Internal Report 2012–08 July 2012

Universiteit Leiden

Opleiding Informatica

Dynamic Ant Colony Optimization for the Traveling Salesman Problem

Sjoerd van Egmond

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Dynamic Ant Colony Optimization for the Traveling Salesman Problem

Sjoerd van Egmond
svegmond@liacs.nl
LIACS, Leiden University

Supervisor
Prof. Dr. T.H.W. Bäck
Second reader
Dr. M.T.M. Emmerich

July 16, 2012

Contents

Abstract

Ant Colony Optimization algorithms are among the best performing heuristics for NP-hard problems. The problem with the current implementations of these algorithms however is that once a solution is found and reinforced it has become very difficult to efficiently explore other possible solutions, limiting these algorithms to unchangeable problem instances. The goal of this thesis is to investigate the possibilities created when hybridizing current Ant Colony Optimization algorithms with the newer Artificial Bee Colony algorithm for use on dynamic problems. Neither algorithm has really been designed for dynamic problems, but this thesis investigates if using the pheromone boundaries of $\mathcal{MM}$AS or ACS and the clustered neighbourhood search of Artificial Bee Colony combined generates algorithms capable of following the optimum of a dynamic problem through the search space.

Chapter 1

Introduction

This thesis investigates the possibility of a hybrid Ant Colony Optimization-Artificial Bee Colony algorithm, capable of following the optimum of a dynamic Traveling Salesman Problem through the search space. This chapter will give short introductions to the following chapters, in which these subjects will be more thoroughly described.

Dynamic optimization problems are currently very difficult to solve. Optimization algorithms and metaheuristics are highly parameterized, and usually different types of problems need different sets of parameters. Even an algorithm for one optimization problem can use different sets of parameters depending upon different properties of the current problem instance (e.g. sparse graph vs. fully connected graph). Dynamic optimization problems are no exception, it is not easy to get good results with the same algorithms and parameters as used for non-dynamic (static) problems. Using other parameters for a dynamic problem might increase the performance and get better results, but it is not guaranteed an algorithm designed for a static problem can successfully optimize dynamic problems. Ant Colony Optimization (ACO) algorithms belong to this last group, because they slowly converge to the optimum solution and are not designed to release it once they find it. Some of the currently known ACO algorithms are better than others, but none have been designed with the restrictions of dynamic problems in mind.

Sometimes aspects of different types of algorithms can be combined to make a new, hybrid algorithm that uses the best aspects of both to solve problems. In this thesis a hybrid algorithm is proposed based on the Ant Colony Optimization algorithm and the Artificial Bee Colony algorithm. Both algorithms work with a selected number of agents, usually more than one but not too many, and both use paths created by agents to determine the paths of the agents following. While Ant Colony Optimization algorithms are designed to work on graphs, Artificial Bee Colony algorithms are designed to work on continuous domains. It can be difficult to translate an algorithm designed for a continuous domain to work on a graph (a discrete domain), and to translate a graph algorithm to a continuous algorithm is even more difficult, though it can be done (see Chapter 3.2.7). Since the chosen problem is the Traveling Salesman Problem, the Ant Colony Optimization algorithm is chosen as the basis and the Artificial Bee Colony algorithm is translated to work on a graph based problem. The most difficult part to translate is defining a neighbourhood of an agent and being able to randomly create neighbours from it (see Chapter 5.2.1).

In Chapter 2 emergent behaviour is explained, which is the observed behaviour that small, autonomous agents with only small instruction sets can generate complex behaviour in a colony of agents. It is important to understand about emergent behaviour because all *natural computing* algorithms are based upon this idea, since in nature a lot of successful examples can be found. Then in Chapter 3 the Ant Colony Optimization will be explained, with the details of the basic algorithm in Chapter 3.1 and the differences introduced by other Ant Colony Optimization algorithms in Chapter 3.2. Followed by an explanation of the Artificial Bee Colony algorithm in Chapter 4, this chapter describes everything needed to know about the Artificial Bee Colony, from the concepts to the pseudocode and the parameters. In Chapter 5 the hybrid algorithm will be introduced, and it also describes special algorithms needed to make the Traveling Salesman Problem work on a dynamic graph. Finally in Chapter 6 and Chapter 7 the experiments and their results can be found.

Chapter 2

Emergent behaviour

Emergent behaviour, or *emergence*, was already an old concept when it got its name in 1875 [25], and is currently used in a wide variety of knowledge fields, from religion to art and from philosophy to science. One of the many possible definitions is given by Goldstein in *Emergence, Complexity and Organization* [16]:

Definition 1. Emergence *is the arising of novel and coherent structures, patterns and properties during the process of self-organization in complex systems.*

Emergent behaviour is an effect that arises from multiple interactions of simple agents with other agents or the environment, and in this way the collective of agents can form complex behaviour. The *collective of agents* is usually called a *swarm* or a *colony* in natural computation. The interactions are often between agents that already did something and their successors that can use that information to perform the same task better. This *feedback*, that usually comes as *positive feedback* or *reinforcement* but can also come as *negative feedback* or *penalties*, is the driving force of emergence. The number of interactions possible in a system increases combinatorially with the number of components of the system [25], and this makes it very hard, or even impossible, to predict the emergent behaviour. Because this emergent behaviour can only be seen as a property of the swarm and cannot be found in any way in a single agent it is called *irreducible.*

2.1 Foraging behaviour

The form of emergent behaviour Ant Colony Optimization algorithms are based on is the foraging behaviour of ants. The underlying mechanism of this foraging behaviour was discovered in 1959 by Pierre-Paul Grassé [17], a French entomologist, which was called *stigmergy.* Ants leave a pheromone trail to a food source, which stimulates subsequent ants to follow the same path with a greater possibility.

Definition 2. Stigmergy *is a mechanism of spontaneous, indirect coordination between agents or actions, where the trace left in the environment by an action stimulates the performance of a subsequent action, by the same or a different agent. [23]*

As can be seen from Definitions 1 and 2 stigmergy describes a subset of possible emergent behaviours, specifically those emergent behaviours where there is no direct interaction between agents but all interactions are with the environment. In Chapter 3.1 an experiment is explained that shows how ants use stigmergy to find the shortest path.

Artificial Bee Colony algorithms do not use stigmergy, but have another form of foraging behaviour called *self-organization*. Bees do not use the environment as a form of indirect communication, but directly communicate with each other in their hive.

Definition 3. *Self-organization is a set of dynamical mechanisms, which result in structures at the global level of a system by means of interactions among its low-level components. [19]*

Even though the definitions given above seem to describe minuscule differences of emergent behaviour, the effect they actually wish to define is mostly the same, but because emergence occurs in many different knowledge areas many different definitions with as many names exist for it.

Chapter 3

Ant Colony Optimization

Decades after the French entomologist Pierre-Paul Grassé discovered *stigmergy*, the use of pheromones on the environment to communicate indirectly, within ant colonies in 1959 [17], Marco Dorigo et al. founded the field of *Ant Colony Optimization (ACO)* with his master thesis on *positive feedback as search strategy* in 1991 [10],[7]. At first he improved the algorithm himself until the publication of his thesis in *IEEE Transactions on Systems, Man, and Cybernetics* [13], after which research in this field takes flight and within five years Ant Colony Optimization belongs to the best algorithms available for *Combinatorial Optimization (CO)* problems.

Ant Colony Optimization was formalized as a *metaheuristic* for Combinatorial Optimization problems by Dorigo et al. in 1999 [11]. A metaheuristic consists of a set of algorithmic concepts used to solve a general class of computational problems, which can be applied to different problems with only a few modifications.

In Table 3.1 a comprehensive list of algorithms and their authors is given, as given in [8], and in Table 3.2 a list of chronologically ordered overview papers can be found.

Year	Algorithm	Authors	References
1991	Ant System (AS)	Dorigo et al.	[10],[7],[13]
1992	Elitist AS	Dorigo et al.	[7],[13]
1995	Ant-Q	Gambardella & Dorigo	[15]
1996	Ant Colony System	Dorigo & Gambardella	[12]
1996	$\mathcal{MAX}$-$\mathcal{MIN}$AS	Stützle & Hoos	[22]
1997	Rank-Based AS	Bullnheimer et al.	[3]
1999	ANTS	Maniezzo	[20]
2000	Best-Worst AS	Cordón et al.	[5]
2001	Hyper-Cube AS	Blum et al.	[1]

Table 3.1: Overview of the more popular or successful Ant Colony Optimization algorithms

Year	Title	Authors	References
2004	Ant Colony Optimization	Dorigo Stützle	[14]
2005	Ant colony optimization theory: A survey	Dorigo Blum	[9]
2006	Ant Colony Optimization: Artificial Ants as a Computational Intelligence Technique	Dorigo Birattari Stützle	[8]

Table 3.2: Existing Survey papers since 2004

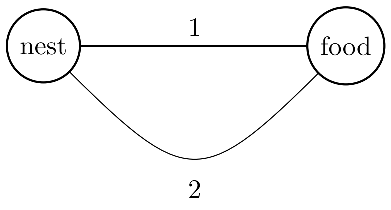

Figure 3.1: Double Bridge experiment, edge 1 is shorter than edge 2

3.1 Ant System

All Ant Colony Optimization algorithms are modeled after the foraging behaviour of ants (see Chapter 2.1), because a swarm of cooperating ants can differentiate between sources of food with variable quantity and quality, and between the lengths of the paths to a food source. If the only objective of the ants is finding the closest source of food then it has been experimentally proven that the majority of ants converge to the shortest path. This experiment is known as the *Double Bridge experiment* (see Figure 3.1) of Deneubourg et al. [6]. For this experiment a colony of ants and a food source are taken and connected by two paths of unequal length. Trying to find a source of food the ants will traverse both paths and deposit an amount of pheromones depending on the length of the path taken. The ants will converge (see Definition 4) to the path with the highest amount of pheromones on them, which corresponds to the shortest path. In the long run ants are unable to remember any other path but the best path, so with two equally long paths the ants will randomly converge to either one path or the other.

Definition 4. Convergence *is the approach towards a definite value, a definite point, a common view or opinion, or toward a fixed or equilibrium state. [24]*

The generic structure of an ACO algorithm is described in Algorithm 1, and its lines will be elaborated in the rest of this section.

