained with $\mathcal{MM}$AS. An additional mechanism, called *pheromone trail smoothing*, may be useful to increase $\mathcal{MM}$AS performance and, more generally, the performance of any elitist versions of AS.

Pheromone trail smoothing, is proposed to increase the exploration of tours. Consider, for example, a situation where $\mathcal{MM}$AS has converged. In such a situation, the iteration-best tour corresponds with a high probability to the one which is obtained by choosing at every node the arc with maximal pheromone trail. Thus, only the arcs which are already at their maximally possible trail level would receive further reinforcement and the pheromone trail matrix does not change at all after the update. Yet, in general, the search progress in ACO algorithms is largest, if the relative pheromone trails on the arcs still change. Therefore, at convergence of $\mathcal{MM}$AS we increase the pheromone trail intensities according to one of the following two possibilities.

- *proportional update:* For each arc the trail intensity is increased proportionally to the difference between τ_{max} and the current trail intensity. The new trail intensity τ_{ij}^* is set to

$$\tau_{ij}^* = \tau_{ij}(t) + \delta \cdot (\tau_{max}(t) - \tau_{ij}(t)) \text{ with } 0 < \delta < 1$$

  .

- *lift-minimum:* In this case $\tau_{ij}(t)$ is set to

$$\tau_{ij}^* = \tau_{min}(t) + (\delta \cdot (\tau_{max}(t) - \tau_{min}(t)) \text{ with } 0 < \delta < 1$$

  .

The basic idea of pheromone trail smoothing is that by increasing the pheromone trail, either proportionally or up to a constant level, the probability distribution induced by Equation 4.1 is influenced biasing it towards cities that previously had small trail intensities. A special case of pheromone trail smoothing is given if $\delta = 1$: this would correspond to a reinitialization of the trails. In this case some information can still be kept by using the global-best solution occasionally to update the pheromone trails.

Pheromone trail smoothing is especially interesting if longer runs are allowed, because it helps to achieve a more efficient exploration of the search space. At the same time, pheromone trail smoothing makes $\mathcal{MM}$AS less sensitive to the particular choice of the lower pheromone trail limit.

4.2.7 Comparison of Ant Algorithms for Longer Runs

In this section we compare the performance of the proposed improvements over AS based on longer runs for symmetric and asymmetric TSP instances proposed for the first international contest on evolutionary optimization [23]. Note that the contest was won by a genetic local search algorithm incorporating the Lin-Kernighan local search algorithm to improve the tours [77]. The comparison allows the same number of tour constructions for all algorithms. The number of tours constructed is $k \cdot n \cdot 10000$, where $k = 1$ for symmetric TSPs and $k = 2$ for ATSPs, and n is the number of cities of an instance. For the two largest instances `att532` and `rat783` 10^6 tour constructions were allowed, the same as for ant colony system in [83].

We compare the performance of $\mathcal{MM}$AS to that obtained with ant colony system (ACS), elitist ant system (AS$_e$), and the rank-based version of ant system (AS$_{rank}$). For AS$_e$ and AS$_{rank}$ no results for applications to the proposed instances are available. Therefore, we coded both algorithms and applied them to instances with $n < 200$. The computational results for ACS are taken from [83], our own implementation of ACS confirms the results presented there.

For $\mathcal{MM}$AS the parameters were set to their default values as described before, except on the instances `att532` and `rat783` we used $\rho = 0.96$ and $m = n/2$ to force convergence within the given number of tour constructions and for ATSPs we set $\beta = 1$, which showed better results on some of the instances. In all runs at every 10th iteration the global-best tour is used to reinforce the trails. Additionally, we use the pheromone trail smoothing mechanism in $\mathcal{MM}$AS using the proportional-update with $\delta = 0.5$. This mechanism has been added in an ad-hoc manner without fine-tuning of the parameters. Pheromone trail smoothing is done if the average branching factor, which is measured every 100 iterations, is smaller than 1.05.

In AS$_e$, we used as default parameter settings $\alpha = 1.0$, $\beta = 5.0$ and the number of elitist ants is chosen as $e = n$. We found these parameter values to yield good performance in correspondence to the results reported in [36]. For AS$_{rank}$ we used the parameter settings proposed in [36], that is, $\eta = 5, \alpha = 1,$ and $e = 6$. To show that pheromone trail smoothing is a general mechanism which may improve the performance of ACO algorithms, we run both, AS$_e$ and AS$_{rank}$ with $\beta = 1.0$ for ATSP and $\beta = 2.0$ for symmetric TSP enhanced by pheromone trail smoothing.

The computational results in Table 4.4 for symmetric TSPs and Table 4.5 for ATSPs show that generally $\mathcal{MM}$AS achieves best performance. Also, on most instances the standard deviation of the tour length is smaller for $\mathcal{MM}$AS, which indicates a more reliable performance. Better average performance is only achieved on ATSP instance `ry48p` with AS$_{rank}$. Yet, AS$_{rank}$ was unable to find the optimal solution, which could be found by $\mathcal{MM}$AS and ACS. On the larger ATSP instances AS$_e$ and AS$_{rank}$ perform rather poorly compared to $\mathcal{MM}$AS or ACS. On problem instance `d198` the best solution found by ACS is slightly better than with $\mathcal{MM}$AS. But, in general, the average performance seems to be the more important performance measure. This is exemplified by the fact that on the two largest symmetric TSP instances, the average solution quality produced by $\mathcal{MM}$AS is even better than the best solution found by ACS.

Table 4.4: Computational results for symmetric TSPs instances from TSPLIB, details on the parameter settings are given in the text. "opt:" indicates the known optimal solution of each instance. All algorithms are allowed the same number of tour constructions. AS has only been applied to instances with $n \leq 100$, AS_{rank} and AS_e to instances with $n < 200$. Results for ACS are taken from [83]. n.a. indicates that an algorithm is not run on the given instance. For each instance are given the average tour length (first row), the percentage deviation from the optimum and the standard deviation of the average tour length (second row) and the best and worst result obtained (third row). Best average results among the algorithms are indicated in **boldface**. Averages are taken over 25 runs for $n < 200$, otherwise 10 runs were done. +ts indicates that pheromone trail smoothing is used by the algorithm.

algorithm	eil51 opt: 426	kroA100 opt: 21282	d198 opt: 15780	att532 opt: 27686	rat783 opt: 8806
$\mathcal{MM}$AS+ts	**427.1** 0.26%- 0.9 426 - 430	**21291.6** 0.05% - 26.7 21282 - 21379	**15956.8** 1.12% - 14.9 15940 - 15982	**28112.6** 1.54% - 148.3 27971 - 28510	**8951.5** 1.65% - 18.4 8920 - 8986
ACS	428.1 0.49%- 2.8 426 - n.a.	21420.0 0.64% - 141.7 21282 - n.a.	16054 1.74% - 71.1 15888 - n.a.	28522.8 3.02% - 275.7 28147 - n.a.	9066.0 2.95% - 28.2 9015 - n.a.
AS_{rank}	434.48 1.99% - 4.8 427 - 445	21746.40 2.18% - 146.1 21429 - 22340	16199.09 2.65% - 110.5 15981 - 16292	n.a.	n.a.
AS_{rank}+ts	428.76 0.648% - 2.99 426 - 435	21394.88 0.53% - 146.07 21282 - 21674	16025.2 1.55% - 67.5 15942 - 16115	n.a.	n.a.
AS_e	428.30 0.54% - 2.21 426 - 434	21522.83 1.13% - 130.78 21320 - 21808	16205.00 2.69% - 96.25 16058 - 16417	n.a.	n.a.
AS_e+ts	427.40 0.33% - 1.43 426 - 431	21431.92 0.70% - 116.62 21282 - 21673	16140.79 2.29% - 151.13 16103 - 16437	n.a.	n.a.
AS	437.3 2.65% - 2.59 435 - 443	22471.4 5.59% - 57.8 22411 - 22560	n.a.	n.a.	n.a.

The experimental results also show the usefulness of the pheromone trail smoothing mechanism. While the performance of AS_e and AS_{rank} without pheromone trail smoothing generally is significantly worse than that of $\mathcal{MM}$AS and ACS, with pheromone trail smoothing both algorithms catch up (with the only exception being AS_{rank} on the ATSP instances) and roughly reach the solution quality obtained with ACS on the symmetric TSPs, but still are worse on most ATSP instances. Thus, the pheromone trail smoothing mechanism may be very useful to increase performance. The pheromone trail smoothing mechanism helps to improve slightly the performance of $\mathcal{MM}$AS, too. Yet, the results would have been the best on most instances even without pheromone trail smoothing. For example, with the same number of tour constructions on `d198` an average solution quality of 1.22% above the optimal solution is obtained, only slightly worse than the result given in Table 4.4. Similarly, with only $2500 \cdot n$ tour constructions, on `eil51` and `kroA100` a solution quality of 0.48% and 0.26% are obtained, better than most of the results achieved with the other variants with many more tour constructions. Regarding AS, it can be observed that AS performs very poorly compared to the other algorithms.

Table 4.5: Computational results for ATSP benchmark instances from TSPLIB. The table entries are explained in the caption of Table 4.1.

algorithm	ry48p opt: 14422	ft70 opt: 38673	kro124p opt: 36230	ftv170 opt: 2755
$\mathcal{MM}$AS	14523.4 0.70% - 96.7 14422 - 14693	**38922.7** 0.65% - 201.5 38690 - 39436	**36573.6** 0.97% - 272.1 36230 - 37443	**2817.7** 2.27% - 13.9 2761 - 2896
ACS	14565.4 0.99% - 115.2 14422 - n.a.	39099.0 1.10% - 170.3 38781 - n.a.	36857.0 1.73% - 521.9 36241 - n.a.	2826.5 2.59% - 33.8 2774 - n.a.
AS_{rank}	**14511.4** 0.62% - 24.6 14459 - 14570	39410.10 1.91% - 186.41 39019 - 39634	36973.50 2.05% - 484.83 36445 - 37960	2854.20 3.60% - 30.72 2820 - 2915
AS_{rank}+ts	14644.56 1.54% - 96.64 14507 - 14850	39199.24 1.36% - 197.06 38837 - 39540	37218.0 2.73% - 458.3 36502 - 38107	2915.6 5.83% - 52.0 2794 - 2969
AS_e	14685.2 1.82% - 148.4 14495 - 15123	39261.8 1.52% - 176.9 38820 - 39644	37510.2 3.53% - 387.1 36928 - 38282	2952.4 7.14% - 93.8 2845 - 3156
AS_e+ts	14657.9 1.63% - 159.5 14446 - 14949	39161.04 1.26% - 263.9 38719 - 39667	37417.7 3.28% - 838.1 36416 - 38759	2908.1 5.56% - 61.5 2822 - 2999
AS	15296.4 6.06% - 92.65 15125 - 15397	39596.3 2.39% - 94.1 39408 - 39740	38733.1 6.91% - 229.7 38315 - 39024	3154.5 14.50% - 34.2 3075 - 3188

In [66] it was shown that ACS is competitive with other nature-inspired algorithms applied to the TSP. Since $\mathcal{MM}$AS reaches a better solution quality than ACS on most instances, the computational results also show the competitive performance of $\mathcal{MM}$AS when compared to other improvements on AS as well as other nature-inspired algorithms. Yet, to obtain results competitive with the best performing algorithms for the TSP, local search has to be added to improve solutions. This will be the topic of the next section.

4.3 $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System and Local Search for the TSP

Several extensive computational studies for the TSP have shown that constructive algorithms usually return solutions of rather poor quality when compared to local search algorithms [21, 211, 129]. Yet, it has been noted that repeating local search from randomly generated initial solutions still results in a considerable gap to the optimal solution for most problems. Hence, to find very high quality solutions, some hybrid technique is needed which exploits the previously found local minima to guide the search. A popular idea for the design of hybrid algorithms is to combine a probabilistic, adaptive construction heuristic with local search [27, 73]. ACO algorithms, in general, are such constructive heuristics. They can be used to identify promising regions of the search space with high quality solutions, using the pheromone trails as an adaptive memory of solution components which have been part of the best local minima found so far. These solution components are combined in subsequent iterations by a probabilistic construc-

tion mechanism which is biased by the pheromone trails and a local heuristic function. Another advantage of using such a constructive algorithm like $\mathcal{MM}$AS is that by generating good initial solutions, the subsequent local search needs less steps to reach a local optimum. Thus, for a given time limit more often a local search can be applied than by starting from randomly generated solutions. In this section we investigate a hybrid between $\mathcal{MM}$AS and local search. We choose $\mathcal{MM}$AS, because it was shown in the previous section to be the best performing ACO algorithm on a wide variety of instances.

The idea of using a mechanism to generate initial solutions which are improved by a subsequent local search, is also applied in other metaheuristics like iterated local search and genetic algorithms. Differing from ACO algorithms, in these metaheuristics the initial solutions are not generated from scratch, but are obtained by modifications of a previously found locally optimal solution or by combining two solutions via a crossover operator, respectively. Computational results for both kinds of heuristics have shown that such hybrid algorithms may be very effective for obtaining near optimal solutions for the TSP in reasonable computation times [169, 186, 251, 129, 174].

Important for the design of such hybrid algorithms is the interaction between the different parts, here specifically the ants and the local search. We first investigate which is the best way of adding local search algorithms to $\mathcal{MM}$AS. As we will see, the best strategy is to use a constant number of ants and to increase the frequency of choosing the global-best ant for the pheromone trail update. We also show that the necessary computation time for finding very high quality solutions can be significantly reduced if the number of ants which apply local search after each iteration increases gradually. We end this section by an extensive experimental study of $\mathcal{MM}$AS, showing that $\mathcal{MM}$AS is currently the best performing ACO algorithm for the TSP.

4.3.1 Local Search for the TSP

The most widely known iterative improvement algorithms for the TSP are certainly `2-opt` [55] and `3-opt` [155]. They proceed by systematically testing whether the current tour can be improved by replacing 2 or at most 3 edges, respectively. Both local search algorithms are widely studied in the literature [211, 129] and have been shown empirically to yield good solution quality. Depending on the initial tour chosen, `2-opt` averages between 8% to 15% and `3-opt` 3.5% to 8% percent above the optimal solution for TSPLIB instances [211]. Local search algorithms using $k > 3$ edges to be exchanged are not used commonly, due to the high computation times involved and the low additional increase in solution quality.

A straightforward implementation for `2-opt` and `3-opt` would require to examine $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ exchanges, respectively. This is clearly infeasible when trying to solve instances with several hundreds of cities in reasonable time. Fortunately, there exist quite a few speed-up techniques [21, 211, 129] achieving, in practice, run-times which grow sub-quadratically. This effect is obtained by examining only a small part of the whole neighborhood.

We use three techniques to reduce the run-time of our `2-opt` and `3-opt` implementations. One consists in restricting the set of moves which are examined to those contained in a candidate list of the nearest neighbors ordered according to non-decreasing distances [21, 156, 211]. Using candidate lists, for a given city c_i we only consider moves which add a new edge between c_i and one of the nodes in the candidate list. Hence, by using a neighborhood list of bounded

length, an improving move can be found in constant time. An additional speed-up is achieved by performing a fixed radius nearest neighbor search [21]. For `2-opt` at least one newly introduced edge has to be shorter than any of the two removed edges. Let $\langle c_1, c_2, c_3, c_4 \rangle$ be the nodes involved in a move, then for an improving move it has to hold that either $d(c_1, c_2) > d(c_1, c_3)$ or $d(c_3, c_4) > d(c_4, c_2)$ or both. Due to symmetry reasons it suffices to check only the first condition. A similar argument also holds for `3-opt` [21]. To yield a further reduction in run-time we use an array of *don't look bits*; one don't look bit is associated with each node. Initially, all don't look bits are turned off (set to 0). If for a node no improving move can be found the don't look bit is turned on (set to 1) and the node is not considered as a starting node for finding an improving move in the next iteration. If an edge incident to a node is changed by a move, the node's don't look bit is turned off again. The effect of the optimizations described above is the reduction of the complexity of a single iteration of `2-opt` or `3-opt` to $\mathcal{O}(n)$.

For asymmetric TSPs, `2-opt` is not directly applicable because one part of the tour has to be traversed in the opposite direction and the length of this partial tour would have been to be computed from scratch, leading to high run-times. Yet, one of the three possible `3-opt` moves does not lead to a reversal of a subtour. We call this special form of `3-opt` `reduced 3-opt`. The implementation of `reduced 3-opt` uses the same speed-up techniques as described before.

The local search algorithm producing the highest quality solutions for symmetric TSPs is the Lin-Kernighan heuristic (`LK`) [156]. The Lin-Kernighan heuristic considers in each step a variable number k of edges to be exchanged. For the benchmark instances from TSPLIB, the average deviation from the optimal solution is between 1–3% for most instances. Thus, improved solution quality should be obtained in $\mathcal{MM}$AS when using the `LK` heuristic instead of `2-opt` or `3-opt`. Yet, a disadvantage of the `LK` heuristic is that its implementation is much more difficult than that of `2-opt` and `3-opt`, and careful fine-tuning is necessary to get it to run fast and produce high quality solutions [211, 129, 179]. We used an implementation of the `LK` heuristic that has been provided by Olivier Martin. We will give a short indication of the performance obtainable when using $\mathcal{MM}$AS in combination with the `LK` heuristic in the local search phase.

4.3.2 Fastening the Pheromone Update

When applying ant algorithms to the TSP, the pheromone trail matrix stores n^2 entries. All the entries are updated due to the trail evaporation implemented by Equation 4.2. Obviously, this is a very expensive operation when applying $\mathcal{MM}$AS to large instances (the update of the whole trail matrix is not done in ACS). To avoid this, we apply the pheromone evaporation only to edges connecting a city c_i to cities of c_i's candidate list and, hence, the pheromone trails can be updated in $\mathcal{O}(n)$. The pheromone trail on edges leading to cities not contained in the candidate list, is initialized to some very low value. If during the solution construction all nodes of a cities' candidate list have already been visited, we choose the node with maximal value of $\tau_{ij} \cdot \eta_{ij}^{\beta}$. The nodes not contained in the candidate set are still affected by the pheromone trail update.

4.3.3 How to Add Local Search to $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ?

For the design of hybrid algorithms the most important question is how the two parts are best combined, that is, in our case this concerns the question how local search is best added to $\mathcal{MM}$AS. In particular, we have to address the question how many ants should be used, which ants should apply local search and which ants should update the pheromone trails. Note that when applying $\mathcal{MM}$AS to the TSP without considering local search, the best results are obtained by increasing the number of ants proportionally to the instance size. Yet, this choice need not be the best when additionally applying local search. Regarding the number of ants we investigated two possible settings for the number of ants. In one variant, denoted as `10+ls`, we use 10 ants and every ant applies local search to its tour. In another version, denoted as `prop+ls`, we increase the number of ants proportionally to the problem size using $n/3$ ants, and only allow the iteration-best ant to apply local search; if in `prop+ls` all ants would apply local search, the computation times would grow too high. Additionally, we investigate whether it is best to use the iteration-best ant to update the pheromone trails or if it is better to use the global-best ant with a high frequency. Therefore, two additional series of experiments are performed. In a first experiment only the iteration-best ant is chosen, denoted by the addition of `-ib`, (for example, `10+ls-ib` denotes the variant which uses 10 ants and only allows the iteration-best ant to update the pheromone trails); in a second experiment at every second iteration the global-best ant reinforces the trails (denoted by the addition of `-gb`). Hence, in total four series of experiments are run with the variants `prop+ls-ib`, `prop+ls-gb`, `10+ls-ib`, and `10+ls-gb`. In some preliminary experiments appropriate parameter values for ρ were determined. We found that values slightly lower than those used without local search provide better solutions. In particular, for `10+ls` we set ρ to 0.8 and for `prop+ls` we set ρ to 0.95 for instances with less than 400 cities and otherwise we set $\rho = 0.925$.

In the experiments we use the step count defined in the first ICEO (see Section 4.2.7) as the stopping criterion to make the results of $\mathcal{MM}$AS with local search comparable to those without local search. For the smaller instances with less than 400 nodes we use $1/4$th, for the larger symmetric TSP only $1/10$th of the maximal number of steps. For symmetric TSPs, we use a `2-opt` local search with a best-improvement pivoting rule and a fixed radius nearest neighbor search. Every tour construction and every improving move found by `2-opt` is counted as one step. Thus, one `2-opt` application typically counts more than one step, unless the initial solution is already a local minimum. When applying $\mathcal{MM}$AS to ATSPs, we count every iteration of `reduced 3-opt`, which uses a first-improvement pivoting rule, as n steps.

The results are given in Table 4.6 for the variants `prop+ls-ib` (left side) and `prop+ls -ib` (right side) and in Table 4.7 for variants `10+ls-ib` (left side) and `10+ls-gb` (right side). When comparing the results for all the four variants, the overall best performance is obtained by `10+ls-gb`. Using variant `10+ls-gb`, the average excess over the optimal solutions for all the tested instances is 0.29% for symmetric TSPs, and 0.046% on the asymmetric TSPs, while for variant `10+ls-ib` the average excess for symmetric TSPs is worse (0.59%), but slightly better for asymmetric TSPs (0.036%). `prop+ls` gives worse results than `10+ls` on almost all problems, the only exception being the computational results on instance `rat783` when comparing to variant `10+ls-ib`. In general, using occasionally the global-best update seems to increase overall performance. Having a closer look at the results in Table 4.7, one might observe that `10+ls-ib` is slightly better for smaller instances up to 442 cities, but significantly worse on the larger ones. An examination of the results showed that if only the iteration-best

Table 4.6: Results for symmetric and asymmetric TSPs with $\mathcal{MM}$AS+ls. Left side: results with pheromone update by the iteration-best ant version. Right side: results if every second iteration the global-best tour adds pheromone. Averages are taken over 10 runs. Given are the best solution found, the average tour length (the percentage deviation from the optimum is given in parenthesis) and the standard deviation of the found tour lengths.

Problem	prop+ls-ib		
	best	average (% dev)	σ
d198	15806	15821.2 (0.26%)	12.7
lin318	42155	42159.0 (0.31%)	8.4
fl417	11874	11884.8 (0.20%)	8.4
pcb442	51148	51189.2 (0.81%)	24.7
d493	35476	35645.5 (1.84%)	73.4
att532	27841	27895.2 (0.76%)	52.0
u574	37310	37500.6 (1.61%)	144.5
rat783	8841	8950.5 (1.64%)	70.9
p43	5620	5620.0 (0.0%)	0.0
ry48p	14422	14429.6 (0.30%)	43.75
ft70	38673	38684.2 (0.029%)	18.8
kro124p	36230	36242.8 (0.035%)	32.8
ftv170	2758	2782.4 (0.99%)	12.7

Problem	prop+ls-ib		
	best	average (% dev)	σ
d198	15780	15791.9 (0.075%)	6.9
lin318	42029	42130.2 (0.24%)	60.5
fl417	11865	11873.0 (0.20%)	6.7
pcb442	50956	51133.0 (0.70%)	95.7
d493	35212	35358.5 (1.02%)	66.4
att532	27716	27829.1 (0.52%)	68.9
u574	37092	37254.3 (0.95%)	114.4
rat783	8841	8884.3 (0.89%)	22.9
p43	5620	5620.0 (0.0%)	0.0
ry48p	14422	14422.0 (0.0%)	0.0
ft70	38673	38679.5 (0.020%)	19.4
kro124p	36230	36237.6 (0.021%)	31.1
ftv170	2755	2765.5 (0.38%)	10.1

tour updates the pheromones, very good solutions are found in some runs, whereas in others the solution quality is rather poor. At the same time, in the runs which gave only poor solution quality, the average branching $\bar{\lambda}$ at the end of a run are still rather high, indicating that the run has not converged. Hence, by choosing the global-best ant for the pheromone update a faster convergence of the algorithm is forced.

A further inspection of the experimental results revealed that prop+ls performs rather poorly, because only the iteration-best ant applies local search. In later stages of the search the iteration-best solution will often be already locally optimal and cannot be further improved by a local search. Hence, we suspected that improved performance is obtained by increasing the number of ants which apply local search. Following this idea, we increased this number depending on $\bar{\lambda}$ and performed some experiments on symmetric TSP instances with the following scheme. If $\bar{\lambda} > 2.0$, only the iteration-best ant applies local search. If $\bar{\lambda} < 2.0$ we set the number of ants to perform local search as $(1 - (\bar{\lambda} - 1)^{0.5}) \cdot m + 1$. Hence, the smaller $\bar{\lambda}$ the more ants apply local search. With this scheme, we obtained better results, but still could not reach the performance of 10+ls-gb. Hence, for the following we will only focus on $\mathcal{MM}$AS hybrids using a constant number of ants.

The computational results presented in this section also suggest that, especially on smaller instances, using only the iteration-best ant for the pheromone trail update may be advantageous. Yet, for larger instances occasional update with the global-best ant gives better solution quality. As discussed in Section 4.2.2, an important aspect of exploitation of the search experience in $\mathcal{MM}$AS is given by allowing only one ant to update the pheromone trails. Yet, updating the pheromone trails only with the global-best solution, would concentrate the search very early around that solution. Using the iteration-best solution for the pheromone trail update helps to better explore which solution components (edges) are contained in high quality solutions and

Table 4.7: Results for symmetric and asymmetric TSPs for `10+all-ls`. Left side: results for the iteration-best version. Right side: results if every second iteration the global-best tour adds pheromone. Averages are taken over 10 runs. Given are the best solution found, the average tour length (the percentage deviation from the optimum is given in parenthesis) and the standard deviation of the found tour lengths.

Problem	`10+ls-ib`			Problem	`10-ls-gb`		
	best	average (% dev)	σ		best	average (% dev)	σ
d198	15780	15780.8 (0.005%)	1.8	d198	15780	15782.5 (0.016%)	4.7
lin318	42029	42044.5 (0.037%)	26.3	lin318	42029	42079.3 (0.12%)	58.1
fl417	11861	11861 (0.0%)	0	fl417	11861	11864.5 (0.029%)	6.7
pcb442	50935	51070.6 (0.57%)	78.9	pcb442	50938	51091.7 (0.62%)	87.7
d493	35168	35200.5 (0.57%)	19.2	d493	35043	35107.7 (0.30%)	34.5
att532	27714	27825.4 (0.50%)	22.2	att532	27722	27744.4 (0.21%)	16.5
u574	36952	37189.1 (0.77%)	200.1	u574	36938	37043.8 (0.37%)	84.8
rat783	8828	9005.1 (2.26%)	65.1	rat783	8822	8851.4 (0.52%)	16.8
p43	5620	5620.0 (0.0%)	0.0	p43	5620	5620.0 (0.0%)	0.0
ry48p	14422	14422.3 (0.0%)	1.4	ry48p	14422	14422.0 (0.0%)	0.0
ft70	38673	38675.6 (0.0%)	8.4	ft70	38673	38673 (0.0%)	0.0
kro124p	36230	36230 (0.0%)	0.0	kro124p	36230	36236.7 (0.018%)	31.2
ftv170	2755	2760.0 (0.18%)	6.7	ftv170	2755	2760.8 (0.21%)	6.5

helps to avoid too early convergence of the algorithm (see also computational results in Section 4.2.5). Only later in the search more emphasis should be given to the best solution found so far. To achieve this goal, we increase the frequency of updating the pheromone trails with the global-best ant according to a specific schedule. Let u_{gb} indicate that every u_{gb} iterations the global-best ant is used to update the pheromone trails and let i be the iteration counter of the algorithm. Then, we apply the following schedule for u_{gb} in the following sections.

$$u_{best} = \begin{cases} \infty & \text{if } i \leq 25 \\ 5 & \text{if } 26 \leq i \leq 75 \\ 3 & \text{if } 76 \leq i \leq 125 \\ 2 & \text{if } 126 \leq i \leq 250 \\ 1 & \text{otherwise} \end{cases} \tag{4.16}$$

The following experiments all use a `3-opt` local search algorithm which uses a fixed radius nearest neighbor search within a candidate list of size 40 (chosen according to recommendations of Johnson and McGeoch [129]) and don't look bits. The don't look bits are all turned off at the start of the local search. The local search algorithms are always used using a first-improvement pivoting rule.

4.3.4 Adapting the Number of Ants which Perform Local Search

As shown in the preceding section, a constant number of ants with occasional global-best update seems to be best for $\mathcal{MM}$AS applications to the TSP. Here we show that by gradually increasing the number of ants which apply local search after each iteration a very good compromise between solution quality and computational speed is obtained. In particular, we perform experiments in which we start with a fixed number of ten ants and first allow only one of the ants, the iteration-best ant, to apply local search. The number of ants which apply local search is then successively increased by one after every `incr-ls` iterations. Hence, with the parameter `incr-ls` the number of iterations which can be performed in a given amount of time can be influenced; the lower `incr-ls` the more ants apply local search after a given number of iterations, resulting in higher computation times per iteration.

In Figure 4.2 we compare different settings of `incr-ls` by plots of the average tour length versus the CPU-time on four symmetric TSP instances. The plots use values for `incr-ls` taken from $\{0, 5, 10, 25, 50, 100\}$. If `incr-ls` is zero, all ants are allowed to perform local search from the first iteration on. The development of the average solution quality in Figure 4.2 shows that for short CPU-times the smaller the value of `incr-ls`, the worse the solution quality obtained (the differences between the `incr-ls` settings are generally more notable with increasing instance size). Yet, with increasing run-time, the curves cross and the best tour length is typically obtained for low values of `incr-ls`. In general, a value for `incr-ls` of 25 seems to give a reasonable compromise between the solution quality obtained versus the run-time. This result is a consequence of the fact that in early iterations the local search starts from rather bad initial solutions, taking a longer time to find local minima. Additionally, the number of iterations to identify promising search space regions is too limited. Yet, after longer run-time the best ant may often have an initial solution that is very close to the previously best one and the local search is more likely to end again at the previously best found solution.

In summary, by adapting the number of ants which apply local search a good compromise between convergence speed and solution quality can be obtained.

4.3.5 Modified $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

Another possibility to reduce the run-time necessary for finding high quality solutions is to exploit more strongly the best solutions found during the search. In fact, the pseudo-random proportional rule given in Equation 4.5, which was proposed for ant colony system (ACS), uses a stronger search exploitation, because with a probability of p_0 the best possible choice as indicated by the heuristic information and the pheromone trails is made. Additionally, when using a large value of p_0, in the first iterations the constructed tours are close to the nearest neighbor tour for the TSP which, generally, is significantly better than the tours constructed by $\mathcal{MM}$AS in the initial search phase. Additionally, better initial tours usually lead to better locally optimal tours for the TSP. Thus, especially in the initial search phase we expect that ACS finds better solutions than $\mathcal{MM}$AS.

The pseudo-random proportional rule can also be used during the solution construction of $\mathcal{MM}$AS. When using large values of p_0, the trail limits should be tighter to facilitate exploration. The reason for doing so is that with large p_0 the best deterministic choice is almost always taken. If additionally we use rather wide intervals for the possible pheromone trail, it may

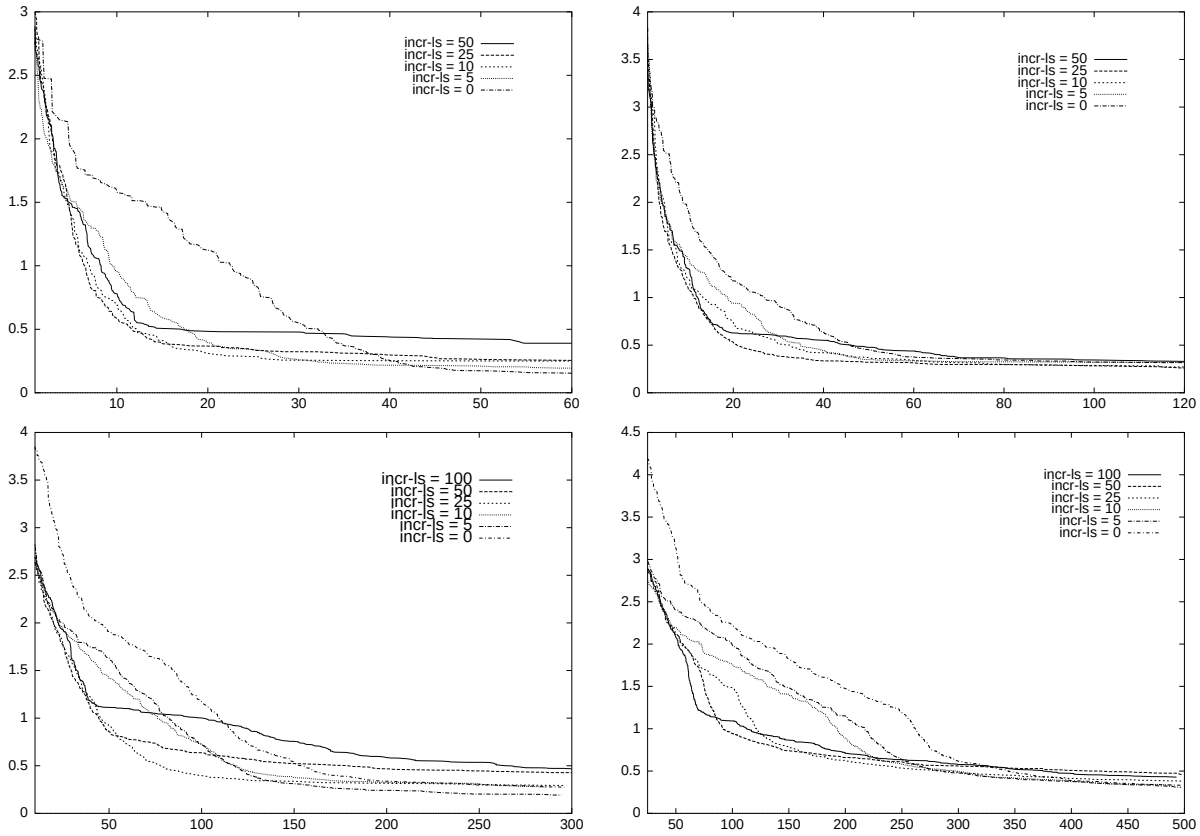

Figure 4.2: Given is the development of the average percentage deviation from the optimum (averages are taken over 25 trials) on the y-axis in dependence on the computation time in seconds (x-axis) for different values of `incr-ls`. Given are the experimental results for `lin318` (upper left side), `pcb442` (upper right side), `rat783` (lower left side), and `pcb1173` (lower right side). See the text for a more detailed description.

happen that also in the probabilistic selection almost always a city with largest $\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}$ is chosen leading to an insufficient exploration of new tours. Hence, tighter pheromone trail limits should be used to provide a balance between exploitation and exploration. To determine the tightness of the trail limits, we introduce a parameter *min-factor* such that $\tau_{min} = \tau_{max}/$*min-factor*; τ_{max} is again chosen as in the standard $\mathcal{MM}$AS version.

In principle, many combinations of possible settings for p_0 and appropriate trail limits could be tested experimentally. Yet, when using the pseudo-random proportional rule we are interested in obtaining a very aggressive algorithm which early concentrates around the best tours found. Hence, we set p_0 to some high value and, in particular, use $p_0 = (n -$ *explor*$/n)$, where *explor* is set to 15 in all experiments. *explor* corresponds to the average number of cities at which a biased exploration takes place. This last parameter setting is motivated by the observation that the number of randomized choices should be constant with respect to instance size. Similarly, in iterated local search applications to the TSP, the kick-move introduces a fixed number of modifications to the current tour, which shows very good performance, independent of instance size (see Section 5.2 for details on ILS algorithms for the TSP).

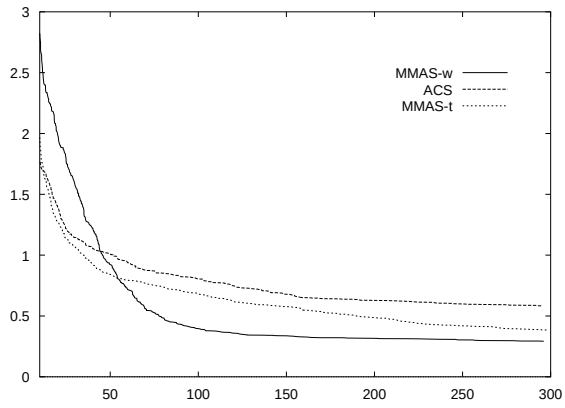

Figure 4.3: Comparison of the development of the average percentage deviation from the optimal tour length (averages are taken over 25 trials) on the y-axis in dependence of the computation time in seconds (x-axis) for $\mathcal{MM}$AS-w, $\mathcal{MM}$AS-t, and ACS for TSPLIB instance `rat783`.

In Figure 4.3 we compare the solution quality development of two variants of $\mathcal{MM}$AS and of ACS. We denote the $\mathcal{MM}$AS version using tight trail limits with $\mathcal{MM}$AS-t (t for tight pheromone trail limits), and the standard version is called $\mathcal{MM}$AS-w (w for wide pheromone trail limits). For $\mathcal{MM}$AS-w we use a setting of `incr-ls` to 25, which was found to be good in the previous section.

The development of the solution quality is very similar for ACS and $\mathcal{MM}$AS-t. For short run times, up to roughly 50 seconds in our example, ACS and $\mathcal{MM}$AS-t obtain better solution quality than $\mathcal{MM}$AS-w. Yet, for higher run-times $\mathcal{MM}$AS-w catches up and finally obtains a significantly better solution quality than either ACS or $\mathcal{MM}$AS-t. The general shape of the solution quality development is also typical for other instances we tested. The fact that $\mathcal{MM}$AS-w yields, in general, higher solution quality is also emphasized by the extensive computational comparison among the different variants of $\mathcal{MM}$AS and ACS in the next section.

4.3.6 Experimental Results

In this section we present an extensive experimental comparison of $\mathcal{MM}$AS-w and $\mathcal{MM}$AS-t and compare their performance to ant colony system (ACS) and to an iterated local search (ILS) algorithm.

Goals of the Experiments

The experimental comparison tries to answer the following three main questions.

(1) How do the variants of $\mathcal{MM}$AS compare to other ACO algorithms and ILS?

(2) Does the pheromone trail smoothing mechanism improve the solution quality obtainable with $\mathcal{MM}$AS-w?

(3) Is the heuristic information necessary to obtain high quality solutions?

To answer the first question, we ran our implementation of ACS and an iterated `3-opt` algorithm. We have chosen ACS, because some preliminary experiments have shown that, when combined with local search, ACS performs better than either elitist ant system or the rank-based version of ant system. Independently from our work, ACS has also been improved by adding local search [66] where also a `3-opt` local search algorithm with the same speed-up techniques as in our implementation is used [66]. Additionally, we applied an iterated local search algorithm with the same `3-opt` local search as for $\mathcal{MM}$AS and ACS. The ILS algorithm always applies the kick-move to the best solution found during the search (see Chapter 5 for a detailed description of the algorithm). Because ILS is considered to be one of the best algorithms for the TSP, this comparison gives a very good indication of the performance of the ant-based algorithms with respect to other algorithms. In fact, the iterated-`LK` algorithm [169, 126, 129], which is the same algorithm as iterated `3-opt` except that it uses the more sophisticated `LK` heuristic for the local search, is one of the best algorithm known for the TSP. Yet, the local search algorithm used has a major influence on the final performance and also ant-based algorithms could be improved using the `LK` heuristic. To keep the comparison fair, we use for all algorithms the same local search method.

With respect to the second question, we used $\mathcal{MM}$AS-w once with and once without pheromone trail smoothing (see Section 4.2.6). In the hybrid algorithm, in fact, we re-initialize the pheromone trails on all arcs to τ_{max} (this would correspond to setting the parameter δ, defined in Section 4.2.6, to 1). The trails are re-initialized if the average branching factor $\bar{\lambda}$ indicates that $\mathcal{MM}$AS is near convergence; in particular, we re-initialize the pheromone trails if $\bar{\lambda} \leq 1.0001$. Since $\bar{\lambda}$ does not change much from iteration to iteration, we compute $\bar{\lambda}$ only every 100 iterations. After the re-initialization of the pheromone trails, the schedule given by Formula 4.16 is followed again from scratch.

If no local search is used to improve solutions, it has been shown for AS [68] and ACS [66] that the use of the heuristic information significantly improves performance because it is necessary to direct the search towards shorter edges. Yet, in the hybrid algorithm, the solutions are improved by local search; by reinforcing the pheromone trails with locally optimal solutions an indication is given which edges should preferably be contained in the best solutions. Hence, to answer this third question, we run each of the three ACO algorithms with and without using heuristic information.

In total we have run eight algorithms and algorithm variants, respectively. These are $\mathcal{MM}$AS-w with (referred to as $\mathcal{MM}$AS-w) and without using heuristic information ($\mathcal{MM}$AS-wnh), $\mathcal{MM}$AS-w without pheromone trail smoothing ($\mathcal{MM}$AS-wnts), $\mathcal{MM}$AS-t with ($\mathcal{MM}$AS-t) and without heuristic information ($\mathcal{MM}$AS-tnh), ACS with (ACS) and without heuristic information (ACS-nh), and an iterated local search algorithm (ILS-`3-opt`). In some preliminary experiments we determined appropriate parameter settings for $\mathcal{MM}$AS. For $\mathcal{MM}$AS-w, and $\mathcal{MM}$AS-wnts we use $\alpha = 1$, $\beta = 2$, $m = 25$, $\tau_{min} = \tau_{max}/2 \cdot n$; $\mathcal{MM}$AS-wnh uses the same parameter settings except that $\beta = 0$. The choice of τ_{min} would correspond to $p_{best} \approx 0.0075$ which is lower compared to the value chosen without using local search. The main reason for setting p_{best} to some lower value (this results in tighter pheromone trail limits) is to generate more diverse tours for the local search algorithm. In $\mathcal{MM}$AS-t and $\mathcal{MM}$AS-tnh we set $m = 10$, and $\tau_{min} = \tau_{max}/3$. The other parameters are the same as for $\mathcal{MM}$AS-w except that always the global-best ant updates the pheromones and no pheromone trail smoothing is used. (In $\mathcal{MM}$AS-t we do not follow the schedule of Equation 4.16.) In aggressive searches like in

ACS and $\mathcal{MM}$AS-t, some preliminary experiments confirmed that it is better to use always the global-best ant for the pheromone update. When running ACS, we use the same parameter settings as those proposed for ACS-`3-opt` in [66]. The iterated local search algorithm (ILS-`3-opt`) is described in more detail in Section 5.2. ILS-`3-opt` always applies the kick-move to the best solution found so far. The kick-move is a specific 4-opt move, called *double-bridge* move, which has the property that it is the smallest change involving four edges which cannot be reversed in a single step by the `LK` heuristic.

The experimental comparison is based on a time equalized basis; the time limits have been increased with increasing instance size. The computation times have been chosen high enough so that a further significant improvement of the solution quality for the applied algorithms is not expected. The runs are performed on a Sun UltraSparc II Workstation with two UltraSparc I 167MHz processors with 0.5MB external cache. Due to the sequential implementation of the algorithm only one processor is used.

Experimental Results on Symmetric TSPs

We used the five smallest instances proposed for the 2nd ICEO'97 for our comparison. These instances are also used by many other authors [66, 102, 174, 181, 257, 238, 187]. Additionally, we have run four algorithms on some larger instances with up to 1577 cities.

The computational results are given in Tables 4.8 and 4.9; we will use the performance of $\mathcal{MM}$AS-w as the basis for the following discussion.

Comparison of the algorithms The computational results show that $\mathcal{MM}$AS and ACS are competitive with the ILS algorithm. While on most instances the variants of $\mathcal{MM}$AS achieve better average solution quality than ILS (the differences are least notable on instances `d198` and `pcb442`), ACS achieves a very similar final solution quality to ILS-`3-opt`. Among the variants of $\mathcal{MM}$AS, those using wide trail limits seem to perform better than the versions using tight trail limits. While this fact is not very noticeable on the three smallest instances, the differences become more visible for the two largest ones (see Table 4.8). Additionally, $\mathcal{MM}$AS-w is the only algorithm which was able to solve instance `lin318` in all runs to optimality[6] and this variant achieves on three out of five instances in Table 4.8 the best average performance. In general, $\mathcal{MM}$AS-t and ACS appear to behave very similar on all the instances, although $\mathcal{MM}$AS-t gives slightly better average solution quality. This fact is not surprising, because both algorithms are showing an aggressive search behavior due to the use of the pseudo-random proportional rule of ACS. Since $\mathcal{MM}$AS-t and ACS perform very similar, we only run ACS on the larger instances (see Table 4.9). Also on the larger instances with more than 1000 cities, the ACO algorithms perform very well. Only on instance `u1060`, ILS-`3-opt` performs better and on `pr1002` comparable to $\mathcal{MM}$AS; the performance of ILS and ACS again is roughly the same. Hence, $\mathcal{MM}$AS scales well also to TSP instances with more than 1000 cities. In summary, the computational results confirm the very good performance of the

[6] In [66] it is reported that ACS found the optimal solution in all runs, which is not the case with our implementation; yet, we imposed apparently tighter time limits. For example, in [66] it is reported that ACS took on average 537 seconds to find the optimal solution, on a SGI Challenge L server using a single 200 MHz processor, a machine faster than ours. Also on instance `att532` the computational results are slightly worse if heuristic information is used while on the other instances the solution quality obtained with ACS is better.

variants of $\mathcal{MM}$AS-w compared to the other algorithms tested. In fact, the computational results obtained with the variants of $\mathcal{MM}$AS-w are the best performance results obtained so far with ACO algorithms for symmetric TSP.

Importance of pheromone trail smoothing The importance of pheromone trail smoothing is most evident on instance `lin318` which could only be solved to optimality in all runs by $\mathcal{MM}$AS-w using the pheromone trail smoothing approach. Without pheromone trail smoothing, the algorithm got stuck in some runs at suboptimal solutions. As explained in [169], this instance contains deep suboptimal local minima and to escape from those, the structure of the solution has to be changed significantly. Hence, the increased exploration of tours achieved by the pheromone trail smoothing mechanism seems to help especially in such a situation. This effect has also been noticed on some other smaller instances of comparable size to instance `lin318`. For the other instances, the solution qualities with or without pheromone trail smoothing are, with exception of instances `pr1002` and `fl1577`, roughly comparable. On these latter two instances the performance without pheromone trail smoothing is better. This observation suggests that with increasing instance size smoothing of pheromone trails is becoming less important or even may cause a decrease of performance. Hence, on large instances the pheromone trail limits seem to suffice to provide sufficient exploration of better tours.

