Otto-von-Guericke-University Magdeburg

Department of Computer Science

Studienarbeit

Determination of Parameter Settings of Ant Colony System for the Traveling Salesman Problem

Author:

Julia Preusse

28. July 2009

Academic Supervisors:

Prof. Saman Halgamuge, Kent C. B. Steer BE/BCS
University of Melbourne
Department of Mechanical Engineering
Parkville, 3052, Victoria, Australia

Prof. Dr. Rudolf Kruse, Dipl.-Inform. Georg Ruß
Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D-39016 Magdeburg, Germany

Abstract

Many computational problems are too complex to be practically solved to proven optimality. This is one of the main reasons why heuristics – algorithms using problem specific knowledge to compute near-optimal solutions – became very popular. Ant Colony Optimization (ACO) is one of the most successful heuristics for the Traveling Salesman Problem (TSP), which is to find a shortest tour through a given list of cities. Unfortunately we cannot use the full potential of ACO algorithms if we cannot determine good parameter settings. Commonly, algorithms use a standard parameter setting, which might be good for many problems. Nevertheless, there are problems for which better settings can be found. This is why we dedicated this work to the search for measures helping us to characterize TSP instances to derive appropriate parameter settings for ACO. We have developed new adaptive settings which have been shown experimentally to be better than the standard version.

Contents

Chapter 1

Introduction

This chapter gives a short introduction to this work, its motivation, the problems we encountered and how these can be tackled.

1.1 Purpose of this Document

The purpose of this document is to give my supervisors a detailed report about what I have done during my six months of internship at the Department of Mechanical and Manufacturing Engineering at the University of Melbourne, Australia. Furthermore, writing this document is mandatory within the conditions of my studies of computer science. Finally, in this document I will describe what I have done in my internship and how I finished my work afterwards.

1.2 Motivation

Since many problems are too complex to solve them to optimality in a reasonable amount of time, finding algorithms that use problem-specific knowledge to achieve a good albeit not optimal solution in an appropriate amount of time, is an important issue.

One of the most-studied problems is the Traveling Salesman Problem (TSP), defines as finding a shortest closed tour through a given list of cities and. For example the problem of finding a shortest closed tour through all 24,978 cities of Sweden was solved in May 2004[1] , but needed more than one year to be computed and proven. One can easily see that this is not a practically usable approach, as the computation time exceeds the time constraints for most problems. We found Ant Colony Optimization (ACO) algorithms to be a promising approach to obtaining good results for the TSP in an appropriate amount of time. Ant Colony Optimization is motivated by the foraging

[1] `http://www.tsp.gatech.edu` (Version from 16th July 2009)

behavior of ants, which enable them to determine the shortest path from the nest to a food source. We notice four major challenges:

Guaranteed optimal solution Even if the two most popular versions of Ant Colony Optimization, $\mathcal{MAX} - \mathcal{MIN}$ Ant System and Ant Colony System, are proven to converge to the optimum [7], the time for this convergence process is not polynomial. Thus in practice, running an algorithm until it converges is not possible, but we want to guarantee to have found a good solution.

Understanding parameter choice A full understanding of the effects of different parameter settings has not yet been developed. This is due to the fact that the parameters interact with each other. So parameter setting is a very complex task.

Problem dependence Different parameter settings were experimentally shown to be good for different problems. So approaches which have proven to yield good results for the Traveling Salesman Problem may not be appropriate for scheduling algorithms [27].

Finding good parameters The problem of finding good parameters for an Ant Colony Optimization algorithm for a problem is usually as difficult as to solve the problem itself. Due to the fact that no full understanding of the parameter choice has yet been developed and the parameters to be chosen are domain-dependent, finding a general approach to determine optimal parameters is a very difficult problem and no breakthrough has yet been made.

We conclude with some tasks and how we tackle them shortly below.

1.3 The Tasks

1. **Developing new approaches:** Inspired by our investigation of literature, we aim to combine approaches already made to develop new dynamic parameter control approaches. We will use the stagnation behavior as the algorithm's feedback. Furthermore, we propose three adaptive approaches that depend on how long no better solution has been found. In order to classify problems, we will build up a collection of relevant attributes, which are based on ideas from statistics, as well as on measures proposed by other researchers.

2. **Conducting experiments:** To validate our proposals, we run experiments with the Ant Colony System. We test all different combinations of the parameter settings we developed before and we determine the values for the respective problem measurements.

3. **Conclusions:** Finally, we will draw conclusions from the results of our simulations. We will do so by analyzing the experimental outcomes of the different measures one by one, finally judging the usefulness of each. We are going to state three new parameter strategies which perform better than the standard settings used and known so far.

1.4 Document Structure

The remainder of this document is organized as follows:

Chapter 2 will provide the knowledge needed to understand the remainder of this thesis. In particular it will introduce one of the most popular combinatorial problems: the Traveling Salesman Problem. We will continue with the concepts of metaheuristics and swarm intelligence. Besides Particle Swarm Optimization, the concept of Ant Colony Optimization will be introduced. We describe how ants are able to determine the shortest path for foraging and explain how this ability is translated into an algorithm. Finally, we will explain different parameter setting strategies.

Chapter 3 addresses Ant Colony Optimization and the three best-known variants. While Ant System was the first Ant Colony Optimization algorithm designed, $\mathcal{MAX} - \mathcal{MIN}$ Ant System and Ant Colony System are improvements which have been experimentally shown to find solutions better than Ant System. As this thesis focuses on Ant Colony System, we explain the parameters of this algorithm in detail and give an overview of strategies for parameter setting. We conclude with improvements of Ant Colony Optimization algorithms in general. Three versions of local search are presented, as well as the often-used speed-up techniques named candidate list and don't-look-at bits.

Chapter 4 presents the design of our testbed and the design choices we have made for parameter settings. We present three new dynamic parameter control strategies for Ant Colony System for the Traveling Salesman Problem and give some graph measures to analyze the chosen instances. Finally, we will explain the setup of the tests.

Chapter 5 presents the results of our tests and conclusions we draw. We evaluate the graph measurement we have done, then we analyze good and bad combinations.

Chapter 6 summaries what we have achieved and shows possible further investigations.

Our starting point was to conduct a literature review on Ant Colony Optimization algorithms and parameter strategies, which will be presented in the following chapter.

Chapter 2

Basics

This basic chapter will provide the knowledge needed to understand the remainder of this thesis. In particular it will introduce the following concepts:

- Traveling Salesman Problem
- Metaheuristics
- Swarm intelligence
- Ant Colony Optimization
- Parameter strategies

2.1 Traveling Salesman Problem (TSP)

The TSP is defined by the following optimization program via an undirected graph:

Given a graph $G = (V, E)$ with weights c_e $(\forall e \in E)$, $\delta(M)$ denotes for a set of nodes M those edges that are incident to nodes in M.

- Definitions:
  $x_e = 1 \quad \leftrightarrow \quad e \in \text{TSP tour} \quad \text{with} \quad x_e \in \{0,1\}^E$
- Objective function:
  $\min \{\sum_{e \in E} c_e x_e\}$
- Constraints:
  - Each node is entered and left once:
    $\sum_{e \in \delta(v)} x_e = 2 \quad \forall\, v \in V$
  - Subtour elimination constraints (no disjunct cycles):
    $\sum_{e \in \delta(S)} x_e \geq 2 \quad \forall\, \emptyset \neq S \subsetneq V$

So the Traveling Salesman Problem involves finding a minimal-weighted tour[1] containing all cities of a given set. It is common to define the weight of an edge $\{i, j\}$ (or respectively a directed edge (i, j)) over the distance from vertex i to vertex j. We will abbreviate the edge $\{i, j\}$ as ij from now on. The three main categories of Traveling Salesman Problems that we are interested in, are introduced below. We consider *complete* (di-)graphs, as each pair of different nodes is connected by an edge (respectively an arc).

Asymmetric and Symmetric Traveling Salesman Problem In the *asymmetric* Traveling Salesman Problem, the distance for traveling from one city to another need not be the same as in the other direction. Therefore, an asymmetric TSP instance can be represented by a directed complete graph, which is a set of vertices and arcs as shown in Figure 2.1a.

For the *symmetric* TSP, the distance between two cities is the same in each direction. Thus, a symmetric Traveling Salesman Problem can be pictured as an undirected complete graph, as it is done in Figure 2.1b.

Euclidean Traveling Salesman Problem In the *Euclidean* Traveling Salesman Problem, each city is represented by a point in the Euclidean plane and the distance between two cities is defined as the Euclidean distance between their coordinates. An example graph is shown in Figure 2.1c.

2.2 Metaheuristics

*"***Heuristic:** *A 'rule of thumb', based on domain knowledge from a particular application, that gives guidance in the solution of a problem. Unlike algorithms, heuristics cannot have proven performance bounds owing to their open-ended dependence on specific application knowledge; [...] Heuristics may thus be very valuable most of the time but their results or performance cannot be guaranteed."*
(A Dictionary of Computing, Oxford University Press 2004)

Speaking loosely, a *heuristic* is an algorithm using problem-specific knowledge to create or improve solutions. Heuristics are often used for the computation of solutions to NP complete problems, as optimal solutions from this problem class are not expected to be found in a suitable amount of time. A *metaheuristic* is a set of heuristic concepts that comprises heuristics which are applicable to more than one problem domain. A particularly interesting and successful metaheuristic is Ant Colony Optimization, which we will discuss in detail in this thesis.