3.1.1 Construction graph

Ant Colony Optimization is an algorithm defined to be executed on a graph, a discrete representation of a problem with a limited amount of solutions. This graph is called the *construction graph* and is created in a way such that the ants can use this graph to walk from node to node over the edges to find a solution

Algorithm 1 Pseudo-code for Ant Colony Optimization

```
1: Create construction graph
2: Initialize pheromone values
3: while not stop-condition do
4:     Create all ants solutions
5:     Perform local search
6:     Update pheromone values
7: end while
```

to the given problem. Usually this graph is *fully connected* to prevent *deadlock* situations in which an ant is stuck on a certain node without possible edges to move on. Several combinatorial optimization problems can be translated into an equivalent graph structure and in [14] the construction graphs for several problems are created and explained.

3.1.2 Solution construction

The *solution construction* step is the part of the algorithm in which the ants create their solutions, in the case of the traveling salesman problem every ant will create a Hamiltonian cycle. An ant starts at a randomly determined node i and traverses the graph according to the *random-proportional rule* by choosing the edge to the next node j with probability

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i{}^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \text{ if } j \in \mathcal{N}_i^k, \tag{3.1}$$

where $\mathcal{N}_i^k$ is the *feasible neighbourhood* of ant k at node i, τ_{ij} is the amount of pheromone on an edge, $\eta_{ij} = 1/d_{ij}$ is a heuristic value known a priori of the algorithm execution (with d being the Euclidean distance between node i and node j as the length of the edge) and α and β two parameters determining the relative influence of τ and η. The feasible neighbourhood of an ant at a certain node is the set of all neighbouring nodes that qualify as a possible next node.

In a fully connected graph the previous step is repeated until a valid tour has been created for the ant. If the construction graph is not fully connected and *loops* are not allowed then it is possible that ants cannot complete their tours and end up with invalid tours. A loop is obtained if any node is visited twice. This possibility arises if an ant arrives on a node from which it cannot leave again without visiting another node for the second time and creating a loop. In general loops are not allowed because it has been seen that ants can get stuck in the loop because of the increasing amount of pheromones.

In [21] a *proof of convergence* is given for ACO algorithms which use the pheromone update rule (see Chapter 3.1.3) on the best solution found so far. The proof shows that the algorithm will always find the global optimum for $t \to \infty$, where t is the elapsed time of the algorithm, and that the ants will converge to the global optimum once it is found.

3.1.3 Pheromone update

The rules ants have to follow when depositing pheromones at the end of their tours are described by the *pheromone update* rule. To have the ants converge to a solution the pheromone amounts on good tours slowly have to increase. This is achieved by having ants deposit an amount of pheromones depending on the quality of their solution, where better tours are reinforced with more pheromones. To increase the effectiveness of dropping pheromones on good tours a little bit of the pheromones is removed at the end of every iteration, which slowly decreases the likelihood of creating bad quality tours. Using these factors of *pheromone evaporation* as *negative feedback*, and the pheromone update from the ants as *positive feedback* the solution construction will slowly converge to the better quality tours. Combined in one function this rule is

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \ \forall(i,j) \in L, \tag{3.2}$$

where L is the set of all edges, ρ is the evaporation rate, m the number of ants. The amount of pheromone dropped on (i,j) by ant k, denoted with $\Delta\tau_{ij}^{k}$, can be computed as:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{L_k} & \text{if ant } k \text{ used edge } (i,j) \text{ in its tour,} \\ 0 & \text{otherwise,} \end{cases} \tag{3.3}$$

where Q is a constant (often 1) and L_k is the length of the constructed tour of ant k.

3.1.4 Pheromone initialization

The de facto standard for the initial pheromone values can be described by

$$\tau_{ij}(0) \leftarrow \Delta\tau_{ij}, \tag{3.4}$$

where $\Delta\tau_{ij}$ is the amount of pheromones deposited by one ant generating a random or unoptimized tour as defined in Equation 3.3. In the case of a fully connected graph the nearest neighbour tour is often used for this purpose. If $\tau_{ij}(0)$ is set too low the ants will immediately converge towards the tours generated in the first iteration. Deviations of the Ant System algorithm however often use other settings, which will be detailed in their respective sections in Chapter 3.2.

3.1.5 Local search

In Algorithm 1 the *local search* step has been defined, but this step is optional as it is not actually needed for the algorithm to work. However, experiments have shown that the results of the Ant Colony Optimization algorithms can be significantly better using the local search step. The idea behind local search is that random search or guided search could benefit from it by optimizing the generated solutions a little bit, and thereby increasing the efficiency of the entire algorithm. Local search tries to find a better solution in the neighbourhood of the generated solution, where neighbourhood is defined as:

Definition 5. *A neighbourhood structure is a function $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ that assigns a set of neighbours $\mathcal{N}(s) \subseteq \mathcal{S}$ to every $s \in \mathcal{S}$. $\mathcal{N}(s)$ is also called the neighbourhood of s [14].*

In the neighbourhood multiple solutions might be better than the original, and the function that chooses the neighbour to pick, the *solution function*, usually uses the *best-improvement* function or the *first-improvement* function. The best-improvement function takes the neighbour with the largest improvement, while the first-improvement function takes the neighbour with the first found improvement. It is however also possible for the local search step to be skipped, as continually taking an improved neighbour has a higher chance of converging the algorithm to a local optimum.

3.1.6 Important principles

The following sections contain a number of interesting facts combined with some key points to increase the understanding of what happens during the execution of an ant algorithm. These points can be found in Dorigo and Stützle's book on Ant Colony Optimization [14].

Heuristic information

Introducing domain specific knowledge into the algorithm can improve its performance impressively, since it usually only has to be computed once and allows the algorithm to make more knowledgeable choices. In the usual case of solving optimization problems on graphs the length of the edge can be used as heuristic information in the form of a weight on the edge, with the weight calculated as $1/d$ for minimization problems, where d is the length of the edge. The shorter the edge is, the greater the weight will be, which will influence the algorithm's choice more favourably than longer edges.

Some key points should be mentioned however to ensure heuristic information is used correctly. The first point is that using heuristic information introduces a bias on the search space, which can decrease the algorithm's performance for problems where the optimal solution lies outside the boundaries of the bias. The second point is that there are problems where it is not possible to have static heuristic information (i.e. there is no heuristic information that only has to be computed once), but heuristic information can be calculated after a partial solution has been found. Sometimes calculating this type of heuristic information can be done with reasonable costs, but other times these calculations can be quite expensive. The last point is that the gain of heuristic information is mostly negligible when a local search algorithm is used. On the other hand this allows expensive heuristic calculations to be removed without much performance loss when the algorithm uses a local search algorithm.

Number of ants

Combinatorial optimization problems can be solved using a colony of only one ant, because the solution construction step (Chapter 3.1.2) can be repeated r times for each iteration to simulate a colony. However using a colony increases the algorithm's performance, because then only once per iteration the pheromones are deposited (instead of r times in a simulated colony) and some operations can be performed more efficiently on the set of ants as a whole as compared to consecutive updates based on single ants. An effect called the *differential path length* is responsible for increased performance introduced by grouping consecutive ants into a colony. With the differential path length effect more pheromones are deposited on shorter tours, and because a group of ants deposit them at once more ants will follow the previously shortest tour than the previously longest tour in the next iteration. One of the most used techniques to increase efficiency is to selectively choose which ants are allowed to deposit pheromones in certain iterations. When a colony contains more than one ant the worst result can be omitted, or the best result or results can be emphasized by increasing the amount of pheromones they will deposit.

There are however only a few guidelines for how many ants to use for an algorithm, usually an amount of ants equal to the number of nodes is used, except for Ant Colony System, where a fixed amount of 10 ants is used.

3.1.7 Exploration and exploitation

Exploration is continuously searching the entire search space for better solutions, and *exploitation* is taking the currently found solutions and using them to find even better solutions. Correctly balancing these two mechanisms must be done carefully and is very difficult. If the effects from exploration are too strong the algorithm will never converge to any solution, while if the effects from exploitation are too strong the algorithm will converge too fast and will not be able to escape local optima.

Balancing exploration and exploitation can be done in two ways. The first way is to update the pheromones on the edges differently as shown by Ant Colony System (ACS; see Chapter 3.2.2) and the $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS; see Chapter 3.2.3). To increase exploration ACS removes some pheromones from a chosen edge to lower the chance it is chosen again, while $\mathcal{MM}$AS resets all pheromone values if the algorithm is estimated to have reached convergence, i.e. has not found a better solution within a certain number of iterations. The second way assumes heuristic information is used and changes the balance of the parameters between using heuristic information and pheromone information. When α is decreased or β is increased the algorithm focuses more on exploration, and when α is increased or β is decreased the algorithm focuses more on exploitation. Another use of these parameters can be achieved by slowly decreasing β from an initial value $\beta > 0$ towards 0.

3.2 Algorithms

Chapter 3.1 introduced the basic algorithm's inner mechanisms, with the pseudo code found in Algorithm 1. This algorithm is the basis for all extensions and modifications, and in the following sections some of them will be described chronologically. In Table 3.1 a reference to the papers of the authors of the extensions and modifications can be found.

3.2.1 Elitist Ant System

The Elitist Ant System (EAS) is the first improvement of the basic Ant System algorithm [7][13] and proposed only one year after the Ant System, in 1992. This improvement introduces a small memory remembering the tour with the best result (T^{bs}, the *best-so-far* tour). Every iteration T^{bs} is reinforced with an extra pheromone update, even if the tour has not been created this iteration. The pheromone update rule (Equation 3.2) changes to

$$\tau_{ij} \leftarrow (1-\rho)\cdot\tau_{ij} + \sum_{k=1}^{m}\Delta\tau_{ij}^{k} + e\Delta\tau_{ij}^{bs},\ \ \forall(i,j)\in L, \tag{3.5}$$

where e is a weight factor determining how much T^{bs} will be reinforced. Experiments in [13] show that a good value for e might be $\frac{1}{4}\cdot$ `#nodes`. Alternately, Dorigo and Stützle [14] use `#nodes` as value for e.

3.2.2 Ant-Q & Ant Colony System

The Ant-Q algorithm, an ACO algorithm combined with reinforcement learning rules from *Q-learning*, was proposed by Gambardella and Dorigo in 1995 [15]. This algorithm however is not used anymore because a year later in 1996 a simplified version was proposed which turned out to be equally powerful, the Ant Colony System (ACS) [12], therefore only the modifications of ACS on the Ant System will be described.

The solution construction step contains one of the changes, in the new version the ants will much more likely follow the best path of high pheromone values. There is a likely chance it will follow the highest pheromone value to the next node, otherwise it will follow the normal computations of Ant System (Equation 3.1).

$$j = \begin{cases} \operatorname{argmax}_{l\in\mathcal{N}_i^k}\{\tau_{il}[\eta_{il}]^{\beta}\} & \text{if } q \le q_0, \\ J & \text{otherwise,} \end{cases} \tag{3.6}$$

where j is the next node selected by the ant, q is a random number between 0 and 1, q_0 a constant with $q_0 \in [0,1]$ and J a path selected according to the probabilities of the random-proportional rule given in Equation 3.1.