Importance of the heuristic information The performance without using heuristic information (indicated by -nh) is for all ant algorithms comparable to the performance when using it. It appears that using heuristic information is not necessary to achieve very good performance, the information contained in the locally optimal solutions is sufficient to guide the algorithms towards high quality solutions. One might argue, that the question whether heuristic information is used or not is just a matter of parameter settings (not using heuristic information is simply achieved by setting $\beta = 0$). Yet, the importance of our computational results is somehow more far reaching. While in the TSP the distance between cities is an obvious heuristic information to be used, in other problems it may be much more difficult to find a meaningful heuristic information which helps to improve performance. Hence, if no such obvious heuristic information exists, our computational results indicate that using an ant algorithm incorporating local search may be enough to achieve a high performing hybrid algorithm.

Experimental Results on Asymmetric TSPs

We applied the same algorithms also to the more general asymmetric TSP; the computational results are given in Table 4.10. In general, the application to asymmetric TSPs confirms the conclusions drawn on symmetric TSPs.

Among the instances tested, `ry48p` and `kro124p` are regularly solved by all ant-based algorithms in a very few seconds. Only the two instances `ft70` and `ftv170` are somewhat harder. For example, only $\mathcal{MM}$AS-w has been able to solve instance `ft70` in all 50 runs to optimality. The main reason for this fact is the use of the diversification features based on pheromone trail smoothing, since $\mathcal{MM}$AS-wnts could not achieve the same performance. Computational results from other researchers suggest that this instance is, considering its small size, relatively hard to solve [174, 257, 66]. Regarding the performance of the ILS algorithm, it is striking that instances `ry48p` and `kro124p` could not be solved to optimality in all runs

Table 4.8: Comparison of $\mathcal{MM}$AS variants, ant colony system (ACS), and iterated `3-opt` (ILS-`3-opt`) applied to selected TSPLIB instances. See the text for a description of the algorithms and the variants used. We give the best solution and (in parenthesis) how many times it has been obtained, the average solution quality (and the percentage deviation from the optimum), the worst solution, the average number of iterations i_{avg}, the average time t_{avg} to find the best solution in a run, and the maximally allowed computation time t_{max}.

algorithm	best	average	worst	$iavg$	t_{avg}	t_{max}
		`d198` opt: 15780				
$\mathcal{MM}$AS-w	15780 (19/25)	15780.4 (0.003)	15784	134.2	61.7	170
$\mathcal{MM}$AS-wnh	15780 (22/25)	15780.2 (0.002)	15784	127.8	59.8	170
$\mathcal{MM}$AS-wnts	15780 (20/25)	15780.2 (0.003)	15781	106.7	59.9	170
$\mathcal{MM}$AS-t	15780 (24/25)	**15780.04** (0.0)	15781	352.8	44.3	170
$\mathcal{MM}$AS-tnh	15780 (22/25)	15780.3 (0.002)	15785	451.8	62.0	170
ACS	15780 (17/25)	15780.3 (0.002)	15781	201.0	37.0	170
ACS-nh	15780 (17/25)	15780.5 (0.003)	15784	229.7	43.2	170
ILS-`3-opt`	15780 (24/25)	15780.2 (0.002)	15784	737.4	18.6	170
		`lin318` opt: 42029				
$\mathcal{MM}$AS-w	42029 (25/25)	**42029.0** (0.0)	42029	139.0	94.2	450
$\mathcal{MM}$AS-wnh	42029 (24/25)	42033.6 (0.011)	42143	164.5	121.6	450
$\mathcal{MM}$AS-wnts	42029 (20/25)	42061.7 (0.078)	42163	77.9	65.9	450
$\mathcal{MM}$AS-t	42029 (24/25)	42031.2 (0.005)	42083	629.7	139.6	450
$\mathcal{MM}$AS-tnh	42029 (22/25)	42038.1 (0.02)	42143	471.4	113.3	450
ACS	42029 (11/25)	42100.4 (0.17)	42208	391.0	86.1	450
ACS-nh	42029 (17/25)	42069.2 (0.096)	42163	386.6	101.3	450
ILS-`3-opt`	42029 (18/25)	42064.6 (0.085)	42163	1527.1	55.6	450
		`pcb442` opt: 50778				
$\mathcal{MM}$AS-w	50778 (1/25)	50911.2 (0.26)	51047	522.0	308.9	600
$\mathcal{MM}$AS-wnh	50778 (3/25)	50903.0 (0.25)	50961	426.2	281.6	600
$\mathcal{MM}$AS-wnts	50778 (2/25)	**50900.9** (0.24)	50931	449.7	319.5	600
$\mathcal{MM}$AS-t	50778 (1/25)	50902.7 (0.25)	51093	1687.2	403.8	600
$\mathcal{MM}$AS-tnh	50778 (1/25)	50901.6 (0.24)	51047	1238.1	261.9	600
ACS	50778 (2/25)	50914.6 (0.26)	51047	1479.9	253.9	600
ACS-nh	50778 (1/25)	50920.5 (0.28)	51060	879.5	159.6	600
ILS-`3-opt`	50778 (1/25)	50917.7 (0.28)	51054	7124.2	180.7	600
		`att532` opt: 27686				
$\mathcal{MM}$AS-w	27686 (2/25)	**27707.9** (0.079)	27756	335.8	387.3	1250
$\mathcal{MM}$AS-wnh	27693 (3/25)	27711.1 (0.091)	27763	373.5	522.3	1250
$\mathcal{MM}$AS-wnts	27693 (2/25)	27708.6 (0.082)	27741	289.4	309.6	1250
$\mathcal{MM}$AS-t	27686 (1/25)	27728.1 (0.15)	27766	1902.4	943.4	1250
$\mathcal{MM}$AS-tnh	27686 (1/25)	27712.6 (0.096)	27746	1335.5	553.3	1250
ACS	27704 (1/25)	27733.4 (0.17)	27830	1289.0	474.3	1250
ACS-nh	27686 (2/25)	27717.6 (0.11)	27758	1141.2	422.5	1250
ILS-`3-opt`	27686 (4/25)	27709.7 (0.086)	27759	8011.1	436.2	1250
		`rat783` opt: 8806				
$\mathcal{MM}$AS-w	8806 (2/25)	**8814.4** (0.095)	8837	631.5	965.2	2100
$\mathcal{MM}$AS-wnh	8808 (1/25)	8818.6 (0.14)	8846	826.6	1337.2	2100
$\mathcal{MM}$AS-wnts	8806 (1/25)	8816.8 (0.12)	8848	805.5	1215.9	2100
$\mathcal{MM}$AS-t	8806 (2/25)	8823.6 (0.20)	8850	2391.6	1226.9	2100
$\mathcal{MM}$AS-tnh	8806 (3/25)	8825.8 (0.22)	8852	2685.2	1464.6	2100
ACS	8810 (1/25)	8830.9 (0.28)	8857	2602.3	1200.9	2100
ACS-nh	8811 (1/25)	8835.6 (0.34)	8857	2309.7	1121.8	2100
ILS-`3-opt`	8806 (1/25)	8828.4 (0.34)	8850	22383.4	1395.2	2100

Table 4.9: Comparison of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system variants, ACS, and iterated `3-opt` on some larger TSP instances from TSPLIB. See Table 4.5 for a description of the entries and the text for a description of the variants.

algorithm	best	average	worst	i_{avg}	t_{avg}	t_{max}
`pr1002` opt: 259045						
$\mathcal{MM}$AS-w	259532 (1/10)	259828.7 (0.30)	260223	1092.1	2433.5	3600
$\mathcal{MM}$AS-wnts	259125 (1/10)	**259506.9** (0.18)	260209	1508.5	2730.7	3600
ACS	259544 (1/10)	260095.7 (0.41)	261925	1508.5	2730.7	3600
ILS-`3-opt`	259045 (2/10)	259591.7 (0.21)	260127	16022.6	1439.6	3600
`u1060` opt: 224094						
$\mathcal{MM}$AS-w	224455 (1/10)	224853.5 (0.34)	225131	538.2	2577.6	3600
$\mathcal{MM}$AS-wnts	224508 (1/10)	224908.5 (0.36)	225350	1665.2	3251.1	3600
ACS	224380 (1/10)	224758.4 (0.29)	225382	5340.7	3794.7	3600
ILS-`3-opt`	224152 (2/10)	**224408.4** (0.14)	224743	12892.7	1210.4	3600
`pcb1173` opt: 56892						
$\mathcal{MM}$AS-w	56896 (1/10)	56956.0 (0.11)	57120	1669.2	3219.5	5400
$\mathcal{MM}$AS-wnts	56892 (1/10)	**56946.3** (0.095)	57040	1138.3	2051.0	5400
ACS -h	56897 (1/10)	57105.4 (0.37)	57311	4354.1	2446.0	5400
ILS-`3-opt`	56897 (1/10)	57029.5 (0.24)	57251	21041.6	1892.0	5400
`d1291` opt: 50801						
$\mathcal{MM}$AS-w	50801 (2/10)	**50821.6** (0.041)	50838	1035.0	1894.4	5400
$\mathcal{MM}$AS-wnts	50801 (3/10)	50828.8 (0.055)	50870	669.6	1206.9	5400
ACS	50820 (1/10)	50873.1 (0.14)	50932	3221.1	2042.8	5400
ILS-`3-opt`	50825 (1/10)	50875.7 (0.15)	50926	16242.6	1102.9	5400
`fl1577` opt: 22249						
$\mathcal{MM}$AS-w	def. 22286 (1/10)	22311.0 (0.28)	22358	690.7	3001.8	7200
$\mathcal{MM}$AS-wnts	22261 (1/10)	**22271.8** (0.10)	22279	1409.2	4473.9	7200
ACS	22309 (1/10)	22326.7 (0.35)	22363	4725.1	5456.8	7200
ILS-`3-opt`	22253 (2/10)	22394.60 (0.65)	22505	44001.9	3447.6	7200

with the ILS algorithm; the algorithm gets stuck at some rather poor quality solutions. Hence, ACO algorithms appear to be very competitive with ILS algorithms on the ATSP instances.

Experimental Results with the Lin-Kernighan Heuristic

The Lin-Kernighan (`LK`) heuristic is regarded as the "uncontested champion" among the local search algorithms for the TSP. For example, in genetic local search algorithms it has been shown that on a time equalized basis, better quality solutions are obtained with the `LK` heuristic than by using `2-opt` [251]. These results suggest, that the solution quality of $\mathcal{MM}$AS can be further increased using the `LK` heuristic, too. We confirm this conjecture by presenting indicative computational results for $\mathcal{MM}$AS-w using the `LK` heuristic. The implementation of the `LK` heuristic has been provided by Olivier Martin and is also used in the large-step Markov chains algorithm proposed in [169, 170]. Since this implementation of the `LK` heuristic is rather run-time intensive, we only tested it on a limited number of TSP instances. Furthermore, we did not test it on instance `fl1577` since this instance is pathologically clustered and it was infeasible to find very high quality solutions using the `LK` heuristic on this instance in reasonable time.

Table 4.10: Comparison of ACO algorithms and ILS on asymmetric TSP instances from TSPLIB. See Table 4.5 for a description of the entries.

Algorithm	best	average	worst	i_{avg}	t_{avg}	t_{max}
`ry48p` opt: 14422						
$\mathcal{MM}$AS-w	14422 (50/50)	14422.0 (0.0)	14422	45.5	2.3	120
$\mathcal{MM}$AS-wnh	14422 (50/50)	14422.0 (0.0)	14422	48.8	3.1	120
$\mathcal{MM}$AS-wnts	14422 (50/50)	14422.0 (0.0)	14422	24.3	4.3	120
$\mathcal{MM}$AS-t	14422 (50/50)	14422.0 (0.0)	14422	195.9	5.1	120
$\mathcal{MM}$AS-tnh	14422 (50/50)	14422.0 (0.0)	14422	107.2	2.2	120
ACS -h	14422 (50/50)	14422.0 (0.0)	14422	355.5	5.1	120
ACS -nh	14422 (50/50)	14422.0 (0.0)	14422	153.0	2.4	120
ILS-`3-opt`	14422 (48/50)	14425.6 (0.025)	14507	8157.6	24.6	120
`ft70` opt: 38673						
$\mathcal{MM}$AS-wh	38673 (50/50)	38673.0 (0.0)	38673	140.5	37.2	300
$\mathcal{MM}$AS-wnh	38673 (46/50)	38675.4 (0.012)	38707	248.6	73.9	300
$\mathcal{MM}$AS-whnr	38673 (30/50)	38686.6 (0.035)	38707	214.4	47.3	300
$\mathcal{MM}$AS-th	38673 (35/50)	38683.2 (0.0)	38707	403.7	35.2	300
$\mathcal{MM}$AS-tnh	38673 (36/50)	38682.1 (0.005)	38707	641.1	66.0	300
ACS -h	38673 (15/50)	38696.8 (0.17)	38707	181.2	9.4	300
ACS -nh	38673 (18/50)	38694.4 (0.095)	38707	260.4	13.3	300
ILS-`3-opt`	38673 (14/25)	38687.9 (0.038)	38707	577.0	3.1	300
`kro124p` opt: 36230						
$\mathcal{MM}$AS-w	36230 (50/50)	36230.0 (0.0)	36230	24.1	7.3	300
$\mathcal{MM}$AS-wnh	36230 (50/50)	36230.0 (0.0)	36230	30.3	11.3	300
$\mathcal{MM}$AS-wnts	36230 (50/50)	36230.0 (0.0)	36230	23.9	7.4	300
$\mathcal{MM}$AS-t	36230 (50/50)	36230.0 (0.0)	36230	118.4	5.5	300
$\mathcal{MM}$AS-tnh	36230 (50/50)	36230.0 (0.0)	36230	158.9	9.2	300
ACS	36230 (50/50)	36230.0 (0.0)	36230	147.4	6.1	300
ACS-nh	36230 (50/50)	36230.0 (0.0)	36230	95.5	4.5	300
ILS-`3-opt`	36230 (14/25)	36542.5 (0.94)	37114	5867.7	23.4	300
`ftv170` opt: 2755						
$\mathcal{MM}$AS-w	2755 (25/25)	2755.0 (0.0)	2755	333.5	56.2	600
$\mathcal{MM}$AS-wnh	2755 (25/25)	2755.0 (0.0)	2755	262.2	53.1	600
$\mathcal{MM}$AS-wnts	2755 (17/25)	2757.8 (0.082)	2764	124.5	39.9	600
$\mathcal{MM}$AS-t	2755 (8/25)	2758.0 (0.15)	2774	1344.3	230.6	600
$\mathcal{MM}$AS-tnh	2755 (22/25)	2756.1 (0.040)	2764	331.3	42.5	600
ACS	2755 (21/25)	2756.4 (0.051)	2764	328.2	18.3	600
ACS-nh	2755 (23/25)	2755.7 (0.025)	2764	805.6	53.2	600
ILS-`3-opt`	2755 (20/25)	2756.8 (0.065)	2764	2713.6	21.7	600

Table 4.11: Experimental results for $\mathcal{MAX}$–$\mathcal{MIN}$ ant system using the Lin-Kernighan heuristic for the local search. Given are the best (in parenthesis the frequency of finding the optimum), the average (percentage deviation from optimum in parenthesis), and the worst solution. i_{avg} is the average number of iterations needed to find the best solution in each run, t_{avg} is the average time.

instance	best	average	worst	i_{avg}	t_{avg}
lin318	42029 (25/25)	42029 (0.0)	42029	20.7	953.6
pcb442	50778 (25/25)	50778 (0.0)	50778	72.9	2737.8
att532	27686 (4/10)	27695.3 (0.033)	27705	169.3	4155.4
rat783	8806 (7/10)	8806.5 (0.006)	8808	315.0	3172.7
pcb1173	56892 (2/10)	56895.4 (0.006)	56901	220.4	7450.7

When applying the LK heuristic we did not fix a priori a time limit to the algorithm, but allowed on each instance a maximum number of 5000 LK applications. Since not as often a local search can be done as with 3-opt, we used slightly different parameter settings. Most significantly, we use only 10 ants and the schedule for using the global-best for the pheromone update is shortened. Here, we use $u_{gb} = 3$ for $i \leq 25$, $u_{gb} = 2$ for $i \leq 51$, and $u_{gb} = 1$ otherwise. The other parameter settings are the same as before.

The computational results obtained by combining $\mathcal{MM}$AS with the LK heuristic are significantly better with respect to solution quality than those using our 3-opt implementation (compare results of Table 4.11 to that of Tables 4.8 and 4.9). Yet, also the run-times are much higher. If using the LK heuristic, the performance with respect to solution quality is comparable to the best local search algorithms for the TSP. For example, our computational results compare favourably to the genetic algorithm which has won the ICEO'96 [77, 78]. Yet, in the meanwhile that algorithm has been substantially improved, mainly by using a significantly faster local search implementation [174]. With respect to computation time, our computational results also cannot rival those of iterated-LK (see also Chapter 5). Hence, to obtain competitive results to state-of-the-art algorithms for symmetric TSPs with respect to computation time, one important issue would be to use a significantly faster implementation of the LK heuristic.

4.3.7 Related Work

The TSP receives since a long time very strong attention from the metaheuristics research community.[7] Today, the best implementations of iterated local search algorithms are the most efficient approaches to symmetric TSPs for short to medium run-times [129]. Yet since the first international contest on evolutionary computation, the TSP has received considerable renewed interest and many new approaches and improved implementations have been presented recently. Among these algorithms we find the genetic local search approach of Merz and Freisleben [77, 174], an improved implementation of ASPARAGOS (one of the first genetic local search approaches to the TSP) of Gorges-Schleuter [102], a new genetic local search approach using a repair-based crossover operator and brood selection by Walters [257], a genetic algorithm using a specialized crossover operator, called edge assembly crossover, by Nagata and Kobayashi [187], and a specialized local search algorithm for the TSP, called iterative partial transcription,

[7] We restrict our discussion of algorithms for the TSP to the most recent work. For the discussion of earlier work we refer to the overview article by Johnson and McGeoch [129].

by Möbius et al. [181]. We do not go into the details of these approaches, but would rather like to indicate their relative performance compared to $\mathcal{MM}$AS.

Applied to symmetric TSP instances, most of these algorithms outperform our $\mathcal{MM}$AS algorithm, in particular they may obtain better solution quality even at lower computation time. For example, the genetic local search approach presented in [174], which uses the LK heuristic for the local search, reaches on average a solution of 8806.2 on instance rat783 in 424 seconds on a DEC Alpha station 255 MHz. The reasons for this fact are certainly on the one side that these implementations are much more fine-tuned than ours. Yet, when using the LK heuristic for the local search phase, at least the solution quality obtained with $\mathcal{MM}$AS is competitive to these recent algorithms.

Applied to asymmetric TSP instances, our computational results with respect to solution quality compare more favorably to these approaches. The solution quality we obtain with $\mathcal{MM}$AS-w is better than that of the genetic local search approach of [174] and the same as reported in [102, 257], but at the cost of slightly higher run-times for the two harder instances.

4.3.8 Conclusions

In this section we extended $\mathcal{MM}$AS with local search and applied it to symmetric and asymmetric TSP instances. We have shown that if we use a constant number of ants and occasionally give reinforcement to the edges of the global-best solution, a very good performance is achieved. To shorten the run-time for finding high quality solutions we proposed two possibilities. The first is based on a gradual increase of the number of ants that apply local search. The second approach consists in combining $\mathcal{MM}$AS with the pseudo-random proportional rule used in ACS.

The computational results of $\mathcal{MM}$AS confirm that very high quality solutions on symmetric and asymmetric TSP instances can be obtained and that $\mathcal{MM}$AS is competitive with the standard implementation of an ILS algorithm used for comparison. Nevertheless, some recently proposed algorithms [174, 181, 187, 257], the iterated LK algorithm [169, 129], and the improved ILS algorithms presented in the next chapter achieve better performance for the TSP. Yet, we are more interested in generic issues like which shape a hybrid $\mathcal{MM}$AS algorithm should take and not in setting up new records for the TSP. As such, our results are good enough to suggest that $\mathcal{MM}$AS may be very successfully applied also to other combinatorial optimization problems. The experimental results also have shown that $\mathcal{MM}$AS-w is currently the ACO algorithm achieving the highest solution quality on most larger symmetric TSP instances.

4.4 $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System for the Quadratic Assignment Problem

In this section we present the application of $\mathcal{MM}$AS to the quadratic assignment problem (QAP). We first explain how $\mathcal{MM}$AS is applied to the QAP and give details on the local search algorithms which are combined with $\mathcal{MM}$AS. Then we compare the performance of $\mathcal{MM}$AS to that of the best local search algorithms known for the QAP. It is well known that the particular type of QAP instances has a strong influence on the performance of the different algorithmic

approaches proposed for solving the QAP. We define four different classes of QAP instances and show that the best choice for the local search algorithm to be used in the hybrid algorithm depends strongly on the QAP instance class. The computational results obtained with $\mathcal{MM}$AS confirm the generality of the improvements introduced by $\mathcal{MM}$AS and establish that $\mathcal{MM}$AS is one of the best algorithms for the QAP. It performs particularly well on the important class of real-life QAP instances.

4.4.1 The Quadratic Assignment Problem

The quadratic assignment problem is an important problem in theory and practice. Many practical problems arising in backboard wiring [230], campus and hospital layout [61, 70], typewriter keyboard design [40], scheduling [89] and image processing [244] can be formulated as QAPs. The QAP can best be described as the problem of assigning a set of facilities to a set of locations with given distances between the locations and given flows between the facilities. The objective is to place the facilities on locations in such a way that the sum of the product between flows and distances is minimal.

More formally, given n facilities and n locations, two $n \times n$ matrices A and B, where a_{ij} is the flow between facility i and j and b_{rs} is the distance between locations r and s, the QAP is the problem to minimize

$$f(\phi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\phi_i \phi_j} \tag{4.17}$$

where ϕ is an arbitrary permutation of the set of integers $\{1, \ldots, n\}$ (corresponding to an assignment of facilities to locations), ϕ_i gives the location of facility i in ϕ, and $f(\phi)$ is the objective function value of permutation ϕ. Intuitively, $b_{ij}a_{\phi_i\phi_j}$ represents the cost contribution of simultaneously assigning facility i to location ϕ_i and facility j to location ϕ_j.

The QAP is an $\mathcal{NP}$-hard [86, 218] optimization problem, even finding a solution within a factor of $1 + \epsilon$ of the optimal one remains $\mathcal{NP}$-hard. It is considered as one of the hardest optimization problems as general instances of size $n \geq 20$ cannot be solved to optimality. Therefore, to practically solve the QAP one has to apply heuristic algorithms which find very high quality solutions in short computation time. These approaches include local search algorithms like iterative improvement, simulated annealing [41, 51], tabu search [18, 225, 241], genetic algorithms [74, 173, 246], evolution strategies [190], GRASP [154], ant system [165], and scatter search [56].

4.4.2 Applying $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System to the QAP

Construction of Solutions

When applying $\mathcal{MM}$AS to the QAP we first have to define how solutions are constructed. Based on the intuitive formulation of the QAP, one possibility for constructing solutions is to assign facilities in some order to locations. Hence, for the application to the QAP the pheromone trails τ_{ij} refer to specific locations for facilities, that is, τ_{ij} gives the desirability of assigning facility i to location j. The ants are used to construct valid solutions for the QAP assigning every facility to exactly one location and not using a location by more than one facility. The

construction of a solution involves two steps. First, a facility has to be chosen which is assigned next to a location. In a second step this facility is assigned to some free location, that is, to a location which still has not been assigned a facility. In $\mathcal{MM}$AS we implement these two steps as follows. First, a facility is randomly chosen among the still unassigned ones. Then, this facility is put on some free location according to the following probability distribution.

$$p_{ij} = \begin{cases} \frac{\tau_{ij}}{\sum_{l \in U(k)} \tau_{il}} & \text{if location } j \text{ is still free} \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

$U(k)$ denotes the set of still unassigned items. The intuition behind this rule is to probabilistically prefer locations for facility i that have shown to be promising in previously found solutions; the higher τ_{ij}, the more likely it is to put facility i on location j. As shown in the $\mathcal{MM}$AS application to the TSP, the heuristic information is not necessary to achieve high quality solutions when solutions are improved by local search. Hence, to keep the application of $\mathcal{MM}$AS to the QAP as simple as possible, we do not use any heuristic information. Doing so we additionally can eliminate the parameter β, which determined the weight of the heuristic information in the solution construction; the parameter α which regulated the influence of the pheromone trail is set to 1.

For the application of $\mathcal{MM}$AS to the QAP we also will use $\mathcal{MM}$AS-t, which has already been introduced in Section 4.3.5. When applying the pseudo-random proportional rule, for the application to the QAP, Equation 4.5 simplifies to

$$j = \begin{cases} \textit{arg}\max_{l \in U(k)}\{\tau_{il}\} & \text{if } p \leq p_0 \quad \text{(exploitation)} \\ S & \text{otherwise} \quad \text{(exploration)} \end{cases} \tag{4.19}$$

where p is a random number uniformly distributed in $[0, 1]$ and S is a random variable with probability distribution given by Equation 4.18. The parameter p_0 controls the exploitation of the accumulated experience reflected in the pheromone trail matrix versus the biased exploration of new solutions.

Update of Pheromone Trails

After all ants have constructed a solution, the pheromone trails are updated according to Equation 4.10. Here, $\Delta\tau_{ij}^{\textit{best}}$ is defined as

$$\Delta\tau_{ij}^{\textit{best}}(t) = \begin{cases} 1/f(\phi_{\textit{best}}) & \text{if facility } i \text{ is put on location } j \text{ in } \phi_{\textit{best}} \\ 0 & \text{otherwise} \end{cases} \tag{4.20}$$

The solution which adds pheromone may be the *iteration-best* solution ϕ_{ib}, or the best solution found since the start of the algorithm, the *global-best* solution ϕ_{gb}. Thus, if in the best solutions facilities are often put on specific locations, these solution components receive a high amount of pheromone and therefore facilities will be put preferably on these location in future iterations of the algorithm. For the application of $\mathcal{MM}$AS to the QAP we will generally choose the global-best solution for the pheromone update (see Section 4.4.4 for details on the parameter settings). An exception is made after a reinitialization of the trails as described in the following section.

Pheromone Trail Limits

The trail limits for $\mathcal{MM}$AS are chosen the same way as in the $\mathcal{MM}$AS application to the TSP. Details on the their exact values are given in Section 4.4.4.

4.4.3 Local Search for the QAP

For a hybrid algorithm like $\mathcal{MM}$AS, it is important to decide which local search algorithm to use. Important for the choice of a local search algorithm is its speed and the solution quality it produces. We first explain how the neighborhood in local search algorithms for the QAP is defined and then explain the possibilities considered for the local search in the hybrid $\mathcal{MM}$AS algorithm.

Neighborhood Definition and Move Evaluation

The neighborhood of a permutation ϕ is usually defined by the set of permutations which can be obtained by exchanging two facilities, that is $\mathcal{N}(\phi) = \{\phi^{'} \mid \phi^{'}(i) = \phi(i)\, \forall i \neq r, s \wedge \phi^{'}(r) = \phi(s),\ \phi^{'}(s) = \phi(r)\}$. The objective function difference $\delta(\phi, r, s)$ of exchanging two facilities $\phi(s)$ and $\phi(r)$ can be computed in $O(n)$, using the following equation [244].

$$\begin{aligned}\delta(\phi, r, s) = a_{rr} \cdot (b_{\phi_s\phi_s} - b_{\phi_r\phi_r}) + a_{rs} \cdot (b_{\phi_s\phi_r} - b_{\phi_r\phi_s}) + \\ a_{sr} \cdot (b_{\phi_r\phi_s} - b_{\phi_s\phi_r}) + a_{ss} \cdot (b_{\phi_r\phi_r} - b_{\phi_s\phi_s}) + \\ \sum_{k=1,k\neq r,s}^{n} (a_{kr} \cdot (b_{\phi_k\phi_s} - b_{\phi_k\phi_r}) + a_{ks} \cdot (b_{\phi_k\phi_r} - b_{\phi_k\phi_s}) + \\ a_{rk} \cdot (b_{\phi_s\phi_k} - b_{\phi_r\phi_k}) + a_{sk} \cdot (b_{\phi_r\phi_k} - b_{\phi_s\phi_k}))\end{aligned} \tag{4.21}$$

Note that Equation 4.21 holds for QAP instances with symmetric and asymmetric matrices A and B. If both matrices are symmetric with a null diagonal, the following simplified formula can be used.

$$\delta(\phi, r, s) = 2 \cdot \sum_{k=1,k\neq r,s}^{n} (a_{kr} - a_{ks}) \cdot (b_{\phi_k\phi_s} - b_{\phi_k\phi_r}) \tag{4.22}$$

When using a best-improvement pivoting rule, the effect of a particular swap can be evaluated faster using information from preceding iterations. Let $\phi^{'}$ be the solution which is obtained by exchanging facilities r and s in solution ϕ, then for swapping units u and v with $\{u, v\} \cap \{r, s\} = \emptyset$ the move can be evaluated in constant time [244].

$$\begin{aligned}\delta(\phi', u, v) = \delta(\phi, r, s) + (a_{ru} - a_{rv} + a_{sv} - a_{su}) \cdot (b_{\phi_s\phi_u} - b_{\phi_s\phi_v} + b_{\phi_r\phi_v} - b_{\phi_r\phi_u}) \\ (a_{ur} - a_{vr} + a_{vs} - a_{us}) \cdot (b_{\phi_u\phi_s} - b_{\phi_v\phi_s} + b_{\phi_v\phi_r} - b_{\phi_u\phi_r})\end{aligned} \tag{4.23}$$

Again, for symmetric QAP instances this formula can be simplified using the fact that $a_{ij} = a_{ji}$ and $b_{\phi_i\phi_j} = b_{\phi_j\phi_i}$.

Local Search for the Hybrid Algorithm

For the hybrid algorithm we considered two possible local search algorithms. One possibility is to apply an iterative improvement local search based on the above described neighborhood, in the following referred to as `2-opt`. The other is the use of *short* runs of tabu search, referred to as *TS*. Iterative improvement has the disadvantage that it stops at the first local minimum encountered, whereas tabu search allows to escape from local minima and generally finds much better solutions. Yet, `2-opt`, compared to tabu search, runs considerably faster and can be applied much more often in a given amount of computation time. Additionally, `2-opt` benefits from the good solutions generated by $\mathcal{MM}$AS and in the later stages of the search only few improvement steps are necessary to reach a local minimum.

Iterative improvement can be implemented using a *first-improvement* or a *best-improvement* pivoting rule. While in the first case the first improving move found is applied, in the second case the whole neighborhood is examined and a move which gives the best improvement is chosen. With first-improvement usually more moves have to be performed to reach a local minimum, but moves may be identified without examining the whole neighborhood and therefore the time per move is less than with a best-improvement pivoting rule. Additionally, different local optima may be obtained by scanning the neighborhood in a random order. On the other side, for the QAP best-improvements benefits from the fact that the effect of exchanging two facilities can be calculated using the information of the preceding iterations (see Equation 4.23). Some initial experiments suggested that in combination with $\mathcal{MM}$AS it was preferable to use the best-improvement local search and therefore we restricted the following experiments to this local search variant.

For many QAP instances of QAPLIB [39] the best known solutions have been found by tabu search algorithms. Consequently, tabu search appears to be the most effective local search approach to the QAP and we also considered it for the local search phase in $\mathcal{MM}$AS. Such an idea has also been used in a hybrid genetic algorithm for the QAP [74]. As the tabu search algorithm we use short runs of the robust tabu search (Ro-TS) [241]. We re-implemented robust tabu search in C, the original Pascal implementation is available from Éric Taillard at `http://www.idsia.ch/~eric/`. The length of the Ro-TS runs was limited to $4n$ iterations which was also done for the reimplementation of the genetic hybrid algorithm in [244]; this choice facilitates the comparison of computational results across papers. Nevertheless, other choices for the tabu search length may give better performance.

4.4.4 Experimental Results

In this section we report on the experimental results obtained with $\mathcal{MAX}$–$\mathcal{MIN}$ ant system on a wide range of QAP instances of QAPLIB, a QAP benchmark library, covering all the four later defined instance classes of QAP instances. Like for the TSP, we again present computational results obtained with both variants, $\mathcal{MM}$AS-w and $\mathcal{MM}$AS-t. In the following we will refer to both versions simply as $\mathcal{MM}$AS, except when we explicitly discuss aspects specific to one of the two variants.

Parameter Settings

Suitable parameter settings have been determined in preliminary experiments. In particular, we use the following parameter settings. We use five ants and set ρ to 0.8. All ants apply local search after every iteration. The low number of ants is motivated by the fact that the local search is computationally much more demanding for larger QAP instances than for the TSP and only a low number of local search applications can be done in reasonable time.

The pheromone trail limits are determined as in the application of $\mathcal{MM}$AS to the TSP. The maximal possible trail limit is $\tau_{max} = \frac{1}{1-\rho} \cdot \frac{1}{f(\phi_{gb})}$, where ϕ_{gb} is the global-best solution. When applying $\mathcal{MM}$AS-w we set τ_{min} to $\tau_{max}/2n$, and to $\tau_{max}/5$ when using $\mathcal{MM}$AS-t. Since we fix the range of the pheromone trail limits in $\mathcal{MM}$AS-t, we modify the parameter value for p_0 according to the instance size such that on average a constant number of randomly biased decisions is made according to Equation 4.18. We use the same parameter values for the parameter p_0 as in the TSP, that is, $p_0 = (n - 15)/n$. In general we found that the above proposed parameter values yield a rather robust performance over a wide range of instances. Yet, for particular instances other parameter values may give better performance.

If `2-opt` is applied, the pheromone update is always done with the best solution found during the run with $\mathcal{MM}$AS-t, except after a reinitialization of the trails as discussed in the next section. When applying tabu search, preliminary experiments suggested that alternatingly choosing the iteration-best and the global-best solution for the pheromone update gives better results. In $\mathcal{MM}$AS-w we use a schedule for the global-best update similar to that proposed in Section 4.3.3. Recall the u_{gb} indicates that every u_{gb} iterations the global-best ant updates the pheromone trails. For the application to the QAP we set u_{gb} to 3 in the 11 first iterations, u_{gb} to 2 up to iteration 25, and from then on to 1.

Additional Diversification Features

For the application of $\mathcal{MM}$AS-w *and* $\mathcal{MM}$AS-t to the QAP we use an additional technique to increase the diversification of the search. As done in the application to the TSP we re-initialize the trails if the average branching factor $\bar{\lambda}$ indicates that $\mathcal{MM}$AS is near convergence; here the trails are re-initialized if $\bar{\lambda} \leq 1.1$; $\bar{\lambda}$ is measured every fifth iteration of. After the re-initialization, only the iteration-best solutions are allowed to update the pheromones for the next 5 iterations in $\mathcal{MM}$AS-t.[8] When applying $\mathcal{MM}$AS-w we simply apply the schedule for u_{gb} like at the start of the algorithm.

Types of QAP instances

According to [244] the instances of QAPLIB can be classified in four classes.

(i) **Unstructured, randomly generated instances**: Instances with the distance and flow matrix entries generated randomly according to a uniform distribution (among those are instances `taixxa` used in Section 4.4.4). These instances are among the hardest to solve exactly. Nevertheless most iterative search methods find solutions within $1 - 2\%$

[8] When using $\mathcal{MM}$AS-t and the global-best solution would be used immediately to reinforce the pheromone trails after a reinitialization the solutions would resemble already the first iteration after the reinitialization that global-best solution due to the aggressive pseudo random proportional rule (4.19).

from the best known solutions relatively fast [244]. This observation can also be backed up theoretically [38, 212]: it has been shown that asymptotically for increasing instance size, the difference between the upper and lower bounds on the optimal objective function value converges to zero.

(ii) **Grid-based distance matrix**: In this class of instances the distance matrix stems from a $n_1 \times n_2$ grid and the distances are defined as the Manhattan distance between grid points. These instances have multiple global optima (at least 4 in case $n_1 \neq n_2$ and at least 8 in case $n_1 = n_2$) due to the definition of the distance matrices [244] (among those are instances `nugxx` and `skoxx` used in Section 4.4.4).

(iii) **Real-life instances**: Instances from this class are "real-life" instances from practical applications of the QAP. Among those are the instances of Steinberg [230] (instances `ste36x`), a layout problem for a hospital [70, 145] (instances `kra30x`), instances corresponding to the layout of typewriter keyboards [40] (instances `bur26x`), and a new type of instances proposed in [244]. The real-life problems have in common that the flow matrices have many zero entries and the remaining entries are clearly not uniformly distributed. The matrix entries exhibit a clear structure and it is in this main aspect that real-life instances differ from the randomly generated instances described above. In real-life problems often the situation is given that there is a rather high interaction among a part of the facilities. It seems to be intuitive that in good solutions facilities with a high interaction with other facilities should be placed at locations with small inter-location distances.

(iv) **Real-life-like instances**: Because the real-life instances in QAPLIB are mainly of a rather small size, a particular type of randomly generated problems has been proposed in [244] (instances `taixxb`). These instances are generated in such a way that the matrix entries resemble a distribution found in real-life problems.

To differentiate among the classes of QAP instances the flow dominance *fd* can be used. It is defined as the coefficient of variation of the flow matrix entries multiplied by 100.

$$\mathit{fd}(A) = 100 \cdot \frac{\sigma}{\mu}, \text{ where}$$

$$\mu = \frac{1}{n^2} \cdot \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} \text{ and } \sigma = \sqrt{\frac{1}{n^2-1} \cdot \sum_{i=1}^{n}\sum_{j=1}^{n}(a_{ij}-\mu)}$$

A high flow dominance indicates that a large part of the overall flow is exchanged among relatively few facilities. Thus, instances from class (i) will have rather low flow dominance, whereas real-life instances, in general, have a rather high flow dominance. Real life problems often have many zero entries, hence the sparsity of the matrix may also give an indication of the type of an instance. A disadvantage of the flow dominance is that it captures only the structure of one out of two matrices, neglecting that of the distance matrix. Therefore, similar to the flow dominance, also a *distance dominance* (*dd*) can be defined. In Table 4.12 the dominance values for both matrices of some instances are given (ordered according to the four instance classes described above). As can be seen, the instances of classes (iii) and (iv) have considerably higher dominance values for at least one of the matrices.

Table 4.12: Given are the dominance values for some of the QAPLIB instances used in the experimental evaluation of $\mathcal{MM}$AS. The dominance values are calculated for the first (*A*) and the second (*B*) matrix as given in QAPLIB. The problem instances are ordered according to the 4 classes described in Section 4.2. Note that real-life (like) instances have significantly higher flow dominance values for at least one of the two matrices.

problem instance	$dd(A)$	$fd(B)$	problem instance	$dd(A)$	$fd(B)$
unstructured, randomly generated (i)			**real-life instances (iii)**		
`tai20a`	67.02	64.90	`bur26a`	15.09	274.95
`tai25a`	61.81	64.29	`bur26b`	15.91	274.95
`tai30a`	58.00	63.21	`bur26c`	15.09	228.40
`tai35a`	61.64	61.57	`bur26d`	15.91	228.40
`tai40a`	63.10	60.23	`bur26e`	15.09	254.00
`tai50a`	60.75	62.24	`kra30a`	49.22	149.98
`tai60a`	61.41	60.86	`kra30b`	49.99	149.98
`tai80a`	59.22	60.38	`ste36a`	55.65	400.30
`tai100a`	59.34	60.31	`ste36b`	100.79	400.30
unstructured, grid-distances (ii)			**real-life like instances (iv)**		
`nug30`	52.75	112.48	`tai20b`	128.25	333.23
`sko42`	51.96	108.48	`tai25b`	87.02	310.40
`sko49`	51.55	109.38	`tai30b`	85.20	323.91
`sko56`	51.46	110.53	`tai35b`	78.66	309.62
`sko64`	51.18	108.38	`tai40b`	66.75	317.22
`sko72`	51.14	107.13	`tai50b`	73.44	313.91
`sko81`	50.93	106.61	`tai60b`	76.83	317.82
`sko90`	50.91	108.34	`tai80b`	64.05	323.17
`sko100a`	50.75	106.64	`tai100b`	80.42	321.34

Which Local Search?

$\mathcal{MM}$AS is a hybrid algorithm combining solution construction by ants with a local search to improve solutions. For such a hybrid method the instance type of the problem to be solved may have a strong influence on which local search algorithm should be used. This issue is exemplified here applying $\mathcal{MM}$AS to the QAP with computational results for instance `tai50a` of class (i) and for instance `tai50b` of class (iv). Both instances show largely different flow dominance ($fd = 62.24$ in the case of `tai50a` and $fd = 313.91$ in the case of `tai50b`. We run $\mathcal{MM}$AS with `2-opt` ($\mathcal{MM}$AS+`2-opt`) and $\mathcal{MM}$AS with short tabu search runs of length $4n$ ($\mathcal{MM}$AS+TS).[9] Additionally we also applied a multiple descent approach (MD) that restarts the local search from randomly generated initial solutions if trapped in a local minimum. All algorithms are allowed the same computation time.

In Figure 4.4 we give the development of the best solutions found by the algorithms averaged over 10 independent runs. Two important observations can be made. One is that $\mathcal{MM}$AS is able to guide the local search towards better solutions, as for `2-opt` and tabu search, the results with $\mathcal{MM}$AS are better than that of the multiple descent approach. The more important observation is that the local search algorithm which should be combined with $\mathcal{MM}$AS strongly depends on the instance type. For instance `tai50a` the addition of tabu search gives significantly better

[9]The presented results have been obtained with $\mathcal{MM}$AS-t. Qualitatively, $\mathcal{MM}$AS-w shows very similar behavior and, hence, we will not distinguish between $\mathcal{MM}$AS-t and $\mathcal{MM}$AS-w in the following discussion.

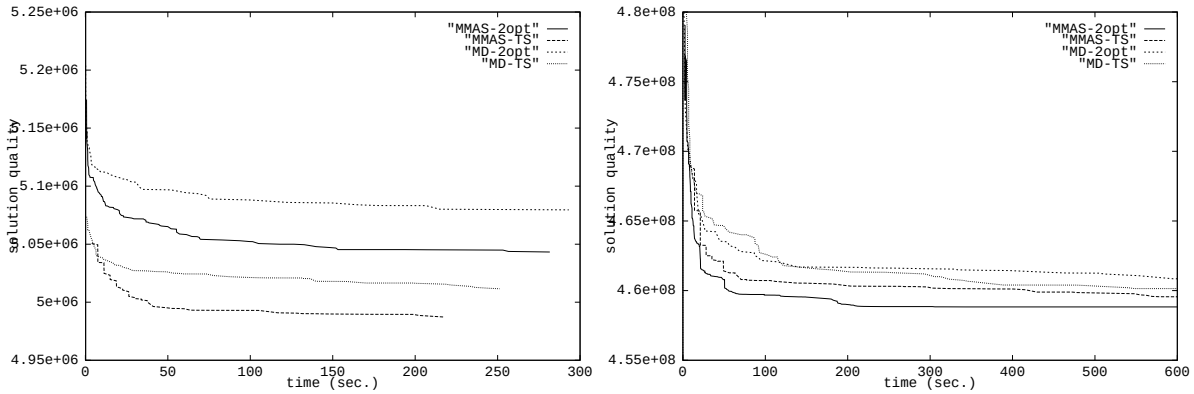

Figure 4.4: Solution quality development on `tai50a` (left side) and on `tai50b` (right side) for $\mathcal{MM}$AS with `2-opt` local search (MMAS-2opt), $\mathcal{MM}$AS with short tabu search runs (MMAS-TS), multiple descent with `2-opt` (MD-2opt), and multiple descent with short tabu search runs (MD-TS). The x-axis indicates the computation time and the y-axis gives the solution quality. The results are based on 10 independent runs for each algorithm.

solution quality than `2-opt`. In contrast, for instance `tai50b` using the simple `2-opt` descent local search suffices to outperform the hybrid algorithm using tabu search. In the case of the structured instance `tai50b` it pays off to use more often a rather simple local search procedure to identify the preferred locations of the facilities instead of the more run-time intensive tabu search procedure which yields slightly better local optima. The computational results of the next section confirm this observation.

Computational Results

We now present computational results for the application of $\mathcal{MM}$AS to a wide range of QAP instances of all the four classes defined above. We only applied the algorithms to instances with $n \geq 20$, because smaller instances are easily solved. For the application of $\mathcal{MM}$AS to the QAP we pay special attention to the best choice of the local search algorithm and its dependence on the QAP instance class. We present computational results using the two different local search procedures as discussed in the previous section. We allow a total of 250 applications of tabu search. For `2-opt` a maximal number of 1000 local search applications is allowed. We applied both variants, $\mathcal{MM}$AS-w and $\mathcal{MM}$AS-t with either of the two local search algorithms. Since on almost all instances $\mathcal{MM}$AS-t with tabu search performed better than $\mathcal{MM}$AS-w with tabu search, we only present results for the first combination. When using $\mathcal{MM}$AS-t with `2-opt`, the computation times are always lower than those obtained by applying 250 short tabu search runs. In fact, the larger the instance size, the smaller is the computation time needed by the 1000 local search applications relatively to the 250 tabu search runs. Yet, when using $\mathcal{MM}$AS-w, applying 1000 times a `2-opt` local search took more run-time than the 250 tabu search runs. (The reason is that $\mathcal{MM}$AS-w converges slower and in the first iterations the local search needs more time to find a local optimum than in $\mathcal{MM}$AS-t.) Hence, we stopped $\mathcal{MM}$AS-w after a time limit equvalent to 250 tabu search runs.

