Natural Algorithms for Optimisation Problems
Final Year Project Report

Dorian Gaertner
dg00@doc.ic.ac.uk

Project supervisor: Prof Keith L. Clark
Second marker: Dr Theodore Hong

June 20, 2004

Abstract

Many computational techniques borrow ideas from nature in one way or another. Neural networks imitate the structure of our human brain, genetic algorithms simulate evolution and swarms of insects inspired algorithms for stochastic combinatorial optimisation. These techniques are characterised by inherent parallelism, adaptivity, positive feedback and some element of randomness.

This report details the investigations into certain features of one such technique: the *Ant System* meta-heuristic proposed by Dorigo. The algorithm is applied to several variations of the well-known Travelling Salesman Problem, including asymmetric and dynamic ones. Furthermore, different approaches to parallelise the algorithm are investigated.

Good parameters are needed to instantiate the generic algorithm efficiently and research to find optimal parameter settings has been conducted in the course of this project. An ontology to classify instances of the TSP is introduced and a thorough analysis of the dependency between parameters and classes of problems is presented. During this analysis, optimal parameters have been found for a given problem instance that led to the discovery of a new shortest tour for this instance.

A novel algorithm, the *Genetically Modified Ant System(GMAS)*, is then proposed that employs ideas from the field of genetic algorithms to eliminate the need to set parameters for a given problem explicitly. An implementation is described and the new system is evaluated with respect to the standard *Ant System* and also compared to other more specialised heuristics.

The project source code and executables together with all documentation, meeting minutes, outsourcing and final report can be found at:

www.doc.ic.ac.uk/∼dg00

Acknowledgements

Firstly, I would like to thank my supervisor, Prof. Keith Clark, for his valuable advice and help throughout the project.

I would also like to thank Dr. Theodore Hong for agreeing to be my second marker.

I am indebted to Dr. Jonathan Dale for igniting my research fever and for his detailed feedback on the draft versions of this report.

Furthermore, I would like to thank Dr. Vicky Mak who researched the RATSP at the University of Melbourne, for providing problem instances and advice on this TSP variant.

Finally, I wish to express my heartfelt gratitude to my parents and grandparents. Without your support this would not have been possible for me.

Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Parameters and genes . . . 2
- 1.3 Parallelisation . . . 2
- 1.4 Objectives . . . 3
- 1.5 Applications . . . 4

2 Background **5**
- 2.1 Swarm intelligence . . . 7
- 2.2 Travelling Salesman Problem . . . 12
  - 2.2.1 Description . . . 12
  - 2.2.2 Complexity . . . 13
  - 2.2.3 Relevance . . . 15
  - 2.2.4 Approximation algorithms . . . 17
  - 2.2.5 Variations . . . 21
- 2.3 Ant System . . . 23
  - 2.3.1 Entomological background . . . 23
  - 2.3.2 Algorithm . . . 24
  - 2.3.3 Ant Colony Optimisation . . . 26
  - 2.3.4 Problems and opportunities . . . 27
- 2.4 Genetic Algorithms . . . 28
  - 2.4.1 Basic structure . . . 30
  - 2.4.2 Encodings . . . 32
  - 2.4.3 Selection methods . . . 32
  - 2.4.4 Genetic operators . . . 35
  - 2.4.5 Example . . . 37
- 2.5 Summary . . . 38

3 Software architecture **41**
- 3.1 Methodology . . . 41
- 3.2 Multi-threaded single processor version . . . 42
- 3.3 Parallel algorithm on multiple processors . . . 46
- 3.4 Asymmetric variant . . . 49
- 3.5 Different kinds of dynamic behaviour . . . 51
- 3.6 Summary . . . 52

4 On the parallelisation of the ant algorithm 53
4.1 Experimental setting . . . 54
4.2 Separating environment and ant processes . . . 55
4.3 Distributing ants over multiple machines . . . 56
4.4 Reduced communication . . . 58
4.5 Sub-colonies . . . 61
4.6 Aggregated results . . . 63
4.7 Summary . . . 64

5 On optimal parameters 65
5.1 Description of parameters . . . 66
5.2 Classification for TSP instances . . . 68
5.3 Optimal parameters for instance Oliver30 . . . 71
5.4 New optimal solution for instance Eilon50 . . . 75
5.5 Generalised results . . . 76
5.6 Summary . . . 78

6 On genetically modified ants 79
6.1 Motivation . . . 79
6.2 Implementation . . . 80
6.2.1 Encoding . . . 81
6.2.2 Selection . . . 82
6.2.3 Recombination . . . 84
6.2.4 Mutation . . . 84
6.2.5 Generation gap . . . 85
6.3 Stagnation problems . . . 86
6.3.1 Scaling methods . . . 88
6.4 Comparison with the standard ant system . . . 91
6.5 Summary . . . 92

7 Conclusions 93
7.1 Contributions . . . 93
7.2 Future work . . . 94

A Game of Life 97

B Swarm platforms and modelling environments 99
B.1 StarLogo . . . 101
B.2 NetLogo . . . 102
B.3 AgentSheets . . . 103
B.4 Swarm . . . 104
B.5 Ascape . . . 105
B.6 RePast with SimBuilder . . . 106

C Implementation languages and technologies 107
C.1 April . . . 108
C.2 InterAgent Communication Model . . . 112
C.3 DialoX . . . 113

D Oliver30 and other instance data 117

E Parallelisation data **119**
E.1 Local optimiser 119
E.2 Splitting the environment from the ant processes 120
E.3 Distributing the ant processes 121
E.4 Reduced communication 122
E.5 Subcolonies 123

F Sample execution trace **124**

List of Figures

1.1 Improvement for one of the parallelisation strategies 3
1.2 Screenshot of the ant system for the asymmetric TSP 4

2.1 Ants following a pheromone trail 8
2.2 Swarming robots 10
2.3 Travelling Salesman tours for a 10-city problem. 13
2.4 Complexity classes 14
2.5 Simulated Annealing trying to find the global optimum 19
2.6 Execution flow for tabu search 20
2.7 Ant experiment 24
2.8 Ants and the Travelling Salesman Problem 26
2.9 Evolution of homo habilis to homo sapiens 29
2.10 Flow diagram of a basic genetic algorithm 31
2.11 Roulette Wheel and Stochastic Universal Sampling 34

3.1 Ant algorithm architecture 43
3.2 Parallel architecture - Experiment 1 46
3.3 Parallel architecture - Experiment 2 47
3.4 Parallel architecture - Experiment 3 48
3.5 Symmetric vs. asymmetric problem instance 49
3.6 Screenshot of the ant application for the Asymmetric TSP 50

4.1 Time taken for 100 iterations given a number of processors . . . 57
4.2 Time taken for 100 iterations given a number of local updates . . 59
4.3 Quality of solutions given a number of local updates 60
4.4 Tradeoff between execution time and accuracy 60
4.5 Comparison of performance with and without sub-colony threads 62

5.1 Two identical problem instances of the TSP modulo scaling . . . 70
5.2 Two different problem instances of the TSP 70
5.3 Screenshot of the new best solution for Eilon50 77

6.1 Family tree of ants 79
6.2 Simplified design of the Genetically Modified Ant System 80
6.3 DNA double helix 81
6.4 Selection and recombination for the GMAS 83
6.5 Mating ants 84
6.6 Three stage steady-state approach 85
6.7 Sigma scaling 89

A.1 Screenshot of the GameOfLife application 97

B.1 A very crowded screenshot of the MadKit platform 100
B.2 Termites gathering wood chips simulated with StarLogo 101
B.3 A 'school of fish' simulation in AgentSheets 103
B.4 A simulation of the Prisoner's Dilemma in Ascape 105

C.1 The message flow between processes on different host machines . 109
C.2 Example of a DialoX window 115

D.1 Screenshot of the best solution for Oliver30 117

List of Tables

2.1 Comparison of terms used in computer science and biology . . . 28
2.2 Comparison between GAs and ant system from [DMC96] 29
2.3 First generation of example genetic algorithm 37
2.4 Second generation of example genetic algorithm 38

4.1 Parameters used for the parallelisation experiments 54
4.2 Influence of local search on execution time 55
4.3 Separating ants and environment processes 56
4.4 Results of reduced communication experiments 58
4.5 Advantage of using sub-colonies 61
4.6 Summary of different parallelisation strategies 63

5.1 The constant parameters 68
5.2 The variable parameters 68
5.3 Metrics for two scaled problem instances 70
5.4 Metrics for two different problem instances 71
5.5 Metrics for a variety of standard problem instances 71
5.6 Constant parameters for the Oliver30 instance 72
5.7 Variable parameters and their ranges for Oliver30 72
5.8 Parameters for optimal solutions for Oliver30 73
5.9 Parameters for very good solutions for Oliver30 74
5.10 Parameters for good solutions for Oliver30 74

6.1 GMAS Experiment 1 86
6.2 GMAS Experiment 2 87
6.3 Results of the GMAS with sigma scaling for different k 90
6.4 Results of the improved GMAS 91

D.1 Coordinates of cities for three famous problem instances 118

E.1 Influence of local search on execution time 119
E.2 Running ants and environment on the same machine 120
E.3 Running ants and environment on separate machines 120
E.4 Running ant process on several machines 121
E.5 Results of reduced communication experiments 122
E.6 Using sub-colony communication one-way 123
E.7 Two-way sub-colony communication and ants on two machines . 123
E.8 Two-way sub-colony communication and ants on three machines 123
E.9 Two-way sub-colony communication and ants on four machines . 123

Chapter 1

Introduction

1.1 Overview

Social insects, like ants, bees or termites are generally conceived as simple, non-intelligent animals. However, collectively they exhibit impressive problem-solving skills. Inspired by these insects research in the past decade has led to some fascinating progress in the field of *natural algorithms*.

These algorithms imitate nature in one way or another. Neural networks imitate the structure of our human brain and genetic algorithms simulate evolution to name just two. They are characterised by inherent parallelism, adaptivity, positive feedback and some element of randomness.

Research led by Marco Dorigo produced a new member of this class of algorithms: the 'ant system' algorithm. Here, the way these insects find the shortest path from a food source to their nest is imitated in an attempt to solve hard combinatorial optimisation problems.

The research undertaken during the course of this project is motivated by the opportunities and shortcomings of this natural algorithm. Certain aspects, like optimal parameterisation, have not been studied sufficiently and others, like the inherent parallelism have not been exploited enough, leaving much room for further investigations.

While the ant algorithm has been applied to a variety of optimisation problems there are still important variants of the Travelling Salesman Problem(TSP) that are not covered.The TSP describes the problem of finding a closed tour through a given set of cities where every city is visited by the salesman exactly once. Especially the dynamic version with changing city landscapes promises good result when tackled with this new algorithm.

The ant system algorithm, the TSP and genetic algorithms are introduced in Chapter 2 to set the scene for the remaining chapters in this report. Chapter 3 then describes the various implementations before the experimental results are presented.