Another change is that pheromone evaporation and pheromone update is only applied to the edges in the best-so-far tour, and the pheromone update is factored by ρ resulting in a weighted average of the pheromone values.

$$\tau_{ij} \leftarrow (1-\rho)\cdot\tau_{ij} + \rho\Delta\tau_{ij}^{bs},\ \ \forall(i,j)\in T^{bs}, \tag{3.7}$$

The two previously described changes rely heavily on the exploitation of the best-so-far tour. The last change increases exploration of the graph by slightly decreasing the pheromone value of an edge traversed by an ant by a percentage (e.g. 5% equals $\xi = 0.05$), with a minimum equal to the initialization value τ_0.

$$\tau_{ij} \leftarrow (1-\xi)\tau_{ij} + \xi\tau_0. \tag{3.8}$$

This means that there are two different ways for the pheromone values to change. Equation 3.7 describes the weighted pheromone update that happens at the end of every iteration once T^{bs} is known, while Equation 3.8 describes an evaporation that happens immediately after an ant has chosen that edge for its tour, to decrease the chance of the next ant following the same tour.

3.2.3 $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System

In the same year as the Ant Colony System (Chapter 3.2.2) was proposed, Stützle & Hoos proposed the $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [22], which exploits the best-so-far tour even stronger than the Elitist Ant System (Chapter 3.2.1). Every iteration either the best-so-far tour or the iteration-best tour (T^{ib}) is allowed to deposit pheromones, and the balance between these two determine the algorithms greediness. The pheromone update rule (Equation (3.2)) for the $\mathcal{MAX}$-$\mathcal{MIN}$AS becomes

$$\tau_{ij} \leftarrow (1-\rho)\cdot\tau_{ij} + \Delta\tau_{ij}^{best}, \ \forall(i,j)\in L \tag{3.9}$$

where ever iteration $\Delta\tau_{ij}^{best}$ is either $\Delta\tau_{ij}^{bs}$ or $\Delta\tau_{ij}^{ib}$. Some other changes however were necessary to increase exploration, because $\mathcal{MM}$AS converges too fast.

Firstly the range of pheromone values is bound to $[\tau_{min}, \tau_{max}]$, and instead of initializing the trails to τ_{min}, where $\tau_{min} > 0$, they are initialized to τ_{max}. If the speed of the pheromone evaporation is low enough this ensures the exploration of the search space at the start of the algorithm because the relative difference between the pheromone values of the updated tour and the rest of the graph will grow less fast. There are some experimental values for the parameters τ_{min} and τ_{max} based on the length of the current best-so-far tour, which can be found in [14].

To get out of local optima the algorithm can reset its pheromone values to the initial values if it has converged to a single solution or if its best-so-far tour has not changed in a given amount of iterations.

3.2.4 Rank-based Ant System

The Rank-based Ant System (AS_{rank}) [3] is proposed by Bullnheimer et al. in 1997, which is a modification of EAS and uses only the best $w-1$ tours of the iteration and the best-so-far tour. Experimentally w usually is set equal to 25% of the number of ants [3][13].

Furthermore the amount of deposited pheromones depends upon the tours rank, where T^{bs} has a weight factor of w, the iteration-best tour a weight factor of $w-1$, the second ranked tour a weight factor of $w-2$ etc. The pheromone update rule now is

$$\tau_{ij} \leftarrow (1-\rho)\cdot\tau_{ij} + \sum_{k=1}^{w-1}(w-r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{bs}, \ \forall(i,j)\in L, \tag{3.10}$$

where w is the amount of ranked ants including T^{bs}, and the lower bound of $(w-r)$ is 0, to ensure that no pheromones are subtracted.

3.2.5 ANTS

ANTS, or the *Approximate Nondeterministic Tree Search*, is proposed in 1999 and influenced by mathematical programming. It dropped the use of heuristic values derived from domain knowledge a priori, but computes lower bounds on completing a *partial solution* (the part of the tour the ant has already constructed) after temporarily adding a node (i, j_1) and uses all these lower bounds of (i, j_1) to (i, j_n), with n the number of neighbours, as heuristic values. The *lower bound* (LB) on the expected result of a tour is the sum of the lengths of the partial solution, the chosen edge (i, j_k) and an estimate of the edges needed to complete the solution from node j_k. The solution construction and pheromone update have also been changed, with solution construction (Equation 3.1) changed to

$$p_{ij}^k = \frac{\zeta\tau_{ij} + (1-\zeta)\eta_{ij}}{\sum_{l \in \mathcal{N}_i{}^k} \zeta\tau_{il} + (1-\zeta)\eta_{il}}, \text{ if } j \in \mathcal{N}_i^k, \tag{3.11}$$

where α and β of Equation 3.1 have been replaced by one parameter ζ and multiplication has been replaced by addition. In the pheromone update rule explicit pheromone evaporation has been removed, changing Equation 3.2 into

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k, \tag{3.12}$$

with $\Delta\tau_{ij}^k$ given by

$$\Delta\tau_{ij}^k = \begin{cases} \vartheta(1 - \frac{C^k - LB}{L_{avg} - LB}), & \text{if edge } (i,j) \text{ belongs to } T^k, \\ 0, & \text{otherwise,} \end{cases} \tag{3.13}$$

where C^k is the length of the tour of ant k, L_{avg} the average length of the last l iterations and ϑ a parameter usually set to τ_0. The result of Equation 3.13 is that the paths for tours better than L_{avg} are reinforced, while if the path is worse than L_{avg} pheromones will be subtracted from the tour.

3.2.6 Best-Worst Ant System

In 2000, the Best-Worst Ant System (BWAS) is proposed, which uses elements from Evolutionary Computation, especially the *mutations*. But also elements from ACS and $\mathcal{MM}$AS have been included. From ACS the pheromone update mechanism (Equation 3.7) is used, and from $\mathcal{MM}$AS the restart and reset of pheromones.

As in ACS, T^{bs} receives pheromones each iteration, but in addition the *iteration-worst* solution (T^{iw}) removes pheromones from edges it contains that are not in T^{bs}. Equation 3.2 is changed to

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij} + \rho \cdot \Delta\tau_{ij}, \; \forall (i,j) \in L, \tag{3.14}$$

where

$$\tau_{ij} = \begin{cases} f(C^{bs}), & \text{if } (i,j) \in T^{bs}, \\ 0, & \text{otherwise,} \end{cases} \tag{3.15}$$

where $f(C^{bs})$ is a function the length of T^{bs} (usually $f(C^{bs}) = \frac{1}{C^{bs}}$). Moreover, to penalize T^{iw} it is evaporated one more time after the global evaporation

$$\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij}, \forall (i,j) \in T^{iw} \text{and} (i,j) \notin T^{bs}. \tag{3.16}$$

Taken from $\mathcal{MM}$AS is the reinitialization of the pheromones to τ_0 when the algorithm has converged too much. This occurs when the difference between T^{bs} and T^{iw} becomes less than a pre-defined percentage, where *difference* is defined as the amount of different edges.

New to ACO however are the mutations to the pheromone values which increase the explorative behaviour. There exists a probability P_m for an edge to be mutated, and the mutation algorithm can be described with

$$\tau'_{ij} = \begin{cases} \tau_{ij} + mut(k, \tau_{threshold}), & \text{if } a = 0, \\ \tau_{ij} - mut(k, \tau_{threshold}), & \text{if } a = 1, \end{cases} \tag{3.17}$$

where a is a random value in $\{0,1\}$, k is the current iteration of the main loop and $\tau_{threshold}$ is the average of the pheromone values of the edges in the best-so-far solution as

$$\tau_{threshold} = \frac{\sum_{(i,j)\in T^{bs}} \tau_{ij}}{|T^{bs}|}, \tag{3.18}$$

The operation *mut* calculates the size of the mutation, which slowly will get bigger during the execution of the algorithm,

$$mut(k, \tau_{threshold}) = \frac{k - k_r}{N_k - k_r} \cdot \sigma \cdot \tau_{threshold}, \tag{3.19}$$

where k_r is the last iteration of a restart, N_k is the maximum number of iterations and σ is the mutation power.

The mutation power defines how fast the mutation reaches $\tau_{threshold}$, and how much higher it can go (e.g. if $\sigma = 2$ then after half of the remaining iterations the mutation is $\tau_{threshold}$, and at the end the mutation will be close to $2 \cdot \tau_{threshold}$).

The problem with this algorithm is that there are many parameters to be adjusted properly, and if it is done incorrectly could greatly decrease the algorithms performance or overfit the algorithm to the benchmark.

3.2.7 Hyper-Cube Ant System

A bit different from the algorithms described above is the Hyper-Cube framework [1], which like ANTS is inspired by mathematical programming. It does not change the rules for the ants or the update mechanisms, but it changes the representation of the construction graph such that solutions can be made by binary vectors.

A binary vector, $\vec{v} = (v_1, \ldots, v_n)$ with every vector component representing the pheromone value of an edge, represents a solution by having every variable v_1 to v_n take a value in $\{0,1\}$, and each unique ordering of 0's and 1's is a solution. This way an n-dimensional hyper-cube is generated, where each corner is a solution, but if values in the entire interval $[0,1]$ are used instead of values in $\{0,1\}$ this hyper-cube represents the entire search-space equivalent to the other Ant System algorithms. Pheromone values have to be scaled to the interval $[0,1]$ for every created vector $\vec{\tau}$, and a binary version of $\vec{\tau}$ is a solution. This means that by creating a vector $\vec{v}$ and creating a one-to-one relationship between the indexes of $\vec{v}$ and the pheromone values τ_{ij} on the edges of the graph a vector $\vec{\tau}$ is created which will not only contain the pheromones of the graph, but also can be viewed as an ant containing a tour when its values are limited to $\{0,1\}$ and a decision function has been defined mapping its current value to either 0 or 1. The vector $\vec{\tau}$ will slowly converge to the optimum using Equation 3.20.