Table 4.13: Experimental results for $\mathcal{MM}$AS with different types of local search procedures, tested on instances of class (i) and (ii). We give the average solution quality found over 10 independent runs (given as the percentage excess over the best known solutions) of the algorithms. The computational results are compared to robust tabu search (Ro-TS) and reactive tabu search (RTS). The best results are indicated in **bold-face**.

Problem instance	**Ro-TS**	**RTS**	$\mathcal{MM}$AS-t + 250 TS	$\mathcal{MM}$AS-t + 2-opt-b	$\mathcal{MM}$AS-w + 2-opt-b
random problems with entries uniformly distributed					
tai20a	0.108	0.169	**0.066**	0.633	0.428
tai25a	**0.274**	0.365	0.507	1.297	1.751
tai30a	0.426	0.301	**0.259**	1.376	1.286
tai35a	0.589	**0.457**	0.610	2.213	1.586
tai40a	0.990	**0.569**	0.782	1.899	1.131
tai50a	1.125	**0.832**	1.147	1.960	1.900
tai60a	1.203	**0.822**	0.981	2.212	2.484
tai80a	0.900	**0.497**	0.741	1.593	2.103
tai100a	0.894	**0.361**	0.685	1.660	2.078
random flows on grids					
nug20	**0.0**	0.763	**0.0**	**0.0**	**0.0**
nug30	**0.013**	0.892	0.0196	0.059	0.042
sko42	0.025	0.683	**0.004**	0.130	0.104
sko49	0.076	0.590	**0.039**	0.156	0.150
sko56	0.088	0.656	**0.072**	0.188	0.118
sko64	0.071	0.654	**0.036**	0.067	0.171
sko72	0.146	0.558	**0.103**	0.258	0.243
sko81	0.136	0.406	**0.077**	0.215	0.223
sko90	**0.128**	0.530	**0.128**	0.355	0.288
sko100a	**0.108**	0.429	0.132	0.258	0.190

We compare the performance of $\mathcal{MM}$AS to two of the most efficient local search algorithms for the QAP. One is the robust tabu search (Ro-TS) algorithm, also used for the hybrid with $\mathcal{MM}$AS, the other is the reactive tabu search (RTS) algorithm [18]. Since the original implementation of RTS runs only on symmetric instances, we modified the code to allow also the application to asymmetric instances. Both tabu search algorithms, Ro-TS and RTS, are allowed $1000 \cdot n$ iterations, resulting in similar run-times to $\mathcal{MM}$AS+TS.

The computational results are presented in Table 4.13 for instances of classes (i) and (ii), and in Table 4.14 for the instances of classes (iii) and (iv). Given is the percentage deviation from the best known solution averaged over 10 trials. In general, which algorithm performs best is strongly dependent on the instance class. For the instances of class (i), RTS is the best algorithm (actually, in [244] variants of strict tabu search performed slightly better than RTS on these instances) but it performs poorly on all the other instance classes. Both, $\mathcal{MM}$AS+TS and Ro-TS perform similar on the instances of classes (i) and (ii), but $\mathcal{MM}$AS+TS achieves for most of these a slightly better average performance. Hence, the computational results indicate that $\mathcal{MM}$AS may be useful as a diversification mechanism for tabu search by restarting the search at promising new initial solutions. The hybrid $\mathcal{MM}$AS variants with `2-opt` local search perform significantly worse than Ro-TS and $\mathcal{MM}$AS+TS on these instances, with $\mathcal{MM}$AS-w performing slightly better on most instances.

Table 4.14: Experimental results for $\mathcal{MM}$AS with different types of local search procedures, tested on instances of class (iii) and (iv). We give the average solution quality found over 10 independent runs (given as the percentage excess over the best known solutions) of the algorithms. The computational results are compared to robust tabu search (Ro-TS) and reactive tabu search (RTS). The best results are indicated in **bold-face**.

Problem instance	**Ro-TS**	**RTS**s	$\mathcal{MM}$AS-t + 250 TS	$\mathcal{MM}$AS-t + 2-opt-b	$\mathcal{MM}$AS-w + 2-opt-b
real life instances					
bur26a-h	0.002	0.092	**0.0**	**0.0**	**0.0**
kra30a	0.268	1.048	**0.135**	0.418	0.314
kra30b	0.023	0.128	**0.014**	0.117	0.049
ste36a	0.155	2.043	**0.061**	0.184	0.181
ste36b	0.081	0.081	0.014	**0.0**	**0.0**
randomly generated real-life like instances					
tai20b	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
tai25b	**0.0**	4.427	**0.0**	**0.0**	**0.0**
tai30b	0.107	0.357	**0.0**	**0.0**	**0.0**
tai35b	0.064	0.576	0.024	0.094	**0.0**
tai40b	0.531	0.401	0.201	**0.0**	**0.0**
tai50b	0.342	0.426	0.125	0.029	**0.002**
tai60b	0.417	0.364	0.043	**0.014**	**0.005**
tai80b	1.031	1.033	0.725	**0.318**	**0.096**
tai100b	0.512	0.555	0.336	**0.142**	0.188

On the instances of classes (iii) and (iv), the performance characteristics of the algorithms are different. Here, both $\mathcal{MM}$AS variants using `2-opt` have a much improved performance and are the best algorithms for instances `taixxb` and `bur26x`. For example, for all instances `bur26x` the best known solution value is found in every run. $\mathcal{MM}$AS-t using `2-opt` achieves this on average in 2.5 seconds on a SUN UltraSparc I processor (167MHz). $\mathcal{MM}$AS-t+TS took on average 7.4 seconds to solve these instances and Ro-TS could not solve the instances to optimality in all runs and the best solutions in the runs were only found on average after 18.5 seconds. For the instances `taixxb` the tabu search algorithms Ro-TS and RTS perform, apart from the smallest instances, significantly worse than the $\mathcal{MM}$AS hybrids. Only on the `kra30x` and the `ste36x` instances Ro-TS achieves a similar performance as the $\mathcal{MM}$AS hybrids. The very good performance of $\mathcal{MM}$AS+2-opt is also backed up by the fact that we were able to find a new best solution for the largest QAPLIB instance `tai256c` with an objective function value of 44759294 [134].

Interestingly, using a simple local search procedure is sufficient to yield very high quality solutions on the structured instances. Hence, for these instances it seems to be better to apply more often a local search procedure in order to identify promising regions of the search space. In fact, the ACO algorithms are able to exploit the structure of the real life (like) QAP instances and are able to guide the local search towards high quality solutions. Once such a promising region is identified, it is rather easy to find very high quality solutions. For example, for the instances `taixxb` with $n \leq 60$ in almost every run the best-known solutions – conjectured to be optimal – are found. The computational result presented also suggest that $\mathcal{MM}$AS-w performs better than $\mathcal{MM}$AS-t. Yet, as detailed before, $\mathcal{MM}$AS-w was stopped after a given

time limit and could not perform the maximally allowed 1000 local search applications in the given computation time. $\mathcal{MM}$AS-t already terminates at roughly 75% of that time on the larger instances. On a time equalized basis $\mathcal{MM}$AS-w performs still somewhat better on the instances of class (iii) and (iv), but the performance gap between $\mathcal{MM}$AS-w and $\mathcal{MM}$AS-t gets smaller.

One might conjecture that the flow (distance) dominance could be used to identify which algorithm should be applied to a particular instance. If the flow and distance dominance are low, the best choice appears to be to apply algorithms using tabu search like $\mathcal{MM}$AS+TS or Ro-TS, while for high flow and/or distance dominance, the best would be to apply a hybrid algorithm with a fast local search. Although such a simple rule would work reasonably well, exceptions occur. For example, although instance `ste36a` has the highest flow dominance among the real-life instances, $\mathcal{MM}$AS+TS and even Ro-TS give slightly better average performance than $\mathcal{MM}$AS+2-opt. Thus, using the flow dominance alone is not a reliable predictor regarding to which algorithm will perform best.

4.4.5 Related Work

The first application of an ACO algorithm to the QAP has been presented by Maniezzo, Colorni, and Dorigo [165]. They applied a direct extension of ant system enhanced with local search to improve solutions. On some small instances good results have been obtained, yet the computational results presented in that report are rather limited. Considering the computational results of ant system for the TSP, it is also very questionable whether this approach scales well to larger instances like those used in our experiments. Recently, Maniezzo and Colorni [164] and Maniezzo [163] have presented an improved version of this ant system application using a `2-opt` local search as we do. They introduce an interesting new feature which is the calculation of the heuristic information based on lower bounds for the completion of a solution. Unfortunately, an exact comparison between the $\mathcal{MM}$AS approach and the second, better performing approach by Maniezzo [163] is difficult due to the different stopping criteria used (in [164, 163] the algorithms are run on instances with $n \leq 40$ allowing a maximal computation time of 10 minutes for each instance).

Another ant-based approach to the QAP, called HAS-QAP, has been proposed in [85]. HAS-QAP differs significantly from $\mathcal{MM}$AS and other ACO algorithms, because the pheromone trails are not used to construct solutions but to modify an ant's current solution, similar in spirit to the kick-moves in iterated local search. For the local search the authors propose a fast first-improvement local search with maximally two complete neighborhood scans. Because the experimental set-up for HAS-QAP and $\mathcal{MM}$AS is quite similar, it is also possible to compare the two approaches. In general, the behavior of the two algorithms is similar. On structured instances, both perform better than the tabu search algorithms Ro-TS and RTS (and also significantly better than the best simulated annealing algorithm [51] as shown in [85]), but for the unstructured instances HAS-QAP and $\mathcal{MM}$AS+2-opt perform worse than the other algorithms. Nevertheless, the performance of $\mathcal{MM}$AS+2-opt seems to be better for the larger structured instances as well as on the unstructured instances of `taixxa`. As shown before, on these later instances, $\mathcal{MM}$AS+TS performs at least as good as Ro-TS and significantly better than HAS-QAP. Interestingly, the performance of $\mathcal{MM}$AS+TS is comparable to the results of a reimplementation of a genetic hybrid algorithm [74], which was previously shown to be one of the best algorithms for QAP instances from classes (iii) and (iv). The better performance

of $\mathcal{MM}$AS+2-opt on many of these instances suggests that the most important contribution of adaptive algorithms is the ability to extract the structure and to find promising regions of the search space.

For the QAP, two other very promising approaches have been presented recently. In [173] a genetic local search algorithm for QAPs is exposed, extending earlier work of the authors on the TSP. Their approach is in the same spirit as our application of $\mathcal{MM}$AS to QAPs, in the sense that they also adapt their successful genetic local search algorithm, initially designed to attack the TSP, to the QAP using only minor problem specific modifications. A new feature they present is a fast local search which dynamically restricts the size of the neighborhood based on the distance between individuals. Especially for larger problems they obtain very good results with rather low computation times. Another new approach to the QAP is based on Scatter Search. This algorithm uses tabu search runs of various lengths (therefore the results are not directly comparable to ours) as the local search operator [56]. They obtain very high solution quality but apparently at the cost of rather high run times.

4.4.6 Conclusions

In this section we have presented an extension of $\mathcal{MM}$AS to the QAP. We have shown that for the QAP the local search method, which is combined with $\mathcal{MM}$AS in a hybrid algorithm, has a strong impact on the performance. We considered two possibilities, one is the application of short tabu search runs, the other is the use of a simple 2-opt iterative improvement local search. For unstructured QAP instances, the use of short tabu search runs has shown to be particularly effective, while for structured and real-life instances $\mathcal{MM}$AS is better combined with a simple 2-opt algorithm. This result suggests that the best possible combination of an adaptive construction heuristic (like $\mathcal{MM}$AS) with local search into a hybrid algorithm depends strongly on the QAP instance type. The comparison of our algorithm with other very good algorithms for the QAP has shown that $\mathcal{MM}$AS is a currently among the best heuristics for structured, real-life QAP instances.

4.5 $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System for the Flow Shop Problem

In this section we present the application of $\mathcal{MM}$AS to the permutation flow shop problem (FSP), a well known scheduling problem [69] which has been attacked by many different algorithmic approaches. It is the first application of an ACO algorithm to this problem and with this application we widen the range of problems which have been successfully attacked by ACO algorithms. We focus on the performance of $\mathcal{MM}$AS for rather short runs, that is, we try to find good solutions as fast as possible. Therefore, we focus on the application of variant $\mathcal{MM}$AS-t (called simply $\mathcal{MM}$AS in the following), since this version achieves faster convergence of the algorithm. Additionally, two new aspects are introduced in this application. One is the use of only a single ant in the $\mathcal{MM}$AS algorithm and the other is the pheromone trail initialization by a heuristic function.

4.5.1 The Permutation Flow Shop Problem

The FSP can be stated as follows: Each of n jobs $1, \ldots, n$ have to be processed on m machines $1, \ldots, m$ in the given machine order. The processing time of job i on machine j is t_{ij}; the processing times are fixed, nonnegative, and may be 0 if a job is not processed on some machine. Further assumptions are that each job can be processed on only one machine at a time, the operations are not preemptable, the jobs are available for processing at time zero and setup times are sequence independent. Here we consider the permutation flow shop problem, in which the same job order is chosen on every machine. The objective is to find a sequence, that is, a permutation of the numbers $1, \ldots, n$ that minimizes the completion time (also called the makespan) of the last job. The problem is $\mathcal{NP}$-hard and only some special cases can be solved efficiently [131]. Therefore, many algorithms have been proposed to find near optimal schedules in reasonable time. These algorithms can be classified as either constructive or based on local search. Several constructive heuristics have been proposed for the FSP [42, 57, 188]; of these the NEH heuristic [188] seems to be the best performing for a wide variety of problem instances. The local search algorithms include simple iterative improvement algorithms [57] and metaheuristics like simulated annealing [193, 192], tabu search [207, 191, 240, 261], and genetic algorithms [208]. A path algorithm approach based on a FSP specific neighborhood structure is presented in [260].

4.5.2 Applying $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System to the FSP

The FSP is like the TSP a permutation problem in which a permutation of the integers $\{1, \ldots, n\}$ has to be found such that some objective function is minimized. A difference between the two problems is that in the TSP only the relative order of the cities is important; the permutation $\pi_1 = (1\ 2\ \ldots\ n)$ represents the same solution as the permutation $\pi_2 = (n\ 1\ 2\ \ldots\ n-1)$, giving the same objective function value. Yet, in the FSP the absolute position of the jobs is of importance, that is, the permutations π_1 and π_2 represent different solutions with, most probably, strongly different makespan.

Definition of pheromone trails

Since in the FSP the absolute position of a job is of importance, this has to be reflected in the definition of the pheromone trails. In our approach we use the absolute position of a job as a solution component. Hence, the pheromone trails τ_{ij} represents the desire of setting job i on the jth position in a sequence. In a first approach we used the pheromone trails as in the TSP, that is, with τ_{ij} meaning that job j should be scheduled after job i. Yet, some preliminary experiments showed that the performance with such a trail definition is rather poor. This observation is backed up by a recent fitness distance correlation analysis of the FSP search space [209] (see Section 2.9 for a discussion of fitness distance correlation). There it was shown that with a distance measure based on job positions, a significantly higher correlation between makespan and distance to the optimal solution could be found than with an adjacency metric which counts the number of times a pair of jobs is adjacent in two solutions. This indicates that if the search takes into account the positions of jobs, the solution quality correlates better to the optimal solution and the search for better solutions should be easier. Hence, this also gives an indication why a position-based definition of the pheromone trails improves performance.

Construction of solutions

For the construction of a sequence we introduce a dummy job 0 on which the ants are set initially. The ants construct a sequence by first choosing a job for the first position, then a job for the second position and so on until all jobs are sequenced. We will use the pseudo-random-proportional rule for the solution construction (see Equation 4.5). In the pseudo-random-proportional rule an ant makes with a probability p_0 the best possible decision, that is, it chooses for the current position a not yet sequenced job with maximal τ_{ij}; with probability $1 - p_0$ an ant k chooses for position j a job according to the following probability distribution

$$p_{ij} = \begin{cases} \frac{\tau_{ij}}{\sum_{l \in U(k)} \tau_{lj}} & \text{if job } i \text{ is not yet scheduled} \\ 0 & \text{otherwise} \end{cases} \tag{4.24}$$

where $U(k)$ is the set of not yet sequenced jobs. To keep track of the jobs already used in the partial solution constructed so far, the ants maintain a list in which they store the partial sequence.

Update of Pheromone Trails

After a complete sequence is constructed and improved by a local search, the pheromones are updated according to Equation 4.10. $\Delta\tau_{ij}^{best}$ is defined as

$$\Delta\tau_{ij}^{best}(t) = \begin{cases} 1/f(\pi_{best}) & \text{if job } i \text{ is put at position } j \text{ in } \pi_{best} \\ 0 & \text{otherwise} \end{cases} \tag{4.25}$$

$f(\pi_{best})$ is the makespan of the pheromone trail updating ant which again may be chosen as the iteration-best or the global-best ant. Thus, positions which are often occupied by certain jobs receive a higher amount of pheromone. The trail limits again are chosen in a straightforward way (see Section 4.5.4 for details on the parameter settings).

4.5.3 Local Search for the FSP

For the FSP we have considered iterative improvement algorithms based on the following neighborhood definitions which have been proposed in the literature.

(i) Swaps of two neighboring jobs at position i and $i+1$, $i = 1, \ldots, n-1$ [57] (*swap-moves*),

(ii) exchanges of jobs placed at the ith and the jth position, $i \neq j$ [193, 240] (*interchange-moves*), and

(iii) removal of the job at the ith position and its insertion in the jth position (*insertion-moves*)

Local search based on swap-moves is very fast, yet the solution quality achieved is very low and therefore we did not consider it further. In [240, 193] it was shown that the neighborhood based on insertion-moves can be evaluated more efficiently than the one based on interchange-moves and gives at least the same solution quality. Therefore, we use a local search algorithm based on the insertion neighborhood. The moves are defined as follows. Let (i, j) be a pair of

positions. The new permutation π' is obtained by removing the job $\pi(i)$ at position i and inserting it at position j. If $i < j$ 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 $\pi' = (\pi(1), \ldots, \pi(j-1), \pi(i), \pi(j), \ldots, \pi(i-1), \pi(i+1), \ldots, \pi(n))$. The size of the neighborhood is $(n-1)^2$; using the fast neighborhood evaluation of [240], the set of possible moves can be examined in $O(n^2m)$.

When applied to large FSP instances the computation time for the local search still grows fast and therefore we use a modified first-improvement strategy which resulted in significantly reduced run-times. For a position i we examine all possibilities for inserting job $\pi(i)$ and if an improved schedule is found, we apply the best insertion-move found during the neighborhood scan for job $\pi(i)$. Using the proposed first-improvement pivoting rule, local minima are found much faster than with a best-improvement pivoting rule and in some initial experiments we could verify that, when starting from random initial solutions, for the same amount of computation time the first-improvement local search gives significantly better results than the best-improvement version.

4.5.4 Implementation Details and Parameter Choices

For the sequence construction steps we consider candidate lists if jobs are chosen probabilistically according to Equation 4.1. The candidate lists of size *cl* are defined by an analogy to the global-best sequence π_{gb}. Within the candidate list are the first *cl* jobs of π_{gb} which are not yet sequenced. The next job for a specific position is then chosen among the jobs of the candidate list according to Equation 4.1. We found that $cl = 5$ gives a reasonably good performance.

As we allow only short run times for $\mathcal{MM}$AS, we make the search procedure more aggressive by favoring very strongly the exploitation of π_{gb}. To find good solutions fast, we use the sequence returned by the NEH heuristic [188] as an initial solution and apply local search to this sequence. This sequence is then chosen for the first pheromone update. With this choice our algorithm is guaranteed to perform at least as well as the NEH heuristic. In the subsequent iterations we always choose π_{gb} for the pheromone update. In our algorithm we use only one single ant, that is, in each iteration only one ant constructs a solution and applies local search. This latter parameter setting is motivated by the fact that for most problem instances $\mathcal{MM}$AS can perform only very few iterations and using more ants would reduce the algorithm almost to a multiple descent algorithm. To achieve a high exploitation of π_{gb} we choose p_0 very large, setting it to $\frac{n-4}{n}$, that is, with a very high probability the best possible choice is made. This makes $\mathcal{MM}$AS similar to an iterated local search algorithm [126] since the sequences are constructed in such a way that they present slight changes with respect to the best solutions found so far. The other parameters are set throughout all experiments as follows. The trail limits are chosen as $\tau_{\max} = \frac{1}{(1-\rho)} \cdot \frac{1}{\pi_{gb}}$, and $\tau_{\min} = \tau_{\max}/5$. Initially, the trails are set to their maximally possible value $\tau_{\max}$ and the persistence of the pheromone trails is set to $\rho = 0.75$.

4.5.5 Experimental Results

In this section we present an experimental comparison of the proposed $\mathcal{MM}$AS algorithm to other, well known algorithms for the FSP. In particular, we have implemented and applied the NEH heuristic [188], NEH followed by a subsequent phase of local search (NEH+ls) using the same local search algorithm as $\mathcal{MM}$AS, a simulated annealing approach proposed in

[193] (SAOP), and a multiple descent approach starting from randomly generated solutions (MD). The simulated annealing algorithm of [193] gives a good indication of the performance of simulated annealing on the FSP because it is among the best performing simulated annealing algorithms for the FSP (the latter proposed simulated annealing variant of [125] gives only a very minor improvement in solution quality and for some instances even performs worse than SAOP); the comparison of $\mathcal{MM}$AS to the other algorithms shows the improvement obtained with $\mathcal{MM}$AS over these simpler algorithms (note that NEH is used for the trail initialization and MD uses the same local search as $\mathcal{MM}$AS). For SAOP the maximal number of iterations is predefined as $\max\{3300 \cdot \ln n + 7500 \ln m - 18500, 2000\}$ and for $\mathcal{MM}$AS and MD we allowed the same computation time as taken by the simulated annealing algorithm. Since the resulting time limits are rather low, the iterative improvement algorithm can be applied only a few times by $\mathcal{MM}$AS and MD. The algorithms were run on a standard benchmark set which comprises randomly generated problem instances which have been originally proposed by Taillard [242]. The benchmark instances are generated for different numbers of jobs and machines; the processing times are randomly generated numbers in the interval $[1, ..100]$. For each problem size 10 instances are available.

The computational results are presented in Table 4.15. Note that the computation times for NEH and NEH+ls are much lower than those needed by SAOP and $\mathcal{MM}$AS. As can be observed, $\mathcal{MM}$AS performs best for all problem sizes. A surprising result is that even the MD approach outperforms the simulated annealing algorithm, contradicting computational results presented, for example, in [193]. We could verify that this is due to the first-improvement pivoting rule used in our local search algorithm and the speed-up techniques applied in the implementation of the local search algorithm. Note, that the first-improvement and the best-improvement version of our local search based on insertion-moves yield roughly the same solution quality, on average, but the first-improvement approach allows to apply much more often a local search and so better results are obtainable. If using a best-improvement pivoting rule the results for MD are significantly worse than those of SAOP for most problem instances. Thus, an important part of the success of $\mathcal{MM}$AS is due to the fast first-improvement local search procedure. The initialization of the pheromone trails, as discussed in Section 4.5.4, seems to be especially helpful for the larger instances with $n \geq 100$. We verified that for the smaller FSP instances this kind of trail initialization does not improve the solution quality obtained by $\mathcal{MM}$AS.

The best performing heuristic algorithm for the FSP is the sophisticated tabu search algorithm of Nowicki and Smutnicki [191], which outperforms our $\mathcal{MM}$AS algorithm with respect to computation time and solution quality produced. Yet, this tabu search algorithm is fine-tuned to the FSP and exploits special features of the FSP to speed up the local search. In contrast, our approach is only a minor variation of an earlier application of $\mathcal{MM}$AS to the TSP using a rather straightforward local search implementation.

4.5.6 Conclusion

We have introduced a new application of ant algorithms to a classical scheduling problems, the permutation flow shop problem. This application, based on $\mathcal{MAX}$–$\mathcal{MIN}$ ant system, introduces new aspects like the use of a single ant and the trail initialization by a heuristically generated solution. We compared the algorithm to local search algorithms and construction

Table 4.15: Given are the results of $\mathcal{MM}$AS applied to FSP instances. The maximally allowed run times (indicated by t_{max}) is given as seconds on a Sun Sparc 5 Workstation for SAOP, $\mathcal{MM}$AS and MD; NEH and NEH+ls need less computation time than the other algorithms. The results are given as the percentage excess over the best known solutions averaged over 10 instances of each size. The size is indicated as $(n \times m)$, where n is the number of jobs and m the number of machines. The best results are indicated in **boldface**.

instances	NEH	NEH+ls	SAOP	$\mathcal{MM}$AS	MD	t_{max}
ta001 - ta010 (20×5)	3.300	1.687	1.061	**0.408**	0.765	0.3
ta011 - ta020 (20×10)	4.601	2.451	1.462	**0.591**	0.955	1.3
ta021 - ta030 (20×20)	3.731	2.418	1.116	**0.410**	0.628	3.3
ta031 - ta040 (50×5)	0.727	0.261	0.597	**0.145**	0.255	0.9
ta041 - ta050 (50×10)	6.453	3.456	3.012	**2.122**	2.873	3.3
ta051 - ta060 (50×20)	5.971	4.658	3.533	**2.855**	3.524	9.0
ta061 - ta070 (100×5)	0.527	0.324	0.509	**0.196**	0.219	1.9
ta071 - ta080 (100×10)	2.215	1.250	1.720	**0.896**	1.287	7.0
ta081 - ta090 (100×20)	5.106	4.089	4.076	**3.138**	3.822	20.5

heuristics proposed for the FSP. Among the algorithms compared the ant algorithms obtains the best results. Yet, it cannot compete with a specialized tabu search algorithm designed specifically for the FSP [191]. Despite this fact the computational results are encouraging enough to suggest that ant algorithms can be applied successfully also to more complex scheduling problems.

4.6 Parallelization Strategies for Ant Colony Optimization

Ant colony optimization is a population-based search metaphor and is as such inherently parallel. Each individual ant can be conceived as a computational agent which tries to solve a problem. Hence, one straightforward possibility would be to assign each ant to a single processor. Yet, such a parallelization scheme has the disadvantage that the communication overhead between the single processors would be extremely high, since the agents communicate indirectly via pheromone trails which would have to be exchanged between the processors. This example shows that there is no golden rule for parallelizing ACO algorithms and an efficient parallelization of ACO algorithms depends strongly on the available computing platform and the problem to which the algorithm is applied. Today a very common way to parallelization is given by MIMD architectures as given by, for example, a cluster of workstations. In our discussion we concentrate on parallelization possibilities in such an environment. In particular, we motivate and investigate the execution of multiple independent runs of ACO algorithms. We show that high quality solutions can be achieved by such an approach presenting indicative computational results for the $\mathcal{MM}$AS applications to the TSP. Additionally, the investigation of the effectiveness of multiple independent runs led us to the design of an improved sequential $\mathcal{MM}$AS-version which is presented in Section 4.6.3.

4.6.1 Parallelization Strategies

Parallel Runs of One Algorithm

The simplest way to obtain a parallel version of an algorithm is the multiple independent execution of the sequential algorithm on k processors and take as the final result the best one obtained on any of the processors. Using multiple independent runs is appealing because basically no communication overhead is involved and nearly no additional implementation effort is necessary. Of course, independent runs of an algorithm in parallel is only useful if the underlying algorithm is randomized, that is, if the search process relies on random decisions. ACO algorithms and $\mathcal{MM}$AS, in particular, are such algorithms.

As argued in Chapter 3, the run-time distributions arising from the application of probabilistic algorithms provide important hints on the effectiveness of a parallelization based on multiple independent runs. (Recall that there we have shown that when the empirical run-time distribution can be closely approximated by an exponential one, then the parallelization by multiple independent runs is extremely efficient: the speed-up will be close to optimal.) Therefore, we have measured run-time distributions arising from the $\mathcal{MM}$AS application to the TSP and give two examples. In Figure 4.5 are given the run-time distributions for the application of $\mathcal{MM}$AS to instance `lin318` (left side) and instance `rat783` (right side) for finding the optimal solution, a solution within 0.1% and within 0.25% of the optimum on instance `lin318` and for finding the optimum on instance `rat783`, respectively. Additionally, we plotted exponential distributions (indicated by $f(x)$) which give an indication of possible stagnation behavior of the algorithm (recall that stagnation can be observed if the empirical run-time distribution falls below the plotted exponential distribution; for more details we refer to the discussion in Section 3.2.3 of Chapter 3). In fact, the empirical run-time distributions of $\mathcal{MM}$AS show such a stagnation behavior and we can conclude that, as argued in Chapter 3, in the simplest case the sequential algorithm can be improved by occasionally restarting it from scratch or, equivalently, by running multiple copies of the algorithm in parallel. This latter fact is illustrated by the following numerical example. The run-time distribution for finding the optimal solution to instance `lin318` reaches a solution probability of $\widehat{G}(60) = 0.57$ after 60 seconds. If five independent runs are performed in parallel, we would empirically obtain a probability of 0.985 of finding the optimal solution (this corresponds to the probability that at least one of the five independent runs finds the optimum). Using the same overall run-time in the sequential algorithm (without restarts), we empirically observe a probability of $\widehat{G}(300) = 0.88$. Consequently, by using multiple independent runs, the probability of finding the optimal solution (in the same overall run-time) even increases with respect to the sequential version.

It is also noteworthy that an initialization time t_{init} is needed until $\mathcal{MM}$AS has a reasonable chance of finding a high quality solution. For example, the first optimal solution for instance `lin318` is only found after roughly 24 seconds. This effect limits the obtainable speed-up by performing multiple independent runs. If only very short computation times are allowed, parallelization has to be used to speed up the execution of a single run to get a reasonable solution quality (see the next section for a short discussion of this issue).

Instead of running multiple copies of one algorithm with *one* particular parameter setting independently on k processors, it may also prove advantageous to run one algorithm with different search strategies or different parameter settings in parallel. Such an approach is especially appealing if an algorithm's performance on different problem instances depends on the algorithm's

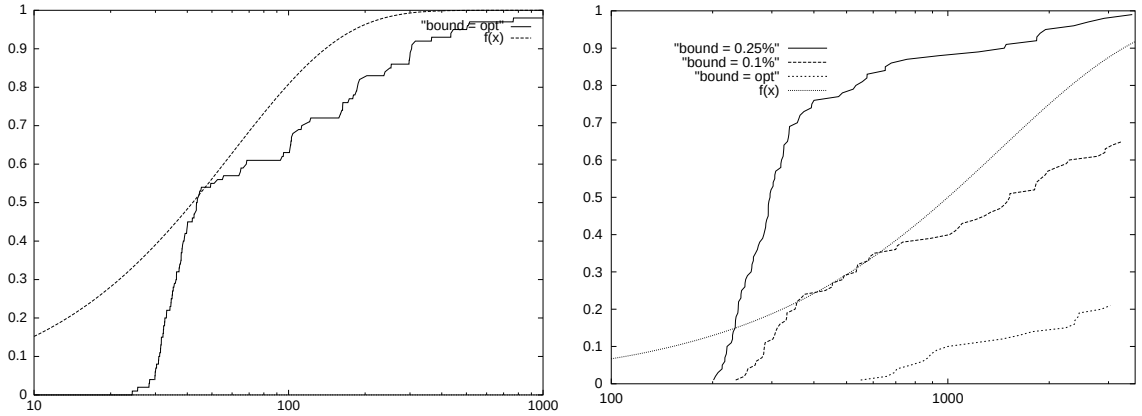

Figure 4.5: Run-time distributions for `lin318` and `rat783`. On the x-axis is given the CPU-time and on the y-axis is given the solution probability of finding a solution with the required quality. The run-time distribution are plotted with respect to the optimal solution for `lin318` and for bounds of 0.25%, 0.1% and the optimal solution for instance `rat783`. Both distributions are based on 100 independent runs of $\mathcal{MM}$AS. See the text for further details.

parameter settings. In the application of $\mathcal{MM}$AS to the TSP, we did not notice strong dependencies of the algorithm's parameter settings on particular problem instances; one fixed setting appeared to perform well on a large set of instances. Yet, running several $\mathcal{MM}$AS variants which use different local search algorithms in parallel should prove successful for applications to the quadratic assignment problem, because for this case the algorithm's performance for an instance type depends strongly on the local search algorithm used to improve the ants' solution (see Section 4.4).

Still more improvement over using multiple independent runs may be gained by cooperation between the ant colonies assigned to the single processors. Such a cooperation among the ant colonies could be implemented by exchanges of ants with very good solutions between different ant colonies, similar to parallel approaches for genetic algorithms [43]. These ants then may be used to update the pheromone trails of other ant colonies. Other possibilities would be to exchange or combine whole pheromone matrices among the ant colonies to influence the search direction of the others. A study of these possibilities is intended for future research.

Speeding up a Single Run

If only very low computation time is available, another possibility of parallel processing is to speed up a single run of an algorithm. A first implementation of ant system on a Transputer and a connection machine has been presented in [28]. In [37] parallelization strategies for ant system have been discussed, presenting synchronous and asynchronous master-slave schemes for ant system. Here, we extend the work of [37] by considering alternative schemes of speeding up runs of ACO algorithms like ant colony system [66] or $\mathcal{MAX}$–$\mathcal{MIN}$ ant system [236] which rely strongly on local search applications.

If local search algorithms are used to improve solutions, a master-slave approach could work as follows. One master processor is used to update the main data structures for the ACO

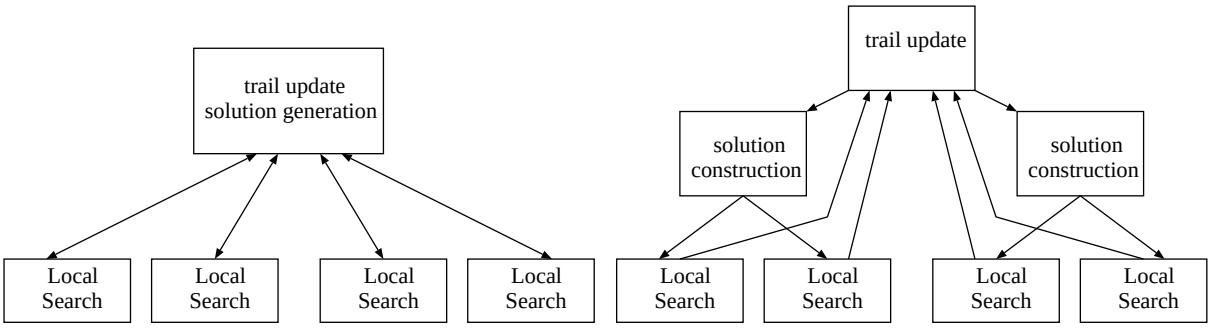

Figure 4.6: Parallelization by a master–slave approach. See the text for more details.

algorithm, to construct initial solutions for the local search algorithms, and to send the solutions to other processors which in turn improve them by local search. The master then collects these locally optimal solutions as they become available. If a sufficient number of such solutions has arrived, the master updates the pheromone trail matrix before constructing more solutions. This situation, also implementable in asynchronous mode, is depicted in Figure 4.6 on the left side. Such a parallelization scheme is particularly interesting if the update of the trails and the construction of solutions consumes much less time than the local search. This, for example, is the case when applying ACO algorithms to the quadratic assignment problem [165, 232, 85]. In this case the construction of solutions and the update of the trail matrix is of complexity $O(n^2)$, whereas the local search is of complexity $O(n^3)$. In fact, an analysis of the sequential algorithm $\mathcal{MM}$AS implementation has shown that roughly 99% of the time is spent by the local search procedure. Yet, for the application of $\mathcal{MM}$AS or ACS to the TSP the situation is different. For the TSP the local search runs very fast using the implementation tricks described in [21]. When using `3-opt` local search only roughly 75% of the time is spent by the local search, around 15% of the time is spent constructing tours and approximately 10% of the time is taken by the pheromone trail update. This situation is less favorable, if `2-opt` local search is used, only when using the `LK` heuristic the run-time data would make the above described approach interesting. If the local search is not very run-time intensive, a situation like that depicted in Figure 4.6 on the right side might be preferable for a parallel implementation. One processor keeps the main data structures for the trail matrix and updates it. One or several other processors can use the trail matrix to construct solutions and send these to processors which apply local search. The improved solutions are then sent back to the main processor which in turn updates the pheromone trails. Again, communication can be done in asynchronous mode. A major disadvantage of such an approach is that communication has to take place frequently. In general, the obtainable speed-up by such an architecture will be suboptimal due to the communication overhead involved.

4.6.2 Experimental Results

In this section we present computational results for the execution of multiple independent runs of $\mathcal{MM}$AS with up to 10 processors. It is difficult to calculate the exact speed-up for multiple independent runs, because the solutions quality has to be taken into account. To circumvent

Table 4.16: Performance of multiple independent runs for $\mathcal{MM}$AS on symmetric TSP instances. $t_{\max}$ is the maximally allowed time for the sequential algorithm, computation times refer to a UltraSparc I processor (167MHz). The results for instances with less than 500 cities are obtained using 10 ants, for the other instances 25 ants are used. Results are given for 1 (sequential case) up to 10 multiple independent runs. Given is the average solution quality measured over 10 runs (except for `d1291`, where only 5 runs were done), the best average solution quality is indicated in bold face.

instance	1	2	4	6	8	10	t_{max}
`d198`	15780.3	15780.0	15780.1	15780.0	15780.0	15780.0	300
`lin318`	42029	42029	42029	42029	42029	42029	450
`pcb442`	50886.5	50873.0	50875.3	**50852.7**	50862.9	50860.6	900
`att532`	27707.4	27702.5	27702.1	27703.5	27702.3	**27699.0**	1800
`rat783`	8811.5	8810.8	**8809.5**	8810.6	8810.8	8813.1	3600
`pcb1173`	56960.3	56960.9	56922.4	**56912.6**	56929.7	56969.1	4500
`d1291`	50845.8	**50809.0**	50821.8	50825.2	50826.6	50830.0	4500

this problem we concentrate on the following experimental setting. We compare the average solution quality of $\mathcal{MM}$AS running k times the algorithm in parallel for time limit t_{max}/k (using different numbers k of processors) to that of a single run of time t_{max}. This experimental setup gives the same overall CPU-time to the sequential version and the parallel versions. As explained in Section 4.6.1, for very short run-times the solution quality for $\mathcal{MM}$AS will be rather poor due to the initialization time t_{init} which increases with increasing instance size. To account for this fact, the computation times were chosen in such a way that $t_{max}/k \geq t_{init}$.

The experimental results for different numbers k of processors are given in Table 4.16. On instance `lin318`, which is regularly solved to optimality for all values of k, the effect of using multiple independent runs can be noted in the reduction of the average time needed to find the optimal solution. For example, with 6 independent runs, the average time to find the optimum is 22.0 seconds as opposed to 132.7 for the sequential version.[10] Thus, the obtained speed-up would be $t_1/t_6 = 6.03$ (for 10 independent runs the speed-up would be 6.7). For larger problem instances, the best average results are always obtained with $k \geq 2$ runs of the algorithm. The results also show the desirability of parallelization by multiple independent runs: By parallel processing the user can expect to obtain in shorter (user) time the same or even better quality solutions at virtually no additional implementation effort.

Note that our experimental setting actually can be used to address an often posed question in the research on parallelization possibilities for simulated annealing algorithms:indexsimulated annealing Is it preferable to execute one long simulated annealing run of time $t_{\max}$ or is it better to execute k short runs of time $t_{\max}/k$ [62]? When posing this question for $\mathcal{MM}$AS, the answer is that, in fact, it is preferable to excute more short runs than one long run of the $\mathcal{MM}$AS algorithm.

[10] The empirical run-time distribution for $\mathcal{MM}$AS on instance `lin318` (given in Figure 4.5) has been obtained using 25 ants, while in the experiments reported here only 10 ants are used. This also explains why in the study of the run-time distributions the first optimal solution has been found after only 24 seconds.

Table 4.17: Performance of a sequential version of $\mathcal{MM}$AS using soft restarts. Averages are taken over 25 runs for instances with less than a 1000 cities, over 10 runs otherwise. Given are the best solution found, the average tour lenght (the percentage deviation from the optimum is given in parenthesis) and the worst solution found. Additionally, we give the average time (t_{avg}) and the average number of iterations to find the best solution in a run (i_{avg}), and the maximally allowed computation time (t_{max}).

instance	opt	best	average	worst	i_{avg}	t_{avg}	t_{max}
d198	15780	15780 (25/25)	15780.0 (0.0)	15780	246.0	110.1	600
lin318	42029	42029 (25/25)	42029.0 (0.0)	42029	131.2	87.8	600
pcb442	50778	50778 (2/25)	50868.7 (0.18)	50912	902.3	539.3	900
att532	27686	27686 (1/25)	27700.0 (0.050)	27706	739.5	791.4	1800
rat783	8806	8806 (8/25)	8808.9 (0.033)	8816	1230.9	1879.7	3600
pr1002	259045	259167 (1/10)	259511.3 (0.18)	260069	1624.3	2924.7	3600
u1060	224094	224342 (1/10)	224642.9 (0.24)	225022	1432.0	2841.3	3600
pcb1173	56892	56892 (1/10)	56906.8 (0.026)	56939	1697.2	3171.2	5400
d1291	50801	50801 (3/10)	50812.9 (0.023)	50909	1747.8	3203.7	5400
fl1577	22249	22289 (1/10)	22305.6 (0.25)	22323	1681.7	3203.7	7200

4.6.3 Improved Sequential $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

Interestingly, the analysis of the run-time distributions arising from the $\mathcal{MM}$AS application to the TSP reveals that the sequential algorithm can, in principle, be improved by restarts. Yet, restarts have the disadvantage that hard cutoff criteria have to be used. As mentioned in Chapter 3, a better idea may be to use soft cutoff criteria related to the search progress of the sequential algorithm. The simplest such criterion is to restart the algorithm if for a certain number of iterations no improved solution could be found. To do so, in $\mathcal{MM}$AS we apply the following rule. If the algorithm has converged, as indicated by the average branching factor $\bar{\lambda}$, *and* no improved solution could be found for 50 iterations, the algorithm is restarted from scratch. Since we use $\mathcal{MM}$AS-w, the schedule for the frequency u_{gb} of the global-best update (see page 82) is applied. Clearly, after the restart of the algorithm the single runs are treated as independent runs and the schedule for the global-best update refers only to the best solution found since the restart. To avoid loosing track of the best solution over all restarts (denoted as s^*_{gb}), we will exploit this solution in the following way. If in the schedule for u_{gb} the restart-best solution is chosen in every iteration, that is, the iteration counter is larger than 250, and for 25 iterations no improved solution has been found, we use solution s^*_{gb} for the pheromone trail update.

The computational results with the soft restart mechanism are given in Table 4.17. We run $\mathcal{MM}$AS on the instances used for the empirical evaluation of $\mathcal{MM}$AS in Section 4.3 but with somewhat longer run-times for the smaller instances. Running the standard algorithm of Section 4.3 for the longer time limits has shown that the computational results presented here could not be reached. For example, using the same time limits as applied here, $\mathcal{MM}$AS-w averages 0.072% and 0.071% above the optimal solution on instances att532 and rat783, respectively. The maximal time allowed for the larger instances with more than a 1000 cities is the same and also for these instances the computational results with the soft restart mechanism in Table 4.17 show a considerable further improvement over those given in Tables 4.8 and 4.9 (pages 89 and 90) for most instances.

4.6.4 Related Work

Parallelization strategies are widely investigated in the area of evolutionary algorithms (see [43] for an overview). In [224] the performance of multiple independent runs of genetic algorithms has been investigated theoretically and for some test problems "super-optimal" speed-up could be observed. Most parallelization strategies for genetic algorithms can be classified into *fine-grained* and *coarse-grained* approaches. Characteristic of fine-grained approaches is that very few (often only one) individuals are assigned to one processor and individuals are connected by a *population-structure*. An example of an implementation of such an approach for the TSP is ASPARAGOS [101] that was able to find very good solutions to the TSP instance `att532`. A typical example of a coarse grained approach is the *island*-genetic algorithm in which the population of a genetic algorithm is divided into several subpopulations. Communication takes place by exchanging individuals at certain time-points among the subpopulations. The parallelization schemes we have discussed for $\mathcal{MM}$AS have strong similarities to the coarse grained approaches; the subpopulations in the genetic algorithm case would correspond to the colonies assigned to single processors in the $\mathcal{MM}$AS case.

Using independent runs is also studied for simulated annealing. The question to answer is whether it is profitable to execute one long run or in the same time several short runs [62]. For an introduction to other concepts of parallel simulated annealing we refer to [2]. Similarly, multiple independent runs are also investigated for tabu search. In [241] it is argued that in case the run-time to find an optimal solution to QAPs is exponentially distributed, optimal speed-up using independent runs of the algorithm can be obtained. In fact, empirical evidence is given that the run-time distribution is very close to an exponential distribution. In [247] multiple independent runs of tabu search are investigated for the job-shop scheduling problem. A classification of parallel tabu search metaheuristics is presented in [53].

Early work on parallel implementations of ant system for the TSP has been presented in [28]. Results with a fine-grained implementation assigning each single ant to one processor have shown that the communication overhead led to a poor scaling behavior. Better results are obtained with a coarse grained implementation by dividing the colony into several subcolonies and only exchanging information after each subcolony had completed a full iteration. In [37] synchronous and asynchronous master-slave schemes for ant system are proposed. In the synchronous case the slaves are used to construct tours while the master updates the pheromone trails and keeps the main data structures. In the asynchronous case subcolonies perform, independently of other processors, a certain number of iterations of the sequential algorithm. Only after the local iterations a global update takes place. An evaluation of the algorithms by simulation has shown that the asynchronous algorithm performs significantly better with respect to speed-up and efficiency. An island-based implementation similar to those for genetic algorithms has been used in the application of ant system to the shortest supersequence problem [177]. Every colony of ants performs, independently from the others, the sequential algorithm and after a certain number of iterations the best solutions are exchanged among the colonies.