1.2 Parameters and genes

This project will try and disprove Dorigo's claim that intuition, experience and luck are required to find the right rules and parameters for a given problem. The widely applicable TSP was chosen as the problem to be considered in detail mainly because it has been analysed by many researchers in the past and benchmarks of other algorithms exist.

The standard ant system proposed by Dorigo in [DMC96] has to be parameterised to control the way solutions are generated and to prune the search space efficiently. All the sensible, potential settings for most of these parameters for a well-known thirty-city problem instance were analysed exhaustively using six weeks of high performance computing power.

The optimal parameters thus found are presented in Chapter 5 and tested on other problem instances to determine whether they are problem independent.

While experimenting with a well-known fifty-city problem instance, the algorithm with my optimised parameters found a new shortest tour for this problem that was not known before. This result has been reported to TSPLIB - the keeper of all famous TSP instances and their shortest known solutions.

Having found a problem dependency between certain parameters and classes of problems, an ontology of problem instances was developed. It was hoped that this would help to specify correlations between certain types of problems with respect to the optimal parameters. Some weak correlations were found but not all parameters had statistically relevant dependencies between them and attributes of the ontology.

A new hybrid algorithm is therefore proposed in Chapter 6 combining ideas from the ant system with methods from genetic programming. Parameters are encoded as genes and evolved over time in order to breed ants that can find high quality solutions to the TSP. This allowed the algorithm to run without any manual parameterisation.

This revolutionary algorithm is tested on several well-known problem instances as well as some randomly composed ones. Its performance is then compared to that of the standard ant system and other heuristics from the literature. This work is to be submitted for the Fifth International Workshop on Ant Colony Optimisation and Swarm Intelligence.

1.3 Parallelisation

A secondary aim of this project was to investigate how ideas from parallel computing could be incorporated into the algorithm to speed up the computation. Several existing and novel strategies have been employed. These range from separating ants and environment onto different machines to using sub-colonies with sparse inter-colony communication. The results of these experiments can be found in Chapter 4.

An increase in performance of more than 90% over the standard ant system running on one machine was finally found using just six machines. Furthermore, the optimal ratio between local computation and inter-process communication was found thus limiting the number of machines needed for the fastest version of the algorithm.

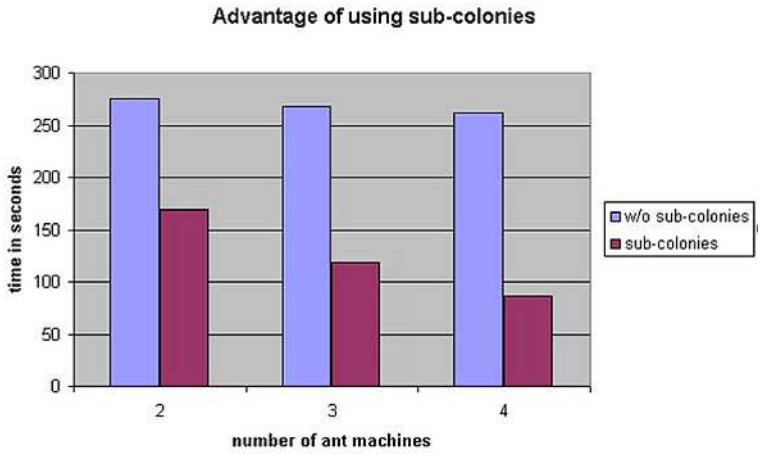


Figure 1.1: Improvement for one of the parallelisation strategies

1.4 Objectives

The aims of this research can be stated as follows:

- To find the optimal parameters for an ant algorithm for a given problem instance of the Travelling Salesman Problem.
- To find correlations between certain classes of problem instances and the optimal parameters associated with them or show that such correlations do not exist.
- To find a way to eliminate the need to find optimal parameters.
- To exploit the inherent parallelism of the ant algorithm as much as possible.
- To investigate the applicability of the ant algorithm to variations of the TSP.
- To visualise the workings of the ant algorithm.

1.5 Applications

A side effect of the research undertaken for this project was the development of several applications visualising different variants of the Travelling Salesman Problem. While the original description of the TSP used intercity distances and hence Euclidean geometry other metrics are possible, too. Using travel time rather than distance one can eliminate the Euclidean constraint and arrive at the so-called Asymmetric Travelling Salesman Problem (ATSP).

Taking this metaphor one step further the travel time between cities could change dynamically e.g. due to a traffic jam on the highway that connects two cities. This additional complexity is found in instances of the so-called Dynamic Travelling Salesman Problem (DTSP).

All three variants (Euclidean, asymmetric and dynamic) were visualised by applications implemented using the agent programming language APRIL and the HCI server DialoX.

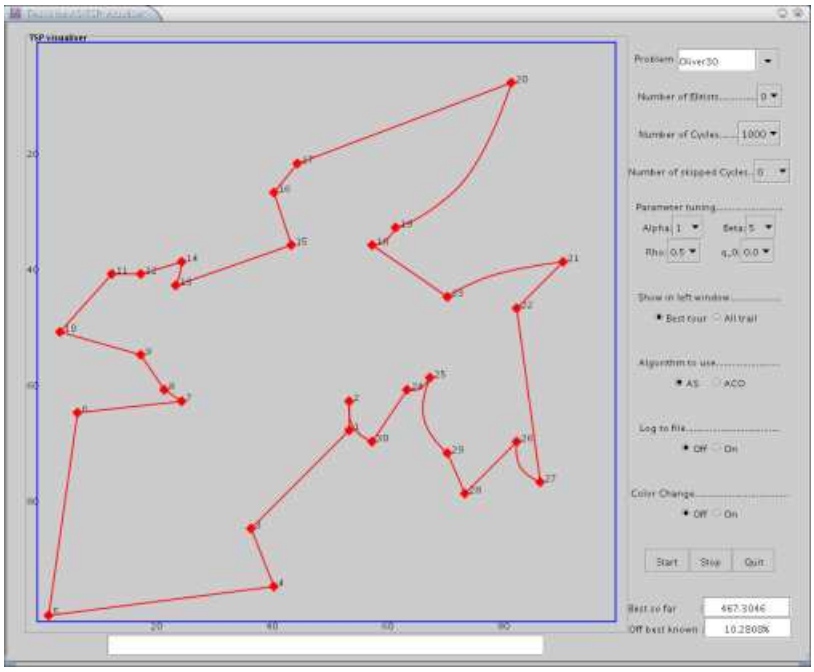


Figure 1.2: Screenshot of the ant system for the asymmetric TSP

Chapter 2

Background

This background chapter will provide the necessary knowledge needed to comprehend the remainder of this report. In particular it will introduce the following concepts:

- swarm intelligence
- the Travelling Salesman Problem
- ant system algorithms
- genetic algorithms

In the late 1940s John Von Neumann and his friend Stanislaw Ulam wanted to design a machine capable of reproduction. Finding it impossible to actually build such a machine they created it 'on paper' and thereby invented what was later termed 'cellular automaton'. The implication of this work is revolutionary. The basis of life is information.

This discovery led to fascinating new research. John Horton Conway invented a game with very simple rules and called it 'Life'[1]. It is a two-dimensional version of the 'cellular automaton' where the state of each cell depends on the state of its neighbours. Less than two or more than three 'live' neighbouring cells cause a cell to die and exactly three 'live' neighbours cause a dead cell to be reborn. These simple rules lead to complex implications. Astonishing patterns like oscillators, gliders and never-ending growths instances emerge from seemingly simple start states of the board.

Craig Reynolds used this notion of artificial life in 1986 to make a computer model of animated bird motion, called 'boids'. Similar to the cells in the 'Game of Life' birds in a flock base their behaviour only on neighbouring birds. Their manoeuvres are based on *separation* (don't get to close to other birds), *alignment*

[1]The author of this report implemented this 'Game of Life' using April and DialoX as a preparation for this project. Screenshots can be found in Appendix A

(steer towards the average direction your local flock-mates are heading into) and *cohesion* (move towards average position of local flock-mates).

Many people believed that birds in a swarm follow a leader. However, with his 'boids' simulation Reynolds could prove that flocks of birds are 'decentralised systems' characterised by the lack of central control. Colonies of ants, traffic jams, market economies and segregation in cities are all examples of systems organised without an organiser.

It remains to define a swarm. In the context of this research we use the standard definition where 'swarm' is taken to mean a set of agents that collectively solve a distributed problem through direct or indirect communication.

Indirect communication can be achieved through manipulation of the local environment. Ants (fairly blind insects) for example can leave pheromones on the path they walk on for other ants to follow. They can also deposit objects (leaves, corpses of other ants or dust particles) and other ants can then perceive this modified environment and adjust their actions accordingly. This is termed *stigmergy* and is used by social insects as an almost exclusive form of communication.

An excellent book - that goes far beyond the 'ant algorithms' described in this report - is *Swarm Intelligence: from natural to artificial systems* by Bonabeau, Dorigo and Theraulaz. It provides an overview of the state of the art in swarm intelligence as well as proposing further directions for the application of biologically inspired algorithms. It is centred around the concept of stigmergy, the indirect form of communication which will be explained in Section 2.3.1.

Steven Johnson provides another take on the subject of emergent behaviour in [Joh01]. He describes how the whole can sometimes be smarter than the sum of the parts and focused especially on the development of cities. His book is centred around the self-organisation of neighbourhoods that emerge without central control out of disconnected groups of shopkeepers, bartenders and tenants.

The research undertaken by the author of this report focused on 'swarms of ants' and their *intelligent* behaviour. This chapter will present the necessary background knowledge needed to understand the body of the report.

Firstly, the concept of *Swarm Intelligence* as found in nature is presented together with some possible applications in computer science. Then the biologically inspired algorithm *Ant System* is described and certain perceived weaknesses highlighted.

The third section provides a quick introduction to the TSP before the final section describes the field of *Genetic Algorithms*.

2.1 Swarm intelligence

Swarm intelligence: the collective intelligence of groups of simple agents.

At first sight, Dictyostelium discoideum is a rather unsophisticated organism. On warm summer days the reddish slime can be found on pieces of bark and fallen leaves. But the primitive fungus is capable of performing an astonishing metamorphosis: it can seemingly disappear. If one looks for it at the same place on a cooler day one will be disappointed since the creature is gone.

Dictyostelium discoideum has gone foraging. It has transformed itself into a miniature hunting community. It consists of billions of unicellular organisms, which will form a super-organism when there is a shortage of food. However, if there is enough food the super-organism will break up and the amoebae will forage individually. Dictyostelium is somewhat schizophrenic in that sometimes it is an organism and at others times it is a swarm.

Scientists have tried to understand this fungus for years. The individual amoeba does not possess any reasoning capabilities and still Dictyostelium displays intelligent behaviour. Rather than starving to death individually, the unicellular organisms unite to fight for survival.

This is just one example of self-organisation in biology. Life forms with limited capabilities solve complicated problems in unison. This so-called swarm intelligence works without a central planner or genetically hardwired master plan. Individuals follow simple local rules, unaware of the swarm's behaviour and probably even unaware of the swarm as such. Only on the global level, intelligent behaviour emerges.