Scaling is done the same way as in ACS (Equation 3.7), changing Equation 3.2 to

$$\tau_{ij} \leftarrow (1-\rho)\cdot\tau_{ij} + \rho\sum_{k=1}^{m}\Delta\tau_{ij}^{k},\ \ \forall(i,j)\in L, \tag{3.20}$$

where $\Delta\tau_{ij}^{k}$ is scaled using

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1/C^k}{\sum_{h=1}^{m}(1/C^h)} & \text{if edge } (i,j) \text{ is used by ant } k, \\ 0 & \text{otherwise.} \end{cases} \tag{3.21}$$

Chapter 4

Artificial Bee Colony

In 2005 Karaboga proposed a swarm-based algorithm based on the behaviour of honey bees [19]. His algorithm is based on the two fundamental concepts of *self-organization* and *division of labour.*

4.1 Concepts

Four mechanisms of self-organization are described by Bonabeau et al. in [2]. *Positive feedback* can be applied very generally, and usually defines the mechanism that rewards good solutions and leads to convergence. On the other hand there is *negative feedback*, which can be implemented to negate some effects from the positive feedback that could negatively effect the outcome. For example if a food source has been emptied a mechanism needs to decrease the amount of agents (ants or bees) visiting that food source. The third mechanism is *fluctuations*, which results in randomness in solutions so new and different solutions can be found. The last mechanism is called *multiple interactions* and require agents to communicate regularly with other agents (or the environment), so every agent can use results from other agents as well as their own results.

Division of labour is a method that allows a swarm to use different types of labourers for different tasks. These labourers will have other instructions and may interact with other types of labourers. Using specialized labourers for specific tasks is supposed to increase the performance of the entire swarm, and *enables the swarm to respond to changed conditions in the search space* [19].

4.2 Honey Bee Swarm

For a honey bee swarm to emerge collective intelligence it needs *food sources* and two types of bees, *employed foragers* and *unemployed foragers.* Two important processes need to be defined, the first describes when bees will start exploitation of a food source, the second describes when they will abandon a food source.

Paths to *food sources* represent the solutions the swarm needs to find, where the quantified quality of the source is equivalent to the inverse length of the solution and used for the TSP minimization problem,

$$Q^F = \frac{1}{L^T} \tag{4.1}$$

with Q^F the quality of the food source F and L^T the length of the associated tour.

Foragers are the labourers of the honey bee swarm and find and exploit the food sources. *Unemployed foragers* are not associated with a known source and act as scouts upon the entire search space or as onlooker in the hive. Scouts generally do not let themselves be influenced by previous results and just go about their own way, the only interactions they have are when they find a new source and the become *employed foragers* to communicate the location to the onlooker bees. Employed foragers are associated with a certain food source and go back and forth between the hive and the food source. In this way they exploit the currently known food source and because of the fluctuations might find a better path to a source close by, which is equivalent to a better solution in the TSP.

Contrary to ants, bees communicate directly with each other in the hive using a *waggle dance*. Depending on the quality of the food source there is a chance that a bee will perform a waggle dance, and depending upon the length of the dance and the dance itself onlooking bees will then decide if they want to become employed foragers for that food source. These mechanisms should ensure that better sources are more thoroughly exploited. Every time an employed forager returns to the hive it can do one of three things, it abandons the source and becomes a scout, it performs a waggle dance to try and get more bees to come to its food source, or it continues to forage the same source. Note that since a scout becomes an employed forager when it finds a new food source, and no other ways exist to become an employed forager, every actively exploited food source will have exactly one employed forager.

4.3 Algorithm

The algorithm based on the behaviour of foraging honey bees is called the *Artificial Bee Colony* (ABC) algorithm. Since only a scout can become an employed forager and only an employed forager can become a scout, every food source has exactly one employed forager and every employed forager has exactly one food source the following equation indicates the number of active food sources

$$\#F = \#S^{\texttt{total}} - \#S^{\texttt{free}}, \tag{4.2}$$

where $\#F$ is the number of food sources, $\#S^{\texttt{total}}$ the maximal number of scouts and $\#S^{\texttt{free}}$ the number of free (unemployed) scouts. Equation 4.2 shows us that the total number of food sources is bound to the total number of scouts. Therefore the mechanism to abandon known sources becomes important to ensure the algorithm does not converge to one of the first solutions it finds.

Bees are the agents of this algorithm, and independent of their role should be able to remember the solution they found. Thus the solution a bee finds consists of a set of values for the input parameters of the function to be optimized. Furthermore every bee knows the role it has and dependent upon that role will perform certain actions.

In Algorithm 2 a global outline of the algorithm is described. In line 1 all the scouts are sent out to find an initial solution. Scouts usually have no guidance since they are meant for exploration and finding any food source, this results in low search costs but usually also low quality food sources. Line 3 sends out the employed bees to the food sources just found, and it is at this moment that the actual quality of the food source is determined. These employed bees are actually the same scouts, but in line 1 they find a solution while in line 3 the quality of the tour is calculated (i.e. the length of the tour). After the solution qualities are calculated probabilities for the onlooker bees to join an employed bee are determined in line 4, followed by sending out the onlookers in line 5. After sending out the onlookers a decision has to be made if, and which, sources are exhausted (line 6), and send out the newly freed scouts to find new solutions (line 7). Before the next iteration of the algorithm the currently best solution (T^{bs}) is memorized in line 8.

Algorithm 2 Pseudo-code for Artificial Bee Colonies algorithm [19]

```
1: Using scouts to find initial food sources
2: repeat
3:     Send employed bees to food sources
4:     Calculate food source probabilities for onlookers
5:     Send onlooker bees to food sources
6:     Stop exploitation of exhausted food sources
7:     Send scouts to discover new food sources
8:     Memorize the best food source so far
9: until Requirements are met
```

4.3.1 Exploration: sending scouts

The scouts that are sent out do not use any information gained by the other bees but try to find new sources of food without being influenced by existing knowledge. In general these solutions will therefore be of too low quality to be optimized using the onlookers. At the start of the algorithm all scouts are send out (line 1), and after existing food sources become exhausted all free scouts are sent out again (line 7).

4.3.2 Determine employed bees

Every scout that found a food source in line 1 or line 7 will automatically become a new employed bee for that food source. For existing food sources the bee with the best solution will become the employed bee for that source, releasing all other bees on that food source as new onlookers. For a minimization problem this results in the following function

$$E_i \leftarrow \texttt{min}\{l(p^k) \mid k \in \{E_{i-1} \cup O\}\} \tag{4.3}$$

where i is the iteration number, p^k the path of bee k and $l(p^k)$ its length, E the employed bee and O the set of onlookers.

4.3.3 Exploitation: sending onlookers

Line 4 and line 5 together determine which onlookers will be sent to which food sources. First the probabilities are calculated for every food source to receive an onlooker

$$P_f = \frac{\frac{1}{L_f}}{\sum\limits_{l \in L} \frac{1}{l}}, \tag{4.4}$$

where P_f is the probability of an onlooker going to food source f, L_f the length of the tour to f and L the set of all tourlengths to the currently active food sources.

Then every onlooker decides which food source to visit and the onlookers get distributed over the available food sources. These onlookers will, by definition, not follow the exact same course as the employed bee and might therefor find a better route to the food source.

4.3.4 Stop exploitation of exhausted food sources

Once a food source has been exhausted exploitation is stopped, the employed bee becomes a new free scout and onlookers are released. Exhaustion in this algorithm is defined by a certain number of iterations without improvement.

4.3.5 Memorize best food source

After every iteration the best food source is memorized by saving its employed bee (containing the best path to this food source). So even if the food source is exhausted in the future it will always be possible to remember the best path found so far.

4.3.6 Parameters

After the employed bee and the onlookers for the same source have found new routes to the source the employed bee will remember the best route, so the employed bee will always know the best known route to the food source. *Exhaustion* is an important mechanism to improve the exploration, because then new scouts are sent and new sources are discovered. There is a parameter called *limit* that describes after how many iterations without improvement a food source is abandoned. Tuning this parameter is important to ensure that the algorithm does not spend too much time in food sources that should be exhausted, or that it exhaust food sources too fast before they can be efficiently searched by the onlookers. In the original ABC algorithm the limit was set dynamically to the number of onlookers times the dimension of the problem.

In Table 4.1 the parameters as used in the simulations of [19] can be found. An interesting fact of these parameters is that scouts and employed bees are mentioned separately, and if every employed bee (and thus food source) gets exactly one onlooker and none of the food sources is exhausted this way there is still at least one scout moving randomly about. If this scout finds a solution better than at least one of the currently employed bees the currently worst employed bee is released (food source is exhausted) and replaced by the scouts solution.

Swarmsize	20
Number of scouts	5% – 10% of number of bees
Number of onlookers	50% of the swarm
Number of employed bees	50% of the swarm
limit	#onlookers · dimension

Table 4.1: Parameters of the ABC algorithm as used in [19]

Chapter 5

Algorithms

This chapter will be dedicated to describing the hybrid *Ant Colony Optimization-Artificial Bee Colony* algorithms used for the experiments. Two of the original Ant Colony Optimization algorithms already contained mechanisms to bind the amount of pheromones to an upper limit which makes it easier for exploration and exhaustion to be adapted and integrated. For this reason these two algorithms, *MMAS* and *ACS*, were chosen to be combined with the ABC algorithm. To complete this chapter an extra section is included that describes the algorithms that deal with making the original Ant Colony Optimization algorithms adapt to a dynamic environment.

5.1 Dynamic ACO

A dynamic TSP works the same as a normal TSP, but the parameters of the problem or the search space in which the algorithm is trying to find the best solution can change. The algorithm should be able to adapt to these changes, and not get stuck in places that used to be optima but are not optima anymore after the search space has changed. A dynamic TSP can be changed in three ways: nodes can be added, nodes can be removed and the length of an edge can change. If a change is great enough, or a number of changes happen at once, such that it is not possible to use at least a part of an already found tour it will be better to restart the entire algorithm. This is needed because existing information is incorrect and will only slow the algorithm down. Small changes that happen too soon will barely have an effect because even a simple Ant Colony Optimization algorithm should be able to adjust if it has not converged very far yet.

Because of the abovementioned reasons the type of dynamic TSP that is investigated contains only small changes, and these changes are performed when the first initial run of the algorithm has already converged to a solution. For every testproblem a number of iterations is determined at which the algorithm has usually converged to a good solution, and the first change does not happen before that number of iterations has passed.

5.1.1 Restoring a tour

A change in the environment can only occur between iterations. Hence only the best-so-far tour T^{bs} (Chapter 3.2.1) needs to be restored if a change happens, because that is the only solution that is remembered across multiple iterations. Since Artificial Bee Colony also remembers every employed ant across iterations the hybrid algorithms need to restore more ants to a valid solution. A change in the TSP problem can introduce new nodes, remove existing nodes or do both, and for both situations a mechanism must be described to make sure that the tour is still valid. It is difficult to ensure that the new valid tour is still as good as the old tour was, but the algorithm should take care of that by itself.