4.6.5 Conclusions

For many modern randomized algorithmic approaches to combinatorial optimization, parallelization strategies have been examined. The parallelization strategy to be used depends on the particular problem one has to solve and on the available hardware. Of special appeal is the

possibility to execute multiple independent runs for randomized algorithms like ant colony optimization approaches because basically no additional implementation effort has to be invested and the communication between processors is the lowest possible. The analysis of the run-time distributions of $\mathcal{MM}$AS and the experimental results have shown that such a simple parallelization scheme can be very efficient for a reasonable number of processors. More elaborate mechanisms are only justified if they give better performance than mutliple independent runs.

One line of possible future research is to investigate the effectiveness of parallel ACO algorithms on other problems like the QAP. Of even greater interest is the investigation of the benefit of using cooperation between the colonies. Here, previous investigations on fine-grained parallelization of genetic algorithms and the island-model seem to suggest that still some performance improvement can be obtained.

4.7 Related Work on ACO for combinatorial optimization problems

The initial work on ant colony optimization is ant system and was presented in 1991 by Dorigo, Maniezzo and Colorni [67] who applied it to the TSP. Since then several algorithmic improvements and new applications of ACO algorithms have been proposed. The algorithmic improvements, all of which have been proposed using the application to the TSP, already have been presented in the first section of this chapter. The ant colony metaheuristic, which embeds the proposed variants in a more general framework, has very recently been proposed by Dorigo, Di Caro, and Gambardella [65].

The most widely studied problems using ant algorithms are the traveling salesman problem and the quadratic assignment problem. We refer to Sections 4.1 and our experimental studies where the previously proposed ant algorithms for the TSP are compared. For related work on ant algorithms for the QAP we refer to Section 4.4.5.

Research on ant algorithms has led to a number of other successful applications to combinatorial optimization problems. The sequential ordering problem is closely related to the asymmetric TSP, but additionally precedence constraints between the nodes have to be satisfied. Gambardella and Dorigo have tackled this problem by an extension of their ant colony system enhanced by a local search algorithm [84]. They obtain excellent results and were able to improve many of the best known results for a standard benchmark set. A first application of ant system to the job-shop-scheduling problem has been presented in [49]. Despite showing the viability of the approach, the computational results are not competitive with state-of-the-art algorithms. Costa and Herz have proposed an extension of ant system to assignment type of problems. They present an application of their approach to the graph coloring problem obtaining comparable results to other algorithms. Applications of the rank-based version of ant system to the vehicle routing problem are presented in by Bullnheimer, Hartl, and Strauss [35, 34]. They obtained good computational results for standard benchmark instances. A further new application of ant system to the shortest common supersequence problem has been proposed by Michel and Middendorf [177]. They introduce the novel aspect of using a lookahead function during the solution construction by the ants. Additionally, they present a parallel implementation of their algorithm based on an island model which is frequently used as a model for parallelizing

genetic algorithms. $\mathcal{MM}$AS has recently been applied to the generalized assignment problem by Ramalhinho Lorençou and Serra [158], obtaining very promising computational results. In particular, their algorithm is shown to find optimal and near optimal solutions faster than a GRASP algorithm which was used for comparison.

Applications of ant colony optimization to communication networks, in particular to network routing problems, have recently received a strongly increased interest in the ACO community. The application of ant algorithms to network routing problems is appealing, since these problems have characteristics like distributed information, non-stationary stochastic dynamics, and asynchronous evolution of the network status. These problem characteristics are well matched by the properties of ACO algorithms which use local information to generate solutions, indirect communication via the pheromone trails and stochastic state transitions. Yet, these applications differ rather strongly from ACO algorithms for static combinatorial optimization problems, which has been the target of this chapter, and we do not go into a detailed discussion of these approaches. We refer to the recent article by Dorigo, Di Caro, and Gambardella [65] for a very good overview of recent developments concerning these applications.

4.8 Summary

Recent research in ACO algorithms has focused on improving the performance of ACO algorithms. In this chapter we have presented $\mathcal{MAX}$–$\mathcal{MIN}$ ant system based on several modifications to AS which aim (i) to exploit more strongly the best solutions found during the search and to direct the ants' search towards very high quality solutions and (ii) to avoid premature stagnation of the ants' search. We have justified these modifications by a computational study of $\mathcal{MM}$AS and have shown that all main modifications are important for obtaining peak performance.

We have shown the excellent performance of $\mathcal{MM}$AS applying it to three standard benchmark problems which are the traveling salesman problem (TSP), the quadratic assignment problem (QAP) and the flow shop scheduling problem (FSP). When applied to the TSP, our results demonstrate that $\mathcal{MM}$AS achieves a strongly improved performance compared to AS and to other improved versions of AS. Furthermore, $\mathcal{MM}$AS is among the best available algorithms for the QAP, which is backed up by the fact that we were able to find a new best solution for the largest benchmark instance `tai256c` [134]. The application of $\mathcal{MM}$AS to the FSP has introduced new aspects like the use of only one single ant and the trail initialization by a heuristic; additionally the computational results have been very good and show the large potential of applications of ACO algorithms to scheduling problems.

Research on ant colony optimization is still in its infancy, but already significant progress has been made and the computational results obtained with $\mathcal{MM}$AS witness the desirability of the approach. It is our hope that our research on $\mathcal{MAX}$–$\mathcal{MIN}$ ant system ultimately improves the performance and the applicability of ant-based algorithms.

Chapter 5

Iterated Local Search

In this chapter we empirically analyze the run-time behavior of iterated local search (ILS) algorithms applying the empirical methodology based on evaluating run-time distributions which has been presented in Chapter 3. In particular, we will use run-time distributions to reveal situations in which ILS algorithms can be improved. Based on this analysis, new variants of ILS, including population-based extensions, are proposed. This is done using the ILS applications to the traveling salesman problems (TSP), to the quadratic assignment problem (QAP), and to the flow shop problem (FSP). ILS has been applied to the TSP in [169, 168, 126, 129]; we analyze the run-time behavior of these algorithms and, based on this analysis, present substantial improvements on them. This is, in fact, a very significant result because ILS algorithms have previously shown to be among the best known approximate algorithms for symmetric TSPs and their further improvement makes them still more desirable. The ILS applications to the QAP and the FSP are new and by applying ILS to these problems we extend the range of successful ILS applications.

The chapter is structured as follows. In Section 5.1 we give a short outline of an algorithmic framework for ILS and discuss application principles of ILS. The subsequent three sections then present ILS applications to the TSP, the QAP, and the FSP. In each of these three sections we first present the implementation details of the ILS algorithm, then analyze the ILS run-time behavior on standard benchmark instances, and experimentally analyze ILS extensions. We end this chapter with a discussion of related work in Section 5.5 and a short summary of our results.

5.1 Introduction

Iterated local search (ILS) is a simple and generally applicable metaheuristic which iteratively applies local search to modifications of the current search point. At the start of the algorithm local search is applied to some initial solution. Then, a main loop is repeated until some stopping criterion is satisfied. It consists of a modification step (kick-move) which returns an intermediate solution s' corresponding to a mutation of a previously found locally optimal solution. Next, local search is applied to s' yielding a locally optimal solution s''. An acceptance criterion then decides from which solution the search is continued by applying the next kick-move. Both, the modification step and the acceptance test may be influenced by the search history, which is indicated in the algorithmic outline of ILS in Figure 5.1 by *history*. (A general outline of ILS has already been presented in Chapter 2, but to make this chapter more self-contained, we give

procedure *Iterated Local Search*
  generate initial solution s_0
  $s = \mathsf{LocalSearch}(s_0)$
  $s_{best} = s$
  while(termination condition not met) **do**
    $s' = \mathsf{Modify}(s, history)$
    $s'' = \mathsf{LocalSearch}(s')$
    if $(f(s'') < f(s_{best}))$
      $s_{best} = s''$
    $s = \mathsf{AcceptanceCriterion}(s, s'', history)$
  end
  return s_{best}
end *Iterated Local Search*

Figure 5.1: Algorithmic skeleton of an iterated local search procedure (ILS)

an algorithmic outline of ILS in Figure 5.1.) We will later give some examples for simple uses of the search history.

ILS is meant to be a unifying framework for algorithms which rely on the same basic idea. Several such algorithms have been proposed so far. All these are very similar and they differ mainly in particular choices of one of the three main parts of our framework, the LocalSearch, the Modify, and the AcceptanceCriterion procedures. The corresponding algorithms are addressed under different terms like large-step Markov chains (LSMC) [169], chained local optimization (CLO) [168], or variable neighborhood search [180] (VNS) in the literature, here we use iterated local search (ILS) to refer to the general framework.

ILS algorithms are among the most powerful algorithms for some of the studied problems. The best known example is the TSP [20, 169, 129], on which the best implementations of the iterated Lin-Kernighan algorithm, a specific ILS algorithm which uses the Lin-Kernighan (LK) local search procedure, provide the best tradeoff between computation-time and solution quality for short to intermediate run-times [129]. Very good performance is also reported for ILS applications to graph partitioning [167], the job-shop scheduling problem [159], the p-median problem [108], weighted MAX-SAT and various others [32, 33, 50]. Besides the very good performance of ILS algorithms for many problems it is particularly worthwhile to stress the ease of the implementation of ILS algorithms. ILS, in principle can be used to improve on any given local search method and if a local search algorithm for the problem to be solved is available, often only a few lines of code need to be added. Typically, it is rather easy to find an appropriate kick-move; for many applications a randomly chosen move of a higher-order neighborhood than used by the local search algorithm is a good first guess. Additionally, ILS algorithms involve only very few parameters and therefore they are much easier to tune for a specific application than many other methods.

Most applications of ILS focus on aspects of the choice of the local search algorithm and the choice of the kick-moves. Although ILS can be used to improve the performance of any given local search algorithm, it is obvious that ILS efficiency depends strongly on the particular

choice of the local search and its implementation. For example, much improved results for ILS applications to the TSP are reported in [169] when replacing the 3-opt local search algorithm by the LK heuristic. Analogously, in [251] an improved performance with a genetic local search algorithm for the TSP is achieved when replacing the 2-opt local search with the LK heuristic Similarly, in [168] a good choice for the local search algorithm and for the kick-move are stressed. In the same spirit, one of the main ideas of variable neighborhood search (VNS), a recently introduced variation of the ILS scheme, is the systematic variation of the kick-move strength.

The importance of the choice of the acceptance criterion has often been neglected in previous ILS applications and no systematic analysis on it have been carried out. Initially, a simulated annealing type acceptance criterion was used and in [169] an improved performance on one TSP instance, lin318, is reported. Improved solution quality by such an acceptance criterion is also reported in [214] for the iterated-LK (ILK) application to a large TSP instance with 18512 cities. In [157] three different acceptance criteria applying ILS to the job-shop scheduling problem are compared. However, in most ILS applications only better quality solutions get accepted. The following analysis of the run-time behavior of ILS algorithms shows that this latter choice may often lead to stagnation phenomena and that other types of acceptance criteria are better suited to avoid such situations.

5.2 Iterated Local Search for the TSP

In this section we start the analysis of the run-time behavior of ILS algorithms for the TSP which has been the target of a significant amount of research in ILS algorithms. To apply ILS to the TSP the three generic operators Modify, LocalSearch, and AcceptanceCriterion have to be defined. In the following we detail the choices for the three operators for ILS applications to the TSP reported in the literature.

5.2.1 Generic Algorithm Choices for the Symmetric TSP

Choice of LocalSearch

In our empirical investigation of ILS applied to the TSP we use two different local search algorithms. One is the 3-opt implementation we also used in the $\mathcal{MAX}$–$\mathcal{MIN}$ ant system application to the TSP. For a description of the implementation details and the speed-up techniques used (they include a fixed radius nearest neighbor search within candidate lists of the 40 nearest neighbors and don't look bits) we refer to Section 4.3.1 on page 78. Yet, the best performance results in the literature are presented using the iterated LK (ILK) heuristic. Thus, to verify the generality of our findings, we also ran the ILK implementation provided by Olivier Martin on most test problems [166]. An advantage of 3-opt – apart from being much easier to implement [129] – is that its run-times are more robust with respect to the instance type than that of the LK heuristic. Especially, on highly clustered instances the run times of the LK heuristic increase significantly [129]. This fact is also illustrated by the implementations we are using. For example, one LK run on the clustered TSPLIB instance fl1577 takes on average 1200 seconds while on a randomly generated Euclidean instance with 2000 cities one LK run takes on average 15 seconds. In comparison, our 3-opt implementation takes, starting from

an initial tour generated by the nearest neighbor heuristic (see Section 2.6 on page 21), tour 0.6 seconds on the first instance and 1.8 seconds on the 2000 cities instance to find a locally optimal solution.

Choice of Modify

As kick-move we use the so called *double-bridge* move which is the standard choice for ILS applications to the TSP [169, 168, 129]. It cuts the current tour at four randomly chosen edges into four subtours $s_1 - s_2 - s_3 - s_4$ contained in the solution in the given order. These subtours are then reconnected in the order $s_4 - s_3 - s_2 - s_1$ to yield a new starting solution for the local search. The double-bridge move changes four edges and is a higher-order neighborhood move than the one used in `2-opt` and `3-opt`. Furthermore, it is a kind of move that cannot be reversed directly by the `LK` heuristic. In our iterated `3-opt` algorithm the four edges to be removed are chosen randomly. Alternatively, restrictions on the length of the introduced edges are imposed in the original ILK code we are using [169] or newly introduced edges are required to be within a fixed number of the nearest neighbors. The last approach has been shown to be effective on large TSP instances with more than a 10000 cities [214]. In fact, in the ILK code applied in our experiments a newly introduced edge has to be shorter than a constant number (here chosen as 25) times the average length of the edges in the current solution. When running `3-opt` local search we apply randomly chosen double-bridge moves which appeared to give best performance in some initial experiments. Similar experience is reported in [126, 129] for the ILK algorithm.

Choice of AcceptanceCriterion

Common knowledge in iterated local search applications to the TSP appears to be that accepting better quality solutions only gives the best performance of ILS algorithms applied to the TSP. Consequently, we will use this acceptance criterion as the basis of our analysis; we will denote it as *Better*(s, s'') and it is given as

$$\textit{Better}(s, s'') = \begin{cases} s'' & \text{if } f(s'') < f(s) \\ \\ s & \text{otherwise} \end{cases} \tag{5.1}$$

This acceptance criterion is the standard acceptance criterion which is typically used in ILS applications to the TSP and to other combinatorial optimization problems [167, 168, 126, 129, 109, 108]. If side-moves are allowed, that is, a new solution is also accepted if it has the same cost as the current one, we will call the acceptance criterion *BetterEqual*(s, s''). The only modification to *Better*(s, s'') is that in Equation 5.1, a new locally optimal solution is accepted if $f(s'') \leq f(s)$.

Occasionally, better performance using a simulated annealing-type acceptance criterion has been reported. The two examples we are aware of concern the application of ILS to the TSP instance `lin318` in [169] and computational results presented for instance `d18512` in [214]. Yet, in the latter work it is mentioned that to obtain better results with such an acceptance criterion requires a fine-tuned annealing schedule, hence, we do not use it here for the application to the TSP.

Initial Solution

The initial solution in our ILS application is generated by the nearest neighbor heuristic (see Section 2.6). This choice, when compared to starting from random initial solutions, reduces the run-time and yields, on average, better solution quality after the first local search application.

5.2.2 Generic Algorithm Choices for the Asymmetric TSP

When applying ILS to the asymmetric TSP (ATSP), the generic algorithm choices are very similar to those for the symmetric TSP. For the local search we use the `reduced 3-opt` implementation which is already described in Section 4.3.1. For the kick-move, again the double-bridge move can be applied since it does not reverse any subtour. The acceptance criterion and the choice of the initial solution are the same as for the symmetric TSP.

Surprisingly, in the literature no results for the application of ILS to the ATSP are reported. The only ILS application to ATSPs is described in [264] in the context of a specific code optimization problem which is formulated as an ATSP. But, there the ATSP instances are solved via a transformation to the symmetric TSP and then they used an iterated `3-opt` algorithm for symmetric TSPs. Consequently, the results presented here are the first evidence that ILS can be applied with considerable success directly to the ATSP.

5.2.3 Resetting of Don't Look Bits for Iterated `3-opt`

Before presenting our analysis of the run-time distributions arising from ILS applications to the TSP, we discuss a specific implementation aspect concerning the use of the `3-opt` local search algorithm. Both, the `3-opt` implementation as well as the `LK` heuristic use don't look bits to speed up local search (see Section 4.3.1 on page 78 for a description of the use of don't look bits). When using the `LK` heuristic in ILS or genetic local search algorithms some authors report that they restrict the start points of the local search to cities which are endpoints of broken edges or edges not shared by both parents when applying crossover [174, 129]. Doing so, a better tradeoff between computation time and solution quality is reported. In an ILS algorithm such a strategy can be implemented by resetting only the don't look bits of cities incident to broken edges to zero. Yet, it is not obvious whether such a strategy gives the best tradeoff between computation time and solution quality for iterated `3-opt`. If only the don't look bits for endpoints of broken edges are reset to zero, possible improving moves may be missed. Yet, in this case, fewer moves have to be examined and the local search runs faster. Hence, a critical question concerns the tradeoff between *how often a local search algorithm can be applied in a given amount of computation time* and *how effective the local search algorithm should be*.[1] To investigate this issue we propose the following three options.[2]

[1] Obviously, this question is also important if we have to choose among various possible local search algorithms like illustrated in the $\mathcal{MM}$AS application to the QAP (see Section 4.4.2). But this is also an important question concerning the particular implementation of a local search algorithm. We will address this issue again for the ILS application to the QAP in the next section.

[2] The comparison between $\mathcal{MM}$AS and ILS for the application to the TSP in Section 4.3.6 uses the computational results of variant (1).

Table 5.1: Results for the comparison of don't look bit resetting strategies on symmetric TSP instances from TSPLIB. The results are averaged over 25 runs for $n \leq 1000$ and over ten runs for $n > 1000$. Given are in the first row of each entry the average solution quality and the percentage deviation from the optimum and in the second row the average computation time and the average number of local search applications to reach the best tour in each trial.

Instance	optimal	(1)	(2)	(3)	t_{max}
d198	15780	**15780.0** (0.0) 27.69 – 1097.9	**15780.0** (0.0) 52.31 – 2897.3	15781.3 (0.008) 47.7 – 6180.2	300
lin318	42029	**42064.6** (0.085) 58.46 – 1527.1	42077.6 (0.116) 73.8 – 3450.9	42104.9 (0.181) 81.9 – 9093.89	450
pcb442	50778	50917.7 (0.275) 180.7 – 7124.2	**50879.04** (0.199) 173.9 – 12296.96	50930.1 (0.299) 210.4 – 24960.7	600
att532	27686	27708.9 (0.083) 511.5 – 9400.0	**27705.6** (0.071) 344.02 – 14611.7	27732.1 (0.167) 521.9 – 47375.9	1800
rat783	8806	8828.4 (0.254) 1395.2 – 22383.4	**8826.0** (0.227) 911.7 – 41516.3	8836.0 (0.341) 985.7 – 81405.3	2100
u1060	224094	224408.4 (0.140) 1210.4 – 12892.7	**224343.2** (0.111) 929.65 – 32519.6	224475.0 (0.170) 2233.6 – 142475.0	3600
pcb1173	56892	57029.5 (0.242) 1892.0 – 21041.6	**56991.7** (0.175) 1983.9 – 78625.3	57167.7 (0.485) 2802.4 – 188650.9	5400
d1291	50801	50875.7 (0.147) 1102.87 – 16242.6	**50868.0** (0.132) 1148.9 – 47044.3	50885.4 (0.166) 1491.32 – 85956.1	5400

(1) Reset all don't look bits to zero.

(2) Reset the don't look bits of the 20 cities preceeding and following the broken edge to zero.

(3) Reset only the don't look bits of endpoints of broken edges to zero.

Of these three options the fewest and the largest number of `3-opt` applications can be performed with (1) and (3), respectively. We present the computational results for symmetric TSPs in Table 5.1. As expected, the number of local search applications is very different for the three versions and therefore the comparison has to be made using the same run-time for all versions. The computational results show that the worst results are obtained with reset option (3). For options (1) and (2), the tendency appears to be that for the smaller instances (`d198` and `lin318`) better results are obtained in shorter time resetting all don't look bits. Yet, on larger instances strategy (2) obtains a slightly better average performance. This observation is also backed up by plots of the development of the average solution quality given in Figure 5.2. The following computational results presented in the context of ILS applications to the TSP with `3-opt` will use don't look bit resetting strategy (2) . When using the `LK` heuristic, we will follow the observation made in [129] and reset only the don't look bits of cities which were connected by broken edges.

5.2.4 Run-Time Distributions for ILS Applied to the TSP

In this section we analyze the empirical run-time behavior of iterated local search on TSP problems. Since the run times needed to reach very high quality solutions on large TSP instances

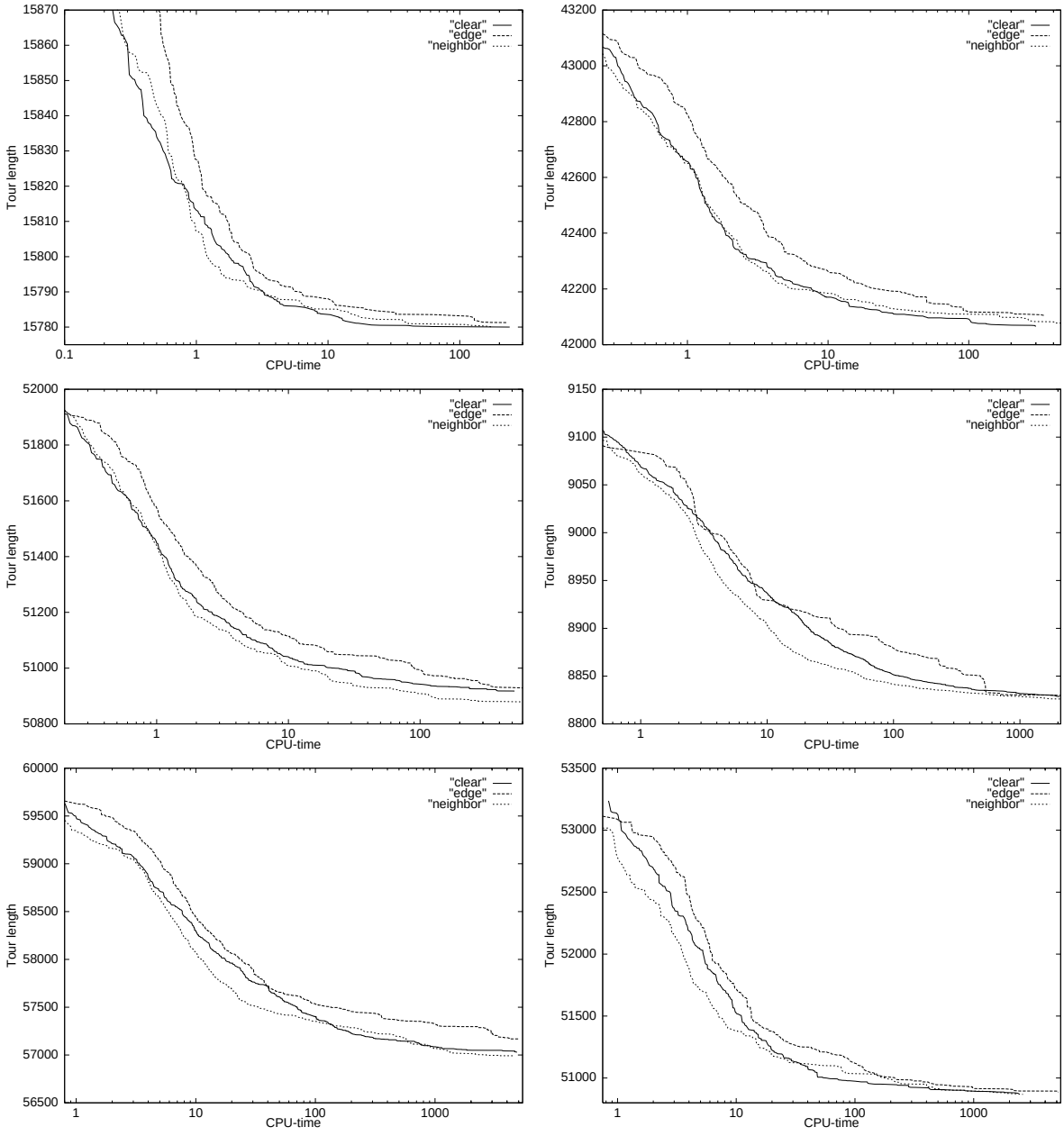

Figure 5.2: Given is the development of the average solution quality with the three don't look bit resetting strategies on 6 TSP instances, `d198` (upper left side), `lin318` (upper right side), `pcb442` (middle left side), `rat783` (middle right side), `pcb1173` (lower left side), and `d1291` (lower right side). In the plots "clear", "edge", and "neighbor" refer to reset option (1), (3), and (2), respectively.

becomes very large, we limit the study to smaller instances with less than 1000 cities. For each instance, run-time distributions have been measured using 100 independent runs of each algorithm. The study comprises both, symmetric and asymmetric TSPs. The algorithms we used are iterated `3-opt` and iterated `LK` (ILK) for symmetric TSPs and iterated `reduced 3-opt` for asymmetric TSPs.

The run-time distributions are given with respect to various bounds on the solution quality. In addition to the empirical run-time distributions we plot the cumulative distribution function of an exponential distribution (indicated by $f(x)$ in the plots) adjusted towards a run-time distribution corresponding to high quality solutions. As explained in detail in Section 3.2.3, the plotted exponential distributions give an indication whether the algorithm can be improved by restarting it from a new initial solution. This is the case if the empirical run-time distribution falls below the plotted exponential one.

Run-time Distributions for Symmetric TSPs with Iterated 3-opt

The empirical run-time distributions for the four TSPLIB instances `d198`, `lin318`, `pcb442`, and `rat783` are given in Figure 5.3. (The empirical run-time distributions for instance `att532` have been used in Chapter 3 in Figure 3.1 (left side) for illustration purposes.) To empirically evaluate the run-time distributions we limited the maximally allowed computation time to a 1000 seconds on the two smaller and to 3600 seconds on the two larger instances. The computation times are chosen high enough to ensure that the algorithm shows limiting performance and that the further increase of solution quality by allowing still longer computation times should be rather small. In the plots, the exponential distribution is fitted to the lower part of the run-time distributions to reach the optimum except in the case of `rat783` where it is fitted to the run-time distribution to reach a solution within 0.25% of the optimum.

From the empirical run-time distributions several conclusions can be drawn. One is that on all instances it is rather easy to get within, say 1%, of the optimal solution; the empirical run-time distribution to reach 1% are always the leftmost ones in the plots of the three larger instances. Yet, if higher accuracy is required, the algorithms suffer from a kind of stagnation behavior. Consider, for example, instance `pcb442`. A solution within 1% of the optimum is obtained after at most 15 seconds in all runs. Yet, there are some few runs of the algorithm that even fail to get within 0.5% of the optimum. After 60 seconds the empirical solution probability reaches 0.93, that is, 93 of the 100 runs have found a satisfying solution, but with 940 additional seconds only two more times a better solution than the required one could be found. A similar situation occurs if still higher quality solutions like the optimum should be achieved. For `pcb442` the first optimal solution is found very fast, reaching a solution probability of 0.1 after 77 seconds, but after 1000 seconds the solution probability only reaches 0.23. Similar observations also apply to the other instances. This stagnation behavior is striking if compared to the exponential distributions plotted.

Run-time Distributions for Symmetric TSPs with ILK

One could object that the observation of a stagnation behavior may be due to the choice of a `3-opt` local search heuristic which shows poorer performance on the TSP than more sophisticated local search algorithms like the `LK` heuristic. To test this hypothesis we have run the ILK heuristic on the same instances used in the previous section, except the smallest one. Instead

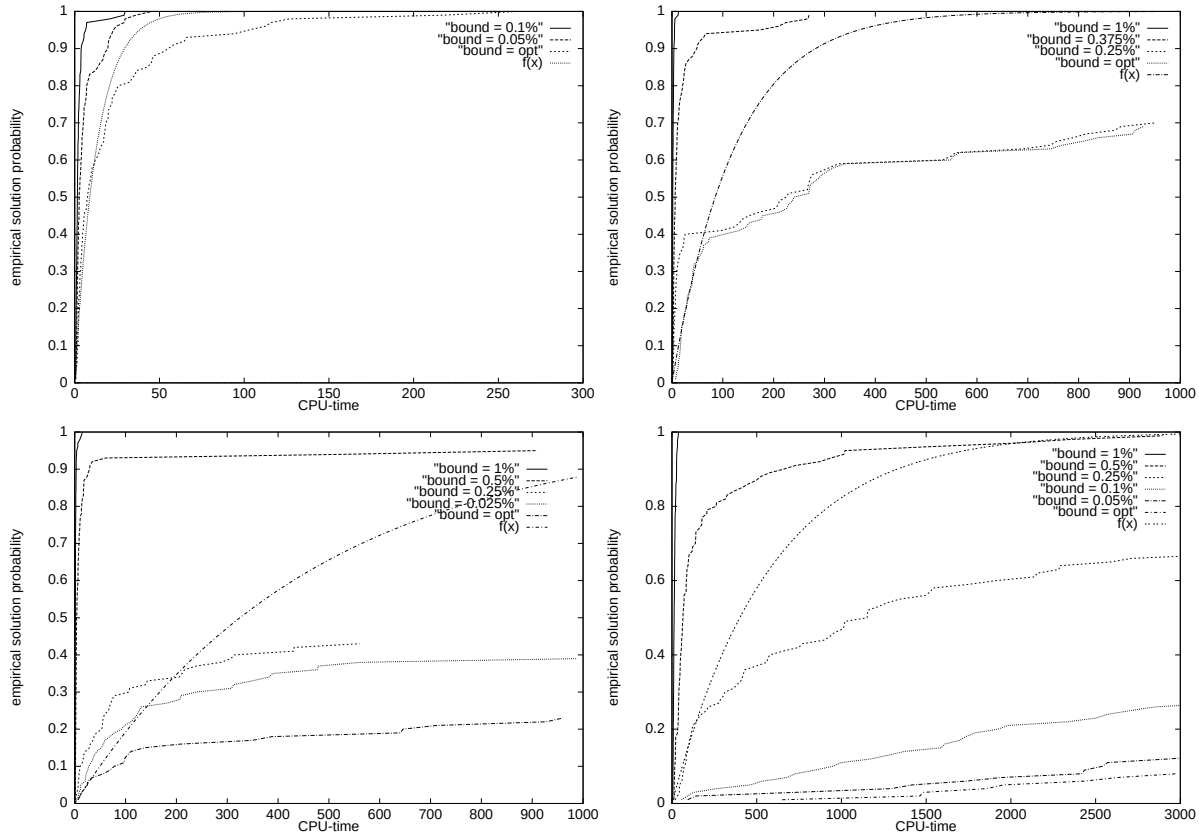

Figure 5.3: Run-time distributions of iterated `3-opt` on four symmetric TSP instances. Given are the distributions for `d198` (upper left side), `lin318` (upper right side), `pcb442` (lower left side), and `rat783` (lower right side).

using run-*time* distributions, here we give run-*length* distributions with the x-axis representing the number of `LK` applications.

Since ILK is known to be one of the best performing approximate algorithms for symmetric TSP instances, we expect higher quality solutions to be obtained more frequently. This expectation is confirmed by the empirical run-length distributions given in Figure 5.4. On three instances the empirical probability to find the optimal solution is significantly higher than using iterated `3-opt`. An exception is instance `lin318` on which both algorithms show clearly a strong stagnation behavior. Signs of stagnation are also shown for instance `att532`, but at a significantly higher probability of finding the optimal solution compared to iterated `3-opt` (compare to Figure 3.1, left side, on page 46). The other two instances are regularly solved to optimality by the ILK algorithm. In particular, the run-length distribution of `pcb442` can be closely fitted by a modified exponential distribution with distribution function $G(x) = \max\{0, 1 - e^{-\lambda(x-\Delta x)}\}$, where Δx adjusts for the fact that a certain minimal number of iterations have to be to performed to obtain a reasonable chance of finding the optimal solution.[3] In general, it appears that this particular instance is easily solved using the `LK` heuristic, but it

[3]The curve in that plot has been fitted with the C. Grammes implementation of the Marquart-Levenberg algorithm available in gnuplot. The fitted parameters are $\Delta x = 51.26$ and $\lambda = 0.00524$.

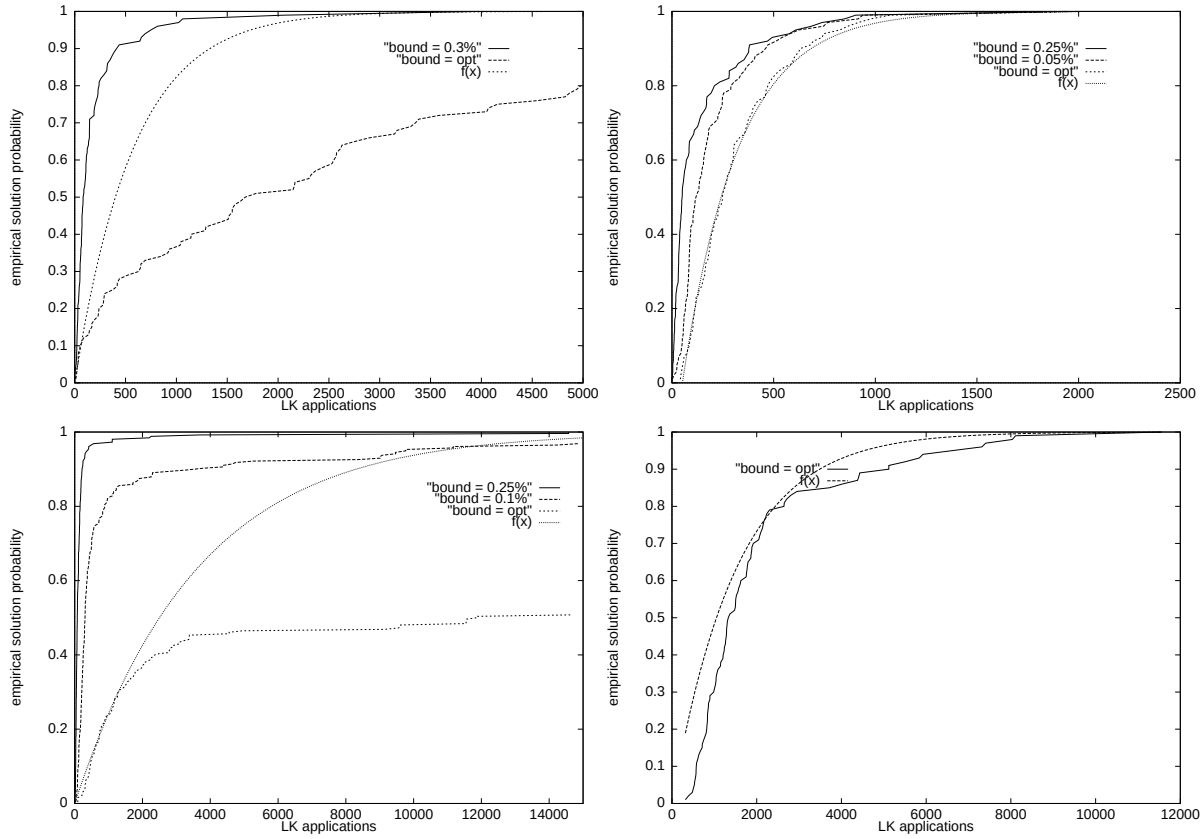

Figure 5.4: Run-length distributions of ILK on four symmetric TSP instances. Given are the distributions for `lin318` (upper left side), `pcb442` (upper right side), `att532` (lower left side), and `rat783` (lower right side).

is "relatively hard" to solve when using "only" `3-opt` (compare the run-time distribution of ILK to that of iterated `3-opt` in Figure 5.3). On instance `rat783`, ILK again falls below the exponential distribution, thus the algorithm's performance could still be improved.

Run-time Distributions for ATSPs with Iterated `reduced 3-opt`

The run-time distribution for the ATSP instances plotted in Figure 5.5 confirm the findings on the symmetric instances. On the ATSP instances the stagnation behavior of iterated `reduced 3-opt` appears to be even more severe. Often the optimal solution is obtained very early in a run or, with a few exceptions, it is not found any more within the given computation time. Surprisingly, also on the smallest instance `ry48p` with only 48 nodes the algorithm shows stagnation behavior, yet with larger run-time it appears that all runs could be terminated successfully.

Interestingly, the largest ATSP instance `rbg443` of TSPLIB with 443 cities is, despite its size, easily solved if the acceptance criterion *BetterEqual*(s, s'') is used. In this case an optimal solution is found, on average, in 35 seconds or 5200 local search applications, with the maximally required run-time to solve this instance being roughly 150 seconds. Interestingly, the

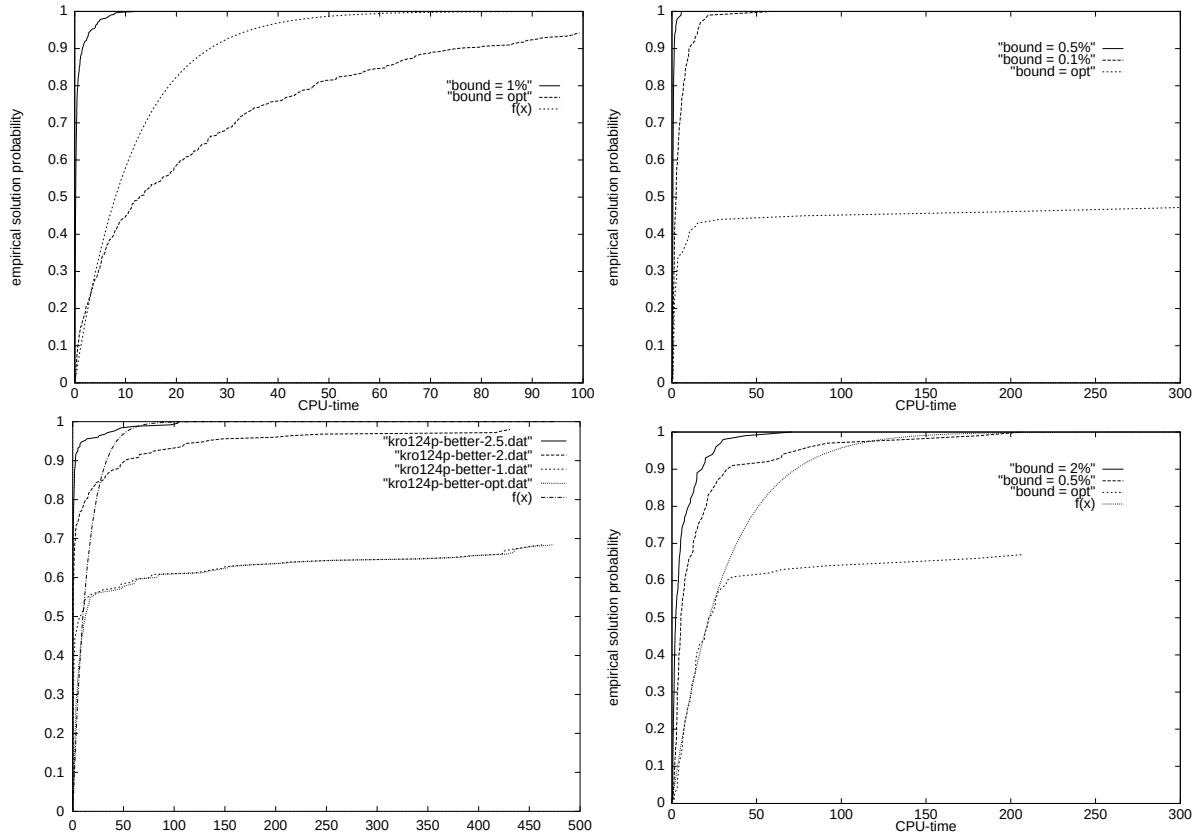

Figure 5.5: Run-time distributions of iterated `reduced 3-opt` on four ATSP instances. Given are the run-time distributions for `ry48p` (upper left side), `ft70` (upper right side), `kro124p` (lower left side) and `ftv170` (lower right side).

same instance could only be solved in 90% of the runs with maximally 1000 seconds if the acceptance criterion *Better*(s, s'') is used. In Figure 5.6 we have plotted the run-time distributions for these two acceptance criteria. The run-time distribution using *BetterEqual*(s, s'') is, in fact, steeper than an exponential distribution, as can be verified in Figure 5.6. Such significant differences between these two acceptance criteria could only be observed on the four ATSP instances `rbg323`, `rbg358`, `rbg403`, and `rbg443`. Considering this fact, in the later experimental investigation we used the acceptance criterion *BetterEqual*(s, s'') for all ATSP instances.

Common Observations from the Run-Time Distributions

A general conclusion from the analysis of the run-time behavior of ILS algorithms is that these algorithms tend to quickly find good solutions to symmetric as well as asymmetric TSPs. But if very high solution quality is required, the ILS algorithms suffer from search stagnation which can be observed in the plots of the run-time distributions. The empirical run-time distributions are, in fact, in nearly all cases less steep than an exponential distribution — and therefore the algorithm can, as argued in Chapter 3, be improved.

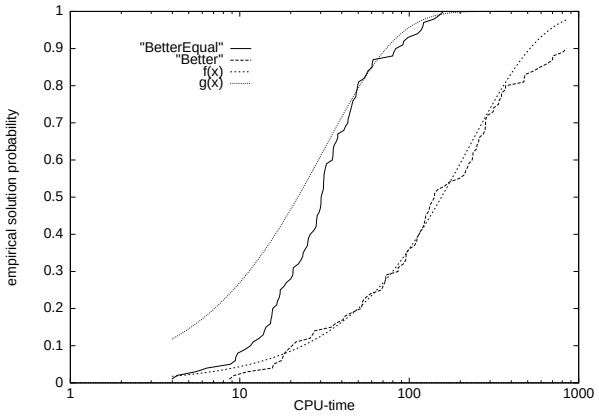

Figure 5.6: Run-time distributions for instance `rbg443` using iterated `reduced 3-opt`. Given are two run-time distribution, one using acceptance criterion *BetterEqual*(s, s''), the other using *Better*(s, s'') for finding the optimal solution. Additionally two exponential distributions are plotted, indicated by $f(x)$ and $g(x)$.

The ILK algorithm does not suffer as strongly from stagnation behavior as iterated `3-opt` on the symmetric TSPs. One interpretation of this fact is that the more powerful an algorithm, the less it will suffer from stagnation behavior. But still, ILK could be considerably improved by using appropriate cutoff values on three of the four investigated instances.

In summary, the run-time distributions arising from the application of ILS algorithms to the TSP suggest that these algorithms can be improved. In the following we will present several possibilities how such improvements can be achieved, followed by an extensive empirical investigation of the proposed possibilities.

5.2.5 Improvements of ILS Algorithms for the TSP

The common idea of the improved ILS algorithms is that an increased performance can be obtained by providing a higher exploration of the search space. In the following we propose improved ILS algorithms which use acceptance criteria which occasionally accept worse solutions as well as population-based extensions of ILS algorithms.

Soft Restarts

As argued in Chapter 3, the simplest strategy to avoid stagnation behavior is to restart the algorithm from new initial solutions after some predefined cutoff value. Yet, optimal cutoff values may depend strongly on the particular TSP instance. A better idea appears to be to relate the restart to the search progress and to apply "soft" cutoff criteria. In particular, we restart the ILS algorithm if no improved solution could be found for i_r iterations, where i_r is a parameter. (Recall that a similar approach has been used to improve the $\mathcal{MM}$AS algorithm in Section 4.6.3.) The soft restart criterion is based on the assumption that after i_r iterations without improvement the algorithm has got stuck in a certain region of the search space and that additional diversification (implemented by restarting the algorithm from a new initial solution) is needed to

escape from there. The restart of the algorithm can easily modeled by the acceptance criterion called *Restart*$(s, s'', \textit{history})$. The *history* component indicates that the use of soft restarts can be interpreted as a very simple use of the search history by the acceptance criterion. Let i_{last} be the last iteration in which a better solution has been found and i be the iteration counter. Then *Restart*$(s, s'', \textit{history})$ is defined as

$$\textit{Restart}(s, s'', \textit{history}) = \begin{cases} s'' & \text{if } f(s'') < f(s) \\ s''' & \text{if } f(s'') \geq f(s) \text{ and } i - i_{last} > i_r \\ s & \text{otherwise.} \end{cases} \tag{5.2}$$

Typically, s''' can be generated in different ways. The simplest strategy is to randomly generate a new initial solution corresponding to a restart of the algorithm.

Fitness-Distance Diversification-based ILS

If an ILS algorithm gets stuck in a certain region of the search space, one should try to escape from there. In such a case the search should be continued from a solution which, in some sense, is distant from the current search point. This can be achieved, as proposed before, by occasional restarts. Yet, especially for larger instances a local search algorithm has to be applied many times to reach very high solution quality as demonstrated by the initialization time t_{init} needed to find high quality solutions (recall the discussion of the run-time distributions in Section 3.2 on page 46). Additionally, previously found high quality solutions get lost by using restarts. To avoid these disadvantages, we propose a new method to escape from the search space region of the current search point. Our goal will be to *find a good quality solution beyond a certain minimal distance from the current search point without using restart*. In fact there are two objectives to be taken into account to find such a solution. The first is to generate a distant solution, the second is that this solution should be of reasonable quality.

To generate such a solution, we proceed as follows. Let s_c be the current solution from which we want to escape and the distance between solutions is measured as the number of edges which are different. Then, the following steps are repeated until we have found a solution beyond a minimal distance d_{min} from s_c.

(1) Set $s = s_c$

(2) Generate p locally optimal solutions. Each is generated by first applying a kick-move to s and then locally optimizing it.

(3) Select the best q of these p solutions, $1 \leq q \leq p$. These are the candidate solutions.

(4) Let s be the candidate solution with maximal distance to s_c

(5) If the distance between s and s_c is smaller than d_{min}, then repeat at (2).