[1] a tour is visiting each city exactly once and then returning to the starting city

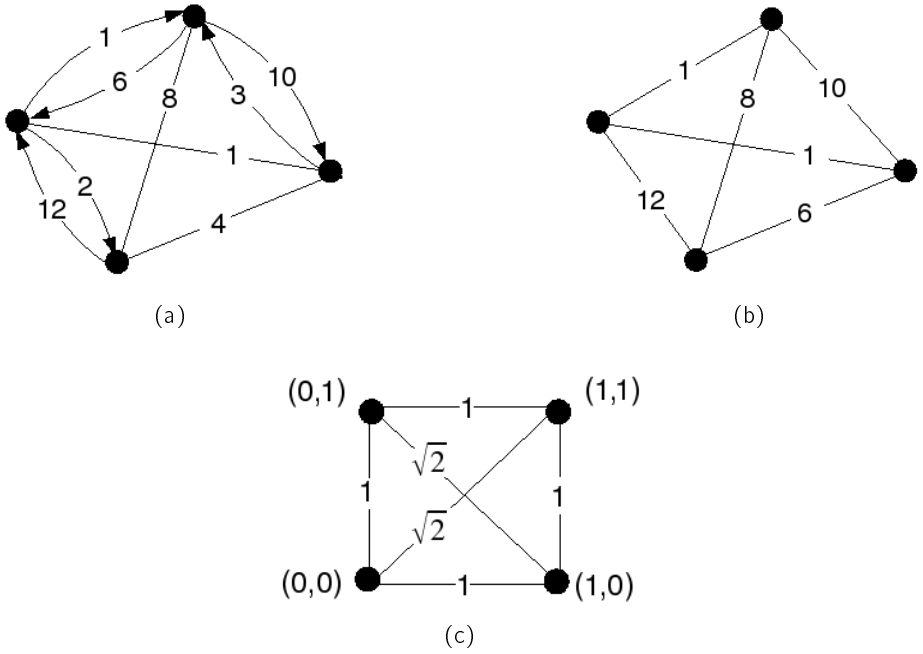


Figure 2.1: (a) asymmetric, (b) symmetric and (c) Euclidean TSP

2.3 Swarm Intelligence

"Swarm intelligence is a property of systems of unintelligent agents of limited individual capabilities exhibiting collectively intelligent behavior"
(White and Pagurek: Towards multi-swarm problem solving in networks p.333, 1998.)

Swarms of bees, an ant colony, a flock of birds, cars being the agents in a swarm of cars called traffic and many more examples could be listed to illustrate the variety of swarms.

The concept of Swarm Intelligence can be described as following: Simple skills of individuals are needed to solve complex problems as a union via rule-based actions. Each individual interacts (directly or indirectly) with others based on simple local rules without being aware of the swarm behavior. The intelligent behavior can be observed on a global level.

At first, we will explain the difference between Swarm Intelligence and Collective Intelligence. Since there are plenty of Swarm Intelligence flavors, we will give a general overview of the two most popular below.

Distinction from Collective Intelligence

Even if Collective Intelligence and Swarm Intelligence are sometimes used synonymously, there is a big difference between both of them [18]. Collective intelligence is a shared or group intelligence that emerges from the collaboration and competition of many individuals. While the single individual in a swarm is rather simple, compared to the task it has to fulfill, each part of the Collective Intelligence might be very complex. Wikipedia is a prominent example of the latter, since each author is a complex intelligence and its contribution to the collective is not replaceable. However, humans do also appear in forms of Swarm Intelligence, e.g. in a Mexican wave. In example, the reaction of each part of this swarm can be described as a simple reaction to its predecessor's action.

Ant Colony Optimization (ACO)

We now introduce the main principles of Ant Colony Optimization. We use ACO for the Traveling Salesman Problem, since Swarm Intelligence and especially Ant Colony Optimization are very promising concepts for combinatorial problems and have shown to perform well on the TSP [7].

Foraging behavior of Biological Ants

The *double bridge experiment* [3] on the foraging behavior of the Argentine ant *Iridomyrmex humilis* explored the ability of these ants to find the shortest path from the nest to a food source (see Figures 2.2 – 2.5).

When following a path, ants lay a trail of a chemical substance called pheromone. The probability of choosing a path depends on the amount of pheromone on the path and some randomness allowing the ants to change their path in order to explore the environment. Pheromone evaporates over time, so long paths have problems to maintain stable pheromone levels.

The starting point of the experiment was to build up a double bridge from the ants' nest to the food source. At the beginning, the ants were freed and started to walk along the paths.

If an ant has to decide between two unseen paths to get to the food, it first chooses randomly which one to take (see Figure 2.2b). As the ant taking the shorter path arrives earlier at the food source (see Figure 2.3a). For the way back there already is pheromone on the path, so it will very likely take the same branch back, as can be seen in Figure 2.3b. If this continues, the amount of pheromone on the shortest branch is increasing and will lead to most of the ants taking the shorter path.

(a) (b)

Figure 2.2: The double bridge experiment is set up. (a) The two branches have different length. (b) At the beginning of the experiment there is no pheromone on the trails, so the ants choose randomly which branch to take. Therefore about half of the ants take the short branch and the others choose to take the longer one.

From Real to Artificial Ants

Inspired by this mechanism, Dorigo *et al.* translated the approach to the computer by using artificial ants laying and following artificial pheromone [7]. In order to adapt these principles to the computer, the main elements of an Ant Colony Optimization (ACO) algorithm according to [5] are:

- A colony of ants that will be used to build a solution in the graph.
- A probabilistic transition rule responsible for determining the next edge of the graph to which an ant will move.
- A heuristic desirability that will influence the probability of an ant moving to a given edge.
- The pheromone level of each edge, which indicates how *good* it is.

Applications

In this thesis, we will examine ACO's application to the Traveling Salesman Problem, but there are also other combinatorial problems using it. Ant Colony Optimization is used for routing problems, clustering algorithms, machine learning, assignment problems and many more [7] (Another fascinating aspect of ants' life, namely their brood clustering, is used for some clustering techniques [1, 15]).

Particle Swarm Optimization (PSO)

The social behavior of flocking birds or fish shoals inspired Eberhart and Kennedy in 1995 to introduce the Particle Swarm Optimization (PSO) algorithm as a method for optimization of continuous nonlinear functions [14].

(a) (b)

Figure 2.3: (a) By using a trail, ants depose pheromone on it. (b) The ant that has taken the shorter path arrives earlier at the food source. On its way back to the nest, it has to make a choice which is now biased towards the amount of pheromone on the branches. Very likely the ant returns via the same branch again, as there is already some pheromone on it.

(a) (b)

Figure 2.4: (a) Finally the second ant arrives at the food source and (b) makes its decision for the return paths biased towards the pheromone: one pheromone unit on the longer branch and two units on the shorter branch. So it is likely that the second ant takes the shortest path, as well.

(a) (b)

Figure 2.5: (a) Via the ants' biased choice the amount of pheromone on the shortest paths is rising steadily. (b) After a certain time most ants take the shortest paths; even though the longer branch is also taken by a small number of ants.

The problem space in which an optimum should be found is initialized with a random population of potential solutions, called particles, that are represented via their coordinates in the space. In PSO the experience of each particle

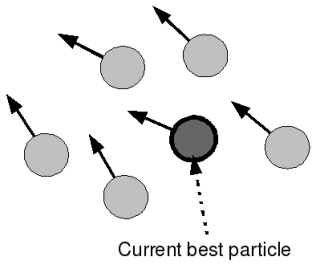


Figure 2.6: Simple illustration of a particle swarm

(its position) plays an important role. Individuals learn from their own past and from the past experiences of others. In order to profit from the swarms' intelligence each particle compares its own position with
a) its neighbors or
b) all other particles
and imitates only those that are superior according to a fitness function. At each time step every particle changes the velocity towards its best performing neighbor or towards the global best performing particle. This movement is weighted by a random term which is generated separately for the best particle of a) and b).

Particle Swarm Optimization was shown to be effective by [17] and has often been applied in research as well as application areas due to its speed and the few parameters to choose. We refer the reader to [17] for a more detailed discussion.

2.4 On Optimal Parameters

Importance of Parameter Setting Strategies

Despite the usefulness of metaheuristics, some difficulties can arise. It is not unusual for a metaheuristic to have ten or more parameters, which need to be tuned or adapted at runtime. Unfortunately the space of possible parameter constructions might not be much smaller than the problem space. This problem arises, since the parameters are not independent of each other so they might influence other parameters. The problem that comes along with this is called the *parameter tuning problem* [23]: The difficulty might not be running the actual program, but the choice of the right parameter values.

Parameter Control versus Parameter Tuning

Since we want to find good parameter combinations, we need to understand the different possibilities to change and adapt parameters. We will explain the basic principles below. Eiben *et al.* divide parameter setting strategies into *parameter control* and *parameter tuning* [10]. The aim of parameter tuning is to find parameters before the run of the program that remain fixed at runtime. Eiben *et al.* give reasons why this method is inappropriate:

- Parameters are not independent, but trying all different combinations systematically is practically impossible.
- The process of parameter tuning is time-consuming, even if parameters are optimized one by one, regardless of their interactions.
- For a given problem the selected parameter values are not necessarily optimal, even if the effort made for setting them was significant.

Therefore the concept of parameter control as a feasible alternative is presented in the following.

Parameter Control

Parameter control is done *during* the run of the program [10]. Furthermore it is divided into:

- *Static* parameter control: The parameter values are varied according to deterministic rules without using any feedback from the program run.
- *Adaptive* parameter control: This parameter control strategy uses feedback from the program to receive a direction for the change of the parameter values to set.
- *Self-adaptive* parameter control: The idea of this strategy is an *evolution of evolution*: The parameters are encoded in chromosomes and undergo an evolutionary process.

In this work we aim at finding appropriate parameter control strategies for Ant Colony Optimization for the Traveling Salesman Problem. So far, just a few adaptive parameter control approaches have been made for ACO, so the standard parameter setting remained a static one. Self-adaptive approaches for this metaheuristic comprise using other metaheuristics or genetic algorithms together with ACO. This makes it even harder to determine good parameter settings, since parameters for two algorithms need to be chosen.

Hence, we focus on adaptive parameter control strategies for some ACS parameters and present them in Section 4.1.

Chapter 3

Ant Colony Optimization for the Traveling Salesman Problem

We have already explained how an ant is able to find the shortest path. This chapter will show how this simple mechanism is used to solve the more complex task of finding a resonable solution to the Traveling Salesman Problem.

The Setup

The setup is the following:
A group of m artificial ants are moving from city to city along the TSP graph. They decide to which city to go next based on:

- The set of cities they have not visited so far on their tour
- The amount of pheromone on the edges between cities
- A probabilistic function of the weight of an edge which enables the artificial ants to determine the closest-by cities

Thus, ants prefer close cities connected by edges with a high level of pheromone. At the beginning of the algorithm, m artificial ants are placed on randomly selected cities. By choosing a path, the artificial ants leave artificial pheromone on it.

After all ants have finished one tour each, the edges of the shortest tour receive some additional pheromone which is inversely proportional to the tour length. It is appropriate to use the inverse of the distances, since nearby cities should be weighted higher than far away ones. In the following sections, we present the three basic ACO versions.

3.1 Ant System (AS)

The Ant System is the first Ant Colony Optimizationalgorithm and has been developed by Dorigo *et al.* [7, 8].

Initialization Phase

At first, all trails are initialized with an amount of pheromone τ_0, chosen as $\tau_0 = \frac{n}{C^{nn}}$, where n is the number of cities and C^{nn} is the tour length computed with the nearest neighbor heuristic [1].