Deleted nodes

When a node is deleted it immediately invalidates the remembered tour, because the tour now contains a node that does not exist. Restoring the tour from deleted nodes is easy to do, because deleted nodes are just removed from the tour and the nodes on either side of it are now connected to each other.

Added nodes

Adding a node does not create the same problem deleting a node creates because the tour can still be followed from start to end. However for the TSP problem it does invalidate the tour since a valid TSP tour must include *all* nodes. The problem is knowing where to insert the newly added node. For this algorithm the choice was made to test the node between all existing concurrent nodes and then insert it at the position where the new solution would be best.

A more intricate solution might be to also shuffle some of the existing nodes to create an even better solution, or to perform a local optimization algorithm after deleting or adding, but because it is not guarenteed that that makes the algorithm perform better as a whole restoring the tour is kept as simple as possible.

5.2 ACO-ABC

Since both Ant Colony Optimization algorithms share a common ancestor and share a lot of functionality this section will first describe the basic hybrid algorithm between an Ant Colony Optimization algorithm and an Artificial Bee Colony algorithm. In Algorithm 3 the pseudocode for the hybrid algorithm is given, and comparing it to Algorithm 1 and Algorithm 2 one can observe how the original algorithms fit together to form this new hybrid algorithm.

In the following sections the lines that have different behaviour specific for the hybrid algorithm are explained. Some of the lines behave exactly the same as in the original algorithms and have already been explained: line 1 can be found in Chapter 3.1.1, line 2 in Chapter 3.1.4, line 6 in Chapter 3.1.5 and line 9 in Chapter 4.3.4.

Algorithm 3 Pseudo code for the hybrid Ant Colony Optimization-Artificial Bee Colony algorithm.

```
1: Create construction graph
2: Initialize pheromone values
3: while not stop-condition do
4:     Create solutions for free scouts
5:     Divide onlookers proportionally and create solutions
6:     Perform local search
7:     Find new employed ants & best-so-far tour
8:     Update pheromone values
9:     Stop exploitation of exhausted food sources
10: end while
```

5.2.1 Solution construction

The first step of every Ant Colony Optimization algorithm is creating solutions. In this hybrid algorithm solution construction goes in two parts, described in the following two sections for line 4 and line 5.

Free scouts

In the original Artificial Bee Colony algorithm solution construction the scouts would create a completely random solution, in the current version of the hybrid algorithm however solution construction for the scouts is performed the same as solution construction for Ant Colony Optimization (Chapter 3.1.2).

Onlookers

Once the scouts have created their solutions probabilities are calculated for the onlookers to use to determine which food source they will visit. Solution construction for the onlookers is different from that of the scouts because the onlookers have to take a route resembling that of the scout, but it is required to be different or it will have no effect. In graph theory however it is not as easy to define a neighbouring solution as it is in a continuous or discrete search space.

One class of defined neighbourhood structures for combinatorial optimization problems is the *k-exchange neighbourhood*.

Definition 6. *The* k-exchange neighbourhood *of a candidate solution s is the set of candidate solutions s' that can be obtained from s by exchanging k solution components.[14]*

The k-exchange neighbourhoods with $k = 2$ or $k = 3$ are often used in the local search of the algorithm [14], but since the hybrid algorithm also allows local search this does not generate neighbours with a difference great enough to really search the neighbourhood. The way neighbours are generated is based on the k-exchange neighbourhood, but cannot be called a k-exchange method. The algorithm for creating neighbours is shown in Algorithm 4. A neighbour is created by following the original tour from a random starting node in a random direction, but at every node there is a chance that the next node to visit is not the next one in the original tour but another random unvisited node. This *jump* *chance* (line 4) is set to 10% to ensure that even smaller TSP's (dimension 10-20) jump at least once, and there is a great chance that larger TSP's jump more than once. Because for every next node for which there is a choice in direction (both nodes before and after have not yet been visited) the direction is chosen by the greatest pheromone value we can describe this algorithm as: cut up the original tour in k places, randomly reorder the pieces and reverse the order of the nodes in some pieces. Where k is set close to 10% of the number of cities of the TSP.

Algorithm 4 Pseudo code to generate a neighbour of solution S.

```
1: Get random node from $S$ as starting point.
2: while unvisited nodes exist do
3:     Get random number in [0,1]
4:     if random number $<$ jump chance then
5:         Jump to an unvisited node using the proportional rule (see Equation 3.1).
6:     else
7:         Determine travel direction (forward or backward).
8:         Get next node in travel direction from $S$.
9:     end if
10: end while
```

5.2.2 Find new employed ants

In line 7 of Algorithm 3 a simple compare and replace operation is performed. The solution of every onlooker is compared to that of the currently employed ant, and if the solution of the onlooker is better they switch roles. This way we end up with a set of employed ants that are currently the best ants for their respective food sources. Then the solution of every employed ant is compared to T^{bs} and the best tour becomes the new T^{bs}.

5.2.3 Pheromone update

The mechanics of the pheromone update in line 8 can be found in Chapter 3.1.3, but as with Ant Colony System (Chapter 3.2.2) the update is only done on the edges of T^{bs}.

5.3 ACS-ABC

One of the two Ant Colony Optimization algorithms that have been used to make a hybrid algorithm with Artificial Bee Colony is the Ant Colony System. Equation 3.7 ensures that there is an upper bound to the amount of pheromones on an edge. The only difference from the hybrid algorithm described in Algorithm 3 is that the solution construction of the scouts is performed using the pseudo-random rule (Equation 3.6), instead of the proportional rule (Equation 3.1).

5.3.1 Parameters

In Table 5.1 the parameters for the ACS-ABC hybrid algorithm as used during the experiments can be found.

Parameter	Value
Local search	off
Node selection (scout)	Pseudorandom
Pseudorandom chance	0.9
Evaporation rate	0.1
Local evaporation rate	0.1
Number of iterations	1500
Pheromones to drop	1
Swarmsize	40
Number of scouts	10
Number of onlookers	20
Number of employed bees	10
Iterations without onlookers before exhaustion	5
Iterations without update before exhaustion	10

Table 5.1: Parameters used for ACS-ABC hybrid algorithm.

5.4 MMAS-ABC

The other algorithm used as basis for the hybrid algorithm is the $\mathcal{MM}$AS. This algorithm runs as an $\mathcal{MM}$AS algorithm with the changes in Chapter 5.2 incorporated. $\mathcal{MM}$AS has explicit boundaries on the pheromone values as shown in Chapter 3.2.3.

5.4.1 Parameters

In Table 5.2 the parameters for the MMAS-ABC hybrid algorithm as used during the experiments can be found.

Parameter	Value
Local search	off
Node selection (scout)	Proportional
Evaporation rate	0.1
Frequency of iteration best deposit	0.9
Iterations without update before reinitialization	50
Number of iterations	1500
Pheromones to drop	1
Swarmsize	40
Number of scouts	10
Number of onlookers	20
Number of employed bees	10
Iterations without onlookers before exhaustion	5
Iterations without update before exhaustion	10

Table 5.2: Parameters used for ACS-ABC hybrid algorithm.

Chapter 6

Experiments

In this chapter the experiments that have been run will be detailed and explained. Four TSP problem instances from the *TSPLib* library have been taken and made into *incremental dynamic* problem graphs. Then the two dynamic algorithms are compared to each other and to the Ant Colony Optimization algorithms they derived from. Furthermore some dynamic algorithms have been run more than once with a difference in the parameter set, these results indicate if a certain parameter would improve the total result or not.

6.1 TSPLib

The TSPLib library contains a large amount of Traveling Salesman Problem instances, with dimensions from only a few nodes to over 10000 nodes. This library is free for use and Traveling Salesman Problems from it are regularly used as benchmark problems. The optimum route, or its length, is given for a number of problems, including three of the chosen tests.

6.1.1 Traveling Salesman Problems

The problems taken from TSPLib are problems for the static Traveling Salesman Problem. To turn them into dynamic Traveling Salesman Problems a set of changes have to be described that tell the algorithm how to change the graph during its execution. These changes could be determined randomly each time the algorithm is executed, but then the algorithms could never be truly reproduced. A possible way to define the changes is a list of operations that should be executed in succession, which would ensure the problem is reproducable. The algorithm could start on the original graph, and then modify it, but then after the changes it is not known what the optimum would be. So for these experiments the algorithm starts with a subset of the graph, a couple of nodes have been taken out, and then the list of operations consists of re-adding the nodes in a certain order.

Because the algorithm first tries to solve a subset of the graph, continuously followed by graphs with one added node, this type of dynamic problem solving is also called *incremental problem solving.*

Problem	Dimension	Minimum
burma14	14	(30.88)
fri26	26	937
berlin52	52	7542
ch150	150	6528

Table 6.1: The chosen TSPLib problems with their dimension and published minimum (as integer). For burma14 no shortest route was published, but using an exact solver (Concorde, see Chapter 6.2) was calculated to be 30.88.

burma14

This problem instance was chosen because it is the smallest TSP included in the library. It is a 14 cities problem, with fourteen cities from Burma. At the start of every experiment three nodes were taken out that were reinserted every 50 iterations. Randomly selected and in order of reinsertion the following nodes had been taken out: 3, 1, 14.

fri26

The fri26 instance was chosen because it was one of the smallest problem instances for which an optimum was given, but also being around twice the size of burma14. This problem is about 26 cities from Fricker, and its shortest route as given by the TSPLib library is 937. For the dynamic graph four nodes were taken out from the beginning and were reinserted one by one every 50 iterations. Randomly selected and in order of reinsertion the following nodes had been taken out: 14, 4, 23, 26.

berlin52

This instance was chosen because it was (again) twice the size of the previous problem and had a given optimum. The origin of this problem is 52 locations in Berlin and its minimum is 7542.

ch150

Ch150 was chosen because it is used more often as a benchmarking problem, and is a 150 cities problem with shortest length 6528. In this instance ten nodes were taken out from the beginning and were reinserted one by one every 50 iterations after 300 iterations. Randomly selected and in order of reinsertion the following nodes had been taken out: 128 24 138 74 40 18 102 69 80 125.

6.2 Concorde

One of the fastest exact algorithms for the Traveling Salesman Problem is *concorde* [4],[18]. Concorde uses an advanced *branch-and-bound* algorithm with a *linear programming solver* to find the optimum, but although it is fast it is still exponential in its worst case complexity. In Table 6.2 the execution time of the concorde algorithm for a number of Traveling Salesman Problems with a wide range of dimensions is shown. This table shows that larger problems take longer to run, although the fluctuations in running time also indicate that besides the size of the problem its complexity also plays a vital role in determining the execution time.