Other examples are the blind termites that can build domes up to seven meters high, where different floors are connected via staircases and rain is led away in gutters. Ant colonies find the shortest path to a food source and bees regulate the temperature in their breeding chamber to pretty much exactly 35 degrees.

Computer scientists are studying these self- organising species to try to understand how simple rules can lead to complex emergent behaviour. They use abstract models for different applications of this principle. Virtual swarms optimise routing in networks, control sensors, mobile robots and even the work flow in companies.

For several decades now scientists are fascinated by the collective behaviour of ant colonies. In the late fifties the Harvard educated biologist Edward O. Wilson showed that the insects communicate with the help of so-called pheromones.

The role of these chemical scents in the cooperation of ants is seen to be the key to understanding the swarm intelligence found in nature. Foraging ants leave a pheromone trail which can be detected by other ants. If they find food and return home they reinforce the trail and thereby encourage other ants to

Figure 2.1: Matabele ants: warriors following a pheromone trail laid by scout worker ants (copyright Dr. B. Marcot)

follow this path. The shorter the path, the stronger the pheromone level, since ants will return faster along the shorter path.

Over time this positive feedback effect, also called stigmergy, will lead to more and more ants following the shorter and stronger smelling path thus reinforcing it even further. So in case there are multiple paths to the food source, the ants will collectively find the shortest, solving a hard combinatorial optimisation problem.

The behaviour of these ants gives a more general template for emergent cooperation. In computer simulations virtual ants are given simple instructions to control their local behaviour. Probabilistic processes ensure that ants are distributed over different food sources.

One of the rules could be that ants should follow the path with the most intense level of pheromone with a high probability. Another rule could ensure that, with a smaller probability, other paths are chosen. These rules lead to the exploitation of the food source by a majority of ants and the discovery of other food sources for the colony by a few.

Software engineers used this principle in a routing algorithm for network traffic for example. The 'Ant Based Control' algorithm imitates nature quite precisely. Virtual ants travel the network on randomly chosen paths. Start and end point of each journey are determined randomly, too. Each node has a routing table, indicating the probabilities with which a neighbouring node is chosen next. While the virtual ants are crawling through the network, they drop virtual pheromone. This increases the probability that the ants arriving at the same node in the future will choose the same path, simulating the stigmergy effect found in nature.

This routing problem is just one member of the class of combinatorial optimisation problems, a class which is of interest to many scientists since its computational demands increase dramatically with increasing complexity.

A classical example of such a problem is the famous 'Travelling Salesman Problem' (TSP). How can a salesman visit N cities on the shortest possible path without visiting a city twice? For small instances of this problem, exhaustive search will give the optimal path. However, even for moderately sized examples iterating through all possible paths is infeasible as there are more than 60,822 trillion different tours through twenty cities!

Algorithms based on the positive feedback mechanism found in the ant world are capable of finding surprisingly good solutions to instances of the TSP with 1,000 and more cities. Again, ants travel from city to city leaving a pheromone trail for others to follow short paths.

Practical applications of these algorithms are the routing of truck fleets, the layout optimisation of highly integrated electrical circuits and search algorithms for web pages to name just a few.

Generally swarm intelligence seems to follow some basic principles. Swarms need a certain size to exhibit intelligent behaviour. Random interactions between members of the swarm are necessary to change the system globally. This requires that individuals are able to communicate locally as demonstrated by the pheromone- dropping ants. Similarly, the primitive amoeba Dictyostelium discoideum emits cAMP - a signalling chemical - which plays a central role in the building of the super-organism.

Bees, in contrast communicate via a waggle dance which tells idle bees the direction and distance in which a food source is located. The more often the dance is performed, the bigger the food source. Even though individual bees usually only watch a single dance before starting their foraging activity and even without a central food source ranking, the swarm is quite capable of adopting to changes in the environment and optimising the collection of food.

Scientists at Oxford University used the ideas found in the animal kingdom to devise an algorithm for optimal load distribution in web server clusters.

Some kinds of swarm are furthermore capable of intelligent task allocation using only local information. Certain ant species build a chain to transport food from the source to their nest - a method that is successfully implemented to solve work flow problems.

Swarm intelligence is also a hot topic in the robotics community. Since the early nineties it has been thought that a swarm based system is much more reliable and robust then a single highly complex machine. Scientists at the AI laboratory of the MIT started in 1995 to experiment with micro robots. They taught them how to communicate and how to behave socially.

In August 2002 researchers at the Idaho National Engineering and Environmental Laboratory presented interesting experimental results involving so called GrowBots. These robots could locate toxic waste in an unknown environment and analyse the size and composition of the spillage. Figure 2.2 depicts

Figure 2.2: Swarming robots forming a perimeter around a spillage (copyright INEEL)

a swarm of these robots surrounding a spillage. The more robots were involved, the quicker the task was achieved.

Recently, even the military has become interested. Swarms of Unmanned Aerial Vehicles (UAVs) could be used in dangerous scenarios to replace conventional communication infrastructures. Only last year DARPA financed a project for a battalion of 120 military robots that were equipped with swarm based software. These robots are to be used in search-and-rescue missions involving minimum human intervention.

Swarm intelligence and self-organisation can neither be described nor controlled with our usual way of thinking. The dependency on randomness makes these systems appear particularly suspicious. However, there exist self-organising systems like the Internet that work fine.

The term *swarm intelligence* was first used in 1988 by Gerardo Beni in [Ben88] to describe cellular robotic systems where simple agents organised themselves through nearest-neighbour interaction. Bonabeau et al. in [BDT99] generalised the definition to:

> *Swarm intelligence is any attempt to design algorithms or distributed problem-solving devices inspired by the collective behaviour of social insect colonies and other animal societies.*

In theory, every possible problem can be solved using a swarm-based system. Less than a decade ago, the mathematician Michael Lachmann proved that a swarm of infinitely many individuals is Turing-complete. That is any Turing machine and therefore any computable function, can be translated into a swarm system.

But there is no general theory on how to design and parameterise a swarm to solve a given problem. Marco Dorigo, one of the pioneers of swarm-based computation, even predicts that such a theory may never be found. He claims that finding the right rules and parameters requires a 'mix of intuition, experience and luck'.

The next section will present the *Travelling Salesman Problem*, a hard combinatorial optimisation problem that has been tackled by swarm-based algorithms. It has been chosen from a pool of combinatorial problems of equal difficulty since it is the benchmark problem in this area and many statistics and comparative results exist.

2.2 Travelling Salesman Problem

This section will first provide a formal description of the *Travelling Salesman Problem (TSP)* as well as the Euclideaness constraint applied to most of the problem instances used in this project.

Then a look at the aspects of complexity theory needed to understand the difficulty of the TSP is taken and the relevance of this problem in the world of complexity is analysed.

This is followed by brief justification for the need of approximation algorithms and some famous examples of this class of algorithms are presented. This section then concludes with a look at several variants of this famous problem.

2.2.1 Description

The origins of the Travelling Salesman Problem (TSP) are somewhat mysterious but have been traced back via [AW83], [BN68] and [AS61] until 1961.

It is a classical combinatorial optimisation problem and can be describe as follows: a salesman, who has to visit clients in different cities, wants to find the shortest path starting from his home city, visiting every city exactly once and ending back at the starting point. More formally:

> Given a set of n nodes and costs associated with each pair of nodes, find a closed tour of minimal total cost that contains every node exactly once.

In computing terms the problem can be represented by a graph where all the nodes correspond to cities and the edges between nodes correspond to direct roads between cities. Throughout this report the terms *road* and *edge* as well as *city* and *node* will be used interchangeably.

Of particular interest is the Euclidean TSP, where the distance between cities is defined as the Euclidean distance. This means that for two cities with coordinates (x_1,y_1) and (x_2,y_2) we have $d_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

The Euclidean TSP has the advantage that a set of cities together with their spatial coordinates provide us with a set of edges between nodes that are always sufficient to build a fully connected graph.

Furthermore it is metric, so the triangle inequality from geometry holds, and symmetric, which is to say, the distance between A and B is equal to the distance between B and A.

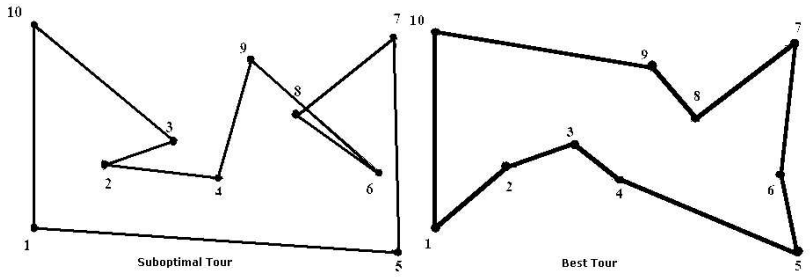


Figure 2.3: Travelling Salesman tours for a 10-city problem.

The problem instance depicted in Figure 2.3 represents a random 10-city problem. On the left hand side, a sub-optimal tour is shown, which can be improved significantly by re-routing the salesman as illustrated on the right hand side.

2.2.2 Complexity

The material in the next two subsections is based on a final year course at Imperial College called *Complexity* which was taught by Iain Phillips. His lecture notes were privately circulated.

Complexity theory divides the space of all problems into complexity classes. Two fundamental classes are called polynomial-time (P) and nondeterministic polynomial time (NP). Problems in the P class can be solved by an algorithm that runs in polynomial time w.r.t. the size of the problem. NP-problems on the other hand are characterised by the fact that we can *check* a solution in polynomial time.

The term *nondeterministic* stems from the original definition in terms of Turing machines. A problem is in NP if there exists a nondeterministic Turing machine that solves it. Such a machine has many choices at each step in the computation and is able to randomly choose a branch of the computation tree. Its running time is measured for the case where all the guesses turn out to be correct.

By definition, P is a subset of NP since we can envisage the polynomial Turing machine for the P-problem to be a nondeterministic one which only ever has one choice at each step. However, it is believed that NP is bigger than P (using set inclusion as an ordering). That is, there exist problems that are in NP but not in P.

While this has not been proven without a doubt[2], one can use a property of the NP class to identify the hardest problems in NP - the so-called NP-complete problems(NPC). These are problems in NP that are more difficult than any other

[2]In fact, it is one of the seven famous open problems, solutions to which are rewarded with a prize of $1,000,000 each by the Clay Mathematics Institute. For a detailed problem description see [URLl].

problem in NP. This requires a polynomial-time translation from any problem in NP to the NP-complete problem. In 1971 Stephen Cook showed that the satisfiability problem (SAT) is NP-complete.

Also, if one has a polynomial-time translation from problem A to problem B and another polynomial-time translation from B to problem C, then one also has a polynomial-time translation from A to C since the class P is closed under composition. The process of translating one problem to another is called *reduction* and is transitive.