We have to be careful with the choice of τ_0 (see Section 3.4 for an explanation). As proposed in [7], the number of ants is set to the number of cities. The m artificial ants are placed on random cities at the beginning of the algorithm and start their tour constructions.

Tour Constructions

Each ant decides, being in city i, whether to go to city j next according to the following probabilistic rule

$$p_{ij}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{x \in N_i} [\tau_{ix}(t)]^{\alpha} \cdot [\eta_{ix}]^{\beta}} & \text{, if j has not been visited yet} \\ 0 & \text{, otherwise} \end{cases} \tag{3.1}$$

where $\tau_{ij}(t)$ is the amount of pheromone on edge ij at time t, η_{ij} is a heuristic function chosen as the inverse of the weight w_{ij}, N_i is the feasible set of neighbors of city i, α and β weight the importance of the pheromone information relative to the heuristic function.

After each ant finished a tour, the pheromone is updated *globally*

$$\tau_{ij}(t+1) \leftarrow (1-\rho) \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t),$$

where ρ is the pheromone evaporation rate, which is bounded between 0 and 1 and $\sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t)$ is the amount of pheromone that all ants deposit on ij, which is calculated by

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} Q/L^{k}(t) & \text{, if } ij \text{ belongs to the tour that ant } k \text{ did at iteration } t \\ 0 & \text{, otherwise} \end{cases}$$

where Q is another user defined parameter and $L^{k}(t)$ the length of the tour that ant k built at iteration t.

[1] This heuristic starts at a city and goes on to the nearest unvisited city until all cities have been visited.

3.2 $\mathcal{MAX} - \mathcal{MIN}$ Ant System

Stützle and Hoos proposed the $\mathcal{MAX} - \mathcal{MIN}$ Ant System as an improvement [25] of the AS. It differs from the AS in four key points:

1. Only the *best* ant updates the pheromone *globally*.
2. The pheromone on the trails is bounded by $[\tau_{min}, \tau_{max}]$.
3. The initial amount of pheromone is set to the upper bound.
4. After a certain number of stagnating iterations, the trails are reinitialized.

Initialization Phase

First, all trails are initialized with τ_0 chosen as $\tau_0 = \tau_{max}$, which is the upper pheromone trail limit (other initialization strategies are discussed in [25]). Together with a low pheromone evaporation rate this causes a slow increase in the difference of pheromone trails and therefore a high exploration of tours at the beginning of the algorithm. For more details on the general requirements for a good setting of τ_0, we refer to Section 3.4. The number of ants is equal to the number of cities [7]. Parameters τ_{min} and τ_{max} were introduced to avoid stagnation since the fixed range of pheromone also alters the probabilities of being chosen, for all edges. Note that both parameters need not remain constant throughout the tour constructions (For further information on how to set these two parameters, we refer to [25]). All ants are placed on random cities at the beginning of the algorithm and start their tour constructions.

Tour Constructions

Ant k decides whether to go from city i to city j also according to the Equation 3.1 from Ant System. The pheromone τ_{ij} is updated *globally* after each iteration by:

$$\tau_{ij}(t+1) \leftarrow \left[(1-p) \cdot \tau_{ij}(t) + \Delta\tau_{ij}^{best}\right]_{\tau_{min}}^{\tau_{\max}}$$

where τ_{min} is the lower pheromone bound, $\tau_{\max}$ is the upper pheromone bound, p is the evaporation rate and $\Delta\tau_{ij}^{best}$ is defined as

$$\Delta\tau_{ij}^{best}(t) = \begin{cases} 1/L^{best}(t) & \text{, if } ij \text{ belongs to } best \text{ tour} \\ 0 & \text{, otherwise.} \end{cases}$$

Depending on the design choice, *best* can either be the *best-so-far* tour or the best tour at iteration t (*iteration-best*).

Function $[x]_a^b$ is defined by

$$[x]_a^b = \begin{cases} b & \text{, if } x > b \\ a & \text{, if } x < a \\ x & \text{, otherwise.} \end{cases}$$

If the quality of the constructed tours stagnates for a certain number of iterations, the trails are reinitialized. This is due to the fact that there are some paths with only a small probability of being chosen, so the tour construction is too biased.

3.3 Ant Colony System (ACS)

Ant Colony System is another modification of the Ant System proposed by Dorigo *et al.* [6]. It differs from the AS in of three main aspects:

1. A local update is done after each traversed edge.
2. The global updating rule is only applied to edges belonging to the *best* tour.
3. The state transition rule is introduced and allows to balance between exploring the search space and exploiting locally good solutions.

We will discuss these changes in the following in more detail.

Initialization Phase

The initialization phase is nearly the same as the one we presented for the AS algorithm, except for the number of ants that are used to construct tours. Dorigo *et al.* propose to set this number to *ten*, as it was experimentally shown to yield good results [7].

Tour Constructions

ACS uses the *pseudo-random proportional* rule: The probability of ant k moving from city i to city j depends on the random variable q, which is uniformly distributed over $[0, 1]$ and a parameter $q_0 \in [0, 1]$

$$p_{ij}(t) = \begin{cases} \max_{x \in N_i} \tau_{ij}(t)[\eta_{ij}]^\beta & \text{, if } q \leq q_0 \\ (3.3) & \text{, otherwise} \end{cases} \tag{3.2}$$

with (3.3)

$$p_{ij}(t) = \begin{cases} \frac{[\tau_{ij}(t)] \cdot [\eta_{ij}]^\beta}{\sum_{x \in N_i} [\tau_{ix}(t)] \cdot [\eta_{ix}]^\beta} & \text{, if } j \text{ has not been visited yet} \\ 0 & \text{, otherwise} \end{cases} \tag{3.3}$$

where $\tau_{ij}(t)$ is the amount of pheromone on edge ij at time t, η_{ij} is a heuristic function chosen as the inverse of the weight w_{ij}, N_i is the feasible set of neighbors of city i and β weights the importance of the heuristic information. Hence with probability q_0, ant k does the next move according to the best possible edge, thus it exploits the learned knowledge. With probability $(1-q_0)$ the ant explores the search space with a bias.

Another difference is that Ant Colony System introduces a *local* pheromone updating rule. After each construction step, every ant changes the pheromone on the last edge traversed in the following way

$$\tau_{ij}(t) \leftarrow (1-\rho_{local}) \cdot \tau_{ij}(t) + \rho_{local} \cdot \tau_0$$

where $0 < \rho_{local} \leq 1$ is the local decay rate and τ_0 is the initial pheromone level on edge ij.

With this decay process it is easier to leave a *local* maximum, because ants are less likely to follow those edges which others have passed before. This enables ants to choose different edges and hence a larger diversity of possible solutions is reached.

The global update is computed differently in the following way

$$\tau_{ij}(t+1) = \begin{cases} (1-\rho) \cdot \tau_{ij}(t) + \rho \cdot \Delta\tau_{ij}(t) & \text{, if } ij \text{ belongs to } \textit{best} \text{ tour} \\ \tau_{ij}(t) & \text{, otherwise} \end{cases}$$

where *best* tour is again either the best tour found so far or the best tour at iteration t and $\Delta\tau_{ij}(t) = \frac{1}{L^{best}}$.

Hence, the global pheromone update is applied only to those edges belonging to the *best* tour found and can be described as a weighted average between the old pheromone value and the pheromone deposit.

$\mathcal{MAX} - \mathcal{MIN}$ Ant System versus Ant Colony System

We focus on one particular ACO version to determine good parameter settings, so the choice between $\mathcal{MAX} - \mathcal{MIN}$ Ant System and Ant Colony System has to be made. Both algorithms yield good results in many tests and are considered as the best-performing ACO variants [7]. Neither one seems to be truly superior over the other. We choose ACS, since there are less parameters to tune (α is fixed to 1 and we do neither need to consider τ_{min} nor τ_{max} and only get ρ_{local} as a new parameter).

3.4 Description of Parameters for Ant Colony System

Since we use Ant Colony System for our simulations, we will give a short overview of the effects of the choice of parameters. In this subsection we will

discuss which settings appear to be good, but the explicit design choices will be explained in Section 4.1.

β : controls the relative weight of the heuristic information or respectively the ratio between heuristic information and pheromone information as α is set to 1 (α does not appear in Equation 3.2, since it is set to 1). If β increases, so does the ratio and thus the difference between all edges increases. This means that, the higher β becomes, the greater the exploitation is and when β is decreases, the exploration is increasing.

We range $\beta \in (0, 5]$. This is a contrast to most proposals in literature [7], where β was ranged between 2 and 5. It is logical to make β close to 0, since this enables the maximal exploration. This is because all distances get close to 1 and therefore the difference between all edges is small, giving even more distant cities a chance to be chosen.

ρ : controls the *global* pheromone evaporation. The value of ρ is naturally bounded between 0 and 1. Dorigo *et al.* [7] propose ρ to be 0.1 for ACS, but we fix ρ in a range between 0.1 and 0.5. If ρ increases, the exploration of the search space is increased. This is due to the fact that only the pheromone on the edges belonging to the best tour is updated. If we increase ρ, there will be less pheromone on these edges, hence the difference to the less desirable edges is diminished, giving the latter a better chance to be chosen.

ρ_{local} : controls the *local* pheromone update and therefore the diversity of different paths. The bigger ρ_{local} is, the less likely it is that ants follow others. Therefore a high value of ρ_{local} forces a high diversity of paths. If q_0 is chosen too large, we would destroy the effect of the ants' pheromone communication.

Since Dorigo *et al.* found 0.1 to be a good value for ρ_{local} and even fixed the parameter in the algorithm to this value, we do not put more research into this parameter and focus on the others instead.

q_0 : defines the percentage of exploration versus exploitation. This is due to the fact that ants perform a biased search with probability q_0 and go for the best next city with probability $(1-q_0)$ (see Equation 3.2). Therefore, the larger q_0 is, the more ants construct their tours through exploitation of knowledge. In contrast, if q_0 is close to zero, the search is rather focused on exploration. Dorigo *et al.* [7] propose a value of 0.9 with obviously a strong focus on those paths with a high probability of being chosen. Especially, if a better solution was not found for a long time, a strong difference between the respective edges should be present. If we continued with a greedy strategy, it would be unlikely to find a better solution. Hence, we range q_0 between 0.5 and 1.0, decreasing q_0 if stagnation occures.