Problem	Dimension	Time
burma14	14	0.02s
fri26	26	0.04s
berlin52	52	0.10s
ch150	150	0.82s
a280	280	2.82s
att532	532	48s
gr666	666	30s
rat783	783	16s
dsj1000	1000	126s
pr1002	1002	11s
pcb1173	1173	135s
d1291	1291	>1h
fl1400	1400	>1h
d2103	2103	>4h

Table 6.2: Concorde experiments, the processes of the results with a > sign were killed so no exact numbers are available.

6.2.1 Traveling Salesman Problems

burma14

Table 6.3 shows the results of running concorde for the burma14 TSP and its generated subgraphs, including their execution time and shortest tourlength. Since concorde is an exact algorithm these tourlengths are the actual optima and can be used to determine the quality of the algorithms compared on this problem.

#nodes	Tourlength	Time
11	28.4097	0.03s
12	28.9273	0.02s
13	29.3352	0.02s
14	30.8785	0.02s

Table 6.3: Results for burma14 with concorde algorithm

fri26

In Table 6.4 the execution time and the actual optima for *fri26* and its subgraphs can be found.

#nodes	Tourlength	Time
22	895	0.02s
23	896	0.02s
24	903	0.01s
25	902	0.04s
26	937	0.04s

Table 6.4: Results for fri26 with concorde algorithm

berlin52

Table 6.5 shows the results of running the concorde algorithm on the *berlin52* problem. No subgraphs of this graph were used during the experiments, therefor only the results of the complete graph are shown.

#nodes	Tourlength	Time
52	7542	0.1s

Table 6.5: Results for berlin52 with concorde algorithm

ch150

The tourlengths of the (sub)graphs and execution time of the concorde algorithm for the *ch150* problem can be found in Table 6.6.

#nodes	Tourlength	Time
140	6381	0.32s
141	6382	0.27s
142	6403	0.30s
143	6430	0.73s
144	6435	0.40s
145	6467	0.47s
146	6467	0.54s
147	6490	0.37s
148	6491	0.54s
149	6503	0.81s
150	6528	0.82s

Table 6.6: Results for ch150 with concorde algorithm

6.3 Traveling Salesman Problem experiments

The following sections describe the experiments as run on the four chosen problems.

burma14

The original six static algorithms are run on this problem, to be able to compare their respective behaviour before the algorithm is changed or the graph is made dynamic. These algorithms are: Ant System, Elitist Ant System, Rank-based

Ant System, $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System, Ant Colony System and Best-Worst Ant System. Furthermore the $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System was also run on the dynamic instance of the burma14 test to test the performance of a non-adapted algorithm on a dynamic problem.

fri26

As with burma14, this problem is run with the original six algorithms: Ant System, Elitist Ant System, Rank-based Ant System, $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System, Ant Colony System and Best-Worst Ant System.

berlin52

This problem is run with five of the original algorithms (Ant System, Elitist Ant System, Rank-based Ant System, $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System and Ant Colony System), to be able to compare the results of the two previous (smaller) tests with a slightly larger test.

ch150

This problem is run with five of the original algorithms: Ant System, Elitist Ant System, Rank-based Ant System, $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System and Ant Colony System. Also the two types of local search algorithms (first-improvement, best-improvement) are run on this problem to see if they should be enabled for all testing. Furthermore the two dynamic algorithms ($\mathcal{MM}$AS-ABC, ACO-ABC) are run on this problem.

6.4 Implementation

The implementation of the algorithms is written in GNU C++, and compiled using GCC 4.4.3. The programs were run on a computer with a dualcore Intel processor on 1.3Ghz with 3GB of DDR3 RAM, running Ubuntu 10.04 64-bit with linux kernel 2.6.32.

Chapter 7

Results

7.1 Parameter results

7.1.1 Local Search

The two different types of local search algorithms have been run on the ch150 problem, and the results for tourlength and execution time can be found in Table 7.1. In the following paragraphs a possible explanation is searched for, but according to these results the algorithm should not be included with a 2-opt local search algorithm.

	Average result	Average run-time
AS	7046.8	11min 25s
AS 2-opt (FI)	7143.1	16min 28s
AS 2-opt (BI)	7086.7	3h 49min 23s
	Minimum run-time	**Maximum run-time**
AS	11min 7s	11min 33s
AS 2-opt (FI)	16min 13s	16min 46s
AS 2-opt (BI)	3h 46min 40s	3h 51min 10s

Table 7.1: Results of various Ant System algorithms on ch150 TSPLib instance (repeated 10 times).

2-opt first-improvement

All types of local search algorithms generate some extra execution time, but the *first-improvement 2-opt* (2-opt (fi)) algorithm generates a much smaller amount of execution time compared to the best-improvement 2-opt (2-opt (bi)), even though it still increases execution time by 30%. In Figure 7.1 the results of the five original algorithms with 2-opt (fi) can be found, and in Figure 7.6 the results of the algorithms without any local search algorithm are reported. Comparing these two figures show that the 2-opt (fi) algorithm increases the performance of ASrank and ACS, but decreases the performance of the other algorithms.

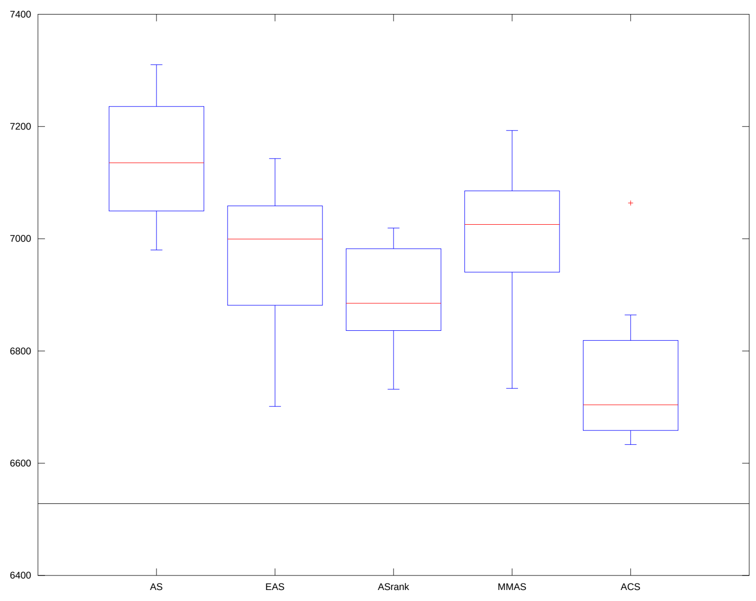

Figure 7.1: Original algorithms using first-improvement 2-opt local search on ch150 TSPLib instance.

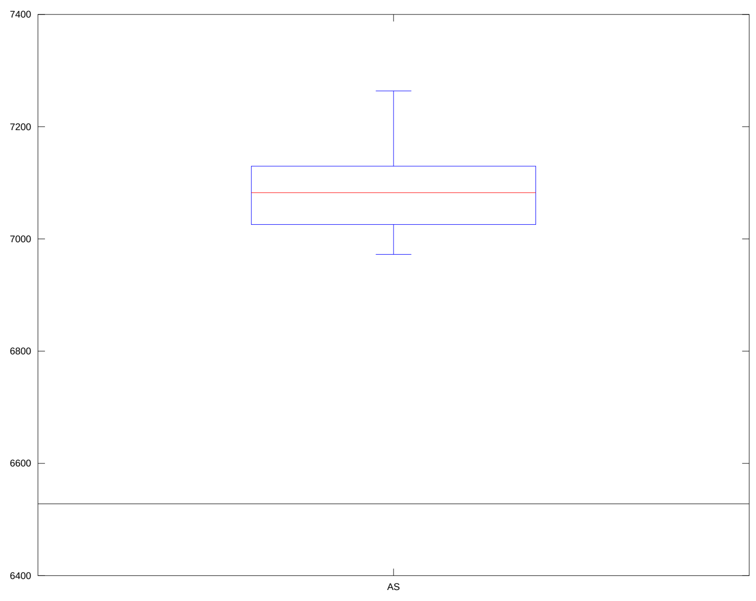

Figure 7.2: Ant System algorithm using best-improvement 2-opt local search on ch150 TSPLib instance.

2-opt best-improvement

Initially both types of local search would have been compared side by side, but after running a set of experiments (10 runs of the Ant System algorithm) it was shown that the algorithms running with the best-improvement solution function (2-opt (bi)) (see Chapter 3.1.5) have a drastic increase in run-time. Without further optimization of and improvements on the best-improvement function it was clear that it would take too long and it was therefor decided not to use this local search procedure. In Figure 7.2 the results are shown of the Ant System algorithm with 2-opt (bi).

7.1.2 Employed ants depositing

To increase the performance of the algorithms it was investigated if allowing all the employed ants to deposit pheromones and the best ant to deposit extra pheromones would generate better results. The reason to allow employed ants to deposit would be that they would increase the pheromones in other promising areas for future exploration, and to make sure the best ant would act different from the employed ants it would deposit twice its pheromones. Running the $\mathcal{MM}$AS-ABC algorithm with these parameters showed a significant decrease in quality of the results after a couple of trials (all results outside the outer bounds of the original results). For now there was decided not to continue researching in this direction because these parameters unbalanced the exploration and exploitation (see Chapter 3.1.7), and the purpose of this research is not about optimizing this algorithm but researching its viability.