Both P and NP are downwards closed under reduction as is illustrated in Figure 2.4.

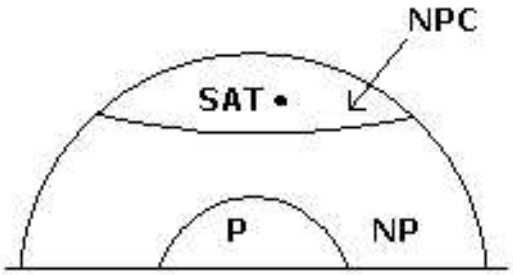


Figure 2.4: Complexity classes P, NP and NPC together with the problem SAT

So to show NP-completeness of a given problem A, all one has to do is show that the problem is in NP and reduce a problem that is known to be NP-complete (e.g. SAT) to the problem A.

The difficulty of NP-complete problems stems from the following characteristics:

- they have a number of different local optima (so local search techniques like steepest-ascent hill-climbing won't work)
- they can't be broken into sub-problems and solved separately (e.g. can't solve sections of the TSP independently)
- they appear statistically random (so statistical methods won't work)

The problems classified into P and NP are all 'decision' problems, that is problems that can be answered with *Yes* or *No*.

A related type of problem are 'function' problems which are concerned with finding something. For example, the decision problem SAT mentioned earlier just asks whether there *exists* a satisfying truth assignment for a given set of clauses. The functional variant FSAT in contrast tries to *find* such a truth assignment.

A further type of problem are called 'optimisation' problems, which are concerned with finding the best solution to a given problem.

In order to measure the strength of the Travelling Salesman Problem we need to define three more classes:

FNP is the class of all *function* problems that can be stated as follows:

> Given a problem instance x, find a solution y such that $R_L(x, y)$. If there is no such y then return 'no'. R_L is a polynomially balanced ($length(y) \leq (length(x))^k$ for some k) and polynomially decidable.

FP is the subset of problems on FNP that can be solved in polynomial-time.

FP$^{\mathbf{NP}}$ is the class of all problems that are in FP with the help of on NP oracle.

An oracle is an imaginary Turing machine which knows the answer to a given problem. For example an instance of FSAT with n variables can be solved by querying a SAT-oracle $n + 1$ times, fixing each variable in turn. So FSAT is in FP^{NP}.

A consequence of a paper by Stephan Mertens[Mer00] on the similarity between finding the smallest element in an unordered list and the NPC problem of number partitioning is that there is no way of telling how close any given solution to a problem is to the optimal solution of the problem.

No possible metric (like the tour length of a solution S to the TSP) will hint at how much of that solution S will be part of the optimal solution. This gives an indication that optimisation problems will be at least as hard and almost always *harder* than the corresponding function problem.

2.2.3 Relevance

In terms of complexity, one has to distinguish between three different versions of the Travelling Salesman Problem. The decision version, TSP(D) can be stated as follows:

> Given a set of cities, the distance between them and a bound B, *does a path exist* such that a salesman travelling along that path will visit each city exactly once and return to the start with a total distance travelled $\leq B$?

FTSP(D) is the functional variant of the TSP(D) and can be formulated as:

> Given a set of cities, the distance between them and a bound B, *find a path* such that a salesman travelling along that path will visit each city exactly once and return to the start with a total distance travelled $\leq B$.

The original TSP is an optimisation problem and can be stated as:

> Given a set of cities and the distance between them, *find the shortest path* such that a salesman travelling along that path will visit each city exactly once and return to the start.

TSP(D) can be shown to be NP-complete. It is clearly in NP - just guess an enumeration of the cities and check that the total path length does not exceed the given bound. One can prove completeness by reducing SAT to 3-SAT, 3-SAT to HP (using ingenious gadgets) and HP to TSP(D). For details of these proofs, the interested reader is pointed to [Pap94]. So we have:

$$SAT \leq \text{3-}SAT \leq HP \leq TSP(D)$$

where $\leq$ denotes reduction.

Since these nondeterministic Turing machines that guess the right choice at every computation step do not exist in real life, we cannot hope to solve TSP(D) in polynomial time. The best known algorithms are exponential, since in the worst case one has to check all possible tours to realise that all of them exceed bound B. For a given problem with N cities, there are $(N-1)!/2$ possible paths if one eliminates identical and mirrored paths. Therefore approximation algorithms are needed to find optimal or at least very good solutions.

FTSP(D) can be shown to be in FP^{NP}. It is a function problem that can be solved in polynomial time with the help of an NP oracle. For n cities, there are $n(n-1)/2$ roads connecting two cities and the algorithm to solve FTSP(D) tries to eliminate each one of them in turn. To find a route with bound B one proceeds as follows:

1. take as an oracle the Turing machine for the NP-complete problem TSP(D)
2. adjust the distance between two particular cities a and b to $B+1$
3. run the oracle on the modified problem; if it still succeeds then the 'road' from a to b was not in the optimal tour, otherwise set the distance back to what it was originally
4. repeat $O(n^2)$ times (for each road)
5. read off the optimal tour which will consist of all roads with a distance of less than $B+1$

Since this polynomial algorithm uses the Turing machine for the NP-complete problem TSP(D), it cannot be easier then NP-complete. However, FTSP(D) can theoretically be solved without the help of an oracle. Using a nondeterministic Turing machine one could guess a solution and check in polynomial time that it is an instance of a polynomially balanced and polynomially decidable relation R_L. Hence FTSP(D) is in the class FNP.

TSP can also be shown to be in FP^{NP}. All one has to do is find the length L of the optimal path and then run the Turing machine for the FTSP(D) with bound B set to L. We know that $0 \leq L \leq 2^k$ where k is the length of the binary input to the Turing machine. Now we ask the NP oracle TSP(D) whether there exist a solution with bound B set to 2^{k-1}. If this is the case we call the oracle again with B set to 2^{k-2}. Otherwise, the length of the optimal path lies between 2^{k-1} and 2^k and we call the oracle with B set to $3 * 2^{k-2}$.

Using binary search one can find the optimal length L within $log(2^n)$ (or $O(n)$) moves by continuously bisecting the search space. So we can find the optimal length L in polynomial time using an NP oracle and then find the optimal path using the algorithm for FTSP(D) which also uses an NP oracle. Hence TSP is in FP^{NP}.

We cannot hope to show that TSP is in FNP since a relation containing just the one pair (x,y) - where x is the problem instance and y is the best path - would not be polynomially decidable.

In fact, Papadimitriou in [Pap94] showed that TSP is FP^{NP}-complete, making it one of the hardest computable problems.

Many other combinatorial problems like Quadratic Assignment, Scheduling, Vehicle and Network routing, Graph Colouring and Multiple Knapsack have been solved using variants of the ant system. The subtle differences lie in the way the problem is formalised to allow the application of the ACO meta-heuristic. One needs to have a solution space so that the ants can travel this space leaving pheromone trails to gear other ants towards good and finally optimal solutions.

All these variations of the ant system share the same parameters that will be looked at in Section 5. Hence finding a way to set these parameters without having to 'guess' them, is a desirable objective.

2.2.4 Approximation algorithms

Exact algorithms such as cutting-plane or facet-finding are very complex and do not run in polynomial time. Hence they are infeasible to use for larger problem instances (those with thousands of cities). The previous section justified the need for approximation algorithms, we now look at some of them more closely.

There are three types of approximation algorithms:

1. *greedy heuristics* that provide a sub-optimal solution used to study theoretical optimality (they generally perform rather poorly)

2. *specialised heuristics* like Lin-Kernighan or the 2-OPT algorithm that use domain specific knowledge. Their disadvantage is that they can only be applied to one problem, the TSP in this case (they outperform all other approximation algorithms)
3. *generic heuristics* based on local search such as Simulated Annealing or Tabu Search that continuously try to improve on a solution (they perform fairly well)

Examples of all three types of approximation algorithms are presented.

The greedy heuristics all restrict the TSP to *Euclidean* instances only. That is, for any three cities A, B and C we have $\delta(A, C) \leq \delta(A, B) + \delta(B, C)$ where δ is the distance. This is a reasonable restriction to make since it allows for greedily picking the best solution locally.

Nearest-Neighbour - Start anywhere and always visit the nearest city that has not yet been visited. A solution thus obtained has been proved to be $\leq (1 + \lceil lg(n) \rceil)/2$ times the optimal solution. This is quite significantly worse than the optimal solution for large problems. The reason for this poor performance lies in the algorithm's dependency on local information only. It may find a good path for $n - 2$ cities say, but is then forced to use inappropriately long connections between the last few cities.

Minimal-Spanning-Tree - Find a minimal spanning tree(MST) of the city-graph in polynomial time. Then double the edges and find an Eulerian circuit of the 'doubled MST' (again, in polynomial time). Visiting all cities using the MST is optimal since removing one link from the optimal tour yields a MST. However, the salesman has to end up where he started, so 'travelling home' along the unused edges of the Eulerian circuit would make the path twice as long as the optimal one. Also, using the Eulerian circuit the salesman would visit some cities more than once. So the algorithm requires to follow the Eulerian circuit but instead of going to the next city, skip to the next unvisited city along the Eulerian circuit.

Others like *Clarke-Wright*, based on a central hub, with savings for avoiding the hub city or the more complicated *Christofides* heuristic

Specialised heuristics involve a local search. That is, given a sub-optimal tour, improve this tour by small local changes. First, one needs to find an initial, sub-optimal tour. This can be done in a variety of ways. One could pick a random permutation of the n cities or use greedy heuristics like the ones mentioned above. Having found an initial tour one can then apply a local search algorithm. The most famous ones are:

2-OPT - an exhaustive exploration of all permutations obtained by exchanging two edges. Choose two non-adjacent edges in a tour, call them $a - b$ and $c - d$ in such a way that they have the same direction on the tour (otherwise we would get two unconnected tours), remove $a - b$ and $c - d$ and add $a - c$ and $b - d$. This can be extended to three edges, resulting in the 3-OPT algorithm.

Lin-Kernighan - generally considered to be one of the most effective algorithms to find optimal or near-optimal solutions to the symmetric TSP. It is a generalisation of the 3-OPT algorithm that borrows heavily from the tabu search (described below). From 1973, when it was first published, until 1989 it was the 'world champion heuristic'.

Both of these local search heuristics are very specialised and only applicable to the travelling salesman problem. There are, however, general algorithms that are useful to finding solutions for a much wider range of optimisation problems. Affected areas range from graph theory (minimum graph colouring, maximum clique, maximum common subgraph), logic (maximum satisfiability), scheduling (job shop scheduling) to network design (TSP, maximum quadratic assignment).