τ_0 : controls the amount of pheromone that the trails are initialized with. If τ_0 is chosen too small, the tour construction is biased towards the very first tour constructions. On the other hand, if it is chosen too large, it takes too many iterations until the pheromone evaporates. This influences when the ants can start using their biased search.

Dorigo *et al.* propose a value of $\frac{1}{nC^{nn}}$ for τ_0, where n is the number of cities and C^{nn} is the length generated by the nearest-neighbor heuristic. That is, to start at a city and to go on to the nearest unvisited city, repeating this step for new cities, until all cities have been visited.

m : determines the number of ants and thus controls the amount of exploration. Obviously the more ants we use, the slower the algorithm gets. On the other hand, if we would not take enough ants, it would take too long to approach good solutions as there are not enough ants to communicate with each other. While for Ant System and $\mathcal{MAX} - \mathcal{MIN}$ Ant System, empirical studies conclude that the number of ants should equal the number of cities, for Ant Colony System a fixed number of ten ants seemed to yield good results. We do not vary the number of ants during our experiments; m is set to ten.

max_it : controls the maximal number of iterations in which all ants build in each case a tour. Obviously, the more iterations we use, the higher the costs will be. If we stop the algorithm too early, we might miss a better solution. Some studies on this subject (e.g. [25]) suggest to choose a strategy where *max_it* only depends on the number of cities.

3.5 Improvements

After the first successes of $\mathcal{MAX} - \mathcal{MIN}$ Ant System and Ant Colony System, researchers put a lot of effort into improving performance of both algorithms. Some of the new ideas that were mentioned in [16] are introduced below.

3.5.1 Candidate List

A candidate list restricts the number of possible choices that need to be considered at each construction step. In the TSP's case a candidate list of city i contains a fixed number nn of cities that have the smallest distance to city i. These cities are called nearest neighbors. In order to determine them, all cities are sorted by their distance to city i in nondecreasing order and the first nn cities are inserted in city i's candidate list. For the tour construction rule (Equation 3.2 or respectively Equation 3.1), the biased search is performed only for cities in the candidate list. If all cities from this list have already

been visited, one of the remaining cities is chosen. Experimental studies have shown that this method results in an improvement of solution quality and a significant performance gain [13]. Most Ant Colony Optimization algorithms use a candidate list of size 20 [7].

3.5.2 Local Search

Local search is used to improve a solution found by looking in its neighborhood for better ones. In the case of the TSP, this means involved by improving a given tour with small local changes. Having found a tour, the order of the cities is slightly changed into another valid tour by applying one of the following three methods.

2-OPT Local Search The simplest way to change a TSP tour is to exchange two edges. To realize this, two non-adjacent[2] edges, call them *ab* and *cd*, are chosen (Note that *ab* means that city *a* is visited before city *b* on the tour). These edges are deleted and new edges *ac* and *bd* are inserted and hence form a new TSP tour (see Figure 3.1). It is important to keep track of the way the cities are reordered since new edges *ad* and *bc* would lead to two disjunct cycles, neither one containing all cities.

2-OPT local search is considering all possible 2-exchanges and computes the best one of those.

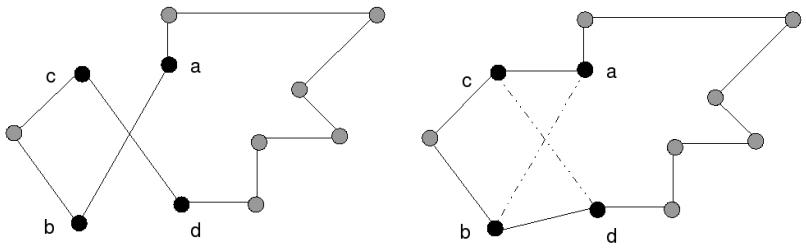


Figure 3.1: This figure illustrates a 2-OPT local search. Edges *ab* and *cd* from the graph illustrated in (a) were deleted and the tour was completed again by inserting edges *ac* and *bd*, thus resulting in the valid tour shown in the graph in (b).

3-OPT Local Search 3-OPT local search considers all exchanges of 3 edges in the tour and finally computes the best one. Three non-adjacent edges *ab*, *cd* and *ef* are chosen and deleted. To result in a valid TSP tour again, there

[2] Two edges *ab* and *cd* are non-adjacent if the endpoints a, b, c, d are pairwise disjoint

are eight possible ways to reconnect the graph correctly. In Figure 3.2 we have taken the best one, in order to reflect the working principle of the 3-OPT local search.

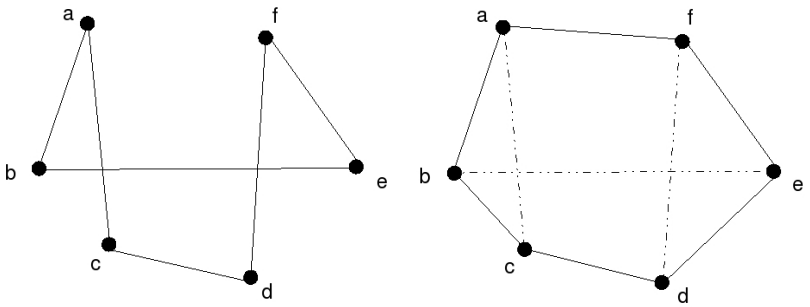


Figure 3.2: This figure illustrates a 3-OPT local search. Edges fd, be and ca were deleted from the graph shown in (a) and the tour was completed again by the insertion of edges bc, de and fa resulting in the valid tour presented in (b).

2.5-OPT Local Search Even if the 3-OPT local search yields better solutions, the complexity of doing such a search is $O(n^3)$ and therefore higher than to do a 2-OPT with complexity $O(n^2)$. The 2.5-OPT local search is more effective than 2-OPT local search but has also complexity $O(n^2)$. It uses the 2-OPT and improves the constructed tour slightly, but does not consider as many possible exchanges as 3-OPT. This is done through trying to insert a city between two nodes, thus changing the edges accordingly. To illustrate the way 2.5-OPT works, we present Figure 3.3.

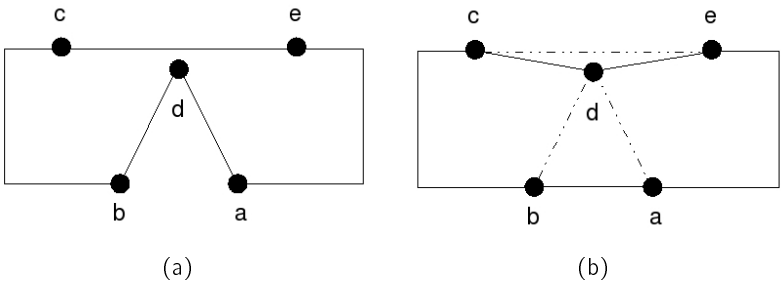


Figure 3.3: This figure gives an example of a 2.5-OPT local search. City d was ordered between city c and e, resulting in a shorter tour.

3.5.3 Don't-look-at bits

In 1992, Bentley introduced the concept of *don´t-look-at bits* [2]. The reason is the following: If we want to try out all possible combinations of 2-OPT, 3-OPT or 2.5-OPT moves after each construction step, the algorithm will be a lot slower. Hence, Bentley proposed considering only those exchange moves which may cause an improvement. Starting at city i, if we fail to find a better move in i's neighborhood it does not make sense to try to find an improving tour starting at i again, since the neighbors have not changed. We make use of this observation by introducing a special flag for each city. At the beginning, the flag is turned off for all cities and is turned on for city i iff a search for an improving move starting at i failed. The bit for i is only turned off again, if an improving move involving i [3], is found. This concept is included in the local search approach by regarding only those cities as a starting point for local search, whose don't-look-at bits are turned off.

[3] that is if i is an endpoint of one of the exchanged edges

Chapter 4

On Optimal Parameters for the Ant Colony System for the Traveling Salesman Problem

Dorigo *et al.* provide a freely available ANSI C implementation of some Ant Colony Optimization versions for the Traveling Salesman Problem[1], which serves as a foundation for our modifications. For the purpose of simplification we restrict our work to symmetric TSP instances, while it can easily be extended to incorporate asymmetric TSP instances. Furthermore, we focus on one particular ACO version, so the choice between $\mathcal{MAX} - \mathcal{MIN}$ Ant System and Ant Colony System had to be made and was carried out in favor of ACS, as already explained in Section 3.3.

The aim of this work is to compare different instances in order to find some structural properties that allow us to determine good parameter settings. Before we can continue with the results, we will discuss the design of the algorithm and the choices we have made.

4.1 Design of the Algorithm

In this section, we discuss the parameter settings that were necessary for running the algorithm. Some of the basic design choices for Ant Colony System's parameters were already discussed in Section 3.4; now we want to go into more detail. As already reasoned in Section 2.4, we want to develop and test new *adaptive* parameter control methods, which we introduce for some parameters in this section.

[1] available at `http://www.aco-metaheuristic.org/downloads/ACOTSP.V1.0.tar.gz`

4.1.1 Trial

A trial is one run of the algorithm, which depends on the time we assign to the ants to build their solutions.

Number of Trials In the given framework the number of trials was set to a default value of ten. After ten trials, the mean of the best solutions found in each trial and the mean of the time needed for it is computed, as well as the standard deviation for both values. We decided to implement it that way since it is easier to deal with the randomness in Ant Colony System and get a feasible view on its performance.

Time for each Trial versus Number of Tour Constructions The question is whether the stopping criterion for the ants' tour construction should be the time or the number of tours which have to be constructed. As we run these experiments to look into some comparisons among completely different instances, time seems to be an inadequate measurement. In [25] it is proposed to perform the comparison based on the same number of tour constructions for all algorithms: This number is chosen as $n \cdot 1000$, where n is the number of cities of an instance.

After running some tests with these values, we observe a long period of stagnation at the end of each trial, where no better solution is found. Therefore, we reduce the number of tour constructions and experimentally found $n \cdot 100$ to be a good trade-off between enabling the algorithm to find better solutions and avoiding excessive stagnation.

Adaptive β

As described in Section 3.4, the higher β is, the more exploitative the algorithm gets. If β is getting smaller, the level of exploration is increasing. We are inspired by [11] to vary β adaptively with the condition of flattening on the stagnation of the algorithm. To make use of this approach, we needed to find a measure to find a way of stagnating. We do so by counting during how many iterations no *better*[2] solution is found (*iterationsWithoutImprovement*). We range $\beta \in (\beta_{min}, \beta_{max}] = (0, 5]$ and link it to the iterations without improvement by

$$\beta = \beta_{min} + \frac{(\beta_{max} - \beta_{min})}{iterationsWithoutImprovement} .$$

Whenever a new best solution is found, *iterationsWithoutImprovement* is set to 1 and therefore $\beta = \beta_{max}$. This is exactly what we want since the algorithm can afford to be greedy at this point and might even try to further exploit the

[2] where *better* is related to the *best-so-far* solution

solution found. The more the algorithm stagnates, the closer β gets to β_{min}, thus ants should follow more diverse paths.