7.2 Traveling Salesman Problems

The following sections describe the results obtained by running the algorithm on the different TSPLib tests.

burma14

Figure 7.3 shows the results of the six original algorithms on the burma14 test. The minimum of the burma14 test is given in Table 6.3 and is 30.88. The plots show that all of the algorithms find the optimum at least once, although for AS the minimum is only found in the lower quartile of the results. Interestingly almost all the results of ACS find the minimum, and those that don't are considered outliers. Furthermore the plots show no defined results between the minimum and the two possible values around 31.2, indicating a gap between the minimum and the nearest other solution, and only the Ant System has trouble bridging this gap.

fri26

In Figure 7.4 the plots for the results of the algorithms obtained from running on the fri26 test are shown. The given minimum for fri26 is 937, and again all the algorithms are able to find this minimum. However $\mathcal{MM}$AS and BWAS do not always find this minimum, but $\mathcal{MM}$AS about half the time and BWAS even less. Normally that would not be something worth mentioning, but in this

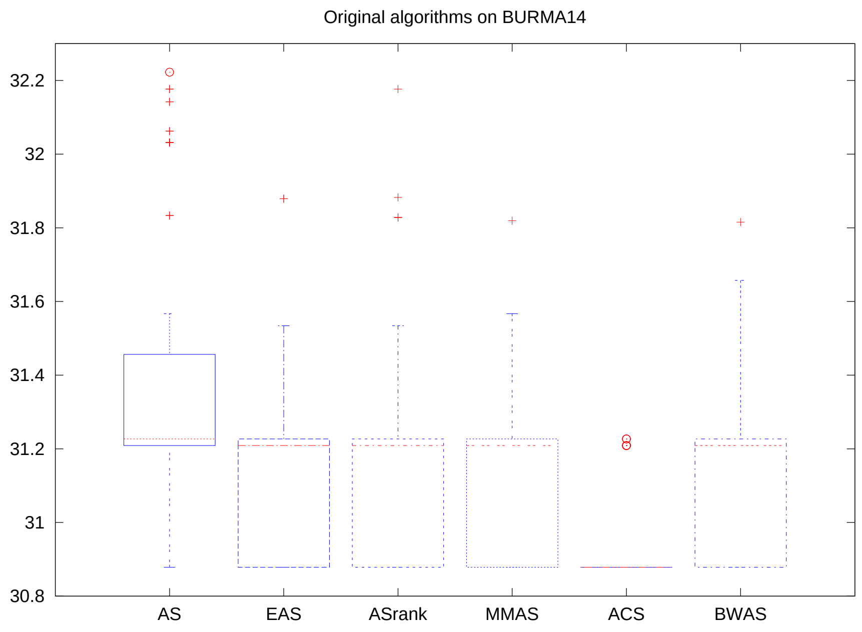

Figure 7.3: Results of the six basic algorithms on the burma14 instance

case even AS, EAS and ASrank almost always find this minimum. This shows that depending upon the complexity of the problem, even for the same type of problem, it is useful to investigate which ACO algorithm to use because there is no algorithm that is always better that the others.

berlin52

The performance of the five algorithms performed on the berlin52 test almost seems switched, since for this test the $\mathcal{MM}$AS algorithm almost always approaches the minimum while the other algorithms do not. Another interesting result is that ACS is now one of the two worst algorithms (the other is ASrank), while in the previous two tests it was the best algorithm in both cases. However none of the algorithms find the minimum, the obtained results approach 7544 but never get below it, while the given minimum is 7542 (see Table 6.5).

ch150

The results for the five algorithms on ch150 can be found in Figure 7.6, with the horizontal black line indicating the minimum value (see Table 6.6). This problem is already more difficult for this implementation of the algorithms with the default set of parameters, since none of the algorithms actually find the minimum. For a problem of this size it shows that the more simple algorithms (AS, EAS and ASrank) are clearly outperformed by the more advanced algorithms ($\mathcal{MM}$AS and ACS). But for this problem ACS seems to perform better, since almost all its results are in the same range as the lower quartile of $\mathcal{MM}$AS.

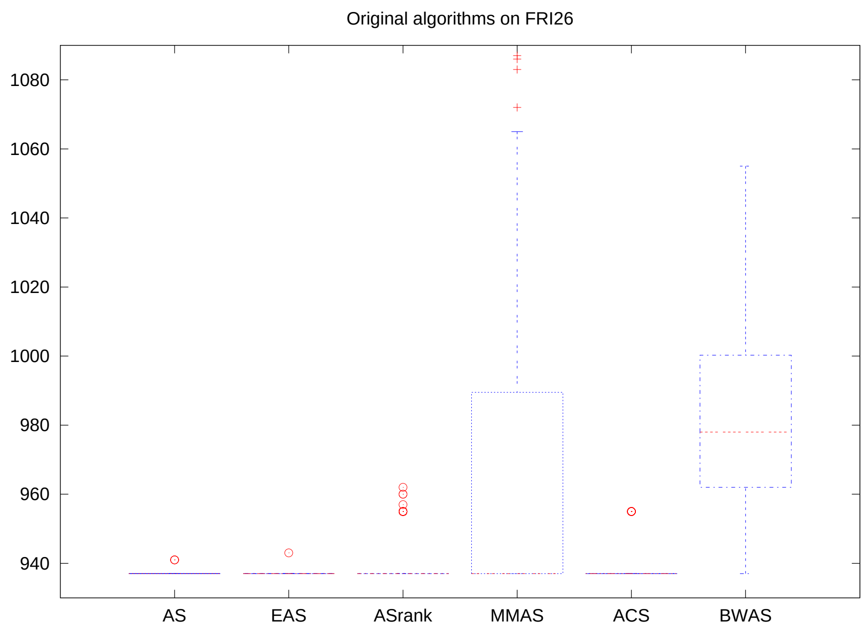

Figure 7.4: Results of the six basic algorithms on the fri26 instance

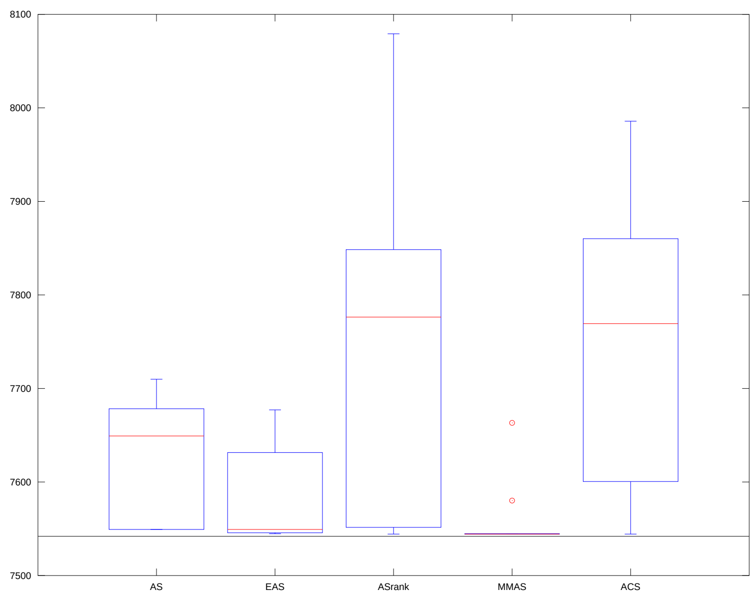

Figure 7.5: Results of five basic algorithms on the berlin52 instance

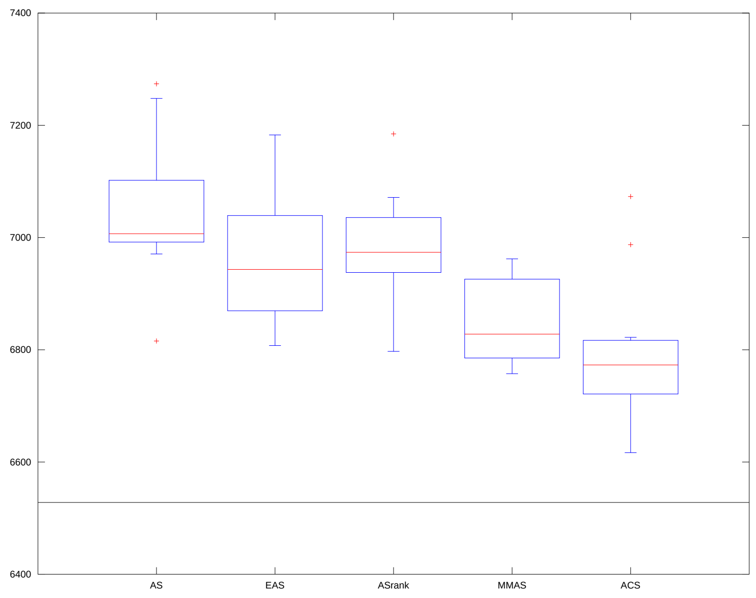

Figure 7.6: Five original algorithms on ch150 TSPLib instance.

7.3 Dynamic Traveling Salesman Problem

7.3.1 Unadapted $\mathcal{MM}$AS on burma14

Figure 7.7 shows the results obtained by running the original $\mathcal{MM}$AS algorithm on the dynamic burma14 test. The black horizontal lines are the calculated minima as given by Concorde (see Table 6.3), where the bottom-most line is for the set of 11 nodes and then in order going up. These results illustrate that this unadapted algorithm is not good in coping with changes in the testset. Every next iteration the distance from the minimum (in tourlength) becomes greater, almost as if the algorithm is diverging.

7.3.2 $\mathcal{MM}$AS-ABC on ch150

In Figure 7.8 the results of the $\mathcal{MM}$AS-ABC algorithm on ch150 are shown. To keep the graph clear only the optimum of the complete graph is shown as a horizontal black line at 6528. The other optima can be found in Table 6.6. Although the graph now seems to diverge from the black line it should be remembered this line is the optimum only for the last iteration and the other iterations actually have lower minima. The graph shows mixed results for the different iterations, iterations 700 and 1200 seem to perform exceptionally well (even finding lower minima than the iterations before them on average), but iterations 900, 1000, 1400 and 1500 perform worse then expected with the average result much higher than its predecessor.

On average however the calculated minimum of the graph increases by

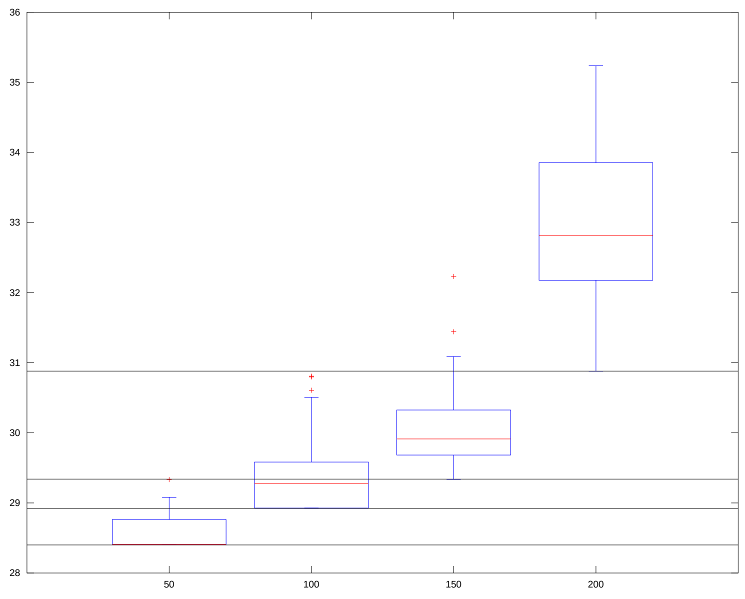

Figure 7.7: Original $\mathcal{MM}$AS on the burma14 testset, beginning with 11 nodes and adding 1 node every 50 iterations. The horizontal lines are the minima as calculated by Concorde.