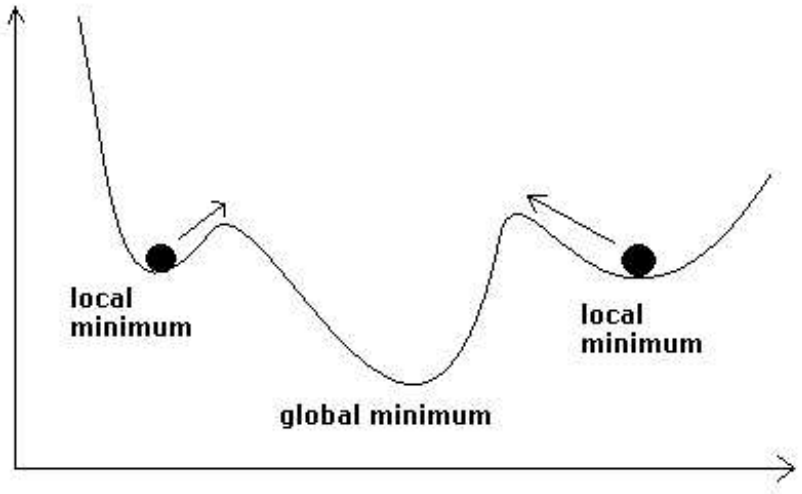


Figure 2.5: Simulated Annealing trying to find the global optimum

Some of the better known general heuristic optimisation techniques are neighbourhood search methods such as:

Simulated Annealing is metaphor related to physics. Hot, molten metal is left to cool down slowly. This allows the atoms to slide progressively to their most stable positions, rather than 'freezing' them in their current position. This makes the resulting metal much stronger. The metaphor of minimising the energy of a block of metal is used in this algorithm to minimise the cost (distance) for the travelling salesman. Simulated Annealing is derived from the search method of steepest local descent (which can similar to the well known 'hill-climbing' result in local optima as shown in Figure 2.5). Rather than generating all neighbouring solutions

and picking the best only one neighbouring solution is generated. If this new solution is better than the old one accept it, if it is worse accept it with a certain probability. Or more formally:

1. Start with an initial solution i
2. Generate *one* neighbouring solution i^*
3. If $f(i^*) < f(i)$ then set $i = i^*$ (f is the cost function)
4. If $f(i^*) > f(i)$ *and* $(rand(1) < prob)$ then set $i = i^*$

The probability with which worse solutions are accepted decreases over time T and is defined as:

$$prob = e^{-[f(i^*)-f(i)]/T}$$

Tabu Search uses a memory mechanism to diversify the direction of search. It prohibits certain actions (makes them 'taboo') for a limited time and forces the search to proceed in another direction. The algorithm can be outlined as follows:

1. Choose an initial solution i.
2. Generate set V of possible improved solutions, excluding 'taboo' solutions and including 'aspiration' solutions (solutions that are taboo but still desired)
3. Choose the best solution i^* from set V
4. If $f(i^*) < f(i)$ then set $i = i^*$ (f is the cost function)
5. Update tabu and aspiration conditions
6. If stopping condition is met then stop, else go to step 2

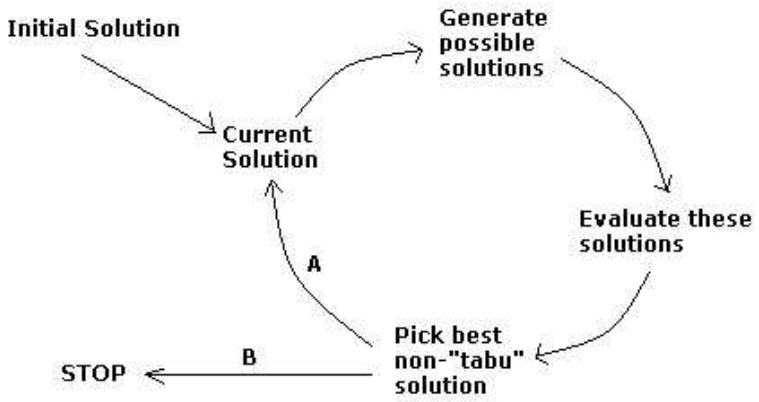


Figure 2.6: Execution flow for tabu search, 'B' stands for 'solution good enough' and 'A' stands for 'solution not good enough'

2.2.5 Variations

Having described the Travelling Salesman Problem in Section 2.2.1 and discussed its complexity thereafter, several variations are now described. Firstly the symmetry constraint is removed resulting in the asymmetric TSP. Then various notions of dynamic TSPs are presented.

Furthermore a related problem called *Black-and-White TSP* adds additional constraints to the problem making certain tours invalid. This and its generalisation, the RATSP, are introduced below.

Asymmetric TSP (ATSP)

The symmetry constraint in the standard Euclidean Travelling Salesman Problem requires that the distance between A and B is the same as the distance between B and A. Removing this constraint opens up the TSP to many more applications.

While 'distance' in a symmetric TSP is usually taken to be the *geographic distance* between two points, it can now represent many other things. For example, it could represent the *time* it takes the salesman to travel between two points. A traffic jam or an uphill struggle could mean that it takes significantly longer to get from B to A than vice versa.

The TSPAnt application developed during the research undertaken for this project proved to be easily extensible to handle this variation of the TSP.

Dynamic versions of the TSP (DTSP)

Dorigo, in a variety of his publications, suggested that the real strength of the ant system algorithm (to be presented in the following section) would show in applications to dynamic problems. However, he never investigated this claim further.

Two different notions of *dynamic* have been described by researchers in Germany and The Netherlands. The first one keeps the number of cities constant but varies the distance between them. This could model traffic jams on motorways connecting the cities. A solution that was optimal before the traffic jam occurred may no longer be optimal and the salesman needs to be re-routed.

A different approach has been taken by Guntsch and Middendorf in [GM01]. They inserted and deleted cities into a given problem instance and proposed strategies on how to modify the pheromone levels on the edges. Generally only edges in the vicinity of the inserted or deleted city need to be modified since remote parts of the tour are unlikely to be affected.

Both approaches will be further investigated in Chapter 3.

Black-and-White TSP (BWTSP)

This variant of the standard TSP adds two extra constraints allowing for more realistic modelling of behaviour in the real world. The set of cities is split into two subsets: black cities (B) and white cities (W). The salesman still has to visit all cities exactly once ending up in the city he started in and the optimal solution is still the shortest one.

A *black-to-black path* is defined as a sequence of cities $c_1, c_2, ..., c_k$ where $c_1, c_k \in B$ and $c_i \in W$ for all i such that $1 < i < k$.

Now, the two additional constraints are:

cardinality constraint: the number of white cities in any black-to-black path can not exceed a certain number Q

cost constraint: the total distance between the two black cities in an black-to-black path can not exceed a certain limit C

Both Q and C are parameters supplied with the problem instance.

TSP with Replenishment Arcs (RATSP)

The RATSP is a more general form of the BWTSP. It is formulated as follows: the set of arcs is split into normal arcs and replenishment arcs. Also each node has a non-negative weight associated with it. The problem still searches for the shortest closed tour through all the nodes.

However, the tour can not accumulate more than a certain limit C of total weight before a replenishment arc must be used where C is supplied as a parameter.

This problem was first formulated by Natashia Boland et al. in [NLG00] as an instance of the Asymmetric TSP. Her student Vicky Mak from the University of Melbourne then presented new and efficient solution techniques for solving it in her PhD thesis *On the Asymmetric TSP with Replenishment Arcs*[Mak01].

She used mathematical concepts like polyhedral results and a Lagrangean Relaxation heuristic to analyse the problem theoretically and also implemented a simulated annealing algorithm.

The practical relevance of this problem comes from the aircraft rotation problem. An aircraft can fly a tour between multiple airports but needs to have a maintenance inspection before flying more than a certain number of miles. This maintaining can only be done at certain airports so that flight legs to and from these maintenance airports can be represented by replenishment arcs.

2.3 Ant System

This section will present the entomological motivations for an algorithm, the algorithm itself, variations of it and its perceived weaknesses.

In 1992, when Marco Dorigo wrote his PhD thesis[Dor92] at the Politecnico di Milano in Italy, he took ideas from robotics and used them for algorithm design.

Four years later he and two colleagues published the 'ant system'[DMC96], the de facto standard work on ant based algorithms. In this paper they propose a model of positive feedback, distributed computation and a constructive greedy heuristic to solve stochastic combinatorial optimisation problems. They apply this model to the TSP and compare it with other general purpose heuristics.

Based upon his work researchers from all over the world have tried to take the metaphor of self-organising behaviour of social insects and apply it to many problems in computer science.

2.3.1 Entomological background

In 1989 Goss et al. [GADP89] undertook an experiment with ant species *Iridomyrmex Humilis*, more commonly known as *Argentine ants*. They used these ants to demonstrate the intriguing ability of social insects to find the shortest path to a food source.

They set up an experiments containing a food source and the nest of an ant colony linked by bridge with two branches of different length. Foraging ants have to choose between the two branches and two interesting findings were made:

- after an initial period most ants will choose the shorter branch
- the probability of choosing the shorter path is proportional to the difference between the two branches

This shortest path selection behaviour is the result of positive feedback (autocatalysis) and the differential path length. It is achieved through stigmergy, an indirect form of communication described below.

Stigmergy is defined as local modifications of the environment. This modification is caused by ants depositing a chemical called pheromone wherever they go.

At each choice point the ants make a decision which is biased by the pheromone levels they detect on each branch. This process is auto-catalytic, since the fact that an ant chooses a path will in turn increase the likelihood of this path being chosen by other ants in the future.

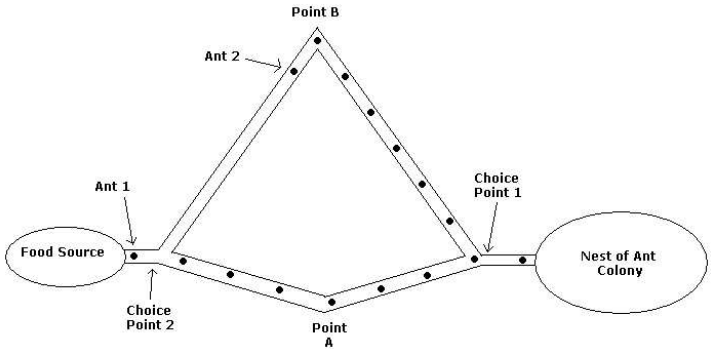


Figure 2.7: The experimental setup used by Goss et al. in their 1989 experiment with *Argentine ants*.

In order to help understanding this phenomenon Figure 2.7 shows a stream of ants leaving the nest of the colony. 'Ant 1' and 'Ant 2' are the first ones to leave and start the foraging activity at the same time. They arrive at 'Choice Point 1' together and make a 50/50 decision as neither branch smells of pheromone.

'Ant 1' which chose the shorter branch reaches the food source first (via 'Point A'), picks up some food and starts the journey back to the nest. On the way back it has to make another decision at 'Choice Point 2'. It detects some pheromone on the shorter branch (the chemical deposited there by itself on the outward journey) and with a high probability it picks that branch again.

The longer branch does not contain any pheromone near the branching point, since the ants that chose it (including 'Ant 2') have not yet arrived at 'Choice Point 2'.