To converge slower to zero, we take the square root of the iterations without improvement. Thus the resulting formula is

$$\beta = \beta_{min} + \frac{(\beta_{max} - \beta_{min})}{\sqrt{iterationsWithoutImprovement}} .$$

Dynamic Pheromone Evaporate Rate

Analogously to parameter β, we link the pheromone evaporation rate ρ to the number of iterations without improvement. We range ρ between $[\rho_{min}, \rho_{max}] = [0.1, 0.5]$.

If no better solution is found for some iterations, ρ should get larger (so close to ρ_{max}) to support a stronger exploration. If, on the other hand, a new best solution is found recently, the exploitation should be stronger and ρ close to ρ_{min}. Finally, to slow down the convergence towards ρ_{max} the square root of *iterationsWithoutImprovement* is taken

$$\rho = \rho_{max} - \frac{(\rho_{max} - \rho_{min})}{\sqrt{iterationsWithoutImprovement}} .$$

Dynamic q_0

We range q_0 between 0.5 and 1.0, where 1.0 should be taken if a better solution was found in the current iteration, otherwise q_0 should converge towards 0.5. We implement this in the same way as we did with β and ρ and hence connect q_0 to *iterationsWithoutImprovement* by

$$q_0 = q_{0_{min}} + \frac{(q_{0_{max}} - q_{0_{min}})}{\sqrt{iterationsWithoutImprovement}} .$$

If a better solution is found recently, a greedy search in city i's neighborhood will be performed, otherwise a biased exploration is enabled.

Iteration-best versus Best-so-far Ant

Surprisingly, it is still unknown whether the use of *iteration-best* ant is better than using the *best-so-far* ant and vice versa. We run our tests with both combinations as an optional input, but at least one of the flags for *best-so-far* or *iteration-best* has to be set. Hence, there is always at least one update for the *best* ant. We assign the same value for the pheromone update to both

versions, so the *best* update is calculated by

$$\tau_{ij}(t+1) = \begin{cases} (1-\rho)\cdot\tau_{ij}(t) + \rho\cdot\Delta\tau_{ij}(t) & \text{, if } ij \in \textit{iteration-best} \text{ tour \& } ib \\ (1-\rho)\cdot\tau_{ij}(t) + \rho\cdot\Delta\tau_{ij}(t) & \text{, if } ij \in \textit{best-so-far} \text{ tour \& } bs \\ \tau_{ij}(t) & \text{, otherwise} \end{cases}$$

where *ib* refers to the set flag for the *iteration-best* update and *bs* denotes that the flag for the *best-so-far* update is set.

Update of Worst Ant

In the original version of the Ant Colony System only the *best* ant is allowed to update the pheromone. In [19] the update of the worst ant was introduced: The worst ant lowers the pheromone level on the bad edges and therefore punishes very bad solutions. Thus, this update makes a bad solution very unlikely to be chosen again.

Since this seems to be a coherent concept, we implement the *worst-so-far* ant update as well. Note that we have the choice between the *worst-so-far* and the *iteration-worst* ant and choose the first one. The main reason for this is because it was proposed in [19] that way and the execution time of all tests would have been doubled if we had taken both versions as a possible input.

Finally the pheromone update can be described by

$$\tau_{ij}(t+1) = \begin{cases} (1-\rho)\cdot\tau_{ij}(t) + \rho\cdot\Delta\tau_{ij}(t) & \text{, if } ij \in \textit{best} \text{ tour} \\ (1-\rho)\cdot\tau_{ij}(t) - \rho\cdot\Delta\tau_{ij}(t) & \text{, if } ij \in \textit{worst-so-far} \text{ tour \& } ws \\ \tau_{ij}(t) & \text{, otherwise} \end{cases}$$

where *best* is either the *best-so-far* tour or the *iteration-best* tour and *ws* denotes that the *worst-so-far* flag is set.

4.2 Testbed Design

Our aim is to examine which structural properties of a TSP instance cause which parameter combination to yield good results. In order to have a variety of TSP instances, we design the following test bed. Note that we consider only symmetric instances.

4.2.1 Problem Sets

TSPLIB

We use symmetric and Euclidean instances from TSPLIB[3] , whereby for most instances the optimal solution is known and can be found at the given URL.

[3] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

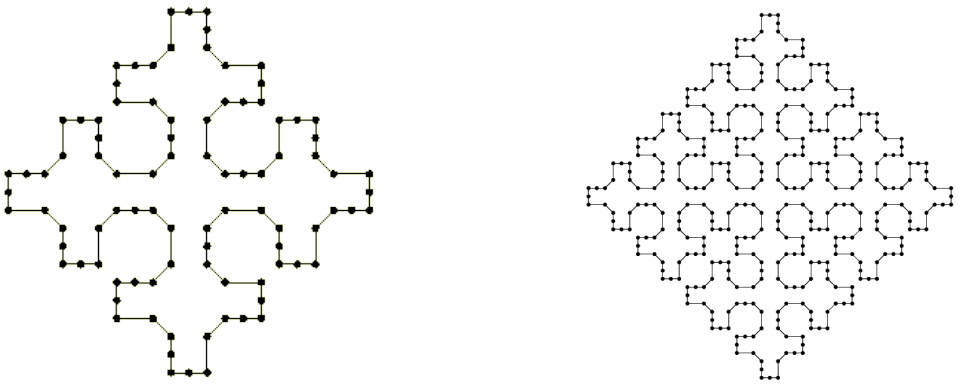
DIMACS TSP Challenge

- **Random Instance Generator:** This random instance generator constructs two-dimensional instances with integer-coordinate points which are uniformly distributed in the 10^6 X 10^6 square. The optimal solutions are unknown for problems produced by this generator.

- **Random Cluster Instance Generator:** This random cluster instance generator constructs integer-coordinate points located in clusters which are uniformly distributed in the 10^6 X 10^6 square. The optimal solution is unknown for this generator.

Fractal Instance Generator

Probably since *L-systems*[4] are quite popular for describing fractal-like recursive structures, they were used in [20] to describe particular instances for the Euclidean TSP. Moreover, due to the recursive structure of the constructed instances, the optimality of all tours generated with a L-system is proven [5]. We decided to generate 3 fractal types: the *David Tour*, the *MPeano* and the *MNPeano Tour*.



(a) (b)

Figure 4.1: Instance *MNPeano* of order two is illustrated in (a) from [21]. (b) from [20] shows the *MNPeano* instance of order three. Note that an order of n means, that n parallel substitutions are made.

[4] An *L-system* or *Lindenmayer system* is a parallel rewriting system (for more information see [24])

[5] The optimal tour is the permutation $(1 \ldots n)$ where n is the number of cities.

Random Symmetric Distance Matrix Generator

Finally, we implement a method for constructing random instances by generating a random distance matrix. The distances are uniformly distributed in (0, 1000).

4.2.2 Settings for the TSP Instances

The next step is to decide which of the instances to take for our simulations. For external reasons, we have to skip numerous instances to finish the simulations soon enough to complete this work.

Size of Instances

The most obvious criterion is the number of cities, often referred to as the *size* of an instance, which is handled as the most significant measurement for complexity of an instance so far. Clearly, the bigger the instance, the more computational effort must be invested to obtain a good solution. We generate the size of all instances with an exponential function. Due to external reasons we have to restrict the number of instances. We create 18 instances sized between 100 and 1,713 cities. For the other testbed classes, we choose 1,883 as the upper bound for the size, since this is the largest instance presented at TSPLIB with less than 2,000 cities. 2,000 is also the limit we set due to the time restrictions.

Resulting Testbed

We present the whole testbed in Table 4.1. Overall we obtain 131 TSP problems from five different testbeds. Since TSPLIB contains the most realistic data, most instances are taken from this repository.

Instance Type	Number of Instances
TSPLIB	65
Random instance generator	18
Random cluster instance generator	18
Fractal	12
Random distance matrix	18
Sum	131 instances

Table 4.1: Testbed

4.2.3 TSP Classifications

To find out which TSP problems can be solved best with which parameter combination, we need to differentiate between different graph instances. Therefore we have chosen six measures to characterize a given instance. Note that we explain all measures only by how we used them.

Measures

Size We call the number of cities in an instance *size*. Since the *number of possible tours* [6] in a graph of size n is $\frac{(n-1)!}{2}$ the difficulty to determine which of all possible tours is the best, is getting larger with the instance size. However problem instances with the same size may be solved in a different number of computational steps [12], so the problem size on its own is not a sufficient criterion.

Arithmetic Mean The arithmetic mean μ (or *mean*) of all graph distances in the distance matrix $D^{n \times n}$ is calculated by $\mu = \frac{\sum_{d_{ij} \in D}}{n^2}$. Thus, the mean is the average distance between all cities. The mean is sensitive to outliers when the samples are small, so in this case the *median* is a better indicator of the center of the data.

Median We call the value positioned in the middle in the sorted list with entries of the distance matrix the *median*. Whereas the mean is vulnerable to outliers, the median does not reflect them.

Standard Deviation The standard deviation (*std*) is a measure of the variability or dispersion of a data set. The higher std gets, the higher the diversity in the data set is. The standard deviation of all entries in the distance matrix D is calculated by $std = \frac{\sqrt{\sum_{d_{ij} \in D} (d_{ij} - \mu)^2}}{n * n}$, where μ is the mean as defined above. In [4, 22] the standard deviation is described as an important measure that goes beyond the typical classification of the problem size. They have shown that problems with the same problem size, which only differ according to the standard deviation of distances, are completely different problems in terms of the required computational effort to solve them.

Std of nNND The nearest neighbors (NN) of a city are those cities that have the shortest distance. In order to make this distance independent of the scaling of an instance, we normalize it by dividing it through the mean

[6] Firstly the starting city is given, so we got $(n-1)$ cities to go to next. As the TSP tour is a cycle it does not matter if we first go to the left or to the right from the starting city; both tours are the same since the graph is symmetric, but they are two different combinations in the set of $(n-1)!$ many combinations. Since this is true for each of the $(n-1)!$ combinations, we have to divide this number by two. □

distance of all edges in the TSP graph. Finally, the standard deviation is computed for the sequence of normalized nearest-neighbor distances (nNND) for all cities [12] and we obtain the *standard deviation of normalized nearest-neighbor distances* (*std of nNND*). This is a measure of the homogeneity of the graph: the more compact all cities are, the smaller this measure is and vice versa. The std of nNND takes only the smallest distances between cities into account. A look beyond the nearest-neighbors is provided by the following measure.