The single steps are motivated by the following considerations. The second objective (find a good quality solution) is attained by keeping only the q best of the p candidate solutions. The first objective (find a solution at minimal distance to s_c) is achieved by choosing a candidate

solution with maximal distance from s_c. The steps are then iterated until a solution s^* is found for which $d(s^*, s_c) \geq d_{min}$. In fact, we stop that process if we have not found a solution s^* with the required minimal distance after a maximal number of repetitions of the diversification steps.

To fix a value for d_{min}, knowledge of the average distance between high quality solutions is important. To get some hints on how d_{min} should be chosen, we measured the average distance (d_{avg}) between 1000 locally optimal solutions obtained with `3-opt` and between 200 solutions obtained by running iterated `3-opt` for n iterations (ILS(n)), n being the instance size. The starting tours have been generated by the nearest neighbor construction heuristic. Additionally, we calculated the ratios of the average distance between local minima of `3-opt` and ILS(n), and the ratio between the average distance and the instance size. The results, given in Table 5.2, show that for most instances the average distance between `3-opt` local minima is larger than between solutions obtained by ILS(n), with the only two exceptions being instances `fl1400` and `fl1577`. A closer examination of the results showed that this latter observation was due to the fact that the initial tour was taken as a nearest neighbor tour. When starting from randomly generated tours, a situation as for the other instances is obtained. The ratio of the average distance between solutions obtained by ILS(n) and the average distance `3-opt` local minima, respectively, is much smaller than the maximally possible distance, which would be the instance size n. (This fact has also been observed before in earlier TSP search space analysis [26, 141, 185].) Reasonably high quality solutions are certainly obtained by the application of ILS(n), thus the average distance between ILS(n) solutions may give an indication for reasonable values of d_{min}. Yet, estimating this value is rather expensive and therefore we use the average distance between 100 `3-opt` local optima ($d_{avg}^{\text{LS(n)}}$) and estimate d_{min} by $d_{min} = 2/3 \cdot d_{avg}^{\text{LS(n)}}$.

A second question is which solution should be taken as s_c. In earlier search space analyses of the TSP search space, a strong correlation between the fitness and the distance to the optimal solution has been found [27, 26] (see also discussion in Section 2.9). Hence, on average, the best found solution (s_{best}) is closer to the optimal one than others. Consequently, it appears to be a good idea to start the diversification from s_{best} since that solution is expected to have many edges in common with the optimal solution and d_{min} may be interpreted as an estimate of the distance to the optimal solution. Taking into account this idea we diversify every second time from s_{best} by setting $s_c = s_{best}$; otherwise, s_c is the solution when the diversification is triggered.

In fact, this new diversification approach for ILS has been inspired by earlier results on the fitness-distance correlation for TSP instances [27, 26]. In this chapter, we only apply this specialized procedure to the TSP, but it is worthwhile to mention that this procedure is also applicable to other problems and we conjecture that it may be particularly successful on problems which share similar search space characteristics with the TSP.

Population-based Extensions

Another convenient way of achieving search space exploration is the use of a population of solutions instead of a single one. Here we propose some simple schemes for extending ILS algorithms by using a population of solutions. The idea is to replace a single ILS run by a population of ILS runs which may interact in some way. Such extensions share similarities to other population-based search metaphors like evolutionary algorithms [12] and, in particular, to those of genetic local search (see page 32). In genetic local search individual solutions are

Table 5.2: Average distance between locally optimal solutions obtained by `3-opt` or n iterations of iterated `3-opt` (ILS(n)). When running `3-opt`, average distances are based on 1000 runs (d_{avg}^{LS}) and for ILS(n) average distances are based on 200 runs ($d_{avg}^{\text{ILS(n)}}$). Additionally, we give the ratio between $d_{avg}^{\text{ILS(n)}}$ and d_{avg}^{LS}, and the ratios of the average distances to the instance size n.

Instance	d_{avg}^{LS}	$d_{avg}^{\text{ILS(n)}}$	$d_{avg}^{\text{ILS(n)}}/d_{avg}^{\text{LS}}$	d_{avg}^{LS}/n	$d_{avg}^{\text{ILS(n)}}/n$
d198	43.83	28.19	0.64	0.22	0.14
lin318	62.02	37.6	0.60	0.19	0.12
pcb442	88.0	77.8	0.88	0.25	0.18
att532	122.4	87.2	0.71	0.23	0.16
rat783	197.8	142.2	0.72	0.24	0.18
pr1002	242.0	177.2	0.73	0.21	0.18
u1060	249.3	186.1	0.75	0.24	0.17
pcb1173	274.6	177.3	0.65	0.22	0.15
d1291	143.9	100.4	0.61	0.11	0.08
fl1400	427.9	452.1	1.06	0.30	0.32
fl1577	149.1	182.6	1.22	0.09	0.11

modified by means of mutation operators and crossover operators, which exchange information between pairs of solutions. Then, a local search is applied to the generated solutions. A deliberate difference of our population-based extensions of ILS with genetic local search is that in ILS solutions are only modified by applications of a kick-move, which, in fact, corresponds to a mutation in the context of evolutionary algorithms.

Our motivation for using a population of solutions is to investigate means of avoiding the stagnation behavior of ILS other than by increasing diversification in single ILS runs. The stagnation behavior in population-based ILS extensions is avoided by, in some sense, delaying the decision on which solution one has to concentrate to find the highest solution quality; by the use of a population, the algorithm is not forced to concentrate the search only around the best solution found as done in single ILS runs. Here, we use three different types of population-based extensions. Certainly many others can be conceived.

No interaction The simplest extension is to not use any interaction between the single ILS runs. Obviously, for a given population size p and a given maximally allowed computation time t_{max} such an approach corresponds to multiple, independent ILS applications, each of runtime t_{max}/p. We consider this possibility, because it gives hints whether interaction among the ILS runs, which is used by the other two extensions, improves performance. Recall that, when presenting the parallelization strategies for ant colony optimization algorithms in Section 4.6, we have argued that multiple, independent runs are the easiest way of achieving a parallel algorithm. Hence, this scheme also gives hints on the performance of multiple, independent runs of ILS algorithms.

Replace-worst This variant proceeds analogous to the first one, except that it occasionally replaces the worst solution with the currently best solution. Here, a parameter u_b indicates

```
procedure Population-ILS
    Generate initial population p
    p = LocalSearch(p)
    while (termination condition not met) do
        s_0 = Select(p)    % select a solution
        s = s_0
        for i = 1 to x do
            s' = Modify(s)
            s'' = LocalSearch(s')
            s = AcceptanceCriterion(s, s'')
        end
        if (f(s) < f(s_0) then
            p = ReplaceSolution(s, p)
        end
    end
end Population-ILS
```

Figure 5.7: Algorithmic skeleton of Population-ILS.

that every u_b iterations such a replacement takes place. The motivation for this scheme is to gradually concentrate the search around the best solution of the population. The parameter u_b determines how fast this concentration takes place. If u_b is very low, the search may concentrate early around possibly suboptimal solutions; if u_b is very high, replace-worst will behave similarly to the first proposed variant without interaction among the ILS runs.

Population-ILS This last scheme shares more similarities than the first two variants to evolutionary algorithms. An algorithmic outline of the population-based ILS is given in Figure 5.7. Initially, a population of local minima is generated and then, in each iteration, one of the locally optimal solutions is selected and a standard ILS algorithm is applied to this solution for x iterations. If the final solution obtained after running ILS is better than the starting solution, one of the solutions in the population is replaced. Different choices of the operators Select and ReplaceSolution will lead to different search behavior. In our experimental study we use the operators as follows: Select chooses randomly one solution of the population and if the solution is improved, ReplaceSolution replaces the worst solution of the population.

5.2.6 Experimental Results

In this section we present an experimental investigation of our proposed ILS extensions for symmetric and asymmetric TSP instances from TSPLIB [210]. We compared all the proposed extensions extensively on the same instances which have also been used to evaluate and compare ant colony optimization algorithms (see Section 4.3.6 on page 85f). The computation time limits we used in our experiments for all instances are the same or less than those of the improved $\mathcal{MM}$AS algorithm presented in Section 4.6.3. Thus, the results obtained by the ILS extensions can be compared to those obtained with $\mathcal{MM}$AS since the runs were all performed on the

same machine, a Sun UltraSparc II Workstation with two UltraSparc I 167MHz processors and 0.5MB external cache. Due to the sequential implementation of the algorithms only one processor is used.

We will refer to the different ILS algorithms as follows. Standard ILS (ILS using acceptance criterion *Better*(s, s'') for symmetric TSPs and *BetterEqual*(s, s'') for ATSPs) will be referred to as ILS, the soft restarts extension as ILS-restart, and the fitness-distance diversification-based extension as ILS-fdd. For the population-based extensions we will use ILS-parallel for version "no interaction", ILS-rep-worst for version "Replace-worst" and ILS-pop for the last one.

The parameter settings for the ILS extensions are as follows. In ILS-restart and ILS-fdd a diversification is triggered if no improved solution is found for $2n$ iterations when applying `3-opt` and after $0.25n$ iterations when using the `LK` heuristic. In ILS-fdd we set p to 20 and q to $0.75p$ when using `3-opt` and determine d_{min} as described before. When applying the `LK` heuristic we use $p = 10$, $q = 7$, $d_{min} = 0.075n$. In the population-based extensions we use a population size of 25 when applying `3-opt` or `reduced 3-opt` and a population size of 10 when running the `LK` heuristic. The acceptance criterion used is *Better*(s, s''). In the version ILS-rep-worst we first run $n/2$ iterations without any interaction (100 when using the `LK` heuristic) and from then on the best solution replaces the worst one every $n/4$ iterations (every 50 iterations when using `LK`). In ILS-pop we first apply to every solution a standard ILS algorithm for $n/2$ generations (differing from the algorithmic outline in Figure 5.7). Then we continue as detailed in Figure 5.7 and a solution chosen by Select is used in the standard ILS algorithms for $n/2$ ILS iterations.

Experimental Results with Iterated `3-opt` Extensions for Symmetric TSPs

The computational results are presented in Tables 5.3 (instances with less than 1000 cities) and 5.4 (instances with more than 1000 cities). Our discussion of the computational results is based on the performance of iterated `3-opt`.

Single run ILS extensions Both extensions, ILS-restart and ILS-fdd, significantly improve ILS performance with respect to average solution quality on all instances. Only for ILS-restart for just two instances (`u1060` and `pcb1173`) is the best result obtained in 10 runs slightly worse than with ILS. On the smaller instances, ILS-restart and ILS-fdd perform comparably. Yet, the larger the instances the more clearly is the performance advantage of ILS-fdd over ILS-restart. ILS-fdd finds better solutions and more often optimal solutions than ILS-restart. This fact is most apparent on instances `att532`, `rat783`, `pcb1173`, and `fl1577`. The only exceptions are `pr1002` and `fl1400`. While for the first one ILS-fdd obtains more often the optimal solution, the higher average is due to one run which stopped at a rather poor quality solution. Instance `fl1400` is, despite its size, very easy to solve. For example, standard ILS finds the optimum in roughly 50% of the runs. In favor of ILS-fdd is also the fact that it obtains on many instancs the best average performance compared to all competitors.

Population-based extensions Also the population-based extensions show significantly improved performance compared to ILS. Of these extensions ILS-rep-worst shows the best average solution quality on most instances. ILS-parallel gives better average solution quality on three of the 11 instances, and, in particular, on the two clustered instances `u1060` and `fl1577`

Table 5.3: Comparison of ILS extensions using `3-opt` local search for symmetric TSPs on some TSPLIB instances with less than 1000 cities. Given are the best solution obtained and the frequency of finding it (in parenthesis), the average solution quality (and, in parenthesis, the percentage deviation from the optimum), the worst solution, the average number of iterations i_{avg} and the average CPU-time t_{avg} to find the best solution in a run, and the maximally allowed computation time t_{max}.

Algorithm	best	average	worst	i_{avg}	t_{avg}	t_{max}
`d198` opt: 15780						
ILS	15780 (25/25)	**15780.0** (0.0)	15780	2897.3	52.3	300
ILS-restart	15780 (25/25)	**15780.0** (0.0)	15780	441.5	8.9	300
ILS-fdd	15780 (25/25)	**15780.0** (0.0)	15780	704.9	16.6	300
ILS-parallel	15780 (25/25)	**15780.0** (0.0)	15780	36.8	18.0	300
ILS-rep-worst	15780 (25/25)	**15780.0** (0.0)	15780	39.1	18.3	300
ILS-pop	15780 (25/25)	**15780.0** (0.0)	15780	23.4	12.4	300
`lin318` opt: 42029						
ILS	42029 (17/25)	42067.7 (0.092)	42163	5423.0	115.2	600
ILS-restart	42029 (25/25)	**42029.0** (0.0)	42029	4370.1	109.9	600
ILS-fdd	42029 (25/25)	**42029.0** (0.0)	42029	4077.2	91.3	600
ILS-parallel	42029 (25/25)	**42029.0** (0.0)	42029	187.6	103.8	600
ILS-rep-worst	42029 (25/25)	**42029.0** (0.0)	42029	270.5	138.9	600
ILS-pop	42029 (25/25)	**42029.0** (0.0)	42029	88.4	92.2	600
`pcb442` opt: 50778						
ILS	50778 (5/25)	50879.0 (0.20)	51069	12296.9	173.9	900
ILS-restart	50778 (18/25)	50782.9 (0.0097)	50860	23013.9	362.9	900
ILS-fdd	50778 (20/25)	50798.8 (0.041)	50912	25887.2	369.1	900
ILS-parallel	50778 (16/25)	50781.6 (0.007)	50802	1286.7	416.4	900
ILS-rep-worst	50778 (22/25)	**50778.8** (0.002)	50785	1131.5	361.5	900
ILS-pop	50778 (11/25)	50825.8 (0.094)	50912	203.2	314.2	900
`att532` opt: 27686						
ILS	27686 (3/25)	27705.5 (0.071)	27741	14611.7	344.0	1800
ILS-restart	27686 (6/25)	27694.4 (0.030)	27705	29512.9	779.8	1800
ILS-fdd	27686 (12/25)	**27691.4** (0.019)	27705	23630.2	572.9	1800
ILS-parallel	27686 (4/25)	27699.3 (0.048)	27706	1328.9	753.4	1800
ILS-rep-worst	27686 (7/25)	27695.7 (0.035)	27706	1438.7	798.1	1800
ILS-pop	27686 (1/25)	27702.8 (0.061)	27719	218.7	746.9	1800
`rat783` opt: 8806						
ILS	8808 (1/25)	8824.3 (0.21)	8864	69620.9	1522.8	3600
ILS-restart	8806 (1/10)	8812.6 (0.074)	8821	57389.9	1791.9	3600
ILS-fdd	8806 (22/25)	**8806.2** (0.0023)	8808	68768.3	1525.8	3600
ILS-parallel	8806 (1/10)	8815.6 (0.11)	8822	4393.1	2343.64	3600
ILS-rep-worst	8806 (2/10)	8810.6 (0.052)	8817	3485.9	1844.6	3600
ILS-pop	8808 (1/10)	8815.4 (0.11)	8828	462.5	2314.6	3600

(see Figure 5.8 on page 141). ILS-pop performs, in general, worse than either ILS-rep-worst or ILS-parallel, yet still significantly better than plain ILS. Yet, the worse performance of ILS-pop may be due to the particular parameter settings used, which were chosen in an ad hoc manner without any fine-tuning.

Overall, ILS-fdd appears to perform best among all extensions. ILS-parallel and ILS-restart perform similarly, yet, ILS-restart is slightly better on most instances. This may be due to the fact that ILS-restart does not use a fixed cutoff time, whereas ILS-parallel corresponds to a restart version with a fixed cutoff time. The similar performance of the two variants indicates that in ILS-parallel, for the computation time given, a reasonable population size (corresponding directly to a fixed cutoff time of t_{max}/p) has been used.

One interesting aspect of the computational results is that the best performing population-based extension for the two clustered instances is in fact ILS-parallel, which does not use any interaction between the search points. Apparently for problems with deep local minima from which standard ILS has major difficulties to escape, one of the best options is not to have any interaction among the individual ILS runs in the population-based ILS extensions. This observation is also backed up by the very good computational results for ILS-restart on these two clustered TSP instances.

Experimental Results with Iterated LK for Symmetric TSPs

We also applied ILS-fdd, ILS-parallel, and ILS-rep-worst using the `LK` local search to some TSP instances; the computational results are given in Table 5.5. When using standard ILK, some of the instances (`pcb442`, `rat783`, and `pr1002`) are rather easily solved to optimality and the contribution of the proposed extensions is not very notable. Yet, on the other instances our extensions showed significantly improved performance with ILS-fdd again performing best. In particular, this extension seems to be the most reliable one, finding the highest number of optimal solutions.

Experimental Results with Iterated Local Search for Asymmetric TSPs

When applied to the ATSP, the computational results of the ILS extensions confirm the significantly improved performance over a standard ILS implementation. With the ILS extensions all the instances could be solved to optimality in all runs, with the only exception being ILS-fdd on instance `ftv170` where in two runs not the optimal solution was found. This improvement is especially notable on the four smaller instances on which ILS showed a strong stagnation behavior. Exceptions are the largest ATSP instances of TSPLIB (those with name `rbgxxx`), on which the standard ILS algorithm performs best (see the computational results for `rbg443` in Table 5.6; the population-based extensions use only 10 solutions). As noted in the discussion of the run-time distributions (page 130), this fact is due to the use of the acceptance criterion *BetterEqual*(s, s''). If instead *Better*(s, s'') is used for these instances, the ILS extensions again yield slightly improved performance compared to standard ILS.

Table 5.4: Comparison of ILS extensions using `3-opt` local search for symmetric TSPs on some TSPLIB instances with more than 1000 cities. See Table 5.3 for a description of the entries.

Algorithm	best	average	worst	i_{avg}	t_{avg}	t_{max}
`pr1002` opt: 259045						
ILS	259045 (1/10)	259759.4 (0.27)	260283	59133.4	1642.1	3600
ILS-restart	259045 (6/10)	**259133.4** (0.034)	259442	79744.9	2536.8	3600
ILS-fdd	259045 (8/10)	259185.7 (0.054)	259634	58493.5	1953.8	3600
ILS-parallel	259045 (4/10)	259270.6 (0.093)	259630	2072.7	1408.6	3600
ILS-rep-worst	259045 (7/10)	259138.7 (0.036)	259426	3475.8	2222.4	3600
ILS-pop	259045 (2/10)	259403.9 (0.14)	259722	398.8	2106.4	3600
`u1060` opt: 224094						
ILS	224113 (1/10)	224343.2 (0.11)	224730	5423.0	115.2	3600
ILS-restart	224168 (3/10)	224212.2 (0.053)	224388	45150.3	2083.6	3600
ILS-fdd	224115 (3/10)	**224139.5** (0.020)	224194	62438.5	1800.8	3600
ILS-parallel	224122 (2/10)	224198.7 (0.047)	224315	3464.0	2398.7	3600
ILS-rep-worst	224152 (1/10)	224256.1 (0.072)	224387	3387.8	2258.8	3600
ILS-pop	224115 (1/10)	224248.7 (0.069)	224377	414.8	2346.4	3600
`pcb1173` opt: 56892						
ILS	56893 (1/10)	56991.7 (0.17)	57186	1983.9	78625.3	5400
ILS-restart	56896 (1/10)	56947.7 (0.098)	57062	34421.7	1525.1	3600
ILS-fdd	56892 (3/10)	56929.0 (0.065)	56993	50964.7	2285.9	3600
ILS-parallel	56892 (2/10)	56938.2 (0.081)	57021	3511.9	2237.1	3600
ILS-rep-worst	56892 (1/10)	**56904.0** (0.021)	56924	3912.2	2352.5	3600
ILS-pop	56892 (2/10)	56920.6 (0.050)	57000	457.2	2525.5	3600
`d1291` opt: 50801						
ILS	50829 (3/10)	50868.0 (0.13)	50913	47044.3	1148.9	5400
ILS-restart	50801 (5/10)	50820.0 (0.037)	50842	56905.8	2257.1	3600
ILS-fdd	50801 (7/10)	**50809.8** (0.018)	50842	51033.5	1267.9	3600
ILS-parallel	50801 (1/10)	50833.1 (0.063)	50861	1328.9	753.4	3600
ILS-rep-worst	50801 (1/10)	50834.8 (0.066)	50864	2794.6	1630.8	3600
ILS-pop	50801 (3/10)	50822.8 (0.043)	50861	415.5	1613.75	3600
`fl1400` opt: 20127						
ILS	20127 (5/10)	20147.9 (0.10)	20175	25255.6	842.9	3600
ILS-restart	20127 (10/10)	**20127.0** (0.0)	20127	9363.2	568.1	3600
ILS-fdd	20127 (10/10)	**20127.0** (0.0)	20127	48410.8	1612.8	3600
ILS-parallel	20127 (10/10)	**20127.0** (0.0)	20127	1173.3	1001.1	3600
ILS-rep-worst	20127 (10/10)	**20127.0** (0.0)	20127	1284.4	1006.1	3600
ILS-pop	20127 (10/10)	**20127.0** (0.0)	20127	366.1	1055.2	3600
`fl1577` opt: 22249						
ILS	22254 (1/10)	22379.8 (0.59)	22523	43187.7	2232.6	7200
ILS-restart	22249 (1/10)	22254.3 (0.024)	22259	80269.8	3664.8	7200
ILS-fdd	22249 (7/10)	**22249.6** (0.0031)	22254	102653.5	4786.8	7200
ILS-parallel	22254 (2/10)	22257.4 (0.038)	22264	6953.2	4425.2	7200
ILS-rep-worst	22254 (2/10)	22271.1 (0.099)	22402	7071.4	4339.3	7200
ILS-pop	22250 (2/10)	22267.9 (0.085)	22400	610.6	4649.2	7200

Table 5.5: Comparison of ILS extensions using the LK local search for symmetric TSPs on some TSPLIB instances. All algorithms are given the same overall number of LK applications. See Table 5.3 for a description of the entries.

Algorithm	best	average	worst	i_{avg}	t_{avg}	i_{max}
lin318 opt: 42029						
ILS	42029 (20/25)	42051.8 (0.054)	42143	1326.0	2090.9	5000
ILS-fdd	42029 (25/25)	**42029.0** (0.0)	42029	142.6	257.5	5000
ILS-parallel	42029 (25/25)	**42029.0** (0.0)	42029	30.0	496.7	500
ILS-rep-worst	42029 (25/25)	**42029.0** (0.0)	42029	33.6	546.9	500
pcb442 opt: 50778						
ILS	50778 (25/25)	**50778.0** (0.0)	50778	325.1	359.2	10000
ILS-fdd	50778 (25/25)	**50778.0** (0.0)	50778	351.2	448.03	10000
ILS-parallel	50778 (25/25)	**50778.0** (0.0)	50778	83.4	984.3	1000
ILS-rep-worst	50778 (22/25)	**50778.0** (0.0)	50778	76.8	914.3	1000
att532 opt: 27686						
ILS	27686 (17/25)	27694.5 (0.029)	27706	1834.6	1484.1	15000
ILS-fdd	27686 (25/25)	**27686.0** (0.0)	27686	3037.4	2672.6	15000
ILS-parallel	27686 (24/25)	27686.5 (0.002)	27698	435.2	3502.0	1500
ILS-rep-worst	27686 (22/25)	27688.1 (0.0077)	27705	233.5	1889.5	1500
rat783 opt: 8806						
ILS	8806 (24/25)	8806.04 (0.0)	8807	2087.9	636.9	10000
ILS-fdd	8806 (25/25)	**8806.0** (0.0)	8806	3676.7	1269.3	10000
ILS-parallel	8806 (23/25)	8806.1 (0.0)	8807	522.5	1632.9	1000
ILS-rep-worst	8806 (25/25)	**8806.0** (0.0)	8806	419.5	1336.0	1000
pr1002 opt: 259045						
ILS	259045 (10/10)	**259045.0** (0.0)	259045	3324.1	4367.8	10000
ILS-fdd	259045 (10/10)	**259045.0** (0.0)	259045	3211.3	4383.6	10000
ILS-parallel	259045 (7/10)	259045.9 (0.0)	259048	728.2	10964.2	1000
ILS-rep-worst	259045 (10/10)	**259045.0** (0.0)	259045	531.2	7333.5	1000
pcb1173 opt: 56892						
ILS	56892 (1/10)	56895.7 (0.0065)	56897	2071.3	1353.7	10000
ILS-fdd	56892 (3/10)	**56892.7** (0.0012)	56893	4932.4	3384.9	10000
ILS-parallel	56892 (2/10)	56893.7 (0.003)	56899	574.8	3828.8	1000
ILS-rep-worst	56892 (4/10)	56894.6 (0.021)	56901	526.5	3519.8	1000
pr2392 opt: 378032						
ILS	378032 (1/10)	378466.2 (0.11)	379025	10924.3	9049.3	20000
ILS-fdd	378032 (6/10)	**378121.4** (0.024)	378471	14013	12752.9	20000
ILS-parallel	378059 (1/10)	378214.0 (0.048)	378558	1596.3	14344.7	2000
ILS-rep-worst	378032 (3/10)	378257.2 (0.059)	378580	1405.4	12998.1	2000

Table 5.6: Comparison of ILS extensions for ATSP instances from TSPLIB. See Table 5.3 for a description of the entries.

Algorithm	best	average	worst	i_{avg}	t_{avg}	t_{max}
`ry48p` opt: 14422						
ILS	14422 (95/100)	14426.3 (0.03)	14507	8157.6	24.7	120
ILS-restart	14422 (25/25)	14422.0 (0.0)	14422	2991.4	9.2	120
ILS-fdd	14422 (25/25)	14422.0 (0.0)	14422	362.5	2.4	120
ILS-parallel	14422 (25/25)	14422.0 (0.0)	14422	87.1	6.9	120
ILS-rep-worst	14422 (25/25)	14422.0 (0.0)	14422	44.1	3.0	120
ILS-pop	14422 (25/25)	14422.0 (0.0)	14422	117.7	7.5	120
`ft70` opt: 38673						
ILS	38673 (14/25)	38694.7 (0.056)	38707	442.8	2.2	300
ILS-restart	38673 (25/25)	38673.0 (0.0)	38673	1423.3	7.4	300
ILS-fdd	38673 (25/25)	38673.0 (0.0)	38673	2219.3	16.7	300
ILS-parallel	38673 (25/25)	38673.0 (0.0)	38673	87.6	12.0	300
ILS-rep-worst	38673 (25/25)	38673.0 (0.0)	38673	90.8	10.8	300
ILS-pop	38673 (25/25)	38673.0 (0.0)	38673	56.4	8.3	300
`kro124p` opt: 36230						
ILS	36230 (17/25)	36451.7 (0.61)	36923	5991.9	22.8	300
ILS-restart	36230 (25/25)	36230.0 (0.0)	36230	2662.3	11.3	300
ILS-fdd	36230 (25/25)	36230.0 (0.0)	36230	1008.1	6.4	300
ILS-parallel	36230 (25/25)	36230.0 (0.0)	36230	162.9	17.5	300
ILS-rep-worst	36230 (25/25)	36230.0 (0.0)	36230	96.4	8.9	300
ILS-pop	36230 (25/25)	36230.0 (0.0)	36230	75.8	9.6	300
`ftv170` opt: 2755						
ILS	2755 (17/25)	2757.8 (0.10)	2764	9057.7	35.3	300
ILS-restart	2755 (25/25)	2755.0 (0.0)	2755	10041.8	45.3	300
ILS-fdd	2755 (23/25)	2755.7 (0.025)	2764	5917.8	32.8	300
ILS-parallel	2755 (25/25)	2755.0 (0.0)	2755	559.9	64.7	300
ILS-rep-worst	2755 (25/10)	2755.0 (0.0)	2755	410.6	39.9	300
ILS-pop	2755 (25/25)	2755.0 (0.0)	2755	157.1	33.1	300
`rbg443` opt: 2720						
ILS	2720 (25/25)	2720.0 (0.0)	2720	1251.3	31.7	900
ILS-restart	2720 (25/25)	2720.0 (0.0)	2720	1265.8	36.3	900
ILS-fdd	2720 (25/25)	2720.0 (0.0)	2720	1154.6	33.4	900
ILS-parallel	2720 (25/25)	2720.0 (0.0)	2720	478.6	105.7	900
ILS-rep-worst	2720 (25/25)	2720.0 (0.0)	2720	397.9	84.9	900
ILS-pop	2720 (25/25)	2720.0 (0.0)	2720	227.8	86.0	900

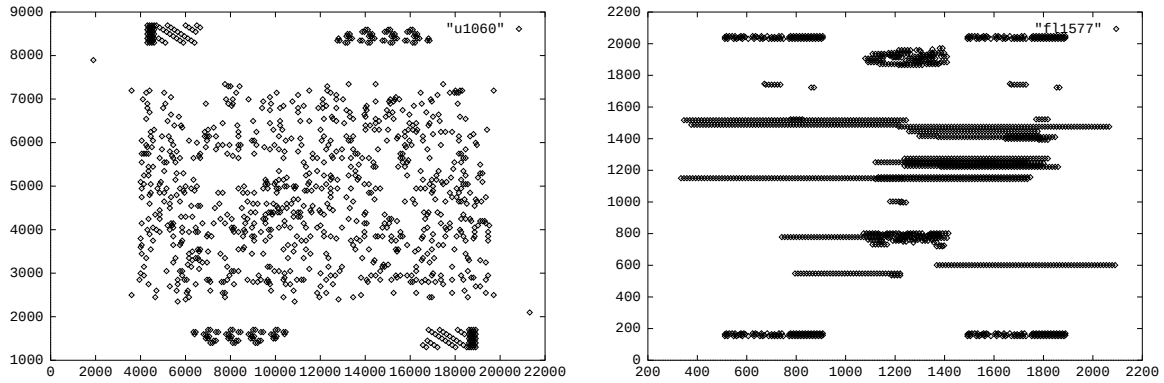

Figure 5.8: TSP instance `u1060` (left side) and `fl1577` (right side) from TSPLIB. Both instances show isolated clusters of cities.

5.2.7 Comparison with $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System and Related Work

An obvious question is how the ILS extensions compare to $\mathcal{MAX}$–$\mathcal{MIN}$ ant system ($\mathcal{MM}$AS), which also has been applied to symmetric and asymmetric TSPs in Chapter 4 and which uses the same local search implementation as the ILS algorithms. So far, $\mathcal{MM}$AS has been compared to our standard iterated `3-opt` implementation and has shown to be competitive on symmetric TSPs and superior on asymmetric TSPs. Yet, the ILS extensions change this first conclusion. On symmetric TSPs the best performing ILS extension, ILS-fdd, performs on most instances significantly better than $\mathcal{MM}$AS (see computational results in Sections 4.3.6 and 4.6.3). Comparable results are obtained with the improved version of $\mathcal{MM}$AS (we will call it $\mathcal{MM}$AS-restart in the following) only on the two smallest instances (`d198` and `lin318`) and the two larger instances `pcb1173` and `d1291`, although at the cost of higher run-times. Compared to the population-based ILS extensions, $\mathcal{MM}$AS-restart shows somewhat better performance on some single instances like `rat783` but overall the computational results are slightly in favor of the ILS-based extensions if we additionally consider that some instances appear to be rather hard to solve for $\mathcal{MM}$AS. Such hard instances for $\mathcal{MM}$AS are `u1060` and `fl1577`, which are plotted in Figure 5.8. These two instances show isolated clusters of cities and one of the main difficulties is to find appropriate connections between the clusters. It is this aspect in which these two instances mainly differ from the others. (These instances are also hard for the `LK` heuristic in the sense that the run-times increase very strongly. This is also the reason that we did not apply ILK to these instances.) On such instances the iterated `3-opt` extensions are the more robust algorithms.

The same conclusion as for the algorithms based on `3-opt` also holds for those applying the `LK` heuristic: our improved ILK versions perform better than $\mathcal{MM}$AS using `LK` local search, too (see results given in Table 4.11 on page 90). ILS-fdd returns better average solution qualities at much lower run-times.

Our ILS extensions also compare more favorably to recently published results on the TSP which have already been discussed in Section 4.3.7 (page 92). The solution quality obtained with ILS-fdd using the `LK` local search is competitive to those reported by the best performing

algorithms for the TSP [174, 181, 187, 257]. Certainly, our computation times are still much higher, yet they could be significantly improved by using a faster implementation of the LK heuristic. In fact, one of the fastest implementations appears to be due to Johnson [129]. For example, he reports that 532 ILK iterations on instance att532 can be run in 29 seconds on a SGI Challenge machine on a 150 Mhz MIPS R4400 processor. Since the performance results of his LK implementation are comparable to the LK implementation we are using [166], we estimate that this instance could, on average, be solved to optimality in approximately 165 seconds. Such a result compares very favorably to the best performance results we are aware of [181]. Hence, one possible issue for future work would be to test our proposed improvements with such a fast LK implementation.

5.3 Iterated Local Search for the QAP

In this section we present the application of iterated local search to the quadratic assignment problem (QAP) which also is the first time that ILS is applied to that problem. (For a short description of the QAP, we refer to Section 4.4.1 on page 94.) When applying ILS to the TSP, the run-time distributions have shown that ILS algorithms often suffer from stagnation behavior. Therefore, we tested the generality of these findings with a straightforward ILS application for the QAP. In the following we first present the details of the ILS algorithm and then give evidence, that ILS algorithms for the QAP show the same kind of stagnation behavior. Last, we present an experimental investigation of ILS algorithms using acceptance criteria which allow moves to worse solutions and of the population-based ILS extensions proposed for the application to the TSP.

5.3.1 Generic Algorithm Choices

In the following we present the generic choices concerning the implementation of the operators LocalSearch, Modify, and AcceptanceCriterion used in our algorithmic framework.

Choice of LocalSearch

For the application of ILS to the QAP we use a 2-opt iterative improvement algorithm (referred to as 2-opt in the following) as well as short runs of the robust tabu search algorithm (Ro-TS) [241], as also done in the application of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system ($\mathcal{MM}$AS) to the QAP. We refer to Section 4.4.3 for the details on the local search algorithms for the QAP.

We tested two versions of the 2-opt algorithm, one with a *first-improvement* pivoting rule, the other with a *best-improvement* pivoting rule. When using 2-opt with a first-improvement strategy we use don't look bits to speed up the local search. Don't look bits have first been used in the context of local search for the TSP [21]. We have extended this technique in a straightforward way to the QAP (see Section 4.3.1 for a description of this technique applied to the TSP). For the QAP we associate with every item a don't look bit. When first applying local search, all don't look bits are turned off (set to 0). If for an item no improving move is found, the don't look bit is turned on (set to 1) and the item is not considered as a starting item in the next local search iteration. If an item is involved in a move and changes its location, the don't look bit is turned off. This technique considerably improves the speed of the first-improvement

local search. Recall from the description of the local search algorithm for the QAP that the first iteration of a best-improvement local search takes time $\mathcal{O}(n^3)$, while the following iterations can be run in $\mathcal{O}(n^2)$ (see Equation 4.23 on page 96). But when using a first-improvement local search algorithm, every iteration would need time $\mathcal{O}(n^3)$. By applying don't look bits, this advantage of a best-improvement local search diminishes and a first-improvement local search may even run faster than a best-improvement local search.

Choice of Modify

Initially, we experimented with different possible kick-moves like exchanging a certain number of pairs or triples of items. Finally, the simplest criterion resulted to be to exchange a fixed number of randomly chosen items which corresponds to a random move of the `k-opt` neighborhood. Since this first implementation, we changed our approach slightly and now we modify the kick-move strength as proposed in simple variable neighborhood search (VNS) [109]. This is done by varying the value of k between some minimal and maximal values k_{min} and k_{max} as follows. Let k be the current neighborhood size. If the application of Modify and the subsequent local search does not yield a better solution, we set $k = k + 1$; if a better solution is found, we set $k = k_{min}$. If we reach the upper limit k_{max} without having found an improved solution, we set k to k_{min} and repeat the same cycle. The main effect of using this scheme of varying k is to give slightly improved results compared to a fixed choice of k and a reduction of the effort for parameter tuning of the kick-move strength. It should be noted here that the systematic change of the kick-move strength is a simple use of the search *history* as proposed in our algorithmic framework for ILS (see Figure 5.1).[4]

Choice of AcceptanceCriterion

As our standard acceptance criterion we use *Better*(s, s''), which was also used for the TSP. This choice is also done in simple VNS [109]. In the context of VNS it is argued that diversification of the search is achieved by allowing larger modifications to the current solution, that is, by large values for k. Yet, a disadvantage of doing so is that for large k, the so modified solution will be rather similar to a randomly generated solution and the local search will need long computation time to find the next local minimum. Also, it is also not very clear whether using a high value for k alone is sufficient to allow the algorithm to escape from bad solutions. An analysis of the run-time behavior of ILS for the QAP will show that, in fact, this is not the case and that the basic ILS algorithm shows stagnation behavior.

Initial Solution

As the initial solution we use a randomly generated assignment of items to locations.

Implementation Details

When applying the `2-opt` first-improvement local search, the don't look bits can be reset in different ways, similar to the variants described in Section 5.2.3. For the QAP we considered

[4] Independently from our ILS approach, preliminary results for a VNS application to the QAP have been presented by Eric Taillard in a talk at the Metaheuristics conference 1997.

two possibilities, one is to reset all don't look bits to zero, the other consists in resetting only the don't look bits of items which are affected by a modification. Some experiments suggested that the latter possibility gives considerably better results, since it allows to run much more often a local search for the same given CPU-time. Hence, this resetting strategy is used in the following when applying first-improvement `2-opt`.

Another implementation aspect concerns the step size to increase the value of k and using the Ro-TS local search. Ro-TS returns, in general, better solutions than `2-opt`. If the kick-move strength is too small, we found that Ro-TS has a high probability of returning the same locally optimal solution from which we started. To avoid this, we calculate the distance between the starting solution and the solution returned by Ro-TS; the distance is measured as the number of items which differ in their position in the two solutions. If the tabu search returns the same solution, we assume that the kick-move strength has to be increased more strongly and we set k to $k + \max\{3, n/15\}$, where n is the instances size. When using the `2-opt` iterative improvement algorithm, k is increased by one if no improved solution has been found. Initially, we set k_{min} to 5 when using Ro-TS and to 3 when using `2-opt`; k_{max} is always set to $0.9n$.

5.3.2 Run-time Distributions for ILS Applied to the QAP

In this section we analyze the run-time behavior of ILS for the QAP with a focus on the variant which uses the first-improvement `2-opt` local search. As discussed in Section 4.4.4 (from page 98 on), the QAP instances of QAPLIB [39] can be classified into four different problem classes. Consequently, we give in Figure 5.9 empirical run-time distributions for one instance from each class. The instances tested are `tai30a` (a randomly generated instance with matrix entries distributed according to a uniform distribution), `sko42` (an instance with randomly generated flows and the distance is the Manhattan distance between points on a two-dimensional grid), `ste36a` (an instance taken from a real-life problem), and instance `tai35b` (an instance which is randomly generated in such a way that it resembles the structure of real-life instances). For each instance we give empirical run-time distributions based on 100 runs of the ILS algorithm with respect to various bounds on the solution quality. Additionally, we also plot the cumulative distribution function of an exponential distribution (indicated by $f(x)$) which indicates whether the empirical run-time distributions show stagnation behavior (recall the discussion in Section 3.2.3, from page 50 on).

The empirical run-time distributions show that the ILS algorithm is able to find the best known solutions with a relatively high probability on three of the four instances. An exception is instance `tai30a`, which is very hard for our ILS algorithm. Only in one of 100 runs the best known solution could be found. Yet, it is well known that for this type of instances it is hard to find optimal solutions [244].

When trying to find high quality solutions or the best known solutions, respectively, the run-time distributions on all instances show stagnation behavior, independent of the problem class. This fact can be observed in the much slower increase of the empirical run-time distributions when compared to the plotted exponential distribution. Hence, as for the application of ILS to the TSP we can conclude that the straightforward ILS algorithm for the QAP can be improved, in the simplest case, by restarting the algorithm after some appropriate cutoff time.

As shown by the application of $\mathcal{MM}$AS to the QAP (see Section 4.4), the type of local search algorithm used has a considerable influence on the performance of a hybrid algorithm.

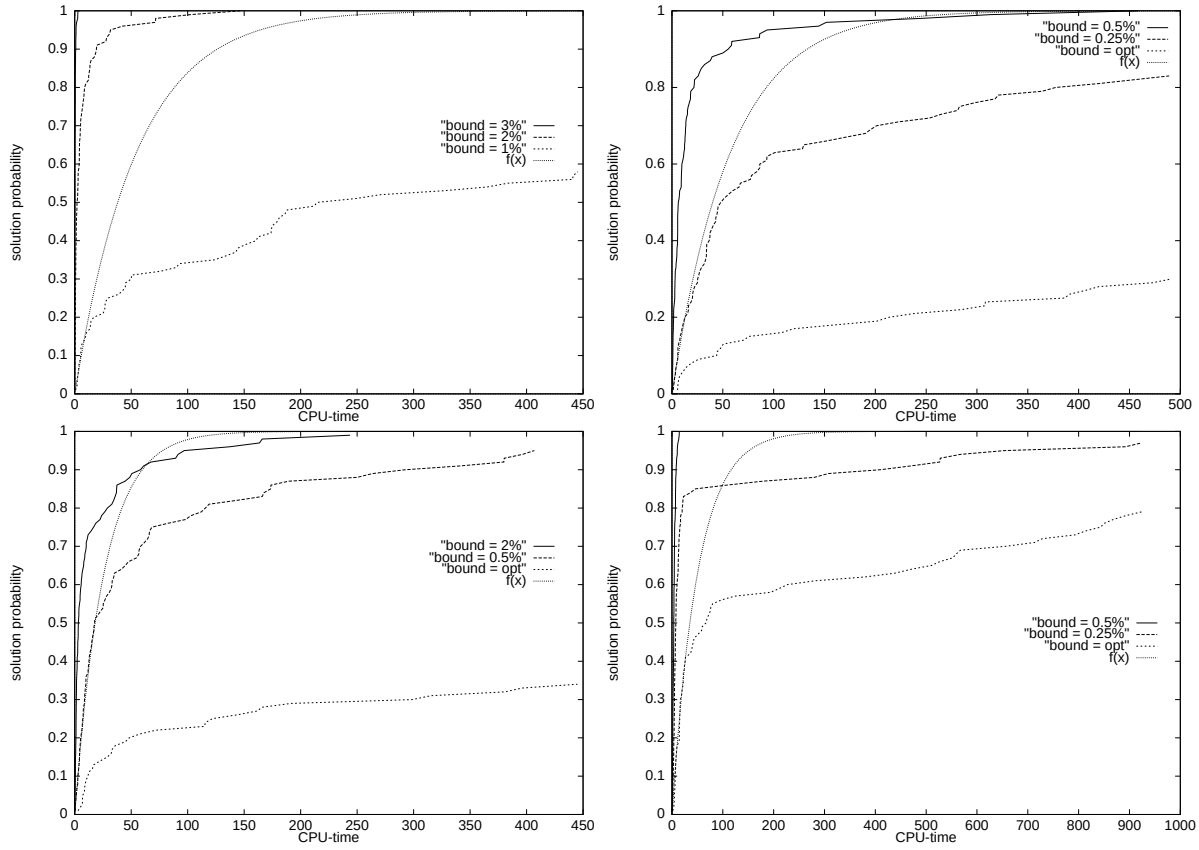

Figure 5.9: Run-time distributions measured for QAP instances with iterated `2-opt` using a first-improvement pivoting rule. Given are the run-time distributions for `tai30a` (upper left side), `sko42` (upper right side), `ste36a` (lower left side) and `tai35b` (lower right side). The ILS algorithm uses first-improvement local search and only accepts better quality solutions. The run-time distributions are based on 100 independent runs of each algorithm; $f(x)$ indicates an exponential distribution.

Therefore, it can be expected that the findings for the $\mathcal{MM}$AS application also carry over to the ILS algorithm, because it is well known that ILS performance depends on the local search algorithm applied. To verify this conjecture, we measured run-time distributions of our ILS algorithm using the three local search variants proposed before. In Figure 5.10 are given the run-time distributions of ILS using first-improvement `2-opt` (ILS-first), ILS using best-improvement `2-opt` (ILS-best), and ILS using short Ro-TS runs (ILS-tabu) of length $4n$. Additionally, we plot the cumulative distribution function of an exponential distribution adjusted to the leftmost run-time distribution.