Picking the shorter branch again will reinforce that path even further. In the future, ants both on the way to and from the food source will pick the shorter path with a higher probability and the pheromone on the longer branch that has not been reinforced for a while, will eventually evaporate and all ants will travel along the shortest path.

2.3.2 Algorithm

In this subsection, the 'ant system' algorithm (AS) for the travelling salesman problem, as originally described in [DMC96] is outlined and then presented in pseudo code notation. It is heavily based on the behaviour described in the previous section.

At each time (t), an ant (in city i) has to choose the next city j it goes to out of those cities that it has not already visited. The probability of picking a certain j is biased by the distance between i and j and the amount of pheromone on the edge between these two cities.

Let τ_{ij} denote the amount of pheromone (also called trail) on the edge between i and j and let η_{ij} be the visibility of j from i (equivalent to $\frac{1}{distance(i,j)}$).

Then the bigger the product of τ_{ij} and η_{ij} is the more likely j is going to be chosen as the next city. The trail and visibility are now weighted by α and β and we arrive at the following formula (where $p_{ij}(t)$ is the probability of ant k choosing city j from city i at time t):

$$p_{ij}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha} * [\eta_{ij}]^{\beta}}{\sum_{k \in allowed_k} [\tau_{ik}(t)]^{\alpha} * [\eta_{ik}]^{\beta}} & if j \in allowed_k \\ 0 & otherwise \end{cases} \tag{2.1}$$

After all the ants have completed their tour, the trail levels on all the arcs need to be updated. The evaporation factor ρ ensures that pheromone is not accumulated infinitely and denotes the proportion of 'old' pheromone that is carried over to the next iteration of the algorithm. Then for each edge the pheromones deposited by each ant that used this edge are added up, resulting in the following pheromone-level-update formula:

$$\tau_{ij}(new) = \rho * \tau_{ij}(old) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{2.2}$$

where m is the number of ants.

The following now presents the AS algorithm in pseudo code notation:

```
// Initialisation phase:
num_of_cycles = 0
set all pheromone levels to 0
place an ant in each city
put the start city of each ant in its tabulist

// Main algorithm:
if num_of_cycles < MAX_cycles and not stagnation_behaviour then
   // One tour cycle
   empty all tabulists
   for all ants do (in parallel)
      while there are unvisited cities (not in its tabulist) do
         pick a city probabilistically according to formula 1
         add this city to the tabulist
      calculate tour_length using order of cities in the tabulist
      if tour_length < best_tour_length_so_far then
         best_tour_length_so_far = tour_length
         best_tour = content of tabulist
      calculate & accumulate pheromones dropped during this cycle
   re-calculate pheromone levels according to formula 2.2
   increment num_of_cycles
else
   print best_tour
   terminate
```

Since the choice of city at each step is partially dependent on the trail level of the connecting edge and this trail level fluctuates (reinforcement adds pheromones to it and evaporation decreases it) this algorithm moves towards good solutions.

However, since the choice is probabilistic it can not be guaranteed that the optimal solution will be found. Perhaps a slightly worse than optimal solution is found at the very beginning (with all trail levels being equal) and some sub-optimal arcs will be reinforced. This reinforcement can lead to stagnation behaviour with the algorithm never finding the best solution.

2.3.3 Ant Colony Optimisation

An improvement called *Ant Colony System (ACS)* has been suggested in [DG97b] that reduces this phenomenon and finds good solutions even faster. Here an additional parameter q_0 with $0 \leq q_0 \leq 1$ is used to control the level of exploration undertaken by the ants.

If a random number q that is uniformly distributed over [0,1] is less than q_0 then we just use rule 2.1 as before. However, if $q \geq q_0$ then we deterministically pick the city j for which the weighted product of trail level and visibility is highest.

Following the successful application of the *ant system* and the *ACS* to the travelling salesman problem, other optimisation problems such as the multiple knapsack problem [Fid02], job-shop scheduling [CDMT94] and quadratic assignment problem [MC99] [GTD98] were tackled by various researchers.

All these variations on the original ant system led to the development of the *Ant Colony Optimisation Meta-Heuristic*. This meta-heuristic, stated by Dorigo et al. in [DC99], describes a class of ant-inspired algorithms. It abstracts away from particular implementations both of the algorithm and the problem instance under consideration. This meta-heuristic was so successful that it was included as 'Chapter 2' in David Corne's book 'New ideas in optimisation' [Cor99].

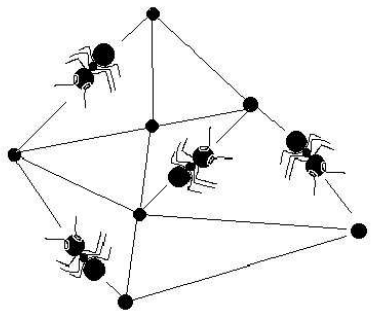


Figure 2.8: Ants and the Travelling Salesman Problem

2.3.4 Problems and opportunities

The ant algorithms presented so far are all parameterised by up to ten parameters. This allows for a lot of fine-tuning and specialising for different applications. However, this is also the weakest point of the presentation since very little research has been conducted into finding **optimal parameters**.

Dorigo et al. in [DMC96] did a rough analysis suggesting some 'robust' parameters which they found to lead to good performance of the ant algorithm. These parameters have been used in all further research on the ant system and its extensions by everyone without questioning.

Dorigo admitted that the parameters he proposed are less than optimal and that optimal parameters are very likely to be problem dependent. Subsequently the ant system for the quadratic assignment problem by Maniezzo and Colorni[MC99] used slightly different parameter values supporting Dorigo's claim that the parameters are problem dependent.

Part of the research described in this report was motivated by the belief that the optimal parameters are not just dependent on the problem but even dependent on the *problem instance*. That is to say, one needs different parameters to find the best tour in a set of cities that are lined up to form a grid than to find the best tour in a set of cities that are placed randomly.

Another interesting aspect of the ant system algorithm and its successor, the ant colony optimisation algorithm, is the inherent **parallelism**. This feature has not been analysed sufficiently. Bullnheimer et al. in [BKS97] presented two obvious parallelisation strategies and noted, that there is a lot of room for further research. Nobody has picked up on this so far.

This report presents a variety of parallelisation strategies in Chapter 4. The strategies from [BKS97] are analysed in detail, improvements are suggested and two new strategies are presented.

Further opportunities lie in the adoption and application of the ant algorithm to other variations of the Travelling Salesman Problem. The author of this report has implemented several applications that visualise variations of the TSP such as the **asymmetric TSP** and the **dynamic TSP**.

To perform well on dynamic instances of the TSP, the algorithm needs to be able to change the ratio between exploitation and exploration on the fly. This can be achieved by the shaking operator introduced in [CS02] or the η- or τ-strategies presented in [GM01].

In Chapter 3 these variations are discussed further together with the changes needed to adopt the ant system to variants of the TSP such as those described in Section 2.2.5.

2.4 Genetic Algorithms

Inspired by observation of natural processes in the real world John Holland invented the first genetic algorithm in the 1960s. He imitated the insight he got by studying Darwin's theory of evolution which can be summarised as:

- the traits found in the parents are passed on to their offspring during reproduction
- new traits are produced by variations or mutations that are naturally present in all species
- a process termed *natural selection* chooses those individuals that are best adapted to the environment
- variations can accumulate and produce new species over long periods of time

According to Darwin, natural selection can be rephrased as survival of the fittest. The characteristics of the fittest individuals, encoded in their genes, are passed on to their offspring and keep on propagating into new generations.

In sexual reproduction, the chromosomes of the offspring are a mix of their parents' genes. But an offspring's characteristics are only partly inherited from parents and partly the result of new genes created during the reproduction process.

Genetic algorithms (GAs) in their most general form are methods of moving from one generation of *chromosomes* to another. Each chromosome can be thought of as a bitstring, representing the encoding of one particular solution to a problem. Other encodings are also possible and will be looked at later.

computing term	biological term
solution to a problem	individual
set of solutions	population
quality of a solution	fitness
encoding of solution	chromosome
part of the encoding of a solution	gene
search operators	crossover and mutation
reuse of good solutions	natural selection

Table 2.1: Comparison of terms used in computer science and biology

These algorithms can be applied to machine learning, complex optimisation problems or to simulations of real-life phenomena as has been done in projects like *ALife*. Other simulation experiments involve ecological concepts such as symbiosis and host-parasite co-evolution.

In machine learning, GAs can be used to evolve weights in neural networks or in general classification tasks. They have been applied to weather prediction, aircraft landing scheduling and stock market analyses.

A further field where evolutionary principles are used in a computer science context is *genetic programming*, the automatic creation of computer programs. This field researches the ability of programs to evolve given a specific task.

Genetic algorithms were used heavily for approximations to optimisation problems such as scheduling or the Travelling Salesman Problem described in Section 2.2. However, Dorigo et al. in [DMC96] showed that the *ant system algorithm* presented in Section 2.3.2 outperforms the *genetic algorithm* in all the problem instances of the TSP that were investigated.

Problem instance	GA	AS	AS with local search
Nugent15	1160	1150	1150
Nugent20	2688	2598	2570
Nugent30	6784	6232	6128
Elshafei19	17640548	18122850	17212548
Krarup30	108830	92490	88900

Table 2.2: Comparison between GAs and ant system from [DMC96]

The results in Table 2.2 above represent the lengths of the shortest tours for each problem instance and were averaged over 5 runs. The ant system with local search found the best known solution in all cases but one (for the problem instance Nugent30, the best known solution is 6124). It clearly was superior to the genetic algorithm for every problem instance considered.

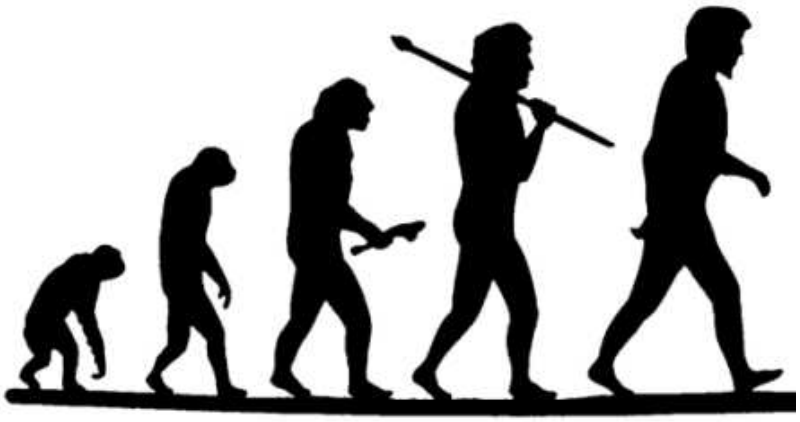


Figure 2.9: Evolution of homo habilis to homo sapiens

2.4.1 Basic structure

The basic structure of a standard genetic algorithm found in [Mit96] is given by the following five steps:

1. start with a randomly generated population of n individuals (bitstrings) that each represent a solution
2. calculate fitness(x) for each individual x in the population
3. repeat $\lceil \frac{n}{2} \rceil$ times, each time generating two individuals:
   (a) select a pair of parents (put them back after selection); the probability of selection increases with fitness of the individual
   (b) with crossover probability p_c, do crossover to create two offspring, otherwise just clone parents (take an identical copy of each parent)
   (c) mutate both offspring at each gene with mutation probability p_m

   if n is odd, discard one member of new generation at random
4. replace current population with new population
5. if best individual is not fit enough, go to step 2

This basic genetic algorithm displayed above models the generational approach found in nature. Once the new generation is created it replaces the old generation (step 4) which then ceases to exist. This is desirable in nature, where the objective is to establish healthy populations.