Variation Coefficient In [12] the variation coefficient is defined as the standard deviation of all distances of the graph divided by the mean of all distances given. It is a measure of dispersion which is independent of the scaling of a graph.

Running the Tests

We run the simulations on four AMD Opteron 254 E with 2×2.8GHZ and 4GB RAM. We split the simulations into four groups and run each group on a different machine. Table 4.2 gives an overview over the possible parameter configurations.

To obtain the best configuration for the respective instances, we test *all* possible combinations of the strategies. Finally, we obtained *six* possible combination of the first three parameters, resulting in: $6 \times \underbrace{2}_{\beta} \times \underbrace{2}_{\rho} \times \underbrace{2}_{q_0} = 48$ parameter combinations for all parameters.

Parameter Strategy	Flag Set	Flag not Set
best-so-far	apply best-so-far update	set iteration-best flag
iteration-best	apply iteration-best update	set best-so-far flag
worst-so-far	apply worst-so-far update	–
dynamic β	apply dynamic β	β is set to 3.5
dynamic ρ	apply dynamic ρ	ρ is set to 0.1
dynamic q_0	apply dynamic q_0	q_0 is set to 0.9

Table 4.2: Usage of parameter strategies

Chapter 5

Results

In this chapter, we present the results of our tests and the conclusions we have drawn. At first, we evaluate the graph measurement we have done. We continue with the analysis of *good* and *bad* combinations and we end this chapter with a review of what we have achieved.

5.1 Evaluation of Graph Measures

Our primary goal is to determine which parameter settings work well in conjunction with particular graph characteristics. After having implemented six graph measures (Section 4.2.3), we plot each of them against all instances and all different combinations. We aim to find a link between a graph measure and some parameter combinations, telling us for which measure range which parameter combination fits best.

Firstly we focus only on the *five* best parameter combinations for each instance. We obtain those by comparing the average best tour, taking the average time for a solution as the second criterion if otherwise no choice can be made.

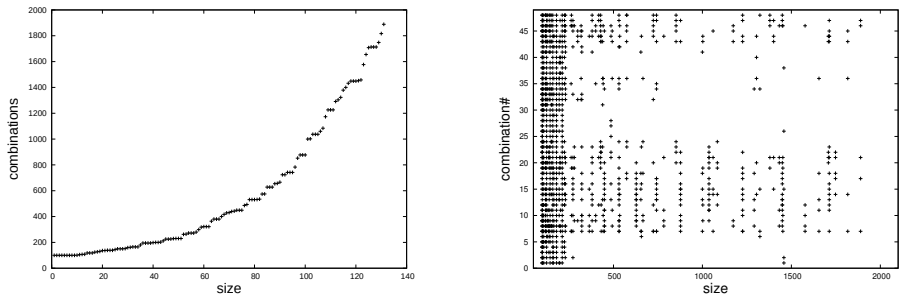
Some experiments lead us to conclude that there are some instances where more than five (for some of them even all) combinations yield the same best result in the same time. However, if we continued taking only the best five combinations for these instances, the combination of parameters might be skewed. In order to avoid this distorted picture on which combination fits best, we will do the following: If we obtain more than five combinations with the same best solution and the same time, we call it *nBest*, we consider all those *nBest* combinations as best. Otherwise, we take the best five combinations and consider them best.

We present the figures and our conclusions in the following sections. Note that we do not split the instances in the five testbed groups, since we are interested in how to find the best parameter setting for a new and unknown instance. In this case we do not know where the instance was taken from.

Therefore, the distinction between different testbed classes is not in our interest and will be disregarded.

Size of an Instance

Firstly, to get an impression on the distribution of the instances in the testbed according to their size, we present Figure 5.1a. The second Figure 5.1b is a plot of the problem size against all parameter combinations. The instance sizes



(a) (b)

Figure 5.1: Size versus parameter combinations

are ranged between 100 and 1,889. As one can already see, we have taken more instances with smaller size than with a bigger one, since the smaller instances require less computational effort. The figure suggests that the difficulty is increasing with the instance size. Unfortunately, this cannot be a sufficient measure as, e.g., two instances both have 100 nodes. While for the first problem *twelve nBest* solutions were found, there was only *one* parameter combination yielding the best solution for the latter.

Evaluation

Even if the problem size seems to be a good measure, we cannot use it to characterize graphs. Two graphs might have the same number of cities, but as shown their difficulty differs a lot. Therefore, we propose to measure the problem's computational difficulty not only with the problem size, but in combination with further measurements. It is, however, unclear which measure to choose.

Mean, Median and Standard Deviation of all Distances

By means of Figure 5.2, one can infer that mean, median and standard deviation vary a lot, not only in their range, but also with the parameter settings

that are *easiest* to solve (according to the number of *nBest*). They are easier to solve since many combinations, for some even the worst ones, lead to the best solution in the shortest time. So there must be something structural about these instances that makes them easier to solve than other instances, thus we refer to them as *easy* instances. Looking at Figures 5.2a, 5.2c and 5.2e, a range gap between 1,000 and 10,000 on the y-axes can be seen. Due to the harsh restrictions in number of instances and instance size, we cannot say whether this gap is also apparent among other testbeds and whether it might be seen as an important characteristic.

Evaluation

Regrettably, the data points are not arranged in a way that leads to a conclusion on how the three measures influence the parameter setting. This is since the areas of high density (the areas with many *nBest* points) are too equally distributed.

The standard deviation is another inappropriate measurement, since it does not show any property of the respective instances, in contrast to what was proposed in [4, 22]. We can but establish one property of our test data: only a few outliers are present in the distance matrix. This results from comparing the median of distances for each instance is to the mean. All three figures are rather similar and do not enable us to draw any further conclusion. Therefore, the three measures seem inappropriate to us, since they do not reflect properties which might help us to reason anything about the parameter combinations that suit the respective instances.

Standard Deviation of Normalized-Nearest Neighbor Distances and Variation Coefficient

Figure 5.3 shows the two measures plotted for all instances or against all parameter combinations, respectively.

Standard Deviation of Normalized-Nearest Neighbor Distance (std of nNND) The std of nNND reflects the homogeneity of the graph. Most of the instances are ranged between 0 and 0.06 on the x-axis, as can be see from Figure 5.3b. A division into two parts can be made according to Figure 5.3b: Instances ranged between 0 and 0.05 on the x-axis seem to have many *bestN* and the remaining ones only the minimal number of five *bestN* or slightly more. Nevertheless, this might be misleading since the number of instances in the first part is much higher than in the second. On the contrary, one can see from Figure 5.3b that the rightmost instance has more than five *bestN*.

Variation Coefficient The following tendency for the plot can be seen: The farther we proceed to the right, the fewer instances there are with more than five *bestN*, and thus the more difficult the instances are. Most of the TSP instances are grouped around 0.5 on the vertical axis. Hence, the standard deviation for most instances is around half of the mean value of all distances.

Conclusions

Both approaches might be worth being considered further, even if we do not know how to evaluate them appropriately at the moment. The concept of both measures is a good idea; further measures should be chosen independently of the problem scale. The independence of the instance scaling is achieved for both measures, because they are computed by a ratio based on the mean distances of all cities.

5.2 Ranking of the Combinations

We use some abbreviations for the tables in this chapter and denote them therefore briefly below in Table 5.1. For a more detailed description of these parameters, see Section 4.1 and Section 4.2.3.

Abbreviation	Long Form
ib	iteration-best update
bs	best-so-far update
ws	worst-so-far update
der	dynamic ρ (evaporation rate)
dq0	dynamic q_0
db	dynamic β

Table 5.1: Abbreviations used

5.2.1 Best Combinations

To obtain an overview of which parameter settings are best for most instances, we have listed the best ten combinations below. Note that we use the ✓ symbol to show that we have enabled the adaptive option or the update respectively. The ✗ symbol is used to indicate which update was disabled and a given value represents the static parameter setting with the particular value.

Evaluation

Surprisingly, we found three combinations that yield better results than the standard version proposed by Dorigo *et al.* (only *best-so-far* is set), on our

Pos.	ib	bs	ws	der	dq0	db	in *bestN*
1	✓	✗	✓	✓	✓	3.5	37.40%
2	✓	✗	✗	✓	✓	3.5	29.77%
3	✗	✓	✓	✓	0.9	3.5	28.24%
4	✗	✓	✗	0.1	0.9	3.5	26.72%
5	✓	✗	✓	0.1	0.9	3.5	26.72%
6	✓	✗	✗	0.1	0.9	3.5	25.95%
7	✗	✓	✓	✓	✓	3.5	25.95%
8	✓	✗	✗	✓	0.9	✓	25.95%
9	✗	✓	✓	0.1	0.9	3.5	25.19%
10	✓	✗	✓	✓	0.9	✓	24.43%

Table 5.2: Ranking of the best combinations found for all instances.

test data. This might indicate that either we have a better approach for most instances or that our testbed is not an appropriate representation for most TSP problems.

Dynamic β is only presented twice among the best combinations, so we would not recommend this strategy so far. To answer the question whether the *best-so-far* update is better than *iteration-best*, we have scored the occurrences in the table: *Iteration-best* is ranked six times among the best instances, so only slightly more often than *best-so-far*. It is however set in the two best combinations. Otherwise, the *worst-so-far* update is ranked six times among the best and is even set in the best combination. If we turn the *worst-so-far* update off, we obtain only the second best combination.

No combination is totally superior in a sense that using it is a guarantee for good solutions. Even the best combination achieves the best solution only for 37.40% of all instances.

Unknowingly, we might have chosen instances with characteristics that are biased towards the given combination, so it is hard to tell how valid the results presented here are.