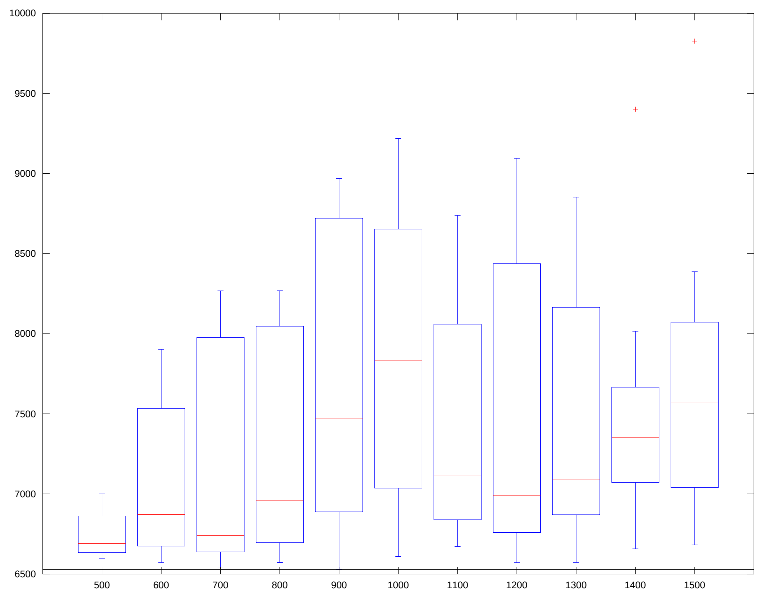

Figure 7.8: Plots for the intermediate and end results for the $\mathcal{MM}$AS-ABC algorithm on ch150. The horizontal line is the minimum for the complete graph (optimum for iteration 1500).

147 (6528 – 6381, see Table 6.6), where the average minimum of the result in Figure 7.8 seem to increase with about 600 (try to draw a line through the averages and then calculate the difference, ±7300 – ±6700).

7.3.3 ACS-ABC on ch150

Figure 7.9 shows the results of the ACS-ABC algorithm on ch150. Again, to keep the graph clear only the optimum for the complete graph, iteration 1500, is shown. These results are a lot better than the results of the $\mathcal{MM}$AS-ABC algorithm, neither fluctuating nor diverging as much as $\mathcal{MM}$AS-ABC. Even though the iterations slowly increase the average minimum after 10 additions with ±100 (±6770 – ±6670), this is less than the calculated minimum increases (147) and can be seen as convergence even with the dynamic behaviour of adding nodes. Another point worth mentioning is the Y-scale of Figure 7.9, where the Y-scale of this entire graph is $\frac{1}{7}^{th}$ that of Figure 7.8.

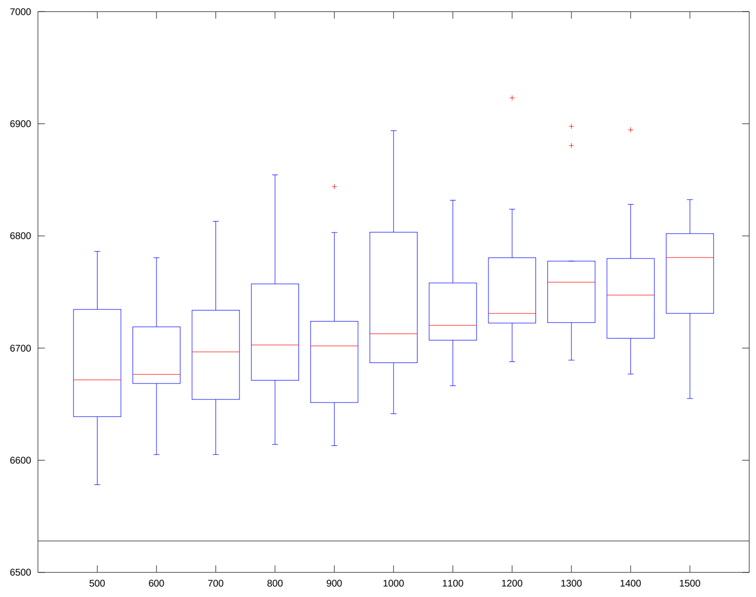

Figure 7.9: Plots for the intermediate and end results for the ACS-ABC algorithm on ch150. The horizontal line is the minimum for the complete graph (optimum for iteration 1500).

Chapter 8

Conclusion

In Chapter 7 the results of this research were described and analyzed, this chapter however will try to draw a more global conclusion from these results. Chapter 7.2 shows us that for different problems different Ant Colony Optimization algorithms should be used, there is not a single algorithm that is always better than the other algorithms. However as a general rule it can be concluded that for any problem either $\mathcal{MM}$AS or ACS will be among the best performing ACO algorithms (see Figures 7.3, 7.4, 7.5 and 7.6).

The results in Chapter 7.3.1 are used as an argument to show that the normal algorithms cannot cope with dynamically changing graphs, and although this was only tested on the smallest available problem the divergence shown is obvious enough that this same argument should hold for the other problems. Following this Chapter 7.3.2 and Chapter 7.3.3 analyze the results of the hybridized algorithms. As is shown in Figure 7.6 the ACS algorithm performs better than the $\mathcal{MM}$AS algorithm, and this is also valid for their hybridized counterparts. While the results of the $\mathcal{MM}$AS-ABC algorithm (Figure 7.8) are bad enough to conclude that this hybridization is inefficient, the results of the ACS-ABC algorithm (Figure 7.9) are hopeful that this hybridization can be used for other problems because it converges despite the changing graph. Regretfully neither the static algorithms on the normal graphs, nor the dynamic algorithms on the dynamic graphs reach the actual minimum for ch150, therefor no conclusion can be drawn about the speed at which changes are allowed before the algorithms cannot keep up and start diverging.

But the final conclusion that can be drawn is that the hybrid ACS-ABC algorithm on the dynamic ch150 problem performs as well as the ACS algorithm on the static ch150 problem, and much better than the hybrid $\mathcal{MM}$AS-ABC algorithm. Even though there are no real exceptional results neither has the research failed to deliver what it set out to find, an algorithm that is better in coping with dynamic traveling salesman problems.

Acknowledgements

I especially would like to thank Prof. Dr. T.H.W. Bäck, for his patience in waiting for this projects completion, his practical advice and the discussions we had about on which area to focus. But my thanks also go to Dr. M.T.M. Emmerich, as second reader of my thesis. Furthermore a general thanks to the other master students and especially Marijn Swenne, who kept me motivated when I wasn't.

Bibliography

[1] C. Blum, A. Roli, and M. Dorigo. HC-ACO: The Hyper-Cube Framework for Ant Colony Optimization. In *MIC'2001 - 4th Metaheuristics International Conference*, pages 399–403, 2001.

[2] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[3] B. Bullnheimer, R. F. Hartl, and C. Strauß. A New Rank Based Version of the Ant System - A Computational Study. Technical Report Tech.Rep., Institute of Management Science, University of Viena, 1997.

[4] Concorde. `www.tsp.gatech.edu/concorde/`, march 2012.

[5] O. Cordón, I. F. de Viana, F. Herrera, and L. Moreno. A new ACO model integrating evolutionary computation concepts: The best-worst Ant System. In *Abstract proceedings of ANTS 2000 – From Ant Colonies to Artificial Ants: Second International Workshop on Ant Algorithms*, pages 22–29, 2000.

[6] J. Deneubourg and S. Goss. Collective patterns and decision making. *Ethology Ecology & Evolution*, 1:295–311, 1989.

[7] M. Dorigo. *Ottimizzazione, apprendimento automatico, ed algoritmi basati su metafora naturale (Optimization, learning and natural algorithms)*. PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, 1992.

[8] M. Dorigo, M. Birattari, and T. Stützle. Ant Colony Optimization: Artificial Ants as a Computational Intelligence Technique. Technical report, IRIDIA, Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle, Université Libre de Bruxelles, 2006.

[9] M. Dorigo and C. Blum. Ant colony optimization theory: A survey. *Theoretical Computer Science*, 344:243–278, 2005.

[10] M. Dorigo, A. Colorni, and V. Maniezzo. The Ant System: An Autocatalytic Optimizing Process. Technical Report 91-016 Revised, Université Libre de Bruxelles, Milano, Italy, 1991.

[11] M. Dorigo and G. Di Caro. The Ant Colony Optimization Meta-Heuristic. Technical report, Université Libre de Bruxelles, 1999.

[12] M. Dorigo and L. M. Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1:53–66, 1997.

[13] 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*, 26:1–13, 1996.

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

[15] L. M. Gambardella and M. Dorigo. Ant-Q: A reinforcement learning approach to the traveling salesman problem. In *Proceedings of the Twelfth International Conference on Machine Learning (ML-95)*, pages 252–260, 1995.

[16] J. Goldstein. Emergence as a Construct: History and Issues. *Emergence, Complexity and Organization*, 1:49–72, 1999.

[17] P.-P. Grassé. La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. La théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6:41–80, 1959.

[18] Hahsler, Michael, Hornik, and Kurt. TSP - Infrastructure for the Traveling Salesperson Problem. *Journal of Statistical Software*, 23:1–21, 2007.

[19] D. Karaboga. An Idea based on Honey Bee Swarm for Numerical Optimization. Technical report, Engineering Faculty, Erciyes University, Kayseri, Türkiye, 2005.

[20] V. Maniezzo. Exact and Approximate Nondeterministic Tree-Search for the Quadratic Assignment Problem. *INFORMS Journal on Computing*, 11:358–369, 1999.

[21] T. Stützle and M. Dorigo. A short convergence proof for a class of ACO algorithms. *IEEE Transactions on Evolutionary Computation*, 6:358–365, 2002.

[22] T. Stützle and H. Hoos. Improving the Ant System: A Detailed Report on the $\mathcal{MAXMIN}$ Ant System. Technical Report AIDA-96-12, FG Intellektik, FB Informatik, TU Darmstadt, Germany, 1996.

[23] Stigmergy. `http://en.wikipedia.org/wiki/Stigmergy`, november 2009.

[24] Convergence. `http://en.wikipedia.org/wiki/Convergence`, april 2010.

[25] Emergence. `http://en.wikipedia.org/wiki/Emergence`, march 2012.