However, this model is not always suitable for optimisation problems. In optimisation one wants to find a single superior individual representing the optimal solution to a given problem.

Another disadvantage of this basic genetic algorithm lies in the representation of individuals. Bitstrings, as used most often, imitate the DNA structures from nature very well. However, if the parameters that the GA is going to optimise are not binary, the algorithm performs poorly, since the natural distances between individuals are destroyed when transforming them to bitstrings.

Both of these problems arise from following the evolutionary process found in nature too closely. There are several improvements that can be applied to the basic algorithm where the analogy with natural evolution breaks.

For example, the *generational* approach from step 4 of the basic genetic algorithm can be replaced with a *steady state* algorithm. Rather than computing a whole new generation, one could just create one offspring . If this new offspring is better (fitter according to the fitness function) than the worst member of the parent generation, then just replace this worst member. Otherwise discard the new offspring.

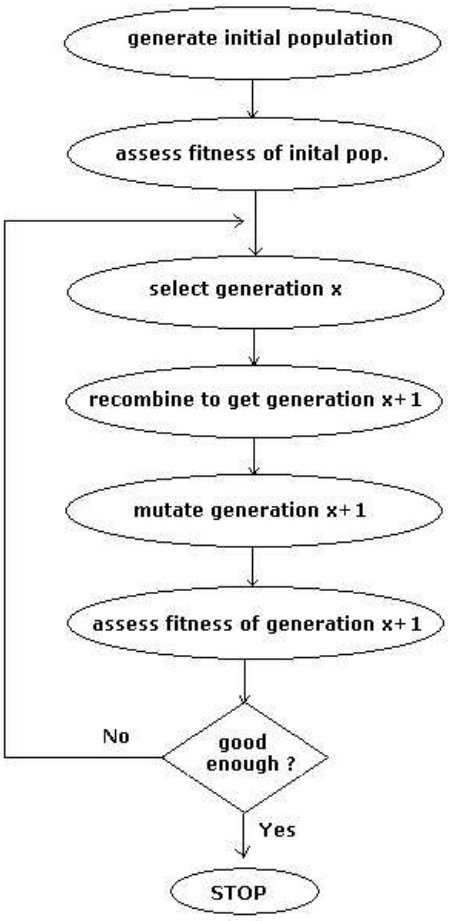


Figure 2.10: Flow diagram of a basic genetic algorithm

Generally, genetic algorithms differ from traditional approximation methods like hill-climbing in four aspects:

- they work with encodings of the parameters rather than travelling the parameter space directly
- search proceeds from sets of solutions to sets of solutions rather than from one individual solution to the next
- they use probabilistic transition rules rather than deterministic ones
- fitness is used directly rather than derivatives of the objective function

Alternative encodings and selection methods will be looked at in the following two subsections.

2.4.2 Encodings

Candidate solutions need to be encoded appropriately. This is one of the most important steps during the design of a genetic algorithm for any kind of problem.

The most common encoding is that of fixed-length, fixed-order bitstrings. A binary parameter can be represented as a string of zeros and ones like this example of length 25:

0101110101010101010000111

This encoding is popular because of its history as well as information theoretic superiority. Holland, when he first proposed genetic algorithms in [Hol75], used a binary encoding. He used an argument involving schemata that is beyond the scope of this introduction. A lot of further research has been conducted using bitstring encoding and it remains the most popular encoding. A comprehensive summary of this research can be found in [Mit96].

One big advantage is the close distance between neighbouring individuals. In most cases, the next individual in a sequence can be created from its predecessor by toggling one bit.

Translated real values do not have this property. The real number 175 is represented by 10101111. The next number, 176, is 10110000 which differs from the predecessor in more than half of the bit locations.

However, real-valued encodings are more natural for many problems though much harder to implement. Some real-valued representations have been used by researchers in the past but not many results are known. Another experimental representation is that of trees used mainly in genetic programming, where computer program are evolved.

There are no guidelines for predicting which encoding is most appropriate for a given problem and the consent in the GA community is that one should pick the encoding that most naturally fits the problem at hand and then design the genetic algorithm based on the chosen encoding.

2.4.3 Selection methods

Similar to the problem of finding a good encoding is that of choosing the right selection method. There are several possibilities and again, no rigorous guideline. The most common methods will be described presently.

Note, that all selection methods emphasise the fitter members of the population in order for the offspring to be of equal or higher fitness. Here, a balance has to be found between restricting selection to too few fittest parents (which results in early stagnation) and too little restriction (in which case evolution will be very slow).

Selection methods choose a number of individuals, some repeatedly, to make up the pool of parents, also called *intermediate population*. During the mating process, individuals from this pool will be chosen, so this pool should contain multiple copies of the fit individuals and no copies of the unfit ones.

Fitness-proportionate selection is used in Holland's original presentation. It says that the number of times an individual is expected to reproduce is the individual's fitness divided by the average fitness of the population. The same scheme is found in nature and biologist refer to it as *Viability Selection*.

There are two sampling methods for this selection known as *Roulette Wheel Sampling* and *Stochastic Universal Sampling*. The former assigns slices of a wheel to each individual. The size of each slice is proportionate to the individual's fitness. Spinning this wheel N times will select N individuals (with repetitions) which will make up the pool of possible parents.

However, due to the stochastic nature of the sampling, it is possible, that the wheel stopped on the small slice of the worst individual (in the worst case, this happened N times). Also, the big slice of the fittest individual could have been missed every time.

This led to the proposal of the stochastic universal sampling method. Here, the roulette wheel will be split up as before, but instead of spinning the wheel N times with one marker, it is spun once only but with N equally spaced markers. This ensures that good individuals can't be missed in the pool by a freak of nature.

If the fitness of an individual divided by the average fitness of the population is denoted by x and x_i is the integer part of x, then stochastic universal sampling guarantees to pick this individual at least x_i and at most $x_i + 1$ times.

Figure 2.11 shows the pictorial representation of the roulette wheel sampling on the left and the stochastic universal sampling on the right. One can clearly see two very fit individuals, six fit ones, five less fit ones and seven unfit ones. The left wheel will have to be spun 20 times in order to get an intermediate population of size 20, whereas one spin of the right wheel suffices.

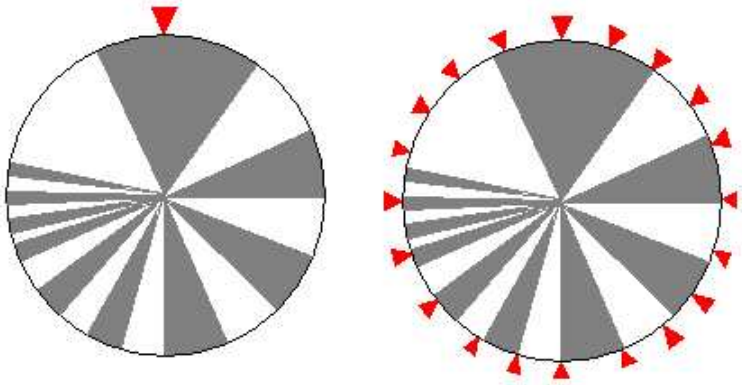


Figure 2.11: Roulette Wheel and Stochastic Universal Sampling

There universal stochastic sampling could easily be implemented like so (note, this sample algorithm does not guarantee a fixed size for the intermediate population but this has no negative effect on the genetic algorithm):

```
for each individual i do
   x   = fitness(i) / average_fitness_of_population;
   x_i = integer_part_of(x);
   x_f = fractional_part_of(x);
   put x_i copies of it into the intermediate population
   with probability of x_f add an extra copy
```

A problem with fitness-proportionate selection is that normally it converges too quickly. A few very fit individuals are creating many offspring and their genes dominate the population after a few generations. The lack of diversity that occurs essentially halts the evolution.

In **Rank Selection**, all individuals are ranked by their fitness. The number of times an individual is expected to reproduce is no longer proportionate to their fitness but instead proportionate to their rank. This ranking can be simply based on a linear ordering of fitness values or on more sophisticated orderings as required.

Tournament Selection is another widely used selection method. It's advantage is efficiency. The fitness-proportionate selection had to do two passes through the individuals, one to calculate the mean and one in the main loop. Rank selection had to rank all individuals which again is quite a computationally expensive operation.

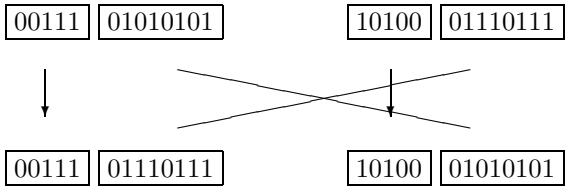
In tournament selection, two individuals are chosen at random. A copy of the fitter one is put into the intermediate population and both individuals are returned to the original population so that they can be selected again. A modification of this scheme is to pick the less fit individual with a small probability which may be a parameter to the system.

2.4.4 Genetic operators

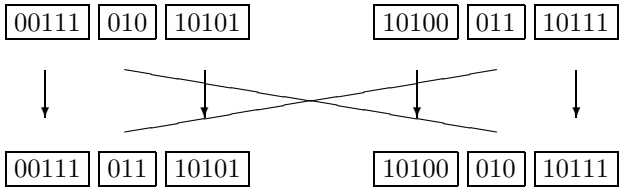
The main distinguishing feature of genetic algorithms over other stochastic methods is the use of genetic operators. Recombination, a combining of genes different from what they were in the parents, is most often done using *crossover*. A second standard genetic operator is *mutation* which will also be described presently.

Crossover

The four most popular recombination operators will be presented in this section, starting with *one-point crossover*. One position in the chromosome is chosen randomly and the parents are split in a front and a back part. Then two offspring are generated by using the front part from one parent and the back part of another and vice versa.



The advantage of this operator, is that large sequences of genes are kept in the order they are found in the parents. Another useful way of combining two parents is *two-point crossover*. Here, two locations in the chromosome are chosen randomly and the middle parts are exchanged to create two offspring.



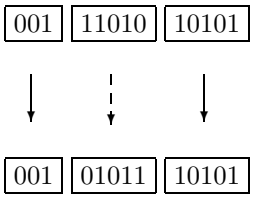
The need for this operator arises from an argument based on schemata which is beyond the scope of this introductory chapter. Essentially, it allows for a more varied population, since sequences can be replaced that are neither at the beginning nor at the end of the chromosome but somewhere in the middle.