Comparison among three best Combinations

After having found the best combinations, we ask ourselves whether the best combinations achieve the best solutions on the same instances or whether they have been ranked best at different ones. The results are illustrated in Figure 5.4. Surprisingly, the three best combinations have not been ranked together among the best for most instances; so they appeal to different problems. This is unexpected, since the first combination and the second only differ in one parameter setting. Overall, this confirms the complexity of the interaction of all different parameters, reported by Dorigo *et al.*

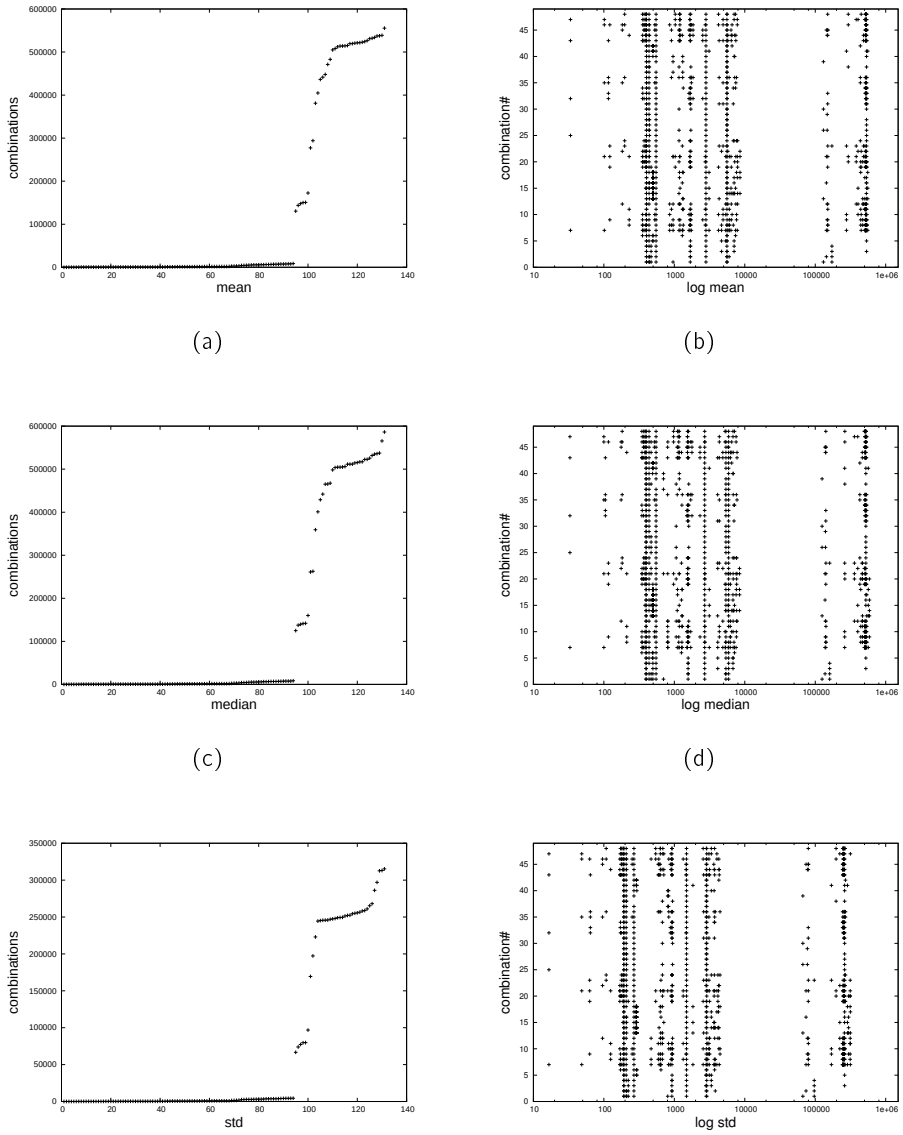


Figure 5.2: The distribution of mean, median and standard deviation for all instances is presented in (a), (c) and (e). The plot of the respective measure against all parameter combinations is given in (b), (d) and (f).

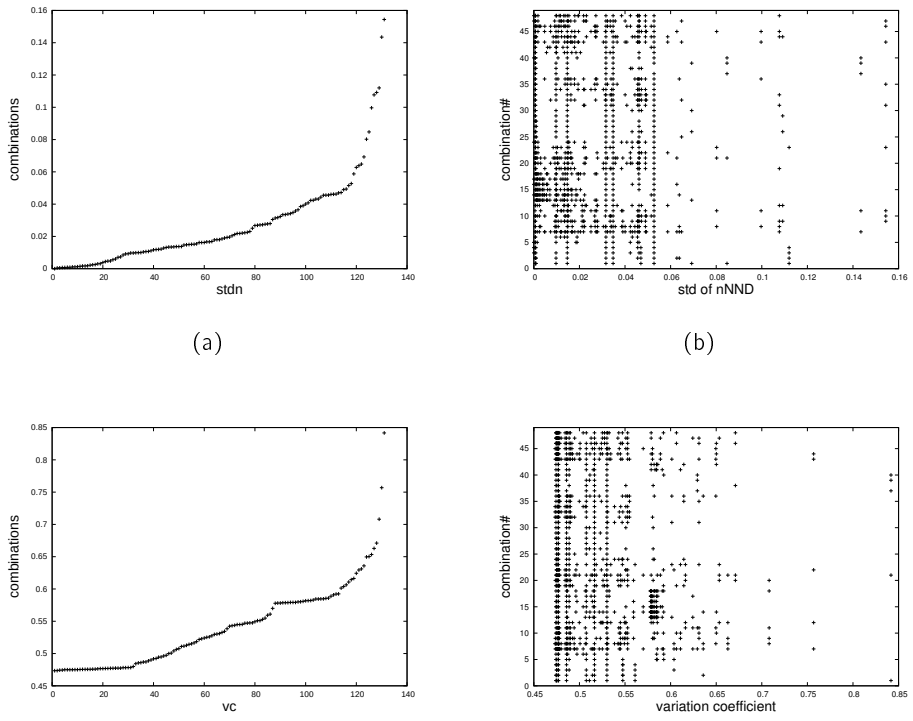


Figure 5.3: The distribution of the standard deviation of normalized-nearest neighbor distances for all instances is presented in (a). (c) shows the same plot for the variation coefficient. (b) and (d) illustrate the plot of the respective measure against all parameter combinations.

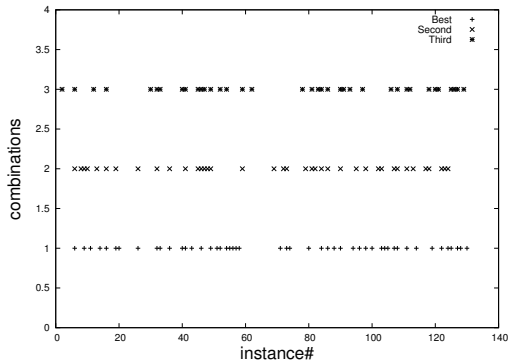


Figure 5.4: Three best parameter combinations

5.2.2 Worst Combinations

Without the knowledge of the worst combinations, the best ten combinations are not meaningful enough to draw conclusions based on the frequency of each parameter setting. This is why we also consider the worst combinations and list them in Table 5.3.

Pos.	ib	bs	ws	der	dq0	db	in *bestN*
39	✗	✓	✓	✓	✓	✓	9.16%
40	✗	✓	✗	✓	✓	✓	8.40%
41	✓	✗	✗	0.1	0.9	✓	8.40%
42	✓	✓	✗	0.1	✓	✓	8.40%
43	✗	✓	✓	0.1	0.9	✓	7.63%
44	✓	✗	✓	0.1	0.9	✓	6.87%
45	✓	✗	✓	0.1	✓	✓	6.87%
46	✗	✓	✗	0.1	✓	✓	6.87%
47	✓	✓	✓	0.1	✓	✓	6.11%
48	✗	✓	✓	0.1	✓	✓	6.11%

Table 5.3: Ranking of the worst combinations found for all instances

Evaluations

One of the results we can derive from both scores is that the usage of *dynamic* β is highly discouraged. Table 5.3 reveals that *dynamic* β is always ranked among the last combinations and only twice among the best combinations. Thus, due to these results it is not a good strategy. One possible explanation for this is that, as the distances approach one, the search is only biased towards the pheromone. Concerning the amount of pheromone, trails which have been ranked best often are thus likelier to be chosen again. We can thus say that β should be bigger than zero, otherwise the pheromone information is overweighted.

Dynamic evaporation rate does, on the other hand, seem to be a promising parameter control approach. Even if it is only listed three times among the best instances (but there among the two best strategies), together with only two listings from Table 5.3, we suggest the use of *dynamic evaporation rate*.

Finally, we would prefer the *iteration-best* update over the *best-so-far*, because the latter is listed seven times in this table and only scored four times among the best. The difference between both is not as high as expected, but overall *iteration-best* is superior in our tests.

It is hard to tell, whether the use of the *worst-so-far* update is recommendable, since it is scored six times for the most successful, as well as six

times for the least successful combinations. Nevertheless, it is set for the best combination, so we would encourage the further use of it.

Comparison among five worst Combinations

After having ranked the worst parameter combinations, we are interested again in whether those parameter combinations are ranked among the same instances as best-performing ones. As one can see from Figure 5.5, they mostly belong to the best combinations for the same instances. This can be explained by

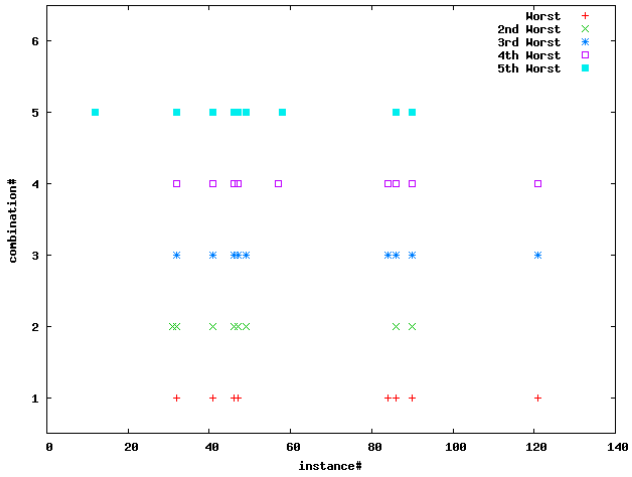


Figure 5.5: Comparison of five worst combinations

counting how many combinations achieved the best solution in the best time for each instance: *Three* instances were found for which *all* combinations result in the best tour and *five* instances for which more than 30 best combinations were found. Together this accounts for 6.11% of all instances, which explains the percentages found for the last two combinations listed in Table 5.3. Those eight instances have a size of 100 to 212 cities; relatively small.