A surprising result is that for each instance ILS-tabu obtains the highest probability of finding the required solution quality on the long run. Yet, this observation has to be rectified, if we consider the possibility of adaptively restarting ILS algorithms when signs of search stagnation are encountered. In such a case, on instance `tai35b` (lower right side in Figure 5.10) it would be preferable to use a simple first-improvement `2-opt` algorithm. In fact, the run-time distributions of ILS-first and ILS-tabu cross over after ca. 100 seconds. If restart with optimal cutoff values would be used, the plotted exponential distribution (indicated by $f(x)$) gives an

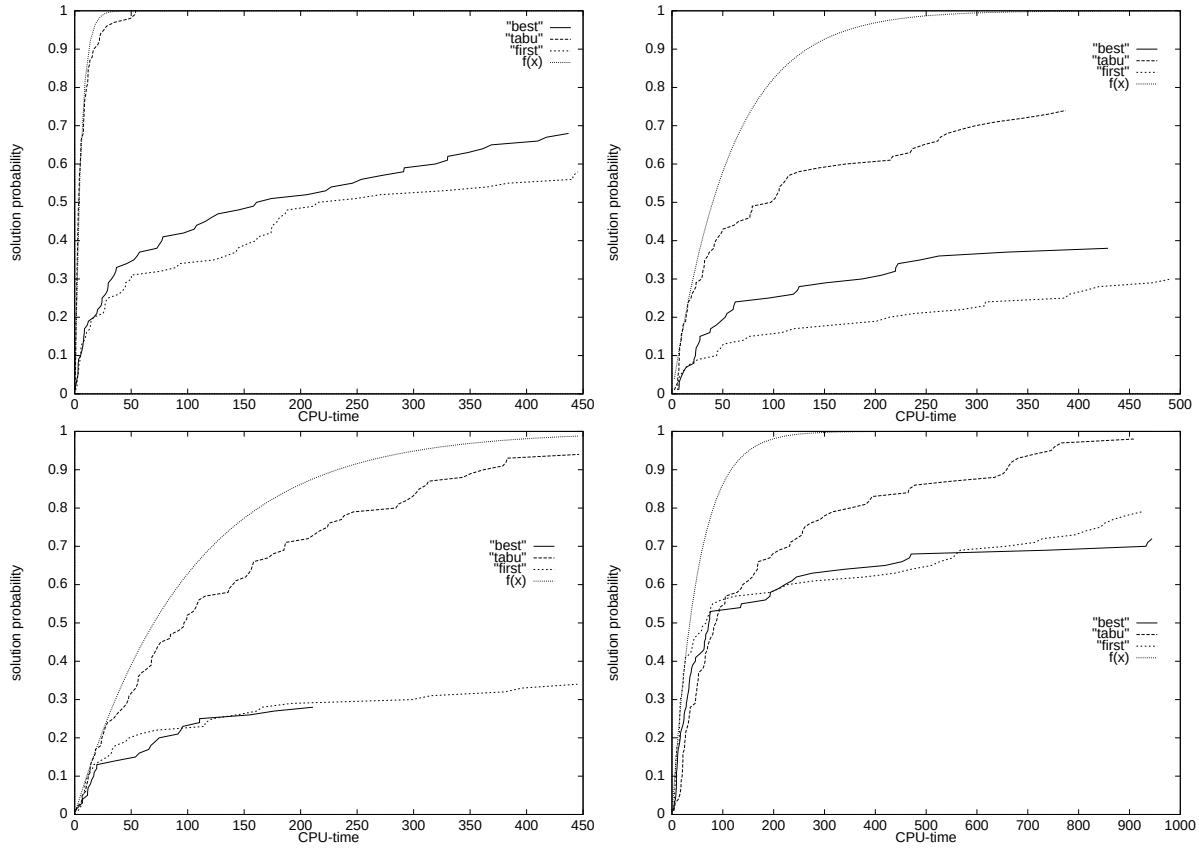

Figure 5.10: Comparisons of local search algorithms (`2-opt` with a first- and best-improvement pivoting rule and Ro-TS) in ILS (the different local search algorithms are indicated by "first", "best", and "tabu"). Given are the run-time distributions for `tai30a` (upper left side), `sko42` (upper right side), `ste36a` (lower left side) and `tai35b` (lower right side) with respect to the best known solution, except on instance `tai30a` where we required to find a solution within 1% of the best known. $f(x)$ indicates an exponential distribution.

indication of the possible development of the solution probability for ILS-first. Certainly, also ILS-tabu can be improved by using restarts, but the resulting run-time distribution would be below the one obtained when using ILS-first with restarts.

The run-time distributions show that additional diversification in the ILS algorithm does not have such a strong impact on the performance of ILS-tabu as on ILS-first or ILS-best; ILS-tabu appears to be much less affected from the stagnation behavior than ILS-first and ILS-best. Yet, for ILS-tabu occasional restart would still yield a better performance on the long run. This fact suggests, as observed when using the `LK` heuristic in ILS for the TSP, that the more powerful a local search algorithm is, the less the ILS algorithm will be affected by stagnation behavior.

ILS-first and ILS-best show very similar behavior on all the instances. Their run-time distributions are very similar and reach comparable final solution probabilities. Yet, some initial experimental results with ILS extensions, which are described in the next section, have shown that a slightly better performance as been obtained with the first-improvement local search. Therefore, we will only present experimental results using that `2-opt` variant in the following.

5.3.3 Extended Acceptance Criteria

The analysis of the run-time behavior of the ILS algorithms for the QAP indicates that they suffer from stagnation behavior which may severely compromise their performance. This is particularly true if for the local search `2-opt` is used while with the use of Ro-TS the stagnation phenomena are less notable. As argued also for the TSP, the ILS algorithms for the QAP can be substantially improved by a stronger exploration of the search space which can be achieved by acceptance criteria which accept moves to worse solutions. The most straightforward of these has been an acceptance criterion, called *Restart*(s, s''), corresponding to an occasional restart of the algorithm. Here, we will present an experimental investigation of several other possible acceptance criteria which accept worse solutions, yet, without resorting to restart the algorithm from a new, randomly generated solution.

Exploration of the search space is extremely favored if s'' is always accepted as the new solution, in the following denoted as *RandomWalk*(s, s''). Such an acceptance criterion may have the advantage that after small modifications, the following local search will return the next local minimum very fast. Hence, for a given amount of computing time, much more often a local search can be applied than when starting from random initial solutions.

A bias towards better solutions can be introduced by accepting worse solutions only with a certain probability. This is achieved by a simulated annealing type criterion which has been used in [169, 157] for applications of ILS to the travelling salesman problem and the job-shop scheduling problem, respectively. We denote this acceptance criterion *LSMC*(s, s''), reminiscent of the term *large step Markov chains* for one of the first ILS algorithms proposed in [169]. In particular, s'' is accepted with probability $\exp\{(f(s) - f(s''))/T\}$ if s'' is worse than s; if s'' is better than s, then s'' is always accepted. T is a parameter called temperature and it is lowered during the run of the algorithm (see Section 2.7.4 on page 29 for a short description of simulated annealing). In fact, when applying *LSMC*(s, s''), the temperature schedule we use is a non-monotonic temperature schedule as those proposed in [124] for simulated annealing or those used in tabu thresholding [93]. We increase the temperature to an intermediate level if the minimal temperature is reached.

A disadvantage of using *LSMC*(s, s'') is that a schedule for reducing the parameter T has to be given. A simpler idea may be to use a constant temperature T_c. We refer to this acceptance criterion as *ConstTemp*(s, s''). Hence, one only has to fix one single parameter.[5]

Similar to non-monotonic temperature schedules is the acceptance criterion which is called *Reheating*(s, s''). It is derived from *LSMC*(s, s'') by setting the parameter T close to zero, that is, basically only better local minima are accepted. If no better solution is found for a number of iterations the parameter T is increased to some large value for a number of iterations and then is reset to a value close to zero.

All the acceptance criteria proposed, apart from *Better*(s, s''), have in common that worse solutions are accepted with some probability and therefore the ILS algorithm may avoid getting trapped. The computational results we present in Tables 5.7 and 5.8 show that accepting occasionally worse solutions improves considerably the computational results.

[5] In fact, we have tested the acceptance criteria *Reheating*(s, s'') and *ConstTemp*(s, s'') also on the symmetric TSP. Although they showed good results, they did not show better results than *Restart*(s, s'') and had major problems to find improved solutions on the clustered instances like those which are given in Figure 5.8.

5.3.4 Experimental Results

The comparison of the different possible choices for LocalSearch has shown that the ILS variants which use `2-opt` local search can be expected to profit most by other acceptance criteria. Hence, we focus on ILS using the first-improvement `2-opt` algorithm. In particular, we give computational results obtained with the standard ILS algorithm (ILS using acceptance criterion *Better*(s, s''); we refer to this variant as ILS), ILS using acceptance criteria *LSMC*(s, s'') (referred to as LSMC), *RandomWalk*(s, s'') (referred to as RW), *Restart*$(s, s'', history)$ (referred to as Restart), and with the population-based extensions "replace-worst" and "population-ILS" (see page 133 for a description of these two extensions); the population-based extensions are referred to as ILS-rep-worst and ILS-pop, respectively. Additionally, we also give results obtained with ILS and ILS-restart when using Ro-TS for the operator LocalSearch; the latter two are referred to as ILS-tabu and Restart-Tabu, respectively.

The parameter settings of the different versions are as follows. In LSMC an annealing schedule like for simulated annealing has to be defined. We do this by first generating 40 locally optimal solutions and then calculating the standard deviation σ of their solution quality. The initial temperature T_0 is then set to $\sigma/3$ and the final temperature T_{end} to $\sigma/15$. The temperature is lowered every 50 iterations according to a geometric cooling, that is, we set $T_{n+1} = 0.95 \cdot T_n$. If T is smaller than T_{end}, T is set to T_0. In Restart, we generate a new random initial solution if no improved solution is found for $2n$ iterations. Both population-based extensions use a population size of 40. In ILS-rep-worst every 3 iterations the best solution replaces the worst one. In ILS-pop in every iteration a solution is chosen randomly and one single ILS iteration is applied. If a better solution is found, it replaces the worst solution of the population.

The computational results are given in Table 5.7 for the unstructured instances and in Table 5.8 for the structured ones. The computational results obtained with *ConstTemp*(s, s'') have been very similar to those of *LSMC*(s, s'') and those of *Reheating*(s, s'') have been close to those of *Restart*(s, s'') and therefore we do not explicitely give them in the tables. A first conclusion from the computational results is that considerable improvements over standard ILS are possible. On almost all problem classes, the ILS-extensions obtain a significantly better average solution quality. Yet, whether or how much an extension of ILS does improve the performance of standard ILS strongly depends on the problem class.

Unstructured instances For the unstructured QAP instances, all extensions achieve a much improved performance over ILS. Among the ILS variants using the `2-opt` local search, LSMC and RW are the best performing ones. The good performance of RW on the unstructured instances is certainly due to the fact that no bias towards higher quality solutions is used in RW; for the unstructured instances it appears to be a good idea to sample as many local minima as possible, which is exactly done when using RW. Yet, on these instances ILS algorithms using Ro-TS local search perform best, with RS-TS showing slightly better results than ILS-tabu. Yet, an interesting fact is that the average solution quality obtained with LSMC and RW are close to those of the ILS variants using Ro-TS on the instances with grid-distance matrices (upper half of Table 5.7). Additionally, we run an ILS algorithm with acceptance criterion *RandomWalk*(s, s'') using Ro-TS local search on the instances `taixxa` and an even better performance could be obtained (averaging 0.493% above the best known solution compared to 0.732% and 0.638% for ILS-tabu and RS-TS, respectively). Hence, also when using tabu search in the ILS algorithm, the acceptance criterion can have a significant influence on the performance.

Table 5.7: Experimental results for ILS and several extensions on QAP instances taken from QAPLIB for unstructured instances. Given is the percentage deviation from the best known solution over 10 independent runs of the algorithms. Additionally we have computed the average percentage deviation for each algorithm on the specifi c problem class, indicated by Average. See the text for a description of the variants used.

instance	ILS	LSMC	RW	Restart	ILS-rep-worst	ILS-pop	ILS-tabu	RS-TS	t_{max}
random flows on grids									
nug30	0.219	0.049	0.052	0.036	0.026	0.026	0.026	0.026	30
sko42	0.269	0.019	0.027	0.084	0.115	0.043	0.020	0.022	92.0
sko49	0.226	0.068	0.186	0.115	0.143	0.182	0.054	0.053	135
sko56	0.418	0.110	0.133	0.237	0.194	0.212	0.132	0.088	211
sko64	0.413	0.100	0.086	0.179	0.137	0.323	0.083	0.068	308
sko72	0.383	0.180	0.175	0.225	0.276	0.272	0.166	0.124	455
sko81	0.586	0.130	0.155	0.304	0.164	0.413	0.097	0.159	656
sko90	0.576	0.221	0.187	0.356	0.262	0.453	0.268	0.176	895
Average	0.374	0.109	0.125	0.192	0.165	0.241	0.106	0.089	
random problems with entries uniformly distributed									
tai20a	0.723	0.661	0.542	0.614	0.596	0.517	0.217	0.229	9
tai25a	1.181	0.919	0.896	0.733	0.989	0.706	0.525	0.446	17
tai30a	1.304	0.975	0.989	0.955	1.017	0.964	0.367	0.458	30
tai35a	1.731	1.184	1.113	1.066	1.357	1.191	0.746	0.595	51
tai40a	2.036	1.077	1.490	1.547	1.496	1.454	0.863	0.559	75
tai50a	2.127	1.434	1.491	1.761	1.669	1.780	0.988	0.988	150
tai60a	2.200	1.744	1.692	1.974	1.930	2.573	1.321	1.053	265
tai80a	1.775	0.789	1.200	1.461	1.420	1.394	0.827	0.778	670
Average	1.634	1.098	1.177	1.264	1.309	1.323	0.732	0.638	

Structured instances On the structured instances, again much better performance could be achieved with the ILS extensions. On the real life instances, the best performance is obtained with the population-based extensions ILS-pop and ILS-rep-worst followed by Restart; the same is true for the randomly generated real-life like instances. Especially on these latter instances the relatively poor average performance of LSMC and RW compared to ILS is striking. For RW the explanation is that these instances contain structure and by accepting indiscriminately every new local minimum, this structure cannot be exploited. Yet, LSMC has a bias towards better solutions and the relatively poor performance on instances `taixxb` is rather astonishing. The main reason may be that LSMC has problems escaping from large attraction areas in the search space. The worse performance on the latter instances is mainly due to the very bad performance on one single instance (`tai40b`). A closer look at the development of the kick-move strength showed that LSMC did not reach large values for k and therefore probably got stuck at poor local minima. On the other side, ILS could avoid some very poor solutions by finding improved solutions after relatively large kick-moves. Another interesting fact is that the two ILS algorithms which use Ro-TS perform worse than the other extensions (except RW) and on instances `taixxb` even worse than the standard ILS algorithm using `2-opt`.

The experimental results for the QAP show that the right choice of the acceptance criterion is important for ILS algorithms to achieve peak performance. Yet, there is also one further aspect of our results. Recall that we modify the kick-move strength during the algorithm's run

Table 5.8: Experimental results for ILS and several extensions on QAP instances taken from QAPLIB for structured instances. See Table 5.7 for a description of the entries.

instance	ILS	LSMC	RW	Restart	ILS-rep-worst	ILS-pop	ILS-tabu	RS-TS	t_{max}
real life instances									
bur26a-h	0.0	0.003	0.001	0.0	0.0	0.0	0.004	0.0	44
kra30a	0.672	0.276	0.0	0.090	0.090	0.0	0.535	0.291	30
kra30b	0.094	0.023	0.046	0.026	0.029	0.036	0.026	0.015	30
ste36a	0.377	0.011	0.451	0.109	0.086	0.128	0.197	0.099	54
ste36b	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	54
Average	0.229	0.062	0.099	0.045	0.041	0.033	0.152	0.081	
randomly generated real-life like instances									
tai20b	0.045	0.0	0.045	0.0	0.0	0.0	0.0	0.0	10
tai25b	0.0	0.0	0.007	0.0	0.0	0.0	0.0	0.0	41
tai30b	0.0	0.0001	0.093	0.0	0.0003	0.0	0.0	0.035	73
tai35b	0.131	0.0	0.081	0.048	0.019	0.018	0.094	0.108	117
tai40b	0.0	0.604	0.204	0.0	0.0	0.0	0.202	0.403	177
tai50b	0.203	0.116	0.282	0.035	0.018	0.015	0.215	0.181	348
tai60b	0.029	0.001	0.645	0.015	0.025	0.016	0.417	0.036	633
tai80b	0.785	0.618	0.703	0.330	0.420	0.285	1.011	1.118	1510
Average	0.149	0.167	0.257	0.053	0.060	0.042	0.242	0.235	

as proposed in simple variable neighborhood search. Therefore, our experimental analysis also reveals that an increase of the kick-move strength alone is not sufficient to obtain best possible performance in an ILS algorithm. The better idea apears to be to choose the right acceptance criterion.

5.3.5 Comparison with $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System and Related Work

Similar to ILS, the choice of the local search algorithm has a considerable influence on the performance of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system ($\mathcal{MM}$AS). When using $\mathcal{MM}$AS, the performance has been best with tabu search on the unstructured and the small real-life instances, while on the randomly generated real-life like instances, the best solution quality was obtained was when using a best-improvement `2-opt` local search. $\mathcal{MM}$AS, in general, obtains better results than the standard ILS algorithm, but compared to the ILS extensions, the advantage of $\mathcal{MM}$AS gets smaller. Still, $\mathcal{MM}$AS+`2-opt` local search achieves the overall best results on instances `taixxb` among all algorithms compared. On the unstructured instances, $\mathcal{MM}$AS+`2-opt` performs slightly worse than, for example, ILS-restart, while when using tabu search, the solution quality achieved with $\mathcal{MM}$AS is better than that of ILS-tabu or RS-TS. Hence, it appears to be better to combine tabu search in a hybrid algorithm with $\mathcal{MM}$AS rather than with the ILS algorithms. In general, $\mathcal{MM}$AS compares much more favorably to ILS algorithms for the QAP than for the TSP. Especially for large, structured QAP instances, $\mathcal{MM}$AS seems to be the better choice.

5.4 Iterated Local Search for the FSP

In this section we present the application of iterated local search to the permutation flow shop problem (FSP). (For a short description of the FSP, we refer to Section 4.5.1 on page 94.) It is the first time that this problem is tackled with ILS and the computational results are very promising. For example, with the proposed algorithm we could improve the best known solutions for some benchmark instances which have often been used to test other algorithms [191, 206, 208, 242, 260]. We analyze the impact of the acceptance criterion on ILS performance for the FSP and compare the effectiveness of ILS to the best local search algorithms known for the FSP. But first, we give details on the choices made for the three generic procedures Modify, LocalSearch, and AcceptanceCriterion in our ILS approach.

5.4.1 Generic Algorithm Choices

Choice of LocalSearch

In our ILS approach we use the same iterative improvement algorithm as in the $\mathcal{MM}$AS application to the FSP which has shown to be fast and to produce very good quality solutions. The details of the implementation have been presented in Section 4.5.3 on page 107.

Choice of Modify

For Modify we consider simple modifications that cannot be reversed directly by the local search algorithm. We proceed by applying a number of swap-moves or at least one interchange-move. In a swap-move $\pi = (\pi(1), \ldots, \pi(i), \pi(i+1), \ldots, \pi(n))$ is modified to $\pi' = (\pi(1), \ldots, \pi(i+1), \pi(i), \ldots, \pi(n))$; in an interchange-move $\pi = (\pi(1), \ldots, \pi(i), \ldots, \pi(j), \ldots, \pi(n))$ is modified to $\pi' = (\pi(1), \ldots, \pi(j), \ldots, \pi(i), \ldots, \pi(n))$. Initially, we tried also moves by simultaneously exchanging jobs at three or four randomly chosen positions, but these moves did not improve the results. Experimentally, we found that rather small modifications are sufficient to yield very good performance. Yet, the appropriate kick-move strength seems to be instance dependent. For the experimental results in Section 4.6.2 the kick-move is composed of two swap-moves at randomly chosen positions $i = 1, \ldots, n-1$ and one interchange-move with $|i - j| \leq \max\{n/5, 30\}$. The restriction on the distance of two jobs in an exchange-move is imposed to avoid a strong disruption of an already good solution.

Choice of AcceptanceCriterion

As the standard acceptance criterion we again use *Better*(s, s'') which only accepts better solutions. Additionally, we applied also the following ones, which have also been used for the ILS application to the QAP in Section 5.3.3. In particular, we applied *RandomWalk*(s, s'') (accept every new local minimum), *LSMC*(s, s'') (a simulated annealing type acceptance criterion), *ConstTemp*(s, s'') (similar to the simulated annealing type acceptance criterion but using constant temperature T_c), and finally *Reheating*(s, s'') (accept after a number of non-improving moves worse solutions with a high probability). Yet, for the FSP we do not present such an extensive analysis of the run-time behavior as done in the previous two sections. Nevertheless, we will give evidence that also for the FSP the acceptance criterion *Better*(s, s'') is not the best possible choice (see Section 5.4.2).

Initial Solution

We use the NEH heuristic [188] to construct an initial solution. The NEH heuristic is the best performing construction heuristic for the permutation flow shop problem on a wide variety of problem instances [250, 144]. For the NEH we use the efficient implementation due to Taillard [240] such that NEH can be run in $\mathcal{O}(n^2m)$. Obviously, the ILS algorithm could also be started from random initial solutions. Yet, for short run times on large instances $n \geq 100$ we observed that with the initial solution generated by the NEH-heuristic a slightly better solution quality could be obtained.

5.4.2 Experimental Results

We compare the proposed ILS algorithm to a previously proposed simulated annealing algorithm by Osman and Potts [193] (SAOP), to the $\mathcal{MM}$AS approach to the FSP presented in Section 4.5, and to tabu search algorithms of the literature [191, 242]. The tabu search implementation of Nowicki and Smutnicki (Tabu-NS) [191] is the best performing local search algorithm with respect to solution quality produced for a fixed number of local search iterations as well as computation time. The comparison with SAOP and $\mathcal{MM}$AS is done for short computation times as taken by SAOP. For the comparison to the tabu search algorithms we concentrate on the same number of local search iterations as we are more interested to show that the proposed ILS algorithm is an easy way to improve local search performance and not to set new records in computational speed for the solution of the FSP. Before presenting the computational results, we investigate the dependence of ILS performance on the acceptance criterion next.

Experimental Comparison of Acceptance Criteria

The analysis of the run-time behavior of ILS for the TSP and the QAP suggest that the choice of the acceptance criterion is especially important for long runs. To verify whether this is also true for the FSP, we experimentally tested the proposed acceptance criteria on some instances. In Figure 5.11 we present indicative plots of the run-time's cumulative distribution for different acceptance criteria on two instances and one plot of the development of the average solution quality with increasing run-time. The run-time distributions are given for the instances, `ta021` with $n = 20$ jobs and $m = 20$ machines and `ta053` ($n = 50, m = 20$). For `ta021` the optimal solution is known and we plot the empirical run-time distributions to find an optimal solution. Instance `ta053` is harder to solve and two run-time distributions are plotted, one for reaching a solution within 2% and another for obtaining a solution within 1% of the best known upper bound.

Except for *Better*(s, s''), all the other acceptance criteria allow moves to worse local minima. The graphs in Figure 5.11 show, that acceptance criterion *RandomWalk*(s, s'') performs worst and we therefore conclude that a bias towards better local minima is necessary to yield best ILS performance. Among the tested acceptance criteria with such a bias, *Better*(s, s'') resulted to give the worst performance as can be observed in all plots of Figure 5.11. (Interestingly, the lower the bound is chosen for the run-time distributions, the stronger visible is the performance difference between the acceptance criteria; this fact can be observed when comparing the run-time distributions for instance `ta053` for bounds 2% and 1%.) Thus, only accepting better solutions seems to lead to stagnation of the search around suboptimal solutions and the

experimental results for the FSP confirm the findings of the ILS applications to the TSP and the QAP.

For the following experiments we choose the acceptance criterion *ConstTemp*(s, s''), because it has shown very good performance and it involves only one single parameter.

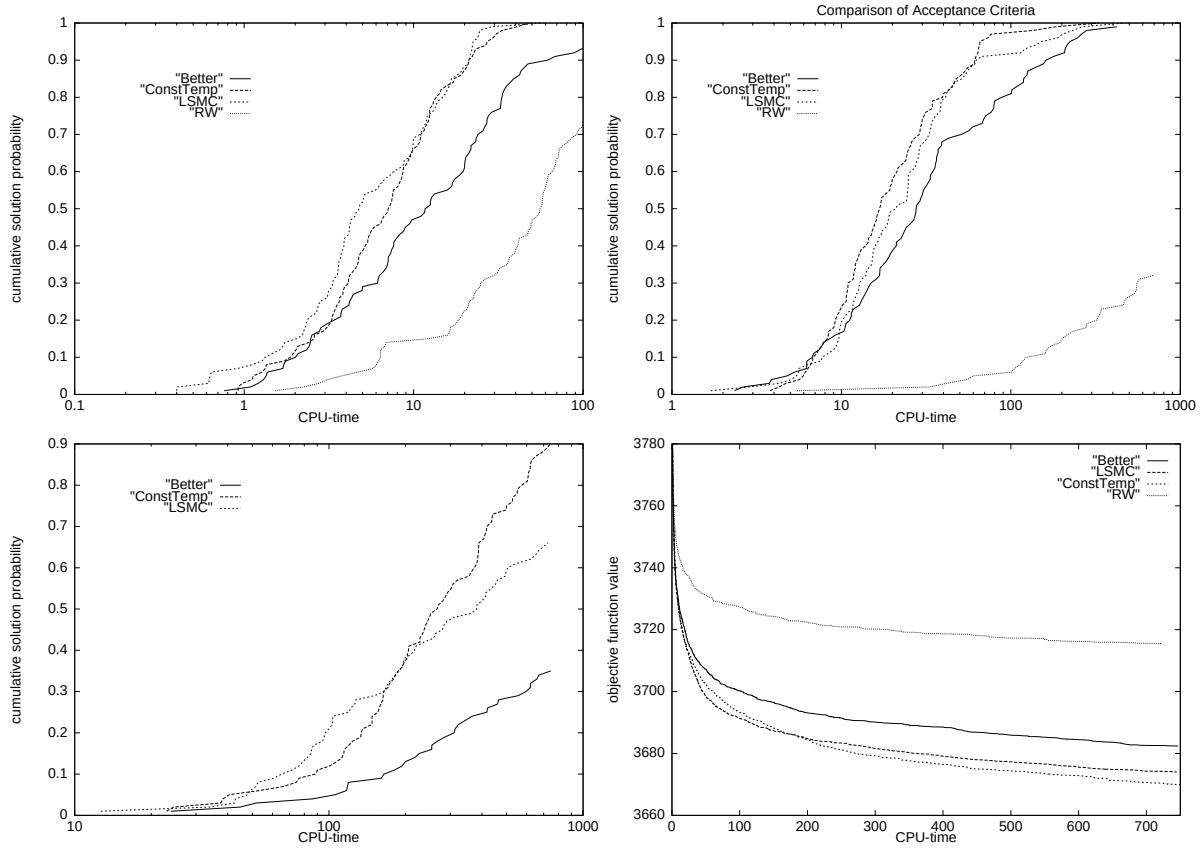

Figure 5.11: Comparison of acceptance criteria for ILS on long runs. Upper left side: Run-time distributions to reach the optimal solution for instance ta021. Upper right side: Run-time distributions to reach a solution within 2% of the best known upper bound on instance ta053. Lower left side: Run-time distributions to reach a solution within 1% of the best known upper bound on instance ta053. Lower right side: Development of the average solution quality for ILS on instance ta053. See the text for more details.

Experimental Comparison – Short Runs

For short run times we compare the performance of the ILS approach on the benchmark instances proposed by Taillard [242] to our implementation of the simulated annealing approach proposed in [193] (SAOP), a multiple descent approach starting from randomly generated initial solutions (MD), and the $\mathcal{MAX}$–$\mathcal{MIN}$ ant system approach to the FSP proposed in Section 4.5. The simulated annealing algorithm of [193] gives a good indication of the performance of SA on the FSP, because it is still among the best performing simulated annealing algorithms for the FSP. The comparison to $\mathcal{MM}$AS is of interest, because $\mathcal{MM}$AS has been shown to give

Table 5.9: Results for short runs of ILS on Taillard's benchmark instances compared to SAOP, $\mathcal{MM}$AS and MD. t_{max} indicates the maximum run-time allowed on a Sun Sparc 5 Workstation. The results are given as the percentage excess over the best known solutions averaged over 10 instances of each size. The size is indicated as $(n \times m)$, where n is the number of jobs and m the number of machines. The best results are indicated in **boldface**. See the text for more details.

instances	SAOP	$\mathcal{MM}$AS	ILS	MD	t_{max}
ta001 - ta010 (20×5)	1.061	0.408	**0.348**	0.765	0.3
ta011 - ta020 (20×10)	1.462	0.591	**0.390**	0.955	1.3
ta021 - ta030 (20×20)	1.116	0.410	**0.272**	0.628	3.3
ta031 - ta040 (50×5)	0.597	**0.145**	0.182	0.255	0.9
ta041 - ta050 (50×10)	3.012	2.122	**1.672**	2.873	3.3
ta051 - ta060 (50×20)	3.533	2.855	**2.324**	3.524	9.0
ta061 - ta070 (100×5)	0.509	**0.196**	0.201	0.219	1.9
ta071 - ta080 (100×10)	1.720	0.896	**0.807**	1.287	7.0
ta081 - ta090 (100×20)	4.076	3.138	**2.509**	3.822	20.5

significantly better results than SAOP. For SAOP the maximal number of iterations is chosen as $\max\{3300 \cdot \ln n + 7500 \ln m - 18500, 2000\}$. For $\mathcal{MM}$AS, ILS, and MD the same computation time as needed by the simulated annealing algorithm are allowed. To make the comparison easier we reproduce in Table 5.9 the results of $\mathcal{MM}$AS, SAOP, and MD given in Table 4.15.

From the results presented in Table 5.9 it can be observed that ILS performs best for almost all instance sizes. Only the $\mathcal{MM}$AS algorithm shows on the smaller instances with few machines slightly better results. Yet, on the larger instances the ILS algorithm shows significantly better performance than either of the other algorithms and obtains high quality solutions in moderate CPU-times.

Our (general purpose) ILS approach also performs well compared to other algorithms proposed specifically for the FSP. The path algorithm proposed by Werner [260] has been applied to the instances above, of size $n = 20, m = 10$; $n = 20, m = 20$; $n = 50, m = 10$; $n = 50, m = 20$ averaging 1.46%, 1.30%, 1.97%, and 2.76% above the best known upper bound at that time. Adjusting our results to the then best known solutions, ILS averages 0.39%, 0.27%, 0.77%, and 1.57% above those. Yet, ILS needs significantly longer times (adjusting for the differences in computer speeds in [260] and here) on the instances with 20 jobs. On the instances with 50 jobs the path algorithm takes roughly 70% of the time needed by ILS.

Comparing the ILS results with Tabu Search algorithms [261, 240, 207, 191], ILS appears to outperform the approaches of [261, 240] (see also next section), similar to that of Reeves [207] and worse with respect to computation time than Tabu-NS [191]. (Recall that we use the simplest form of local search in our ILS.) For the instances solved, ILS gives better solution quality on instances with up to 100 jobs than the Tabu Search of [207] and slightly worse results for the larger benchmark instances with 200 and 500 jobs (adjusting our results to the then best known solutions for the benchmark instances in [242]). For the same computation time, Tabu-NS, the best performing local search algorithm for the FSP, gives significantly better results than ILS because their local search is very fine-tuned and therefore much faster than ours. Yet, as shown in the next section, if the comparison is made using the same number of local search iterations, our ILS approach performs similarily to Tabu-NS. Nevertheless, our point here is that

Table 5.10: Results for long runs of ILS on instances with $n = 50$, $m = 20$. 30000 local search iterations are allowed for Tabu-NS and for each of 10 ILS runs. For Tabu-T is given the best result of 3 runs with 50000 local search iterations. For the ILS algorithm the best makespan, the average makespan, and the worst makespan obtained in the trials is reported. See the text for more details.

Instance	best-known	ILS			Tabu-NS	Tabu-T
		best	average	worst		
ta051	3856	3873	3887.4	3893	3875	3886
ta052	3707	3712	3716.4	3721	3715	3733
ta053	3643	3647	3668.0	3676	3668	3673
ta054	3731	3744	3748.0	3758	3752	3755
ta055	3619	3616	3631.4	3641	3635	3648
ta056	3687	3694	3705.1	3713	3698	3719
ta057	3706	3722	3723.8	3743	3716	3730
ta058	3700	3717	3721.7	3734	3709	3737
ta059	3755	3760	3766.6	3782	3765	3772
ta060	3767	3767	3776.2	3784	3777	3791

given any local search algorithm, ILS is easy to implement, and will cheaply do better than the corresponding local search.

Experimental Results – Long Runs

In this section we compare the effectiveness, that is, the solution quality obtainable with our ILS algorithm to published results of the best tabu search algorithms for the FSP. In particular, we compare the ILS algorithm to published results of the tabu search algorithm of Nowicki and Smutnicki (Tabu-NS) [191] and the tabu search algorithm of Taillard (Tabu-T) [240]. The ILS algorithm is allowed the same number of local search iterations as Tabu-NS, for Tabu-T the number of local search iterations is at least twice as much as for Tabu-NS and ILS. We run ILS 10 times on each instance with $n = 50$, and 5 times on larger instances. Given are the best, the average and the worst makespan obtained for each instance. For Tabu-T only the best results over several runs are given in [242]. We only present results for instances with $n \geq 50$ and $m = 20$ since these instances are the hardest instances to solve and they are the only instances for which the exact optimum is still not known.

The results given in Tables 5.10 to 5.12 show that ILS gives for most instances a better solution quality – even in the worst of the runs – than Tabu-T, although for Tabu-T at least twice as many local search iterations were allowed. Therefore, we can conclude that ILS is significantly more effective on the FSP instances used for comparison than Tabu-T. When comparing ILS to Tabu-NS, both seem to give rather similar solution quality: on some instances the average solution quality obtained with ILS is better than the solution given by Tabu-NS, while on others it is the other way around. Additionally, the best solutions found by ILS are in most cases better than the solutions returned by Tabu-NS.

Concerning the run-time, an important point is that a single iteration of Tabu-NS can be performed in significantly less time than a single iteration of ILS. One complete scan of the neighborhood for ILS takes $\mathcal{O}(n^2m)$, the same as for Tabu-T. But for Tabu-NS it is reported that the neighborhood, empirically, can be scanned in $\mathcal{O}(n^{1.114}m^{1.1})$. Therefore the computation times

of Tabu-NS are significantly smaller than for ILS by a factor of roughly 10 to 30. Nevertheless, the techniques to obtain the speed improvements for the local search used in Tabu-NS are quite general and could also be used for the local search employed in our ILS approach. One might argue that the local search algorithm used in our ILS approach searches a larger neighborhood than the Tabu-NS and therefore for some instances better solutions are found. Yet, Tabu-NS finds much better solutions than Tabu-T which uses the same neighborhood size as our local search implementation in a significantly lower number of local search iterations (see Table 3 in [191]). Therefore, the reason for the good performance of our ILS is not simply the larger neighborhood scanned, but is based on the structure of the algorithm itself.

Table 5.11: Results for long runs of ILS on instances with $n = 100$, $m = 20$. 15000 local search iterations are allowed for Tabu-NS and for each of 5 ILS runs. For Tabu-T is given the best result of 3 runs with 10000 local search iterations. For the ILS algorithm the best makespan, the average makespan, and the worst makespan obtained in the trials is reported. See the text for more details.

Instance	best-known	ILS			Tabu-NS	Tabu-T
		best	average	worst		
ta081	6228	6265	6272.6	6291	6286	6330
ta082	6210	6233	6245.4	6257	6241	6320
ta083	6271	6316	6324.2	6334	6329	6364
ta084	6269	6303	6320.2	6366	6306	6331
ta085	6319	6368	6373.8	6379	6377	6405
ta086	6403	6437	6447.0	6487	6437	6487
ta087	6292	6312	6322.0	6344	6346	6379
ta088	6423	6450	6459.2	6471	6481	6514
ta089	6275	6332	6339.6	6353	6358	6386
ta090	6434	6480	6486.0	6499	6465	6534

Table 5.12: Results for long runs of ILS on instances with $n = 200$, $m = 20$. 10000 local search iterations are allowed for Tabu-NS and for each of 5 ILS runs. For Tabu-T is given the best result of 10 runs with 4000 local search iterations. For the ILS algorithm the best makespan, the average makespan, and the worst makespan obtained in the trials is reported. See the text for more details.

Instance	best-known	ILS			Tabu-NS	Tabu-T
		best	average	worst		
ta101	11195	11262	11270.4	11285	11294	11393
ta102	11223	11300	11317.6	11347	11420	11445
ta103	11337	11402	11423.0	11454	11446	11522
ta104	11299	11350	11369.8	11416	11347	11461
ta105	11260	11313	11315.6	11320	11311	11427
ta106	11189	11292	11312.4	11325	11282	11368
ta107	11386	11442	11453.6	11460	11456	11536
ta108	11334	11415	11427.4	11439	11415	11544
ta109	11192	11280	11317.4	11364	11343	11424
ta110	11313	11385	11407.2	11435	11422	11548

In summary, the performance results reported for the ILS approach to the FSP shown that the performance of a basic local search algorithm can be significantly increased with only very small additional implementation effort. For the solution modification moves based on exchanges of two jobs and swaps of neighboring jobs have shown to suffice. We have shown that, as it is the case for the TSP and the QAP, the choice of the acceptance criterion is crucial to obtain best performance. Despite the simplicity of the approach, the experimental results show that very high solution quality can be obtained, similar or better than the best known local search algorithms applied to the permutation flow shop problem. For five of the 33 still open benchmark instances of Taillard [242] we could improve the best known solution.

5.5 Related Work

Like for many metaheuristic algorithms, the first application of an ILS algorithm has been proposed for the TSP. A first application is reported in [20], yet the obtained results were not very convincing due to a poor choice for Modify and LocalSearch. A major improvement in the development of ILS algorithms came from the *large-step Markov chain* (LSMC) algorithm for the TSP proposed by Martin, Otto, and Felten [169]. The term large-step Markov chain stems from the simulated annealing like acceptance criterion initially used in that algorithm. They consider both, the application of a 3-opt local search algorithm and the Lin-Kernighan heuristic (LK) which is known to be the best performing local search algorithm for the TSP. For the modification they applied a particular 4-opt move, called *double-bridge move* which cannot be reversed directly by the applied local search algorithms. The most efficient implementations of this idea for the TSP seems to be Johnson's iterated Lin-Kernighan algorithm (ILK) [129] and the code of Applegate, Bixby, Chvatal, and Cook which is available at `http://www.caam.rice.edu/~keck/concorde.html`. Apart from the more fine-tuned implementation of the LK heuristic this algorithm differs in two details from the Martin, Otto, and Felten implementation. One difference is that the double-bridge move cuts four randomly chosen edges, whereas in the LSMC algorithm restrictions on the length of the newly introduced edges are used. The other is that only better local optima are accepted. In [47] new possibilities for Modify are proposed for the ILS application to the TSP. In particular, they randomly perturb the city coordinates of a TSP instance and starting from the current solution they apply local search using the perturbed coordinates. The so obtained locally optimal tour with respect to the perturbed coordinates is in turn used as a starting tour for the next local search with the original instance data.

An application of ILS to the graph partitioning problem is described by Martin and Otto in [167, 168]. They propose a kick-move specific to the graph partitioning problem and for the local search they use the Kernighan-Lin heuristic. In the acceptance criterion they only accept better quality solutions for the next modification.

ILS has also been applied to several scheduling problems. Ramalhinho Lorençou [157] and Ramalhinho Lorençou and Zwijnenburg [159] describe an application of ILS to the job-shop scheduling problem. They performed extensive computational tests using various possibilities for the kick-move and the acceptance criterion. For the local search algorithm three alternatives were considered, iterated descent, short simulated annealing runs, and short tabu search runs. The highest quality solutions were obtained with the short tabu search runs and embedding the tabu search algorithm inside the ILS approach performed better than running the tabu search

algorithm for the same amount of time. These results also show that not only iterated descent algorithms, but also other, more general local search algorithms can be improved by ILS ideas. In [32, 33] Brucker, Hurink, and Werner apply the principles of ILS to several other scheduling problems. They introduce two types of neighborhoods, one for the local search and a secondary neighborhood defined on the set of locally optimal solutions with respect to the first neighborhood. Very recently, also an application of ILS to the total weighted tardiness problem has been presented with very promising results [50].

A significant part of the ideas introduced by Mladenović and Hansen using the term *simple variable neighborhood search* (VNS) [180, 109] can, in fact, easily be cast into the framework of ILS. In the context of ILS, VNS introduces the idea of systematically exploring higher-order neighborhoods by Modify, that is, the kick-move strength is systematically modified. The moves for Modify are chosen from neighborhoods $\mathcal{N}_k, k = 1, \ldots, k_{max}$ which are ranked according to some criterion, typically their size. In the basic version of VNS, called *simple VNS* in [180], only better solutions are accepted by AcceptanceCriterion. If an improved solution is found, the kick-move is again chosen randomly from the first neighborhood. If no better solution is found, randomly chosen moves from the next higher-order neighborhood are applied to the current solution. Empirical evidence for the very good performance of VNS has been given on various problems like the p-median problem [108], weighted MAX-SAT, clustering problems, and others [109]. In VNS, in some sense, the *search history* is used to adjust the kick-move strength. This is an important aspect, because often it may be difficult to determine an appropriate kick-move strength a priori. Therefore, by varying the kick-move strength, VNS can give more robustness to an ILS algorithm.

Another possibility to dynamically determine an appropriate kick-move strength is introduced in [15] by the application of *Reactive Search* to the MAX-SAT problem. This algorithm is not designed as an ILS algorithm, nevertheless, it may be interpreted as such. In this algorithm, a local search phase is altered with a diversification phase based on tabu search which is used to guarantee that the modified solution has a minimal hamming distance from the initial solution. Depending on the Hamming distance of the local minimum s'' to the starting point s, the strength of the modification, as determined by the tabu search length, is adjusted. As acceptance criterion, the rule to accept every new local minimum is used.

5.6 Summary

In this chapter we have analyzed the run-time behavior of iterated local search algorithms when applied to three combinatorial optimization problems, namely the traveling salesman problem, the quadratic assignment problem, and the flow shop problem. While iterated local search has been applied previously to the traveling salesman problem and is known to be among the best algorithms available for that problem, it is the first time applied the quadratic assignment problem and the flow shop problem. The analysis of the run-time behavior on all the three problems revealed that iterated local search shows a kind of stagnation behavior which may severely compromise its performance if longer run-times are allowed. Based on this observation, we have proposed improved iterated local search algorithms and could show that these improved algorithms (i) achieve a strongly improved performance when compared to the standard way of applying iterated local search and (ii) that these extensions are among the best available approaches for the three problems we attacked.

Chapter 6

Tabu Search

In this chapter we study the application of tabu search to improve a known local search algorithm, the *min conflicts* heuristic, for constraint satisfaction problems (CSPs). We present two possible applications of tabu search features, differing in the choice of the tabu attribute. An extensive experimental investigation shows the improved performance of the *min conflicts* heuristic when using features of tabu search. We also analyze the run-time behavior of the improved algorithm by a functional approximation of the empirically measured run-time distributions for approximately optimal noise parameter settings. This analysis reveals that on the hardest binary CSP instances as well as hard CSP-encoded graph coloring instances the run-time behavior of the local search algorithms running with approximately optimal parameter settings can be characterized by exponential distributions. This result has a number of interesting consequences concerning parameter tuning and the possible speed-up by a parallelization based on multiple independent runs.

The chapter is structured as follows. We first introduce the *min conflicts* heuristic, which is the basic local search algorithm for binary CSPs, and detail our tabu search extension. In Section 6.3 we present an extensive experimental analysis of our improved algorithms which is followed by an analysis of the run-time behavior of our improved algorithms for binary CSPs as well as graph coloring problems in Section 6.4. We end this chapter by a discussion of related work and a summary of the results.

6.1 Introduction

Many practical problems like spatial and temporal reasoning, graph coloring problems, and scheduling can be formulated as constraint satisfaction problems (CSPs). CSPs in general involve finding an assignment of values to variables that satisfies a given set of constraints between these variables. Constraint optimization problems additionally involve an objective function that has to be optimized. These problems are $\mathcal{NP}$-hard, that is, according to current knowledge these problems cannot be solved in polynomial time by a deterministic algorithm.

Especially in the last few years, local search procedures have received considerable attention for the solution of CSPs [178, 58] as well as for the closely related satisfiability problem in propositional logic (SAT) [107, 223, 222] (see page 11 for a description of SAT). The first local search algorithm for the solution of CSPs is the *min conflicts* heuristic which also inspired later work on GSAT, a well known local search algorithm for SAT. Yet, this first local search

heuristics gets stuck in local minima in the search space and subsequently different possibilities to improve the *min conflicts* heuristic have been considered. The simplest way is to run the local search procedure until it is stuck in a local minimum and then simply restart from a new, randomly generated solution. Another very popular approach is the introduction of random moves during the search [222, 136, 256]. In this chapter we will have a closer look on tabu search extensions of the *min conflicts* heuristic (see Section 2.7.3 for an introduction to tabu search). Tabu search is one of the most successful metaheuristics in the operations research community for a wide variety of hard combinatorial problems. Hence, it may be an important technique to enhance also local search performance for CSPs.

6.2 Local Search for Constraint Satisfaction Problems

A CSP is defined by a set of variables $\mathcal{X}$, every variable x_i having associated a discrete, finite domain D_i, and a set of constraints $\mathcal{C}$ (see Section 2.1.2 on page 10 for more details). In this chapter we restrict ourselves to binary CSP, that is, a constraint $C(x_i, x_j) \subset D_i \times D_j$ restricts the set of allowed value pairs for two variables. A variable instantiation is the assignment of a value $d_i \in D_i$ to a variable x_i. If $(d_i, d_j) \notin C(x_i, x_j)$ we call a constraint violated; the two variables participating in the constraint $C(x_i, x_j)$ are in *conflict*. As *conflict set* $K(\mathcal{X})$ we denote the set of variables which are involved in a constraint violation. The task in a CSP then is to find a candidate solution such that all constraints are satisfied.

6.2.1 The *min conflicts* Heuristic

To apply local search to CSPs, an objective function, a neighborhood structure, and a neighborhood examination scheme has to be defined. Often, the number of violated constraints is taken as the objective function to be minimized and a CSP has a solution if a candidate solution with objective function value zero exists. The possible moves are defined by a neighborhood function (see Section 2.4). The simplest neighborhood is a 1-opt neighborhood, that is, two instantiations s and s' differ in one variable instantiation. The *min conflicts* heuristic [178] uses the objective function just defined, the 1-opt neighborhood and a particular neighborhood examination scheme which is described next. In a first step, the *min conflicts* heuristic chooses randomly according to a uniform distribution a variable from the conflict set $K(\mathcal{X})$. This variable is assigned a value which minimizes the number of conflicts. The number of conflicts for a variable value pair (x_i, d_i) is the number of violated constraints in which variable x_i is involved if it takes value d_i. If this minimum value is taken by several values, then one of these values is chosen randomly according to a uniform distribution. In the following we will call the choice of a variable and the choice of a variable instantiation as one *iteration*. The *min conflicts* heuristic then repeats, starting from some initial candidate solution these steps. If after maxSteps iterations no solution is found, the algorithm restarts from a new initial solution. If after a given number maxTries of such tries still no solution is found, the *min conflicts* heuristic terminates unsuccessfully. A general outline of the *min conflicts* heuristic is given in Figure 6.1.

One specific problem for a local search algorithm like the *min conflicts* heuristic is the occurrence of local minima. The easiest way to escape from local minima is to restart the local search heuristic from another random initial solution, which has been already included into the description of the algorithm. Another possibility is to allow moves which may worsen the

```
procedure min-conflicts (𝒳, 𝒟, 𝒞, s)
    for i = 1 to maxTries do
        generate random candidate solution s
        for j = 1 to maxSteps do
            if s is solution of 𝒞 then
                return s
            else
                choose a variable x_i ∈ K(𝒳)
                P = possibleSteps(s, x_i)
                Step = random element of P
                perform Step on s
            end
        end
    end
    return "no solution found"
end
```

Figure 6.1: Pseudocode of the *min conflicts* heuristic

objective function value. This latter possibility is addressed by several approaches to improve local search algorithms for CSPs and SAT like *random walk* [222], simulated annealing [22, 228], the breakout method [182, 58], and weak commitment search [263].

Especially the random walk option has been very popular to improve greedy local search algorithms for SAT, the best known of which are GSAT [223] and WSAT [222]. When applied to SAT, random walk flips with a fixed walk-probability p_w the instantiation of a randomly chosen variable which is contained in a currently unsatisfied clause (variant *rw1*). It is rather straightforward to extend the *min conflicts* heuristic with the random walk option, which was first proposed in [255]. When applying the random walk idea to improve the *min conflicts* heuristic, first a variable is selected like in the *min conflicts* heuristic and then, with a probability w_p, a randomly chosen value is assigned to this variable. Yet, it may be argued that in local search algorithms for SAT, the random walk option has the important effect that at least one previously unsatisfied clause will be satisfied by such a step. Therefore, we mimic this behavior in a newly designed variant *rw2*, in which we first choose a violated constraint and then with equal probability one of the variables involved in the constraint. If possible, we assign to that variable randomly a value which satisfies the chosen constraint; if no such value exists, a value is chosen at random.

6.2.2 Tabu Search and the *min conflicts* Heuristic

Tabu search [107, 97] is a heuristic for the guidance of an underlying local search procedure (see Section 2.7.3 for a more detailed description). In contrast to random walk and simulated annealing, tabu search relies less on probabilistic elements during search, but rather guides the search more deterministically by exploiting the search history. In tabu search at every iteration the best possible move is chosen, although it might worsen the objective function value. Yet,

this strategy may lead to cycles as the search may easily return to a previously visited solution. To avoid such a behavior, attributes of previously found solutions are stored in a tabu list for a number tl of iterations , where tl is called the tabu tenure. A neighboring solution is regarded as forbidden if it has an attribute which is currently in the tabu list. Yet, when using tabu attributes it may happen that very good neighbors are regarded as tabu, although they may be very promising. Therefore, *aspiration criteria* are used to override the tabu status of such solutions.

Tabu search is a *metaheuristic* which is used to guide a greedy local search procedure. For the CSP we use the *min conflicts* heuristic as the underlying greedy local search method. For the application of tabu search an important decision is which solution attributes are used as tabu attributes. We examined two such possibilities. In one version, specific variable-value pairs are forbidden; that is, if the *min conflicts* heuristic changes a variable instantiation from $s(x_i) = d_j$ to $s(x_i) = d_k$, then the variable-value pair (x_i, d_j) is tabu for tl iterations. In this case we also use an aspiration criterion which drops the tabu status of a move, if it improves the best objective function value. Another possibility is to declare variables as tabu and to influence in this way the variable selection strategy. This means that a variable which is tabu cannot be chosen for the next tl iterations. Hence, in this version it may happen that a variable not contained in $K(\mathcal{X})$ changes its value.

6.3 Experimental Results

In this section we experimentally compare the proposed tabu search extensions of the *min conflicts* heuristic to the random walk extensions which were previously found to be the most efficient techniques to improve the *min conflicts* heuristic [256]. In the following we will refer to the different extensions algorithms as follows. `tmch` denotes the tabu search extension of the *min conflicts* heuristic which uses variable-value pairs as tabu attributes, and `tvmch` the variant which uses the tabu search version with variables as tabu attributes. The random walk extensions are referred to as `wmch` for variant *rw1* and as `wmchf` for variant *rw2*.

In our experimental analysis we first study the possibility of hybrids which combine elements of the random walk and the tabu search extension. The motivation for this study is that possibly the best performance is achieved by an adequate combination of the two strategies which we consider for the improvement of the *min conflicts* heuristic. As a side effect, also a study of appropriate parameter settings for the "pure" random walk and tabu search extensions is done. The parameters p_w and tl will also be called *noise parameters* in the following.

6.3.1 Benchmark Problems

We tested the different variants of the *min conflicts* heuristic on randomly generated binary CSPs and graph coloring instances from the DIMACS benchmark set which are available via ftp at `dimacs.rutgers.edu/pub/challenge/`. The binary CSP instances are specified by four parameters $\langle n, k, p, q \rangle$, where n is the number of variables, k is the uniform domain size, p is the probability that a constraint between two variables exists, and q is the conditional probability that a value pair satisfies a constraint. (In the literature also often a parameter $q' = 1 - q$ is used, where q' is the probability that a value pair is inconsistent.) It has been shown that

with specific parameter settings of the instance generator on average particularly hard instances for complete as well as incomplete algorithms are generated; for other parameter settings the generated instances are rather easily solved or easily shown to be unsolvable by complete algorithms [203, 227]. The randomly generated instances show a *phase transition phenomenon* as observed also in physical systems [248, 140]. For randomly generated binary CSPs the phase transition is located at a point where the expected number of solutions is roughly one [226]. The expected number of solutions $\mathbf{E}[N]$ can be calculated as

$$\mathbf{E}[N] = k^n \cdot q^{pn \cdot (n-1)/2} \tag{6.1}$$

that is, the number of candidate solutions multiplied by the probability that a randomly chosen candidate solution is satisfiable. Therefore, for fixed n, k and for fixed parameter p or q, the value of the fourth parameter can be determined such that on average hard problems are generated. More recent work on the exact location of the phase transition point for random CSPs has refined this knowledge. Especially in case of rather sparse constraint graphs, the exact location of the phase transition deviates slightly from the theoretically predicted point [227].

In our experimental investigation we used randomly generated CSPs with $n = 20$ variables, a uniform domain size of $k = 10$, different values for p (which determine the constraint graph density) and varied q in steps of 0.01 around $\hat{q}_{\text{crit}}$, where $\hat{q}_{\text{crit}}$ is the estimation of the phase transition point [227]. We only used satisfiable instances, which have been filtered out of the unsatisfiable ones by a complete search algorithm.

6.3.2 Experimental Results for Binary CSP

In this section we present our experimental results of a detailed investigation on the variants of *min conflicts* using tabu search and random walk. We first consider the possibility of hybrid strategies combining elements of tabu search and random walk in a single algorithm for benchmark instances from the phase transition regions and then compare the tabu search and the random walk extension over a wider range of instances which were generated by varying the constraint graph density (determined by parameter p) and the constraint tightness (determined by parameter q).

A Parametric Investigation on Tabu Search and Random Walk

We restricted the study of parameter settings for the tabu tenure tl as well as the walk probability p_w to binary CSPs with $p = 0.5$ and $q = 0.62$ because of the high computational load involved. (For these instances we have $q_{\text{crit}} = 0.616$.) The hybrids between the tabu search and the random walk extension are obtained by using the two strategies at the same time in the algorithm and therefore we examined a wide range of possible parameter settings for p_w and tl. When combining these strategies, the tabu search strategy is always followed, except in the case that a random walk move is done.

The experiments were done using 250 instances and solving each instance exactly 10 times. Each instance is solved several times, because the solution cost to solve one single instance varies very strongly between the trials; we refer to Section 6.4 for an investigation of the run-length distribution when using approximately optimal parameter settings. The average solution

cost for solving one instance is then averaged over the 10 trials and the solution cost for a specific (p_w,tl) combination is the solution cost incurred for solving the ensemble of 250 instances. In Figures 6.2 and 6.3 are given these averages for several (p_w,tl) combinations. It should be noted here that we used a fixed setting for maxSteps of 10,000, hence, here we neglect the influence of this parameter. For a discussion on the influence of the parameter maxSteps we refer to Section 6.3.4.

In Figure 6.2 a hybrid between `tmch` and `wmch` is investigated. For $tl = 0$ we obtain "pure" `wmch`, for $p_w = 0.0$ we obtain "pure" `tmch`. For `wmch` the lowest average solution cost of 10795.5 was obtained with $p_w = 0.11$. For smaller values of p_w the solution cost increases fast, for example, for $p_w = 0.04$ the average solution cost is 22924.4, while with larger settings for p_w the increase is much slower, for example, with $p_w = 0.16$ we obtain an average solution cost of 14515.7. Compared with `tmch`, the lowest solution cost of 8810.8 is obtained using a tabu tenure of $tl = 2$; a much lower value than for `wmch`. If the tabu tenure is chosen lower, that is, $tl = 1$ the solution cost increases to 12358.6 and for $tl = 3$ it is 9985.8. For larger tl the solution cost increases strongly, reaching 19436.5 for $tl = 6$. Thus, the optimal tabu tenure is surprisingly small. Yet, this result may be explained as follows. Suppose, for example, that only one constraint is not satisfied. If for one of these variables a specific value is chosen, this value will be tabu and if the variable is chosen again within the two next iterations it has to change its value so that possibly other constraints are violated and the objective function value is increased again allowing to leave local minima. Note, that the objective function value will increase as little as possible. This mechanism of leaving local minima seems to be more effective than the often used random walk strategy.

Concerning the effectiveness of hybrid strategies on can observe that for not optimal values of either tl or p_w the addition of the other strategy may increase performance; for example, the solution cost for $tl = 1,\ p_w = 0.06$ reaches 9083.5, while without tabu search for $p_w = 0.06$ the solution cost is 14711.7. Generally, we can observe a valley in the average solution cost reaching from the optimal walk probability to the optimal tabu tenure. Yet, if either of the two strategies is used with approximately optimal parameter value, the addition of either one of the strategies does not improve performance. If the parameter values are larger than the optimal parameter settings, the addition of the other strategy worsens the performance even more.

In Figure 6.3 the experimental results for a combination of `wmch` and `tvmch` are plotted; we refer to this combination as `tvwmch`. Using only `tvmch` without adding features of random walk did not yield improvements over the basic *min conflicts* heuristic; hence, in Figure 6.3 we give no results with $p_w = 0$ but only those with parameter settings of $p_w > 0$. In fact, using hybrids between `tvmch` and `wmch` shows benefits and improves the computational results for either of the variants. The best results have been obtained with the parameters $tl = 4, p_w = 0.09$ resulting in an average solution cost of 7788.5. In general the tendency is that the lower the walk probability, the larger (up to some upper limit around 8 for these instances) the tabu tenure should be chosen. Also, when using `wmch` the addition of the tabu search variant is beneficial for all the examined settings of p_w, which was not the case for the `twmch` hybrid discussed before. Hence, influencing the variable selection may uniformly improve performance of `wmch`; however, with a too large tabu list size, performance again drops. Thus, some bias of the variable selection towards variables occurring in violated constraints seems to be necessary.

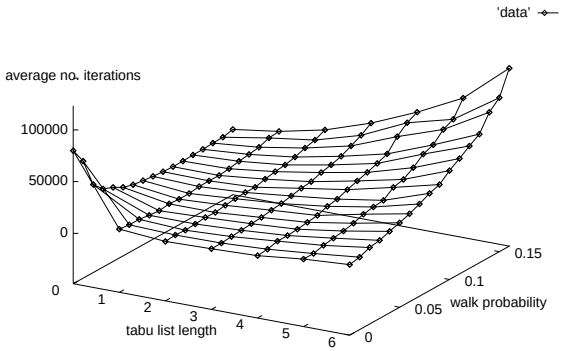

Figure 6.2: Results for a hybrid algorithm using `tmch` and `wmch` for random binary CSP instances, $p = 0.5$, $q = 0.62$. The tabu tenure tl is varied from 0 to 6 (indicated on the x-axis), the random walk probability p_w is varied from 0.0 to 0.16 (indicated on the y-axis). The average solution cost is indicated on the z-axis. Note, that for point $p_w = 0,\ t = 0$ the actual solution cost is much higher than shown here.

Experimental Results for Binary CSP from the Phase Transition Region

The experimental results of the previous section were obtained only on binary CSPs with one fixed parameters setting for p and q. We extend the experimental analysis to instances which were generated over a wide range of parameters p and q, which determine the density of the constraint graph and the constraint tightness, respectively. In particular, the instances were generated with 20 variables and uniform domain size 10 and for the following combinations of p and q. p was chosen $p \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ and q was chosen for each value of p over a range of values around $\hat{q}_{\text{crit}}$ in steps of 0.01. $\hat{q}_{\text{crit}}$ is the estimated phase transition point for fixed p. In our experiments we then compared the performance of `wmch`, `wmchf`, `tmch`, and `tvwmch` over 100 instances and each instance had to be solved exactly 25 times using a cutoff parameter setting of maxSteps $= 10000$. Each variant was run with approximately optimal parameter settings, which were determined in preliminary runs.

In Figure 6.4 we give the results obtained for parameter values of $p = 0.5$ and $p = 0.9$, respectively; for the other settings of p the relative ranking of the algorithms was the same. The results show that actually the variants using some element of tabu search perform better than the "pure" random walk variants over the whole range of constraint tightness values. Overall, the version `tvwmch` seems to be the best performing one, closely followed by `tmch`. Both perform considerably better than the two random walk versions of which `wmchf` appears to be slightly better. Hence, we can conclude that the *min conflicts* heuristic variants using tabu search features are, in fact, superior to the random walk extensions of the *min conflicts* heuristic.

6.3.3 Experimental Results for Graph Coloring Problems

To further test the proposed extensions, we applied them to hard graph coloring instances from the DIMACS benchmark set with up to 450 nodes. Hence, 450 variables are used in the corre-

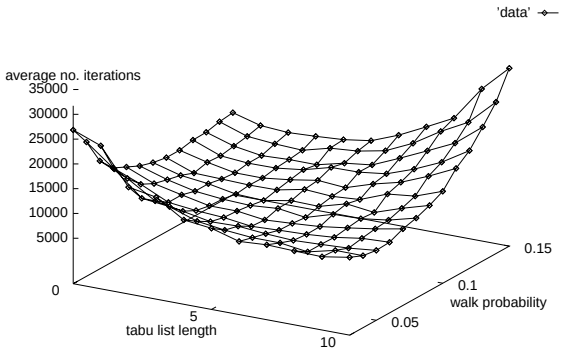

Figure 6.3: Results for a hybrid algorithm using `tvmch` and `wmch` for random binary CSP instances, $p = 0.5$, $q = 0.62$. The tabu tenure is varied from 0 to 10 (indicated on the x-axis), the walk probability from 0.03 to 0.15 (indicated on the y-axis). The average solution cost is given on the z-axis.

sponding CSP formulation and the solution of such large instances indicates that the proposed algorithms scale well to larger instances. The tests were run for each instance with some given number of colors close to the chromatic number. Instances `DSJCx.y` have initially been proposed in [128] and have been used to experimentally evaluate the performance of simulated annealing for graph coloring. The instances are randomly generated graphs where x is the number of nodes and $y/10$ is the probability of including a specific edges into the graph (for example, DSJC125.1 has 125 nodes and an edge is included with probability 0.1). For the `DSJCx.y` instances we allowed a maximum number of 5 million iterations per trial, for the other two instances 1 million iterations. The latter two instances given in Table 6.3.3 contain 300 and 450 nodes, respectively, the second number being the known chromatic number of the graph. These two instances are known to be hard to solve to optimality. For `tmch` again a tabu tenure of two showed to be a good parameter setting over all instances and the results presented here are all obtained with this parameter value. Yet, for `wmch` the optimal parameter setting of w_p depends strongly on the particular instance and has to be fine-tuned. In fact, the best value for p_w seems to depend on the density of the graph; yet, for binary CSP we could not detect such a dependence of the best walk probability on the density of the constraint graph. Concerning the performance, again the tabu search variants of the *min conflicts* heuristic, in particular `tmch`, show significantly better results than the random walk version (see Table 6.3.3) and have the additional major advantage that good parameter values are much easier to find. On these instances, `tmch` performs also much better than the basic *min conflicts* heuristic which, for example, could solve instance DSJC125.5.col with 18 colors within a 2 million iterations using maxSteps= 1250 (this corresponds to 1600 restarts) only in 2 out of 25 independent experiments.

6.3.4 Experimental Results on the Influence of Max-steps

In the experimental investigation of the tabu search and random walk extensions of the *min conflicts* heuristic we neglected the influence of the maxSteps parameter. Yet, it is known that

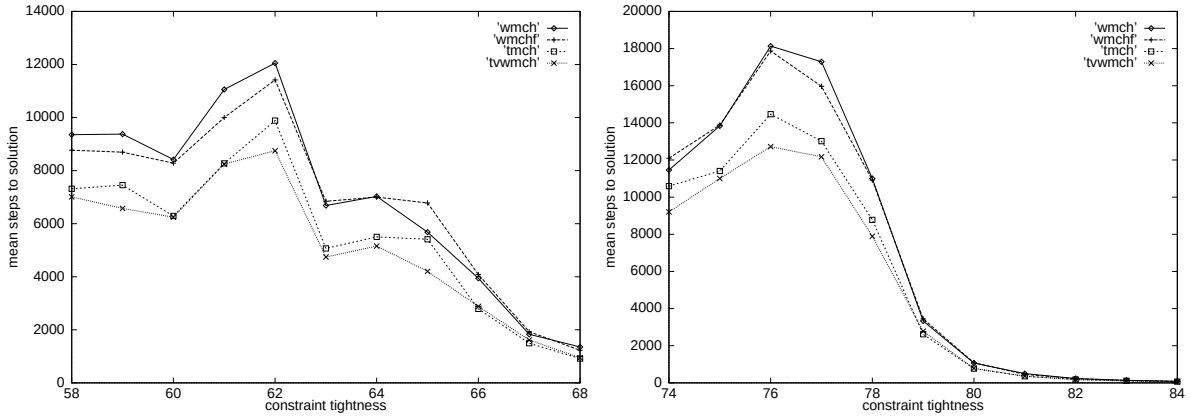

Figure 6.4: Given is the average number of iterations (y-axis) to solve random binary CSPs with $p = 0.5$ (left side) and $p = 0.9$ (right side) for several values for q around $\hat{q}_{\text{crit}}$. The x-axis represents $100 \cdot q$. Each data point gives the average solution cost over 100 instances.

this parameter may have a significant influence on the performance of the basic *min conflicts* heuristic and GSAT, a greedy local search algorithm for SAT [223]. Therefore, here we examine the influence of this parameter for the different *min conflicts* heuristic extensions. We ran the experiments with different settings of maxSteps for `tmch`, `wmch` and the *min conflicts* heuristic. In particular, we were especially interested on the influence of maxSteps if the extensions use approximately optimal, lower than optimal and larger than optimal parameter settings and whether maxSteps has a qualitatively different influence. To limit the experimental burden, we restrict ourselves to some specific parameter value choices for each of the algorithms. The experiments were run again on the instances with $n = 20, k = 10, p = 0.6, q = 0.62$ and we give the average solution cost for solving each of 100 instances exactly 25 times; the computational results are given in Table 6.3.4.

The experimental results give several interesting insights. First, effectively the *min conflicts* heuristic performs significantly worse than the tabu search and random walk extensions if good parameter values are chosen for latter two. Second, for `tmch` and `wmch` the parameter maxSteps does not seem to have a significant influence on performance if approximately optimal noise parameter settings are chosen. The performance only degrades if maxSteps is chosen too low (we refer to the next section for a discussion of the reasons for this phenomenon). Third, if the parameter value for random walk (see $p_w = 0.03$ in the table) or the tabu tenure (see $tl = 1$ in the table) is smaller than the optimal value, maxSteps has a considerable influence and the best performance is obtained for rather small values of maxSteps, but, forth, if the noise parameter values are larger than the optimal ones, the best performance is obtained for large values of maxSteps (see $tl = 5$ and $p_w = 0.15$ in the table). In this latter case, the influence of maxSteps again becomes negligible, if it is chosen large enough (around 2,500 to 5,000). Hence, the influence of maxSteps is qualitatively different depending on whether the noise parameter settings are chosen too large or too small.

An explanation for this latter phenomenon may be given by considering the (im)balance between exploitation and exploration of the search space, which directly depends on the noise

Table 6.1: Results on some graph coloring instances from the DIMACS challenge. In the first column the instance is specified, in the second column the number of colors used. The numbers give the median solution cost to solve the instances. If an instance is not solved in every trial, we give the proportion of successful runs. See the text for more details.

instance	k	tmch	wmch	tvwmch
DSJC125.1.col	5	33452	67879	43846
DSJC125.5.col	18	12980	25290	19773
DSJC125.5.col	17	(36/50)	(17/50)	(12/50)
DSJC125.9.col	44	17026	18704	13770
DSJC250.5.col	30	70570	462421	460533
DSJC250.5.col	29	(13/25)	(1/25)	(1/25)
flat300-28-0.col	34	59208	(5/10)	144975.4
le450-15c.col	21	110346	165361	159405.2

Table 6.2: Investigation of the maxSteps dependence. The first column gives the algorithm and in parenthesis the parameter setting used, the other columns give the average solution cost of solving averaged over 100 instances for several values of maxSteps which are indicated in the first row. See the text for more details.

algorithm	250	500	1000	2500	5000	10,000	50,000	100,000
mch	21437	33358	59674	150128	302182	575841	/	/
tmch ($tl = 1$)	11565	9217	9781	9938	11324	12358	15980	16092
tmch ($tl = 2$)	13368	10187	8792	9560	8972	8810	9961	8990
tmch ($tl = 5$)	19492	19649	16602	16498	15987	16551	16479	15965
wmch ($w_p = 0.03$)	15452	15834	17377	19409	22860	26831	33429	41713
wmch ($w_p = 0.1$)	16183	19361	11056	10894	10888	10802	10656	10982
wmch ($w_p = 0.15$)	23925	16495	14779	13743	13310	12384	12868	12302

parameter settings. If the parameter values for tl or p_w are too small, exploration of the search space is too low to rapidly escape from local minima and to explore new regions of the search space. Exploration of the search space must then be added by some other feature, which in this case is given be the restart from a new, random solution. If the parameter values for p_w or tl are too large, the exploitation of the neighborhood of the current solution is very low. Thus, if additional exploration of the search space by random restart is provided, this is simply the wrong remedy, because the algorithm needs to exploit more strongly the neighborhood of the current solution. Yet, for approximately optimal noise parameter settings the particular value of maxSteps does not seem to have a significant influence on the final performance, provided maxSteps is not chosen too low. In the next section we analyze the run-time behavior of local search algorithms for CSPs and we show that for approximately optimal noise parameter settings the run-time distribution on the hardest instances can be approximated by exponential distributions. As we will discuss later, that result also explains the observations on the influence of maxSteps settings when approximately optimal parameter settings are used.

6.4 Analysis of the Run-time Behavior

In this section we present an empirical analysis of the run-time behavior of extensions of the *min conflicts* heuristic applied to binary, randomly generated CSPs from the phase transition region [248, 227]. Our analysis is based on measuring run-length distributions, where the run-length is the number of iterations the algorithm needs to find a solution. Using the run-length is a very good indicator of the run-time since it correlates very closely to the CPU-time of the algorithm. An advantage of using run-length distributions is that results can be compared more easily over different computer architectures.

To actually measure the run-length distributions we have to take into account that local search algorithms applied to CSPs (and also SAT) have a cutoff parameter like maxSteps which determines after how many iterations the algorithm is restarted from a new, randomly generated solution (see the algorithmic outline in Figure 6.1). Practically, we measure the run-length distributions by running the algorithm k times on a given instance up to some very high cutoff value and by recording for each trial the number of iterations required to find a solution. The empirical run-length distribution is then defined by $\widehat{G}(l) = |\{j \mid rl(j) \leq l\}|/k$, where $rl(j)$ is the length of the jth try. Note that measuring run-length distributions for satisfaction problems is, in fact, a special case of our methodology proposed in Chapter 3, because we are mainly interested in finding a global minimum of the objective function, that is, a solution which satisfies all constraints.

6.4.1 Characterizing Run-length Distributions on Single Instances

In Figure 6.5 we give run-length distributions for `tmch` based on 1000 independent runs of the algorithm, measured on three randomly generated CSP instances of different hardness. All three instances have 20 variables, a uniform domain size of 10, and were generated setting p to 0.5 and q to 0.62. The three instances are taken from a set of 250 randomly generated instances and correspond to the easiest, the median and the hardest instance with respect to the average number of iterations required to find a solution. We refer to these instances as *easy*, *medium*, and *hard* in the following. The run-length distribution, despite being based on a discrete measure, is approximated by a continuous probability distribution, because these are easier to handle mathematically. In fact, we have used exponential distributions for this approximation and, obviously, they closely fit the empirical run-length distributions. (The exponential distribution and its properties have been discussed in Section 3.2.3 on page 51. Recall that its distribution function is given as $G(t) = 1 - e^{-\lambda t}$, where $1/\lambda$ is the expectation value.) Only for the *easy* and *medium* instance, the empirical distributions show a small deviation for short runs.

The goodness of fit of the approximation can be tested by standard statistical tests like the χ^2 test [213]. Basically, for a given empirical run-length distribution this is done by estimating the parameter λ of the exponential distribution and comparing the deviations of the empirical distribution from the predicted exponential distribution. The result of this comparison is the χ^2 value, where low χ^2 values indicate a close correspondence between the empirical and the predicted distribution. This test (the corresponding χ^2 values are given in Table 6.3) has rejected the approximation of the *easy* and the *medium* instance, while the approximation for the hardest instance was accepted by the test at the $\alpha = 0.05$ significance level. (In the same experiment

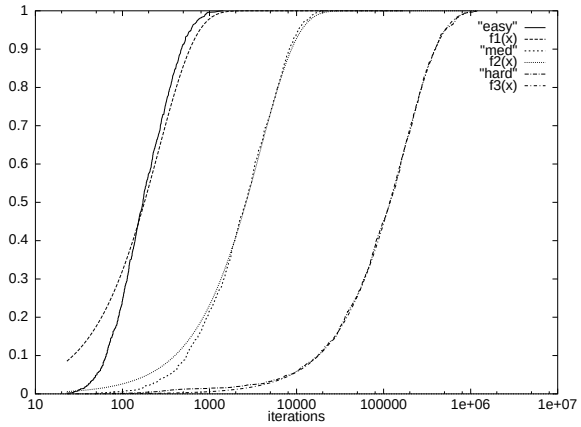

Figure 6.5: Run-length distributions for `tmch` with $tl = 2$ (best setting) on three random CSP instances "easy", "medium", and "hard". On the x-axis the number of iterations is given and the y-axis gives the probability of finding a solution. Every empirical run-length distribution is approximated by an exponential distribution, indicated by $f1(x), f2(x), f3(x)$.

Table 6.3: Test results of the χ^2 test for the approximations of the run-length distributions. Given are the average number of iterations (i_{avg}) needed to find a solution and the χ^2-value of the test (1,000 trials have been performed). The $\alpha = 0.05$ significance level is 42.557, hence the approximations for the easy and medium instance are rejected.

instance	i_{avg}	χ^2-value
easy	209.1	248.2
medium	4686.9	44.84
hard	171164.9	14.2

with `wmch` the test for the hardest and the medium instance was accepted, while for the easiest instance the test was rejected.) The reason for the rejection of the approximations for the two easier instances are the deviations in the lower part of the curve. We conjecture that the deviations are caused by the initial hill-climb phase [88], which is due to the fact that when starting from random solutions the local search algorithm needs a number of iterations to reach the bottom of a valley in the search space and to have a realistically high chance of finding a solution.

The run-length distributions also illustrate the widely differing hardness among the random CSP instances from the phase transition region. While the easiest solvable instance could be solved in an average number of 209.1 iterations, the hardest one took 171164.9 iterations (see Table 6.3). This enormous variability in problem hardness between the instances of randomly generated test sets for CSPs and SAT appears to be rather typical [118, 119, 200].

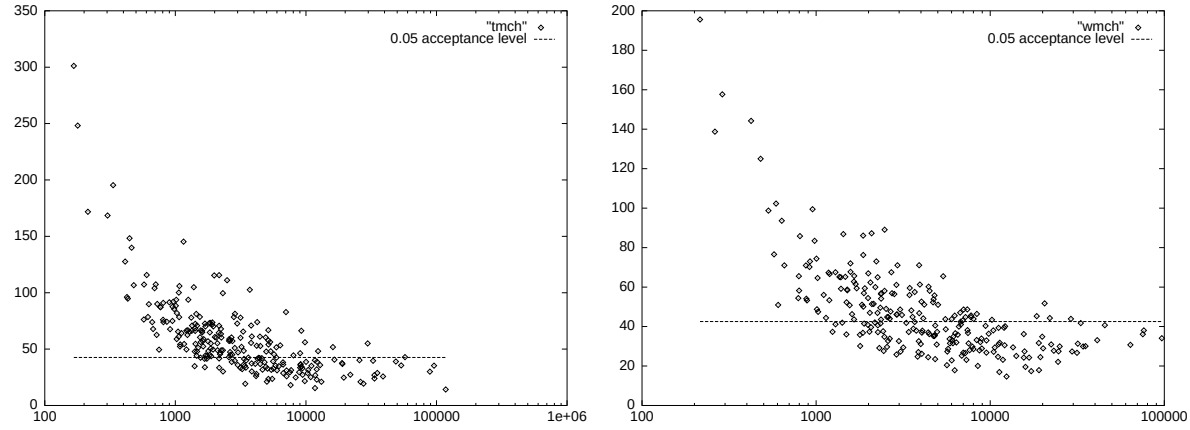

Figure 6.6: Correlation between χ^2 value (y-axis) and problem hardness (x-axis), given as the average number of iterations to find a solution for `tmch` (left side) and `wmch` (right side). The horizontal line indicates the $\alpha = 0.05$ acceptance level of the χ^2-test.

6.4.2 Formulating and Testing Hypothesis on Problem Classes

The observation of the run-length distribution on the three single instances lead us to formulate the following hypothesis:

For hard random CSP instances from the phase transition region, the run-time behavior of the best performing variants of the min conflicts heuristic with approximately optimal noise parameter settings can be characterized by exponential distributions.

To test this hypothesis, we apply the methodology outlined above to the entire test set of 250 instances. The resulting correlation between average search cost and χ^2 values is plotted in Figure 6.6 for the application of `tmch` (left side) and `wmch` (right side). There is a strong negative correlation for both algorithms, indicating that for harder instances, their behavior can be more and more accurately characterized by exponential distributions. The figures also indicate the standard acceptance level for the χ^2 test ($\alpha = 0.05$). As can be seen from the plots, for high average search cost almost all instances pass the χ^2 test.

In summary, the data from the χ^2 tests confirm and refine our hypothesis that for hard random CSP instances from the phase transition region, the run-time behavior of `tmch` and `wmch` with approximately optimal noise setting can be approximated using exponential distributions. By using run-length distributions to characterize the algorithms' behavior on single instances, we could derive and verify a general hypothesis on the run-time behavior of extensions of the *min conflicts* heuristic when applied to hard random CSP instances from the phase transition region.

6.4.3 Run-length Distributions for Graph Coloring Instances

To test the generality of our observations on other test problems, we applied both, `tmch` and `wmch` to graph coloring instances. In Figure 6.7 we present their respective run-length distributions on the graph coloring instance `DSJC125.5.col`. This instance is taken from the DIMACS Benchmark set and has 125 nodes. The best known solution for this instance uses 17

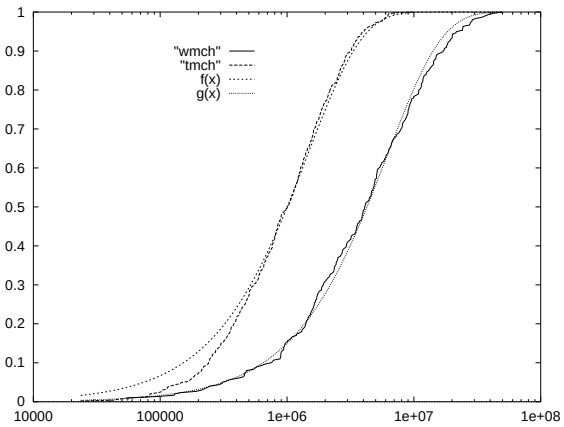

Figure 6.7: Run length distribution for `tmch` and `wmch` on graph coloring instance `DSJC125.5.col`. The x-axis gives the number of iterations, the y-axis the corresponding solution probability. $f(x)$ and $g(x)$ are exponential distributions which approximate the empirical ones.

colors and it is conjectured to be optimal. Consequently, we solved this instance allowing 17 colors. The run-length distributions are measured running both algorithms 250 times. Again, the run-length distributions can be reasonably well approximated by an exponential distribution (although the approximation for `tmch` passed the χ^2-test only at the 0.001 level due to the deviations for short runs). Hence, the experimental results obtained on this large graph coloring instance confirm the generality of our observations.

6.4.4 Interpretation

The empirical results have a number of theoretically and practically interesting consequences.

Random restart For algorithms exhibiting an exponential run-length distribution, the probability of finding a solution within a fixed time interval is independent of the run-time spent before. Based on our results, this holds for various variants of the *min conflicts* heuristic for CSPs when using approximately optimal parameter settings. Thus configured, these local search algorithms are essentially memoryless, as for a given total time t, restarting at time $t' < t$ does not significantly influence the probability of finding a solution in time t. Consequently, for these algorithms random restart is ineffective. In practice this result can easily be verified over a broad range of maxSteps settings. However, due to small deviations at the extreme left end of the run-length distributions, for very low maxSteps settings the algorithms' performance usually deteriorates. We interpret this as an effect of the initial hill-climbing phase of local search algorithms [88]. This observation is also confirmed by the computational results presented in Section 6.3.4 when we investigated the influence of the cutoff value. For smaller-than-optimal settings of the noise parameter, a different situation can be found [122]: The run-length distributions are less steep than exponential distributions, which means that the efficiency of the search process decreases over time and therefore random restart can effectively be used to speed up the algorithm. In some cases we observed run-length distributions which are steeper than exponential distributions. Such a situation occurs, for example, when trying to solve the graph

coloring instance `DSJC125.5.col` for a suboptimal number of colors. In these situations, the algorithm apparently benefits from experience gathered earlier in a run; hence, best performance is obtained by not using restart at all. For some algorithms, like the basic *min conflicts* heuristic without features of tabu search or random walk, the run-length distributions converge to a probability smaller than 1. In these cases, random restart is effective and optimal settings for the cutoff parameter always exist.

Consequences for parallel processing Randomized algorithms lend themselves to a straightforward parallelization approach by performing independent runs of the same algorithm in parallel. Given our characterizations of exponential run-length distributions for CSPs, this approach is particularly effective: Based on the properties of the exponential distribution (see Section 3.2.3 on page 51) one would observe optimal speed-up when using multiple, independent runs. This holds for almost arbitrary numbers p of processors; only for high p, due to the deviations for short runs caused by the initial hill-climb phase of local search, the speed-up would be less than optimal. Note that again this result holds for approximately optimal as well as larger-than-optimal noise parameter settings [122]. For smaller-than-optimal settings, if the overall processing time (number of processors $\times$ time per processor) is fixed, a single optimal number of processors can be derived from the run-length data, for which a super-optimal speed-up can be obtained when compared to the sequential case $p = 1$.

It should be noted, that parallel execution of independent tries is an extremely attractive model of parallel processing, since it involves basically no communication overhead and can easily be implemented and run on almost any parallel hardware platform, from networks of standard workstations to specialized MIMD machines with thousands of processors. Therefore, these results are not only relevant for the application of local search algorithms to hard combinatorial problems with time-critical tasks (like robot control or online scheduling), but also for the distributed solution of very large and hard problem instances.

6.5 Related Work

Local search for CSPs is most often based on variants of the *min conflicts* heuristic which are extended with additional features to avoid getting stuck in local minima. The simplest form to escape from local minima is to restart the local search heuristic from new random initial solutions, a feature which has to be used for the basic *min conflicts* heuristic and GSAT [223] to get reasonable performance. The application of random walk steps in the *min conflicts* heuristic has been introduced in [256], based on the random walk steps which with some fixed probability make "random" move and have been shown to be very useful to improve GSAT's performance [221]. (Actually, [221] distinguishes between *random noise* and *random walk*, where *random noise* flips any variable and with *random walk* one variable occurring in some violated clause is flipped.) Another escape mechanism, the breakout method, is proposed in [182] and is similarly used also in GENET [58]. The escape mechanism introduced by the breakout method works indirectly by assigning increasing weights to constraints that are violated in local minima. Another possibility to escape from local minima is the EFLOP procedure proposed in [265]. If one encounters a local minimum, the variable value for some variable involved in a conflict has to be changed and EFLOP propagates this through the constraint graph as long as previously satisfied constraints become unsatisfied due to the value propagation process. This last

method is actually an instance of iterated local search, because it only applies the diversification if supposedly trapped in a local minimum.

Other approaches using tabu search for SAT or CSP, we are aware of, are [107, 171, 81]. In [171] positive results for the application of tabu search to SAT are reported and it was shown that tabu search put on top of GSAT can outperform the GSAT version with random walk. Additionally the authors studied the scaling behavior of their algorithm and showed that the approximately optimal tabu list length scales linearly with problem size. This scaling does not seem to happen for the approach we presented here for the *min conflicts* heuristic, since for `tmch` the best tabu tenure for the much larger graph coloring problems is still around 2. Similarly, in [231] improved performance by the use of tabu search is reported for WSAT [222]. Since our work on extending the *min conflicts* heuristic with tabu search features [232, 231], in [81] a new tabu search approach to MAX-CSP has been presented (TS-GH). They use a modified neighborhood definition and a dynamic update scheme of the violation count. This update scheme allows that one single iteration of TS-GH can be performed in the same time as one iteration of the *min conflicts* heuristic. Their computational results show significant improvements over those obtained with `wmch` for the MAX-CSP instances tested. We re-implemented their algorithm and run it on satisfiable CSPs. We found that their algorithm also performs significantly better on satisfiable CSP than `wmch` and our proposed extensions. Hence, in fact, TS-GH appears to be the best performing local search algorithm for binary CSPs. Interestingly, the run-length of TS-GH on the hardest randomly generated instances when using approximately optimal parameter settings again can be characterized by exponential distributions.

The generality of our findings on the characterization of the run-length distributions of local search algorithms in the CSP domain is also confirmed by the experimental analysis of local search algorithms for SAT, which has been done in joint work with Holger Hoos [118, 120, 122, 119]. Experimental results for some of the best performing local search algorithms for SAT, in particular those based on the WSAT architecture [222, 172], show for a wide variety of instances a run-time behavior which can be closely approximated by exponential distributions. Exponentially distributed run-times have occasionally been reported for the application of tabu search algorithms to the job-shop scheduling problem [243] and to the quadratic assignment problem [241, 17]. Yet, the run-time distributions for the application of iterated local search algorithms to combinatorial optimizations problems presented in Chapter 5 are obviously not exponentially distributed. Consequently, further research is needed to investigate for which type of algorithms and on which problems one may observe exponentially distributed run-times.

6.6 Summary

In this chapter we have presented improvements of the *min conflicts* heuristic, a local search algorithm for solving constraint satisfaction problems, by using tabu search features. We have extensively tested the proposed improvements on randomly generated binary CSP instances from the phase transition region as well as on hard graph coloring instances from a standard benchmark set. The experimental results have shown the improved performance obtained by the new *min conflicts* heuristic extensions.

Additionally, we could show that, when using approximately optimal parameter settings, the run-time behavior of the best performing *min conflicts* heuristic extensions on the hardest

randomly generated CSP instances as well as on hard graph coloring problems can be characterized by exponential distributions. This result has important consequences for the optimal parameterization of the algorithms showing such behavior as well as for the efficiency of parallel processing.

Chapter 7

Conclusions

In this chapter we highlight the main contributions of our research and summarize the main results. Then, we present some potential avenues for future research.

7.1 Contributions

In this thesis we have studied metaheuristics for the solution of combinatorial optimization and satisfaction problems. These algorithms are general search schemes defining a set of concepts which can be used either to derive approximate algorithms for a large set of different problems, ideally with only minor problem specific adaptions, or to provide ways of extending problem specific algorithms to increase their performance. Recent research in this field has shown that metaheuristics can be used very successfully for finding very high quality solutions to large and hard problems in relatively short computation time. Our contributions to this field cover the development of tools for the analysis of the run-time behavior of metaheuristics, the analysis and improvement of recently proposed metaheuristics, and the exploration of new applications of specific metaheuristics. The main contributions can be summarized as follows.

Empirical Analysis of Metaheuristics (Chapter 3)

- We have developed and presented a novel methodology for evaluating the run-time behavior of metaheuristics, based on the measurement of run-time distributions on single instances of combinatorial optimization problems.

- We have shown that the analysis of run-time distributions is very useful to visualize an algorithm's behavior, to compare different metaheuristics, to characterize a metaheuristics run-time behavior on specific instance classes of combinatorial problems, and to reveal situations in which their performance can be improved.

- We have used this methodology to characterize the run-time behavior of local search algorithms for a specific class of hard constraint satisfaction problems (CSPs) and to considerably improve the performance of iterated local search algorithms.

Ant Colony Optimization (Chapter 4)

- We have introduced a new ant colony optimization (ACO) algorithm, called $\mathcal{MAX}$–$\mathcal{MIN}$ ant system, which significantly improves the performance of ant system, the seminal work on ACO. Comparisons of our algorithm to other recently proposed ACO algorithms have shown that $\mathcal{MAX}$–$\mathcal{MIN}$ ant system is currently the best performing variant for large, symmetric traveling salesman problems (TSPs). Our results have provided substantial evidence that for combinatorial optimization problems ACO algorithms perform best when combined with local search heuristics.

- We have also applied $\mathcal{MAX}$–$\mathcal{MIN}$ ant system to the quadratic assignment problem (QAP). The excellent computational results prove the generality of the improvements introduced by $\mathcal{MAX}$–$\mathcal{MIN}$ ant system. When applied to the QAP, $\mathcal{MAX}$–$\mathcal{MIN}$ ant system is currently among the best algorithms known for the solution of structured QAP instances. Additionally, we have shown that the choice of the local search algorithm, which should be combined with $\mathcal{MAX}$–$\mathcal{MIN}$ ant system, depends strongly on the QAP instance type.

- We also have extended $\mathcal{MAX}$–$\mathcal{MIN}$ ant system to the flow shop problem (FSP). This is the first application of an ACO algorithm to one of the "classical" scheduling problems which has reached a performance competitive with state-of-the-art algorithms.

- With our research we have provided substantial evidence that, when applied to static combinatorial optimization problems, the most successful ACO algorithms are hybrid algorithms which combine solution construction by ants with their subsequent improvement by the application of local search algorithms.

- We have presented parallelization strategies for the best performing ACO algorithms and experimental results obtained with $\mathcal{MAX}$–$\mathcal{MIN}$ ant system have shown that even with the simplest parallelization strategy, the one consisting of multiple independent runs, significant improvements can be achieved.

Iterated Local Search (Chapter 5)

- We have proposed an algorithmic framework for iterated local search (ILS) algorithms into which fit the previously proposed algorithms relying on the same basic principle as ILS.

- We have presented new applications of ILS to the QAP and to the FSP. Additionally, we have provided the first evidence that ILS can also be applied with considerable success directly to asymmetric TSPs.

- Based on the results of the run-time analysis of iterated local search algorithms for the TSP, the QAP, and the FSP we could show that standard ILS applications often suffer from a stagnation behavior which is mainly caused by the choice of the *acceptance criterion*. To avoid this stagnation behavior we have proposed extended acceptance criteria and have introduced population-based versions of ILS algorithms. We have provided extensive computational results which show that our improved ILS algorithms are currently among the best algorithms known for the TSP. Applied to the QAP they obtain very high quality

solutions comparable to the best algorithms and for the FSP we even could improve some of the best known solutions for several benchmark instances.

Tabu Search (Chapter 6)

- We have presented improvements of the *min conflicts* heuristic, a standard local search algorithm for constraint satisfaction problems (CSPs), by extending it with features of tabu search. The experimental results of our improved algorithm for hard binary CSPs from the phase transition region and for large graph coloring problems compare very favorably to previously proposed extensions.

- We have analyzed the run-time behavior of local search algorithms, including our tabu search extensions, for hard CSP instances. We could show that for approximately optimal noise parameter settings the run-time distributions arising on the hardest instances can be described by exponential distributions. This result has important consequences for the parameterization and parallelization of these local search algorithms.

7.2 Future Work

The primary goals of this thesis have been the study and improvement of metaheuristics. Our contributions and the observations made in our work also pose a number of interesting open questions for the specific research issues attacked in this thesis and for the research in the field of metaheuristics, in general.

Empirical Analysis of Metaheuristics

- We have shown that, when using optimal noise parameter settings, the run-time distributions for local search algorithms applied to constraint satisfaction problems can be described well by exponential distributions. Exponentially distributed run-times also have occasionally been reported for applications of local search algorithms to combinatorial optimization problems like the job-shop scheduling problem and the quadratic assignment problem [17, 241, 243]. Yet, the run-time distributions measured when applying iterated local search algorithms to combinatorial optimizations problems in Chapter 5 were obviously not exponentially distributed. Consequently, an open question for future research is for which type of algorithms and on which problems one may observe exponentially distributed run-times. Other important question which can be analyzed via run-time distributions are the scaling behavior of algorithms and algorithm robustness.