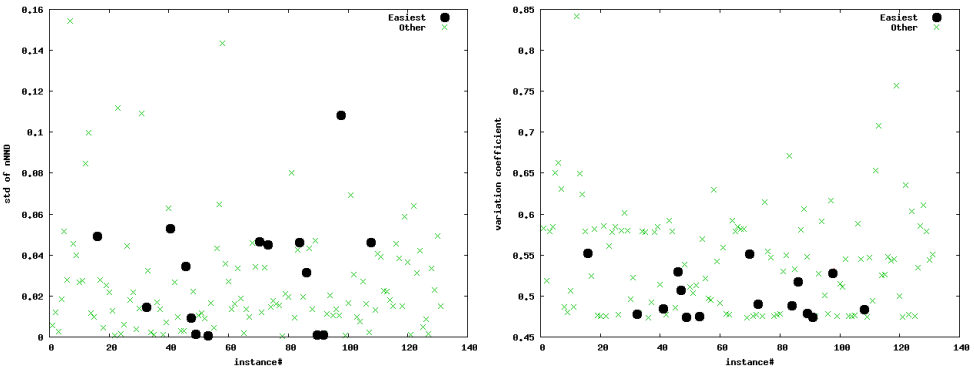
Unfortunately, it is still unclear why these particular instances are easier to solve; it cannot be exclusively due to the size since there are many more instances in this range which are harder to solve.

Hence, we decided to compare all those instances with more than 10 best combinations based on their graph measures. The graph measurements for those 15 instances are presented below in Table 5.4. The range for most of the values is too large, so considering median, mean and standard deviation does not help in revealing similarities among the values. Only for the standard deviation of the normalized nearest-neighbor distances and the variation coefficient, we examine the general range of all instances and aim to draw some conclusions. The two plots are presented in Figure 5.6. In both figures *easiest*

No.	*nBest*	Size	Mean	Median	Std	Std nNND	V.coeff.
15	21	100.0	1.67e3	1537.0	923.71	0.05	0.55
31	48	180.0	402.14	397.0	191.86	0.01	0.48
40	47	100.0	550.11	550.0	267.28	0.05	0.49
45	48	159.0	2.81e3	2.70e3	1.49e3	0.03	0.53
46	37	144.0	5.60E3	5.40e3	2.84e3	0.01	0.51
48	41	108.0	437.95	438.0	207.70	$0 + \varepsilon$	0.47
52	10	225.0	7.05e3	6.80e3	3.35e3	0.0	0.47
69	11	100.0	1.71e3	1.60e3	944.73	0.05	0.55
72	18	118.0	5.22e5	5.12e5	2.56e5	0.05	0.49
83	37	100.0	5.34e5	5.23e5	2.61e5	0.05	0.49
85	46	124.0	5.58e3	5.89e3	2.88e3	0.03	0.52
89	48	212.0	403.18	398.0	192.17	$0 + \varepsilon$	0.48
90	12	428.0	391.74	384.0	186.08	$0 + \varepsilon$	0.48
97	10	118.0	1.50e5	1.41e5	7.97e4	0.11	0.53
107	14	139.0	5.21e5	5.17e5	2.52e5	0.05	0.48

Table 5.4: Graph measurements for 15 easiest instances

refers to the 15 instances we listed in Table 5.4, while *other* refers to the remaining 116 instances.



(a) (b)

Figure 5.6: Plot of normalized-nearest neighbor distances (a) and of the variation coefficient (b) for all instances

Unfortunately the two figures show that both measures do not help in distinguishing between easier instances and others. All occurrences of *easy* instances are uniformly distributed in the given space. Therefore, the question what makes these instances easier to solve remains unanswered since all measures seem to be inappropriate to classify them.

Chapter 6

Conclusions and Further Investigations

Conclusions

We have set up experiments with 131 instances and tested them with 48 different parameter combinations. We have also computed six TSP measures to classify the chosen instances.

After interpreting the results, we have found three parameter strategies that have led to better solutions than the standard parameter setting proposed by Dorigo *et al.* This was obtained concerning the shortest tour found and the time that was needed for the computation.

Parameter Combinations The most successful combination used the newly developed strategies *dynamic evaporation rate* and *dynamic* q_0 as well as the *iteration-best* and *worst-so-far* update. We encourage the use of *dynamic evaporation rate* and the *worst-so-far* ant for further research, since these settings are scored often among the best combinations. All adaptive strategies make use of the stagnation behavior of the algorithm: if no better solution was found for a while, the finding of different paths should be activated. The newly designed strategy of *dynamic* β lead to disappointing results. An important question is, whether the testbed we have chosen is appropriate, or if we might have chosen (unknowingly) a skewed distribution of instances that influences the rankings.

Graph Measures In order to evaluate the graph measures, we visualized all of them and showed that neither median, mean nor standard deviation are appropriate to represent the TSP graph. Size, standard deviation of normalized nearest-neighbor distances and variation coefficient, on the other hand, look promising to help us in classifying a given instance. Nevertheless, they are

not sufficient, so further investigations into possible TSP measures should be undertaken. It can be concluded that the measures have helped us only to characterize the testbed, but they are not sufficient for judging a single instance.

Further Investigations

Additional Tests More experiments should be run to show if our proposed settings can maintain their positions in the ranking of the different approaches. Those tests should be done with further instances from the five different testbeds (or even further testbeds), possibly choosing only the best *ten* parameter settings provided by this work. It might be also taken into account to vary the static values we have chosen for the default version. In our opinion, it is worth considering *dynamic* β in these tests again, as it might be much better when β is ranged between 1 and 5. Our data for the bigger instances was not statistically significant enough. For the next experiments, we propose to choose a uniform distribution of instance sizes.

Multicore Approaches A lot of effort has been put into the development of multi-core-based distributed systems for Ant Colony Optimization [9], which use multicore processors more effectively. For further tests we propose to make use of this new approach.

New Graph Measures More work should be conducted to find graph measures that represent a TSP instance well. We propose to keep to measures which are independent of the problem scale, since dependent measures are not able to discover similarities among a basically identical, but weight-scaled instance. One possible measure might be the number of clusters derived by density-based clustering, which was suggested in [26].

Static vs. Adaptive Parameter Control An interesting question is how our best parameter setting would behave in direct comparison with the (experimentally found) best static parameter setting, which was e.g. investigated in [12]. We therefore propose to run tests to compare the two approaches.

Bibliography

[1] H. Azzag, N. Monmarche, M. Slimane, G. Venturini, and C. Guinot. Anttree: A new model for clustering with artificial ants. In *Proceedings of the 2003 Congress on Evolutionary Computation*, pages 2642–2647, 2003.

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

[3] S. Camazine, J.-L. Deneubourg, N.R. Franks, J. Sneyd, G. Theraulaz, and E. Bonabeau. *Self-Organization in Biological Systems*. Princeton University Press, 2001.

[4] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of the Twelfth International Conference on Artificial Intelligence*, volume 1, pages 331–337. Morgan Kaufmann Publishers, Inc., 1991.

[5] L.N. De Castro. *Fundamentals of Natural Computing: Basic Concepts, Algorithms, and Applications*. Chapman & Hall/CRC, 2006.

[6] M. Dorigo and L.M.P Gambardella. Ant colonies for the traveling salesman problem. *BioSystems*, 1997.

[7] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.

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

[9] Marc Dorigo and Socha Krzysztof. An introduction to ant colony optimization. *IRIDIA Technical Report Series*, 2006. ISSN 1781-3794.

[10] À.E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. In *IEEE Transactions on Evolutionary Computation*, volume 3 of *2*, 1999.

[11] S. Favuzza, G. Graditi, and E. Riva Sanseverino. Adaptive and dynamic ant colony search algorithm for optimal distribution systems reinforcement strategy. *Applied Intelligence*, 24(1):31–42, 2006. ISSN 0924-669X. doi: http://dx.doi.org/10.1007/s10489-006-6927-y.

[12] D. Gaertner. Natural algorithms for optimisation problems. Master thesis, 2004.

[13] L.M. Gambardella and M. Dorigo. Solving symmetric and asymmetric tsps by ant colonies. In *Proceedings of the IEEE Conference of Evolutionary Computation*, pages 622–627, 1996.

[14] X. Hu. Swarm intelligence. Website, 2009. Available online at `http://www.swarmintelligence.org/`; visited on June 23rd, 2009.

[15] V.K. Jayaraman, P.S. Shelokar, and B.D. Kulkarni. An ant colony approach for clustering. Technical report, Chemical Engineering and Process Division, National Chemical Laboratory, India, 2003.

[16] D.S. Johnson and L.A. McGeoch. *The Traveling Salesman Problem: A Case Study in Local Optimization*, chapter 8, pages 215–310. Wiley, 1997.

[17] J. Kennedy and R. Eberhart. Particle swarm optimization, 1995.

[18] Pierre Lévy. *Die kollektive Intelligenz : für eine Anthropologie des Cyberspace*. Bollmann, Mannheim, 1997.

[19] Y. Li and S. Gong. Dynamic ant colony optimisation for tsp. *International Journal of Advanced Manufacturing Technology*, 22(7-8):528–533, 2003.

[20] A. Mariano, P. Moscato, and M.G. Norman. Using l-systems to generate arbitrarily large instances of the euclidean traveling salesman problem with known optimal tours. In *Anales del XXVII Simposio Brasileiro de Pesquisa, Operacional*, 1995.

[21] P. Moscato. Tspbib. Website, 2009. Available online at `http://www.densis.fee.unicamp.br/~moscato/TSPBIB_home.html`; visited on July 16th, 2009.

[22] E. Ridge and D. Kudenko. Determining whether a problem characteristic affects heuristic performance. In *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, volume 153, pages 21–35, 2008. URL `http://www.springerlink.com/content/g22v456585405444/`.

[23] E. Ridge and D. Kudenko. Screening and tuning the parameters affecting heuristic performance. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2007.

[24] G. Rozenberg and A. Salomaa. *The mathematical theory of L systems*. Academic Press, New York, 1980.

[25] T. Stützle and H.H. Hoos. Improving the ant system: A detailed report on the max-min ant system. Technical report, FG Intellektik, FB Informatik, TU Darmstadt, Germany, 1996.

[26] Jano I. van Hemert. Property analysis of symmetric travelling salesman problem instances acquired through evolution. *CoRR*, abs/cs/0502096, 2005.

[27] K.Y. Wong and Komarudin. Parameter tuning for ant colony optimization: A review. In *International Conference on Computer and Communication Engineering*, 2008.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 05. August 2009

Julia Preusse

Appendix A

Glossary

ACO Ant Colony Optimization

ACS Ant Colony System

AS Ant System

PSO Particle Swarm Optimization

TSP Traveling Salesman Problem