- A further research issue relates to the exploitation of the run-time behavior while running an algorithm. One possibility here is to exploit the time intervals it takes to find an improved solution during the algorithm's run. Research questions here are whether this information can be used to estimate the shape of the run-time distribution and whether this information can be used to derive generally useful heuristics as to when to restart an algorithm or when to use additional diversification features.

Ant Colony Optimization

- At the moment, several ACO algorithms show a promising performance on various combinatorial optimization problems. We strongly believe that future ACO applications will combine features of these ACO algorithms. Here, $\mathcal{MM}$AS may be a very good starting point, because it is one of the best ACO algorithms for combinatorial optimization problems which are often used as benchmarks to test algorithmic ideas. Further promising ideas are the use of lower bounds on the completion of partial solutions for the computation of the heuristic values as proposed in the ANTS algorithm [163] or the use of ranking for the trail updates [36].

- Another issue deals with the setting of parameters in ACO algorithms. In our experience, the parameters given here for the TSP and QAP applications performed very well over a wide range of instances. Nevertheless, in other applications adaptive versions which dynamically tune the parameters during algorithm execution may increase algorithm robustness.

- Finally, we definitely need a more thorough understanding of the features the successful application of ACO algorithm depend on and how ACO algorithms should be configured for specific problems. Particularly, the following questions need to be answered: Which solution components should be used for the solution construction? What is the best way of managing the pheromones? Should the ACO algorithm always be combined with local search algorithms? Which problems can be efficiently solved by ACO algorithms? To answer some of these questions, we believe that the investigation of search space characteristics and their relation to algorithm performance may give useful insights.

Iterated Local Search

- Our research has shown that the acceptance criterion has considerable influence on ILS performance and moves to worse solutions or specific diversification schemes can significantly improve performance. On the other hand, in the recently proposed simple variable neighborhood scheme [180, 109], the kick-move strength in the iterated local search algorithm is systematically varied typically by increasing it if no improved solution could be found (this also increases the search space exploration). Hence, an important research question concerns the importance of either of the two extensions and whether improved performance can be gained by a fine-tuned interaction between these two modifications.

- The search history does have only a very limited role in current ILS implementations. Therefore one important possibility for extending ILS algorithms is to exploit the search history as is typically done in more sophisticated tabu search implementations.

- Iterated local search is an astonishingly simple but at the same time powerful metaheuristic when compared to more complex ones like ant colony optimization or genetic algorithms. Hence, a natural question is on which types of problems ILS will show to be competitive with more complex algorithms.

Tabu Search and Local Search for Constraint Satisfaction Problems

- Current tabu search applications to CSPs are often rather simplistic in the sense that they only exploit the short term memory of the search history. Therefore, the investigation of the use of long-term memory strategies provides an interesting avenue for future research efforts in this area and to improve performance.

- Most application of local search algorithms to CSPs actually attack problems which have only binary constraints like the randomly generated binary CSPs or graph coloring problems. Yet, in many applications constraints are of higher arity and an encoding of the problem with only binary constraints leads to a strong increase in the number of constraints and variables. Therefore, an important area for research is the extension of the local search techniques to such higher arity CSPs. To do so additional features like constraint propagation techniques often used in exact algorithms for CSPs may prove to be extremely useful.

Understanding Metaheuristic Behavior

When comparing the results obtained with ACO algorithms and iterated local search for the three TSP, the QAP, and the FSP we can make the following general observations which, in fact, are not only limited to the two types of algorithms applied in this thesis.

- Our computational results for $\mathcal{MAX}$–$\mathcal{MIN}$ ant system and iterated local search applications have shown that there does not exist one single best algorithm for all problems. It is not obvious which metaheuristic should best be applied to a particular problem to be solved.

- For a given problem, the relative performance of metaheuristics may strongly depend on the particular class of problem instances. For example, our computational results for the TSP have shown that ILS algorithms are significantly better than $\mathcal{MAX}$–$\mathcal{MIN}$ ant system on strongly clustered TSP instances, while on more regular instances the differences between both types of algorithms are much smaller. A similar situation occurs for the QAP with $\mathcal{MM}$AS performing better on large real-life instances.

- Once chosen a particular metaheuristic to be applied to a specific problem, there are often many different possibilities how it can be applied. For example, in our application of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system to the QAP the best choice of the local search algorithm depends strongly on the instance type (see Section 4.4). Although some guidelines can be given, it is not obvious how general they are.

These observations exemplify one of the most important issues which have to be addressed in future metaheuristic research. There are several available metaheuristics but it is often not clear to potential users which metaheuristic should be chosen for their specific application or for their problem class of interest, and how a metaheuristic should be configured when applied to a new problem. To answer these questions a more thorough understanding of metaheuristics behavior, the interaction between their design components, and the final performance achieved has to be gained. An understanding of these issues would lead to the development of more hybrid metaheuristics which combine aspects of already known ones and to a more principled

approach to the design of metaheuristic applications. In the end, such a principled approach may lead to an assembly of metaheuristics applications based on an appropriate choice of design components of different metaheuristics and to a more unified view of the field.

The ultimate goal of research in metaheuristics is to provide general and flexible, but at the same time powerful and efficient algorithms to approximately solve large, hard combinatorial optimization problems. By the presented improvements on existing metaheuristics, by the new tools for their analysis and for the characterization of their run-time behavior, and by new applications of existing metaheuristics we add our contribution to this exciting field. We hope that this thesis can be regarded as a further step towards achieving the goals of metaheuristic algorithms.

Bibliography

[1] E.H.L. Aarts. Local Search Algorithms. Invited Lecture at the Combinatorial Optimization 1998 conference, Brussels, Belgium, 1998.

[2] E.H.L. Aarts and J. Korst. *Simulated Annealing and Boltzman Machines – A Stochastic Approach to Combinatorial Optimization and Neural Computation*. John Wiley & Sons, Chichester, 1989.

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

[4] R. K. Ahuja and J. B. Orlin. Use of Representative Operation Counts in Computational Testing of Algorithms. *INFORMS Journal on Computing*, 8(3):318–330, 1996.

[5] H. Alt, L. Guibas, K. Mehlhorn, R. Karp, and A. Wigderson. A Method for Obtaining Randomized Algorithms with Small Tail Probabilities. *Algorithmica*, 16:543–547, 1996.

[6] E. Angel and V. Zissimopoulos. Autocorrelation Coefficient for the Graph Bipartitioning Problem. *Theoretical Computer Science*, 191:229–243, 1998.

[7] E. Angel and V. Zissimopoulos. Towards a Classification of Combinatorial Optimization Problems Relatively to their Difficulty for Generalized Local Search Algorithms. Submitted to *Discrete Applied Mathematics*, 1999.

[8] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding Cuts in the TSP. Technical Report 95-05, DIMACS, March 1995.

[9] S. Arora. Polynomial Time Approximation Schemes for Euclidean TSP and other Geometric Problems. To appear in *Journal of the ACM*, 1999.

[10] G. Ausiello, P. Crescenzi, and M. Protasi. Approximate Solution of $\mathcal{NP}$ Optimization Problems. *Theoretical Computer Science*, 150:1–55, 1995.

[11] L. Babai. Monte Carlo Algorithms in Graph Isomorphism Testing. Technical Report DMS 79-10, Université de Montréal, 1979.

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

[13] R.E. Barlow and F. Proschan. *Statistical Theory of Reliability and Life Testing*. TO BEGIN WITH, Silver Spring, 1975.

[14] R.S. Barr, B.L. Golden, J.P. Kelly, M.G.C. Resende, and W.R. Stewart. Designing and Reporting on Computational Experiments with Heuristic Methods. *Journal of Heuristics*, 1(1):9–32, 1995.

[15] R. Battiti and M. Protasi. Reactive Search, A History-Based Heuristic for MAX-SAT. *ACM Journal of Experimental Algorithmics*, 2, 1997.

[16] R. Battiti and M. Protasi. Approximate Algorithms and Heuristics for MAX-SAT. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 1, pages 77–148. Kluwer Academic Publishers, 1998.

[17] R. Battiti and G. Tecchiolli. Parallel Biased Search for Combinatorial Optimization: Genetic Algorithms and TABU. *Microprocessor and Microsystems*, 16(7):351–367, 1992.

[18] R. Battiti and G. Tecchiolli. The Reactive Tabu Search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[19] R. Battiti and G. Tecchiolli. Simulated Annealing and Tabu Search in the Long Run: A Comparison on QAP Tasks. *Computer and Mathematics with Applications*, 28(6):1–8, 1994.

[20] E.B. Baum. Towards Practical 'Neural' Computation for Combinatorial Optimization Problems. In *Neural Networks for Computing, AIP Conference Proceedings*, pages 53–64, 1986.

[21] J.L. Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA Journal on Computing*, 4(4):387–411, 1992.

[22] A. Beringer, G. Aschemann, H. Hoos, and A. Weiß. GSAT versus Simulated Annealing. In A. Cohn, editor, *Proceedings of the European Conference on Artificial Intelligence*, pages 110–115. John Wiley & Sons, 1994.

[23] H. Bersini, M. Dorigo, S. Langerman, G. Seront, and L. Gambardella. Results of the First International Contest on Evolutionary Optimisation. Technical Report TR/IRIDIA/96-18, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium, 1996.

[24] D.P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[25] K.D. Boese. Cost Versus Distance in the Traveling Salesman Problem. Technical Report TR-950018, UCLA Computer Science Department, Los Angeles, CA USA, 1995.

[26] K.D. Boese. *Models for Iterative Global Optimization*. PhD thesis, University of California, Computer Science Department, Los Angeles, USA, 1996.

[27] K.D. Boese, A.B. Kahng, and S. Muddu. A New Adaptive Multi-Start Technique for Combinatorial Global Optimization. *Operations Research Letters*, 16:101–113, 1994.

[28] M. Bolondi and M. Bondanza. Parallelizzazione di un Algoritmo per la Risoluzione del Problema del Commesso Viaggiatore. Master's thesis, Politecnico di Milano, Milano, Italy, 1993.

[29] E. Bonabeau, M. Dorigo, and G. Theraulaz. *From Natural to Artificial Swarm Intelligence*. Oxford University Press, 1999.

[30] R.M. Brady. Optimization Strategies Gleaned from Biological Evolution. *Nature*, 317:804–806, 1985.

[31] D. Brélaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(4):251–256, 1979.

[32] P. Brucker, J. Hurink, and F. Werner. Improving Local Search Heuristics for some Scheduling Problems — Part I. *Discrete Applied Mathematics*, 65(1–3):97–122, 1996.

[33] P. Brucker, J. Hurink, and F. Werner. Improving Local Search Heuristics for some Scheduling Problems — Part II. *Discrete Applied Mathematics*, 72(1–2):47–69, 1997.

[34] B. Bullnheimer, R. F. Hartl, and C. Strauss. An Improved Ant System Algorithm for the Vehicle Routing Problem. *Annals of Operations Research*, 89, 1999.

[35] B. Bullnheimer, R.F. Hartl, and C. Strauss. Applying the Ant System to the Vehicle Routing Problem. In S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 109–120. Kluwer, Boston, 1999.

[36] B. Bullnheimer, R.F. Hartl, and C. Strauss. A New Rank Based Version of the Ant System — A Computational Study. *Central European Journal for Operations Research and Economics*, 7(1):25–38, 1999.

[37] B. Bullnheimer, G. Kotsis, and C. Strauss. Parallelization Strategies for the Ant System. Technical Report POM 9/97, University of Vienna, Vienna, Austria, 1997.

[38] R.E. Burkard and U. Fincke. Probabilistic Asymptotic Properties of some Combinatorial Optimization Problems. *Discrete Applied Mathematics*, 12:21–29, 1985.

[39] R.E. Burkard, S. Karisch, and F. Rendl. QAPLIB - A Quadratic Assignment Problem Library. *European Journal of Operational Research*, 55:115–119, 1991.

[40] R.E. Burkard and J. Offermann. Entwurf von Schreibmaschinentastaturen mittels quadratischer Zuordnungsprobleme. *Zeitschrift für Operations Research*, 21:B121–B132, 1977.

[41] R.E. Burkard and F. Rendl. A Thermodynamically Motivated Simulation Procedure for Combinatorial Optimization Problems. *European Journal of Operational Research*, 17:169–174, 1984.

[42] H. Campbell, R. Dudek, and M. Smith. A Heuristic Algorithm for the n Job, m Machine Sequencing Problem. *Management Science*, 16(10):B630–B637, 1970.

[43] E. Cantú-Paz. A Survey of Parallel Genetic Algorithms. Technical Report IlliGAL 97003, University of Illinois at Urbana-Champaign, Ilinois, USA, 1997.

[44] V. Cerný. A Thermodynamical Approach to the Traveling Salesman Problem. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.

[45] N. Christofides. Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[46] D. Clark, J. Frank, I. Gent, E. MacIntyre, N. Tomov, and T. Walsh. Local Search and the Number of Solutions. In E. Freuder, editor, *Principles and Practice of Constraint Programming - CP'96*, volume 1118 of *LNCS*, pages 119–133. Springer Verlag, 1996.

[47] B. Codenotti, G. Manzini, L. Margara, and G. Resta. Perturbation: An Efficient Technique for the Solution of Very Large Instances of the Euclidean TSP. *INFORMS Journal on Computing*, 8:125–133, 1996.

[48] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed Optimization by Ant Colonies. In *Proceedings of ECAL91 - European Conference on Artificial Life*, pages 134–142. Elsevier Publishing, 1991.

[49] A. Colorni, M. Dorigo, V. Maniezzo, and M. Trubian. Ant System for Job-Shop Scheduling. *Belgian Journal of Operations Research, Statistics and Computer Science*, 34(1):39–53, 1994.

[50] R.K. Congram, C.N. Potts, and S.L. Van de Velde. Dynasearch — Iterative Local Improvement by Dynamic Programming: The Total Weighted Tardiness Problem. In *Talk presented at CO98, Brussels, Belgium*, 1998.

[51] D.T. Connolly. An Improved Annealing Scheme for the QAP. *European Journal of Operational Research*, 46:93–100, 1990.

[52] D. Corne, M. Dorigo, and F. Glover. *New Ideas in Optimization*. McGraw-Hill, 1999.

[53] T.G. Crainic, M. Toulouse, and M. Gendreau. Toward a Taxonomy of Parallel Tabu Search Heuristics. *INFORMS Journal on Computing*, 9(1):61–72, 1997.

[54] J.M. Crawford and L.D. Auton. Experimental Results on the Crossover Point in Random 3SAT. *Artificial Intelligence*, 81:31–57, 1996.

[55] G.A. Croes. A Method for Solving Traveling Salesman Problems. *Operations Research*, 6:791–812, 1958.

[56] V.-D. Cung, T. Mautor, P. Michelon, and A. Tavares. A Scatter Search Based Approach for the Quadratic Assignment Problem. In T. Baeck, Z. Michalewicz, and X. Yao, editors, *Proceedings of ICEC'97*, pages 165–170. IEEE Press, Piscataway, NJ, USA, 1997.

[57] D. Dannenbring. An Evaluation of Flow Shop Sequencing Heuristics. *Management Science*, 23(11):1174–1182, 1977.

[58] A. Davenport, E. Tsang, C.J. Wang, and K. Zhu. GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement. In *Proceedings of the 14th National Conference on Artificial Intelligence*. MIT press, 1994.

[59] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5:394–397, 1962.

[60] R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.

[61] J.W. Dickey and J.W. Hopkins. Campus Building Arrangement Using TOPAZ. *Transportation Science*, 6:59–68, 1972.

[62] N. Dodd. Slow Annealing Versus Multiple Fast Annealing Runs – An Empirical Investigation. *Parallel Computing*, 16:269–272, 1990.

[63] M. Dorigo. *Optimization, Learning, and Natural Algorithms (in Italian)*. PhD thesis, Dip. Elettronica e Informazione, Politecnico di Milano, Milano, Italy, 1992.

[64] M. Dorigo and G. Di Caro. The Ant Colony Optimization Meta-Heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*. McGraw-Hill, 1999.

[65] M. Dorigo, G. di Caro, and L.M. Gambardella. Ant Algorithms for Distributed Discrete Optimization. *Artificial Life*, 5, 1999.

[66] M. Dorigo and L.M. Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[67] M. Dorigo, V. Maniezzo, and A. Colorni. Positive Feedback as a Search Strategy. Technical Report 91-016, Politecnico di Milano, Milano, Italy, 1991.

[68] M. Dorigo, V. Maniezzo, and A. Colorni. The Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, 26(1):29–42, 1996.

[69] R. Dudek, S. Panwalker, and M. Smith. The Lessons of Flowshop Scheduling Research. *Operations Research*, 40(1):7–13, 1992.

[70] A.N. Elshafei. Hospital Layout as a Quadratic Assignment Problem. *Operations Research Quarterly*, 28:167–179, 1977.

[71] U. Faigle and W. Kern. Some Convergence Results for Probabilistic Tabu Search. *ORSA Journal on Computing*, 4(1):32–37, 1992.

[72] T.A. Feo and M.G.C. Resende. A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem. *Operations Research Letters*, 8:67–71, 1989.

[73] T.A. Feo and M.G.C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[74] C. Fleurent and J.A. Ferland. Genetic Hybrids for the Quadratic Assignment Problem. In P.M. Pardalos and H. Wolkowicz, editors, *Quadratic Assignment and Related Problems*, volume 16 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 173–187. American Mathematical Society, 1994.

[75] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons, New York, 1966.

[76] J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, USA, 1995.

[77] B. Freisleben and P. Merz. A Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problems. In *Proceedings of ICEC'96*, pages 616–621. IEEE Press, Piscataway, NJ, USA, 1996.

[78] B. Freisleben and P. Merz. New Genetic Local Search Operators for the Traveling Salesman Problem. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Proceedings of the* $4th$ *Conference on Parallel Problem Solving from Nature – PPSN-IV*, volume 1141 of *LNCS*, pages 890–900. Springer Verlag, 1996.

[79] E.C. Freuder and R.J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[80] D. Frost, I. Rish, and L. Vila. Summarizing CSP Hardness with Continuous Probability Distributions. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 327–333. MIT Press, 1997.

[81] P. Galinier and J.-K. Hao. Tabu Search for Maximal Constraint Satisfaction Problems. In G. Smolka, editor, *Principles and Practice of Constraint Programming — CP97*, volume 1330 of *LNCS*, pages 196–208. Springer Verlag, 1997.

[82] L.M. Gambardella and M. Dorigo. Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 252–260. Morgan Kaufmann, San Francisco, 1995.

[83] L.M. Gambardella and M. Dorigo. Solving Symmetric and Asymmetric TSPs by Ant Colonies. In *Proceedings of ICEC'96*, pages 622–627. IEEE Press, Piscataway, NJ, USA, 1996.

[84] L.M. Gambardella and M. Dorigo. HAS-SOP: Hybrid Ant System for the Sequential Ordering Problem. Technical Report Technical Report IDSIA 11-97, IDSIA, Lugano, Switzerland, 1997.

[85] L.M. Gambardella, É.D. Taillard, and M. Dorigo. Ant Colonies for the QAP. *Journal of the Operational Research Society*, 50, 1999.

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

[87] S. Geman and D. Geman. Stochastic Relaxation, Gibbs Distribution, and the Bayesian Restoration of Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

[88] I.P. Gent and T. Walsh. An Empirical Analysis of Search in GSAT. *Journal of Artificial Intelligence Research*, 1:47–59, 1993.

[89] A.M. Geoffrion and G.W. Graves. Scheduling Parallel Production Lines with Changeover Costs: Practical Applications of a Quadratic Assignment/LP Approach. *Operations Research*, 24:595–610, 1976.

[90] F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.

[91] F. Glover. Tabu Search – Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[92] F. Glover. Tabu Search – Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[93] F. Glover. Tabu Thresholding: Improved Search by Nonmonotonic Trajectories. *ORSA Journal on Computing*, 7(4):426–442, 1995.

[94] F. Glover. Ejection Chains, Reference Structures and Alternating Path Methods for Traveling Salesman Problems. *Discrete Applied Mathematics*, 65:223–253, 1996.

[95] F. Glover. Tabu Search and Adaptive Memory Programming — Advances, Applications and Challenges. In Barr, Helgason, and Kennington, editors, *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer Academic Publishers, 1996.

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

[97] F. Glover, E. Taillard, and D. de Werra, editors. *Tabu Search*, volume 43 of *Annals of Operations Research*. Baltzer Science Publishers, 1993.

[98] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[99] C.P. Gomes and B. Selman. Algorithm Portfolio Design: Theory vs. Practice. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI'97)*, pages 190–197. Morgan Kaufmann, San Francisco, 1997.

[100] C.P. Gomes, B. Selman, and N. Crato. Heavy-Tailed Distributions in Combinatorial Search. In *Proceedings of CP'97*, number 1330 in LNCS, pages 121–135. Springer Verlag, 1997.

[101] M. Gorges-Schleuter. Explicit Parallelism of Genetic Algorithms through Population Structures. In *Proceedings of the 1*st *Workshop on Parallel Problem Solving from Nature*, number 496 in LNCS, pages 150–159. Springer Verlag, 1991.

[102] M. Gorges-Schleuter. Asparagos96 and the Travelling Salesman Problem. In T. Baeck, Z. Michalewicz, and X. Yao, editors, *Proceedings of ICEC'97*, pages 171–174. IEEE Press, Piscataway, NJ, USA, 1997.

[103] M. Grötschel. *Polyedrische Charakterisierung kombinatorischer Optimierungsprobleme*. Hain, Meisenheim am Glan, 1977.

[104] M. Grötschel and O. Holland. Solution of Large-scale Symmetric Traveling Salesman Problems. *Mathematical Programming*, 51:141–202, 1991.

[105] H.W. Guesgen and J. Hertzberg. *A Perspective of Constraint-Based Reasoning*, volume 597 of *LNAI*. Springer Verlag, 1992.

[106] B. Hajek. Cooling Schedules for Optimal Annealing. *Mathematics of OR*, 13:311–329, 1988.

[107] P. Hansen and B. Jaumard. Algorithms for the Maximum Satisfiability Problem. *Computing*, 44:279–303, 1990.

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

[109] P. Hansen and N. Mladenović. An Introduction to Variable Neighborhood Search. In S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 433–458. Kluwer, Boston, 1999.

[110] W.E. Hart. A Theoretical Comparison of Evolutionary Algorithms and Simulated Annealing. In *Proceedings Fifth Annual Conference on Evolutionary Programming (EP96)*, pages 147–154, 1996.

[111] A. Hertz and D. de Werra. Using Tabu Search Techniques for Graph Coloring. *Computing*, 39:345–351, 1987.

[112] A. Hertz, E. Taillard, and D. de Werra. A Tutorial on Tabu Search. In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 121–136. John Wiley & Sons, 1997.

[113] T. Hogg and C.P. Williams. Expected Gains from Parallelizing Constraint Solving for Hard Problems. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 331–336. MIT Press, 1994.

[114] T. Hogg and C.P. Williams. The Hardest Constraint Problems: A Double Phase Transition. *Artificial Intelligence*, 69:357–377, 1994.

[115] J.H. Holland. *Adaption in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[116] J.N. Hooker. Needed: An Empirical Science of Algorithms. *Operations Research*, 42(2):201–212, 1994.

[117] J.N. Hooker. Testing Heuristics: We Have It All Wrong. *Journal of Heuristics*, pages 33–42, 1996.

[118] H.H. Hoos and T. Stützle. A Characterization of GSAT's Performance on a Class of Hard Structured Problems. Technical Report AIDA–96–20, FG Intellektik, TH Darmstadt, Darmstadt, Germany, 1996.

[119] H.H. Hoos and T. Stützle. Characterizing the Run-time Behavior of Stochastic Local Search. Technical Report AIDA–98–01, FG Intellektik, TU Darmstadt, Darmstadt, Germany, 1998.

[120] H.H. Hoos and T. Stützle. Evaluating Las Vegas Algorithms — Pitfalls and Remedies. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 238–245. Morgan Kaufmann, San Francisco, 1998.

[121] H.H. Hoos and T. Stützle. Satlib - the Satisfiability Library, 1998. Accessible at `http://www.informatik.tu-darmstadt.de/AI/SATLIB`.

[122] H.H. Hoos and T. Stützle. Characterising the Behaviour of Stochastic Local Search. *Artificial Intelligence*, 112:213–232, 1999.

[123] J.J. Hopfield and D. Tank. Neural Computations of Decisions in Optimization Problems. *Biological Cybernetics*, 52:141–152, 1985.

[124] T.C. Hu, A.B. Kahng, and C.-W.A. Tsao. Old Bachelor Acceptance: A New Class of Non-Monotone Threshold Accepting Methods. *ORSA Journal on Computing*, 7(4):417–425, 1995.

[125] H. Ishibuchi, S. Misaki, and H. Tanaka. Modified Simulated Annealing Algorithms for the Flow Shop Sequencing Problem. *European Journal of Operational Research*, 81:388–398, 1995.

[126] D.S. Johnson. Local Optimization and the Travelling Salesman Problem. In *Proc. 17th Colloquium on Automata, Languages, and Programming*, volume 443 of *LNCS*, pages 446–461. Springer Verlag, 1990.

[127] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation: Part I, Graph Partitioning. *Operations Research*, 37(6):865–892, 1989.

[128] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation: Part II, Graph Coloring and Number Partitioning. *Operations Research*, 39(3):378–406, 1991.

[129] D.S. Johnson and L.A. McGeoch. The Travelling Salesman Problem: A Case Study in Local Optimization. In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, 1997.

[130] D.S. Johnson, C.H. Papadimitriou, and M. Yannakakis. How Easy is Local Search? *Journal of Computer System Science*, 37:79–100, 1988.

[131] S. Johnson. Optimal Two- and Three-stage Production Scheduling with Setup Times Included. *Naval Reseach Logistics Quarterly*, 1:61–68, 1954.

[132] T. Jones and S. Forrest. Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms. In L.J. Eshelman, editor, *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 184–192. Morgan Kaufman, San Francisco, 1995.

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

[134] S. Karisch. personal communication, 1997.

[135] S.A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, 1993.

[136] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proceedings of the 13th National Conference on Artificial Intelligence*, volume 2, pages 1194–1201. MIT Press, 1996.

[137] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technology Journal*, 49:213–219, 1970.

[138] S. Khanna, R. Motwani, M. Sudan, and U. Vazirani. On Syntactic Versus Computational Views of Approximability. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 819–836, Los Angeles, CA, 1994. IEEE Computer Society.

[139] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[140] S. Kirkpatrick and B. Selman. Critical Behavior in the Satisfiability of Random Boolean Expressions. *Science*, 264:1297–1301, 1994.

[141] S. Kirkpatrick and G. Toulouse. Configuration Space Analysis of Travelling Salesman Problems. *Journal de Physique*, 46(8):1277–1292, 1985.

[142] A. Kolen and E. Pesch. Genetic Local Search in Combinatorial Optimization. *Discrete Applied Mathematics*, 48:273–284, 1994.

[143] R.E. Korf. Macro-Operators: A Weak Method for Learning. *Artificial Intelligence*, 26:35–77, 1985.

[144] C. Koulamas. A New Constructive Heuristic for the Flowshop Scheduling Problem. *European Journal of Operational Research*, 105:66–71, 1998.

[145] J. Krarup and P.M. Pruzan. Computer-aided Layout Design. *Mathematical Programming Study*, 9:75–94, 1978.

[146] M.W. Krentel. Structure in Locally Optimal Solutions. In *Proc. of the $30th$ Annual Symposium on Foundations of Computer Science*, pages 216–221. IEEE Computer Society, Los Angeles, CA, 1989.

[147] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[148] A. Kwan. The Validity of Normality Assumption in CSP Research. In *Proceedings of PRICAI'96*, LNCS. Springer Verlag, 1996.

[149] G. Laporte and I.H. Osman, editors. *Meta-Heuristics in Combinatorial Optimization*, volume 63 of *Annals of Operations Research*. Baltzer Science Publishers, 1996.

[150] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, New York, 1976.

[151] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Travelling Salesman Problem*. John Wiley & Sons, 1985.

[152] E.L. Lawler and D.E. Wood. Branch-and-Bound Methods: A Survey. *Operations Research*, 14(4):699–719, 1966.

[153] C.M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, volume 1, pages 366–371. Morgan Kaufmann, San Francisco, 1997.

[154] Y. Li, P.M. Pardalos, and M.G.C. Resende. A Greedy Randomized Adaptive Search Procedure for the Quadratic Assignment Problem. In P.M. Pardalos and H. Wolkowicz, editors, *Quadratic assignment and related problems*, volume 16 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 237–261. American Mathematical Society, 1994.

[155] S. Lin. Computer Solutions for the Traveling Salesman Problem. *Bell Systems Technology Journal*, 44:2245–2269, 1965.

[156] S. Lin and B.W. Kernighan. An Effective Heuristic Algorithm for the Travelling Salesman Problem. *Operations Research*, 21:498–516, 1973.

[157] H. Ramalhinho Lourenço. Job-Shop Scheduling: Computational Study of Local Search and Large-Step Optimization Methods. *European Journal of Operational Research*, 83:347–364, 1995.

[158] H. Ramalhinho Lourenço and D. Serra. Adaptive Approach Heuristics for the Generalized Assignment Problem. Technical Report Economic Working Papers Series No.304, Universitat Pompeu Fabra, Dept. of Economics and Management, Barcelona, Spain, 1998.

[159] H. Ramalhinho Lourenço and M. Zwijnenburg. Combining the Larg-Step Optimization with Tabu-Search: Application to the Job-Shop Scheduling Problem. In I.H. Osman and J.P. Kelly, editors, *Meta-Heuristics: Theory & Applications*, pages 219–236. Kluwer Academic Publishers, 1996.

[160] M. Luby, A. Sinclair, and D. Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Information Processing Letters*, 47:173–180, 1993.

[161] M. Lundy and A. Mees. Convergence of an Annealing Algorithm. *Mathematical Programming*, 34:111–124, 1986.

[162] A.K. Mackworth. Constraint Satisfaction. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 205–211. John Wiley & Sons, 1987.

[163] V. Maniezzo. Exact and Approximate Nondeterministic Tree-Search Procedures for the Quadratic Assignment Problem. Technical Report CSR 98-1, Science dell'Informazione, Universitá di Bologna, sede di Cesena, Italy, 1998.

[164] V. Maniezzo and A. Colorni. The Ant System Applied to the Quadratic Assignment Problem. To appear in *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[165] V. Maniezzo, M. Dorigo, and A. Colorni. The Ant System Applied to the Quadratic Assignment Problem. Technical Report IRIDIA/94-28, Université Libre de Bruxelles, Belgium, 1994.

[166] O. Martin. Personal communication, 1997.

[167] O. Martin and S.W. Otto. Partitoning of Unstructured Meshes for Load Balancing. *Concurrency: Practice and Experience*, 7:303–314, 1995.

[168] O. Martin and S.W. Otto. Combining Simulated Annealing with Local Search Heuristics. *Annals of Operations Research*, 63:57–75, 1996.

[169] O. Martin, S.W. Otto, and E.W. Felten. Large-Step Markov Chains for the Traveling Salesman Problem. *Complex Systems*, 5(3):299–326, 1991.

[170] O. Martin, S.W. Otto, and E.W. Felten. Large-step Markov Chains for the TSP Incorporating Local Search Heuristics. *Operations Research Letters*, 11:219–224, 1992.

[171] B. Mazure, L. Sais, and E. Gregoire. TWSAT: A New Local Search Algorithm for SAT – Performance and Analysis. In *Proceedings of CP'95 Workshop on Solving Hard Problems*, 1995.

[172] D. McAllester, B. Selman, and H. Kautz. Evidence for Invariants in Local Search. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 321–326. MIT press, 1997.

[173] P. Merz and B. Freisleben. A Genetic Local Search Approach to the Quadratic Assignment Problem. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA'97)*, pages 465–472. Morgan Kaufmann, San Francisco, 1997.

[174] P. Merz and B. Freisleben. Genetic Local Search for the TSP: New Results. In *Proceedings of ICEC'97*, pages 159–164. IEEE Press, Piscataway, NJ, USA, 1997.

[175] P. Merz and B. Freisleben. Fitness Landscapes, Memetic Algorithms and Greedy Operators for Graph Bi-Partitioning. Technical Report TR-98-01, University of Siegen, Department of Electrical Engineering and Computer Science, Siegen, Germany, 1998.

[176] Z. Michalewicz. *Genetic Algorithms + Data Structurs = Evolution Programs*. Springer Verlag, Berlin, 2nd. edition edition, 1994.

[177] R. Michel and M. Middendorf. An Island Based Ant System with Lookahead for the Shortest Common Supersequence Problem. In *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature*, volume 1498 of *LNCS*, pages 692–708. Springer Verlag, 1998.

[178] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, 52:161–205, 1992.

[179] L.A. McGeoch M.L. Fredman, D.S. Johnson and G. Ostheimer. Data Structures for Traveling Salesmen. *Journal of Algorithms*, 18(3):432–479, 1995.

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

[181] A. Möbius, B. Freisleben, P. Merz, and M. Schreiber. Combinatorial Optimization by Iterative Partial Transcription. *Physical Review E*, 59(4):4667–4674, 1999.

[182] P. Morris. The Breakout Method for Escaping from Local Minima. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 40–45. MIT press, 1993.

[183] P. Moscato. On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms. Technical Report 790, Caltech Concurrent Computation Program, 1989.

[184] P. Moscato and M.G. Norman. A 'Memetic' Approach for the Traveling Salesman Problem. Implementation of a Computational Ecology for Combinatorial Optimization on Message-Passing Systems. In M. Valero, E. Onate, M. Jane, J.L. Larriba, and B. Suarez, editors, *Parallel Computing and Transputer Applications*, pages 187–194. IOS Press, Amsterdam, 1992.

[185] H. Mühlenbein. Evolution in Time and Space – the Parallel Genetic Algorithm. In *Foundations of Genetic Algorithms*, pages 316–337. Morgan Kaufmann, San Mateo, 1991.

[186] H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer. Evolution Algorithms in Combinatorial Optimization. *Parallel Computing*, 7:65–85, 1988.

[187] Y. Nagata and S. Kobayashi. Edge Assembly Crossover: A High-power Genetic Algorithm for the Traveling Salesman Problem. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA'97)*, pages 450–457. Morgan Kaufmann, San Francisco, 1997.

[188] M. Nawaz, E. Enscore Jr., and I. Ham. A Heuristic Algorithm for the m-Machine, n-Job Flow-Shop Sequencing Problem. *OMEGA*, 11(1):91–95, 1983.

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

[190] V. Nissen. Solving the Quadratic Assignment Problem with Clues from Nature. *IEEE Transactions on Neural Networks*, 5(1):66–72, 1994.

[191] E. Nowicki and C. Smutnicki. A Fast Tabu Search Algorithm for the Permutation Flow-Shop Problem. *European Journal of Operational Research*, 91:160–175, 1996.

[192] F. Ogbu and D. Smith. The Application of the Simulated Annealing Algorithm to the Solution of the $n/m/C_{\max}$ Flowshop Problem. *Computers & Operations Research*, 17(3):243–253, 1990.

[193] I. Osman and C. Potts. Simulated Annealing for Permutation Flow-Shop Scheduling. *OMEGA*, 17(6):551–557, 1989.

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

[195] I.H. Osman and G. Laporte. Metaheuristics: A Bibliography. *Annals of Operations Research*, 63:513–628, 1996.

[196] M. Padberg and G. Rinaldi. A Branch-And-Cut Algorithm for the Resolution of Large-Scale Symmetric Travelling Salesman Problems. *SIAM Review*, 33(1):60–100, 1991.

[197] C.H. Papadimitriou. The Complexity of the Lin-Kernighan Heuristic for the Traveling Salesman Problem. *SIAM Journal on Computing*, 21(3):450–465, 1992.

[198] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[199] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization – Algorithms and Complexity*. Prentice Hall, 1982.

[200] A.J. Parkes and J.P. Walser. Tuning Local Search for Satisfiability Testing. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pages 356–362. MIT press, 1996.

[201] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[202] E. Pesch and S. Voß, editors. *Applied Local Search*, volume 17 of *OR Spektrum*, 1995.

[203] P. Prosser. An Empirical Study of Phase Transitions in Binary Constraint Satisfaction Problems. *Artificial Intelligence*, 81:81–109, 1996.

[204] V.J. Rayward-Smith, editor. *Modern Heuristic Search Methods*. John Wiley & Sons, 1996.

[205] I. Rechenberg. *Evolutionsstrategie — Optimierung technischer Systeme nach Prinzipien der biologischen Information*. Fromman Verlag, Freiburg, 1973.

[206] C. Reeves. Improving the Efficiency of Tabu Search for Machine Sequencing Problems. *Journal of the Operational Research Society*, 44(4):375–382, 1993.

[207] C. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill, 1993.

[208] C. Reeves. A Genetic Algorithm for Flowshop Sequencing. *Computers & Operations Research*, 22(1):5–13, 1995.

[209] C. Reeves. Landscapes, Operators and Heuristic Search. To appear in *Annals of Operations Research*, 1998.

[210] G. Reinelt. TSPLIB — A Traveling Salesman Problem Library. *ORSA Journal On Computing*, 3:376–384, 1991.

[211] G. Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*, volume 840 of *LNCS*. Springer Verlag, 1994.

[212] W.T. Rhee. A Note on Asymptotic Properties of the Quadratic Assignment Problem. *Operations Research Letters*, 7(4):197–200, 1988.

[213] V.K. Rohatgi. *An Introduction to Probability Theory and Mathematical Statistics*. John Wiley & Sons, 1976.

[214] A. Rohe. Parallele Heuristiken für sehr große Traveling Salesman Probleme. Master's thesis, Universität Bonn, Fachbereich Mathematik, Bonn, Germany, 1997.

[215] F. Romeo and A. Sangiovanni-Vincentelli. A Theoretical Framework for Simulated Annealing. *Algorithmica*, 6:302–345, 1991.

[216] N. Sadeh, K. Sycara, and Y. Xiong. Backtracking Techniques for the Job Shop Scheduling Constraint Satisfaction Problem. *Artificial Intelligence*, 76:455–480, 1995.

[217] N.M. Sadeh and M.S. Fox. Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem. *Artificial Intelligence*, 86:1–41, 1996.

[218] S. Sahni and T. Gonzalez. P-Complete Approximation Problems. *Journal of the ACM*, 23(3):555–565, 1976.

[219] G.R. Schreiber and O. Martin. Cut Size Statistics for Graph Bisection Heuristics. To appear in *SIAM Journal on Optimization*, 1998.

[220] H.-P. Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, Chichester, 1981.

[221] B. Selman and H.A. Kautz. Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 290–295. Morgan Kaufmann Publishers Inc., 1993.

[222] B. Selman, H.A. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 337–343. MIT press, 1994.

[223] B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 440–446. MIT press, 1992.

[224] R. Shonkwiler. Parallel Genetic Algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA'93)*. Morgan Kaufmann, San Francisco, 1993.

[225] J. Skorin-Kapov. Tabu Search Applied to the Quadratic Assignment Problem. *ORSA Journal on Computing*, 2:33–45, 1990.

[226] B.M. Smith. Phase Transitions and the Mushy Region in Constraint Satisfaction Problems. In *Proceedings of the 12th European Conference on Artificial Intelligence*, pages 100–104. John Wiley & Sons, 1994.

[227] B.M. Smith and M.E. Dyer. Locating the Phase Transition in Binary Constraint Satisfaction Problems. *Artificial Intelligence*, 81:155–181, 1996.

[228] W.M. Spears. Simulated Annealing for Hard Satisfiability Problems. Technical report, Naval Research Laboratory, Washington D.C., 1993.

[229] P.F. Stadler. Towards a Theory of Landscapes. Technical Report Technical Report SFI–95–03–030, Santa Fe Institute, 1995.

[230] L. Steinberg. The Backboard Wiring Problem: A Placement Algorithm. *SIAM Review*, 3:37–50, 1961.

[231] O. Steinmann, A. Strohmaier, and T. Stützle. Tabu Search vs. Random Walk. In *Advances in Artificial Intelligence (KI97)*, volume 1303 of *LNAI*, pages 337–348. Springer Verlag, 1997.

[232] T. Stützle. Lokale Suchverfahren für Constraint Satisfaction Probleme: Die *min-conflicts* Heuristik und Tabu Search. *Künstliche Intelligenz*, 1/97:14–20, 1997.

[233] T. Stützle. An Ant Approach to the Flow Shop Problem. In *Proceedings of the $6th$ European Congress on Intelligent Techniques & Soft Computing (EUFIT'98)*, volume 3, pages 1560–1564. Verlag Mainz, Wissenschaftsverlag, Aachen, 1998.

[234] T. Stützle. Iterated Local Search for the Flow Shop Problem. Submitted to *European Journal of Operational Research*, 1999.

[235] T. Stützle and H.H. Hoos. Improving the Ant-System: A Detailed Report on the $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System. Technical Report AIDA–96–12, FG Intellektik, TH Darmstadt, August 1996.

[236] T. Stützle and H.H. Hoos. The $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System and Local Search for the Traveling Salesman Problem. In T. Baeck, Z. Michalewicz, and X. Yao, editors, *Proceedings of ICEC'97*, pages 309–314. IEEE Press, Piscataway, NJ, USA, 1997.

[237] T. Stützle and H.H. Hoos. Improvements on the Ant System: Introducing the $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System. In R.F. Albrecht G.D. Smith, N.C. Steele, editor, *Artificial Neural Networks and Genetic Algorithms*, pages 245–249. Springer Verlag, Wien, 1998.

[238] T. Stützle and H.H. Hoos. $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System and Local Search for Combinatorial Optimization Problems. In S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 137–154. Kluwer, Boston, 1999.

[239] J.Y. Suh and D.V. Gucht. Incorporating Heuristic Information into Genetic Search. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 100–107. Lawrence Erlbaum Associates, 1987.

[240] E. Taillard. Some Efficient Heuristic Methods for the Flow Shop Sequencing Problem. *European Journal of Operational Research*, 47:65–74, 1990.

[241] É.D. Taillard. Robust Taboo Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.

[242] É.D. Taillard. Benchmarks for Basic Scheduling Problems. *European Journal of Operational Research*, 64:278–285, 1993.

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

[244] É.D. Taillard. Comparison of Iterative Searches for the Quadratic Assignment Problem. *Location Science*, 3:87–105, 1995.

[245] E.D. Taillard, L.M. Gambardella, M. Gendreau, and J.-Y. Potvin. Adaptive Memory Programming: A Unified View of Metaheuristics. Technical Report IDSIA-19-98, IDSIA, Lugano, Switzerland, 1998.

[246] D.M. Tate and A.E. Smith. A Genetic Approach to the Quadratic Assignment Problem. *Computers & Operations Research*, 22(1):73–83, 1995.

[247] H.M.M. ten Eikelder, B.J.M. Aarts, M.G.A. Verhoeven, and E.H.L. Aarts. Sequential and Parallel Local Search Algorithms for Job Shop Scheduling. In S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer, Boston, 1999.

[248] T.Hogg, B.A. Huberman, and C.P. Williams. Phase Transitions and the Search Problem (Editorial). *Artificial Intelligence*, 81(1–2):1–16, 1996.

[249] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, UK, 1993.

[250] S. Turner and D. Booth. Comparison of Heuristics for Flow Shop Sequencing. *OMEGA*, 15(1):75–85, 1987.

[251] N.L.J. Ulder, E.H.L. Aarts, H.-J. Bandelt, P.J.M. van Laarhoven, and E. Pesch. Genetic Local Search Algorithms for the Travelling Salesman Problem. In H.-P. Schwefel and R. Männer, editors, *Proceedings 1st International Workshop on Parallel Problem Solving from Nature*, number 496 in LNCS, pages 109–116. Springer Verlag, 1991.

[252] R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. A Local Search Template (revised version). Technical Report Memorandum COSOR 92-11, Department of Mathematics and Computing Science, Eindhoven, 1995.

[253] R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. Job Shop Scheduling by Local Search. *INFORMS Journal on Computing*, 8:302–317, 1996.

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

[255] R. Wallace and E. Freuder. Heuristic Methods for Over-Constrained Constraint Satisfaction Problems. In M. Jampel, E. Freuder, and M. Maher, editors, *OCS'95: Workshop on Over-Constrained Systems at CP'95*, 1995.

[256] Richard J. Wallace. Analysis of Heuristic Methods for Partial Constraint Satisfaction Problems. In E. Freuder, editor, *Principles and Practice of Constraint Programming - CP'96*, volume 1118 of *LNCS*, pages 482–496. Springer Verlag, 1996.

[257] T. Walters. Repair and Brood Selection in the Traveling Salesman Problem. In A.E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Proc. of Parallel Problem Solving from Nature – PPSN V*, volume 1498 of *LNCS*, pages 813–822. Springer Verlag, 1998.

[258] C.J.C.H. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8:279–292, 1992.

[259] E.D. Weinberger. Correlated and Uncorrelated Fitness Landscapes and How to Tell the Difference. *Biological Cybernetics*, 63:325–336, 1990.

[260] F. Werner. On the Heuristic Solution of the Permutation Flow Shop Problem by Path Algorithms. *Computers & Operations Research*, 20(7):707–722, 1993.

[261] M. Widmer and A. Hertz. A New Heuristic Method for the Flow Shop Sequencing Problem. *European Journal of Operational Research*, 41:186–193, 1989.

[262] M. Yannakakis. The Analysis of Local Search Problems and their Heuristics. In *Proceedings STACS'90*, number 415 in LNCS, pages 298–310. Springer Verlag, 1990.

[263] M. Yokoo. Weak-commitment Search for Solving Constraint Satisfaction Problems. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 313–318. MIT press, 1994.

[264] C. Young, D.S. Johnson, D.R. Karger, and M.D. Smith. Near-optimal Intraprocedural Branch Alignment. In *Proceedings 1997 Symp. on Programming Languages, Design, and Implementation*, pages 183–193, 1997.

[265] N. Yugami, Y. Ohta, and H. Hara. Improving Repair-Based Constraint Satisfaction Methods by Value Propagation. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 344–349. MIT press, 1994.

Register