The two recombination methods presented so far produce two offspring whereas the next methods only produce one. Rather than exchanging whole sequences of genes, *uniform crossover* selects one of the parents for each gene

separately. So, if both parents agree for a certain bit location then the offspring will have the same value at this bit location.

However, if the parents differ (i.e. one of them has a zero and the other has a one) at a certain bit location then the offspring will be assigned zero or one randomly. This method destroys any sequence found in the parents and is therefore only applicable for certain problems.

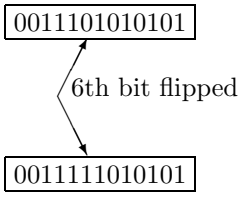
The last recombination operator is called *reverse* or *inversion* and is hardly ever used. It uses only one parent which is split in two random locations. The middle part is then removed and the order of genes in that middle part is reversed before it is placed back.



This keeps some of the fit parent chromosome intact while shuffling some of the genes.

Mutation

Another important genetic operator is known as *mutation*. It is a process of making small changes to genes in a chromosome. In binary chromosomes, this is done by flipping the bit. In the example below, the 6th bit from the left has been flipped.



This mutation leads some members in the next population having different genes than any of the members of the previous population. For example, if all members of the population have a value of zero for the third bit position, say, then thanks to the mutation it is still possible to get an offspring with a value of one at the third bit position.

2.4.5 Example

In 1975, Kenneth Alan De Jong in his PhD thesis[Jon75] on the parameterisation of genetic algorithms presented some optimal parameters for a variety of genetic algorithms. He found that to get good results for the problems he looked at, the following settings are recommendable:

1. population size of 50 to 100 individuals
2. use of single-point crossover as a recombination operator
3. crossover probability of of 0.6
4. mutation probability of 0.001 per bit

Further research involved a genetic algorithm to evolve the parameters for another genetic algorithm. The result of this experiment were similar to De Jong's. However, it has been shown that there are genetic algorithms, for which De Jong's parameter settings are sub-optimal.

This introduction to genetic algorithms concludes with a simple example. Assume the problem is to find the value for x that maximises the following function $f(x) = x^3$ where x is an integer varying from 0 to 63 inclusive.

The obvious encoding for each individual is a bitstring of length six. A population is chosen to contain five individuals which is very small for a genetic algorithm, but suffices for presentation purposes. The fitness function is just taken to be $f(x)$. The initial population is generated randomly.

individual	1	2	3	4	5
generation 1	001011	110000	101101	000100	011100
fitness of generation 1	1331	110592	91125	64	21952
reproduction probability	0.006	0.491	0.405	0.0003	0.098

Table 2.3: First generation of example genetic algorithm

Using roulette wheel sampling, the intermediate population is made up of two copies of individual three and one copy of individual two, four and five. Note, that individual four was 'really lucky' to get picked. From this intermediate generation we now pick two parents and a crossover point at random five times.

Individual two and three cross over after bit two to give offspring 111101 and 100000. Then individual five and two are chosen to cross over after bit one to produce offspring 111100 and 010000. Finally individual three and five cross over after bit three giving 011101 and 101100.

Note, that one of them is dropped at random since only five individuals make up a population and 011101 is randomly chosen not to be in the population. Now the next generation looks like this:

individual	1	2	3	4	5
generation 1	111101	100000	111100	010000	101100
fitness of generation 1	226981	32768	216000	4096	85184
reproduction probability	0.402	0.058	0.382	0.007	0.151

Table 2.4: Second generation of example genetic algorithm

The average fitness of the five individuals from the first generation in Table 2.3 was 45012.8 which improved to 113005.8 in the second generation. The third generation is then generated from the second generation in the same manner.

Note, that no individual has been mutated so far. With a mutation probability of 0.001, the expected probability of a mutation in an individual is 0.006 and the expected probability of mutation in a generation is 0.03, since there are five individuals in a generation. So one can expect a mutated individual in average every 33 generations.

Eventually, the best individual, 111111, is bred or mutated. This individual has a fitness score of 250047 which is the highest possible fitness.

This example was easy for many reasons. Firstly, the encoding was very obvious and the search space was very limited. More importantly, the fitness function was very steep and had no local optima. With a sufficiently large population, the answer is found within very few iterations. The population needs to be large to avoid the following problem.

In the given example, after the second generation, all individuals have a value of zero at the fifth bit location. So mutation is needed in order to find the best individual. Otherwise, the algorithm would never produce an individual with a value of one at bit location five. Mutation on average occurs every 33 generations and not necessarily at bit position five.

This shows, that a small population size may lead to a longer execution time since more generations are needed to find the optimal solution.

2.5 Summary

This chapter provided the necessary background knowledge needed to comprehend the research undertaken for this project. It started with the history of swarms in computer science and an introduction to the emerging field of swarm intelligence.

In addition, the Travelling Salesman Problem was described, its complexity analysed and its relevance compared to combinatorial optimisation problems. This was followed by a description of some famous approximation heuristics and several variants of the TSP.

Furthermore, an introduction to genetic algorithms has been provided, detailing the basic structure, famous encodings, selection methods and genetic operators. This section was concluded with a simple example illustrating the workings of a typical genetic algorithm.

Chapter 3

Software architecture

Real ants find the shortest path between their nest and a food source with the help of indirect communication. They modify the environment they are situated in by dropping pheromone, a signalling chemical. The ants choose where to go next based on the pheromone level they detect on potential paths.

Pheromone levels on a trail increase with the number of ants using the trail but since this pheromone evaporates over time, bad paths will gradually disappear. This leaves a track from the nest to the food source in the environment that will be used by most ants, allowing them to go straight from the nest to the food source and making the whole colony of ants more efficient.

This metaphor from the animal kingdom was used by Dorigo for the *ant system algorithm* which is described in Section 2.3.2 of this report. Its application to the *Travelling Salesman Problem* which was introduced in Section 2.2 is used as the prime example of a combinatorial problem where this algorithm excels.

This project looked into improving various aspect of the standard ant system. Therefore, a basic implementation needs to be provided which is then tweaked to handle several requirements. April[CM94a] was used as the programming language of choice for various reasons that will become clear during the course of this chapter.

3.1 Methodology

Before the different implementations and applications are introduced, a few general comments are deemed to be helpful for the comprehension of the next chapters.

The TSP tries to find the shortest closed tour through a given number of cities without using any city more than once. The ant system algorithm sends ants around the city space, one starting in each city. They drop pheromone

proportional to the reciprocal tour length and their stigmergy behaviour helps them to find short tours.

This ant system algorithm which was introduced in Section 2.3.2 is very generic. It has to be instantiated with various parameters to control certain aspects of the algorithm. A detailed description of the parameters and associated experiments is given in Chapter 5 but some of the more important ones are mentioned presently.

When the algorithm was introduced, what was most important was the ability of the ants to either explore the environment or exploit a food source which was already discovered by other ants. The activity of any given ant is determined probabilistically but their tendency to exploit or explore can be regulated.

Parameters β and q_0 are instrumental for this task. Others like ρ are responsible for controlling the way the pheromone level on each edge is updated.

The main work undertaken for this project was two-fold. On the one hand, applications have been developed in a symbolic programming language that help visualising the working of different variants of the ant system algorithm.

On the other hand, scientific research has been conducted trying to parallelise the algorithm and solving the problems related to the parameters of the system. The results of the parallelisation shortly described in Section 3.3 can be found in Chapter 4 and a detailed analysis of all issues involving system parameters is given in Chapter 5.

A novel approach to finding these parameters is then proposed in Chapter 6 which combines the ant system with ideas from evolutionary computing. It also contains the evaluation of this new algorithm.

For most of the experiments described in the next chapters, well-known and well-researched problem instances have been used. The most famous one is a thirty city problem instance called *Oliver30* after its inventor. Further popular instances are Eilon50 and Eilon75 with fifty and seventy-five cities respectively. The coordinate data for these three instances can be found in Appendix D together with the optimal tour for one of them.

3.2 Multi-threaded single processor version

April facilitates the invocation and management of multiple threads. It also comes with a sophisticated communication infrastructure employing the *InterAgent Communication Model(ICM)* for communication between distributed processes. A more detailed presentation of the language and the ICM can be found in Appendix C.

Trying to simulated the real ants, the most obvious decision is to represent each ant by a separate thread. The diagrams below symbolically represents five ant threads, although more can be used for bigger problem instances.

Another thread is used to implement the environment and message passing is used between ant and environment threads to implement 'pheromone dropping' and the 'sensing of the environment'.

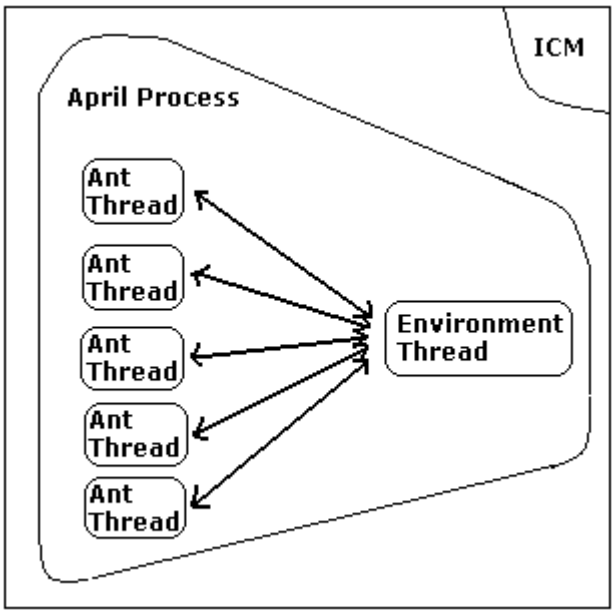


Figure 3.1: Basic ant algorithm architecture with five ant threads and an environment thread within one April process on one machine

The environment thread is invoked in April using the *spawn* command as illustrated below:

```
envHandle = spawn environment(numOfCities,cityList,tau,initTrail,maxCycles);
```

The environment thread receives a number of arguments such as parameters and initialised trails. Some have been omitted here for the sake of clarity. The return value of this spawn operation is a so-called *handle* which can be used to address the thread in future communication. Similarly, a thread is created for each ant and the handle of each ant thread is added to a list.

```
for antNumber in 1 .. numOfCities do
    antHandle = spawn ant(environment,antNumber,numOfCities,distances);
    handleList := [antHandle,.. handleList];
;
```

This code snippet spawns of one thread per city since as many ants are used as there are cities. Each ant thread receives four arguments upon invocation. The environment handle (used to identify the environment process for communication), its number in the ant colony, the total number of cities it has to visit and a static array of distances[1].

The environment process has a *message receive loop* scanning the message queue for incoming messages. The skeleton below shows the three kinds of messages that the environment can receive.

The first and most important one arrives from an ant after it found a tour. It contains the symbol *'UpdateEnvironment* to indicate the type of message together with the length of the tour and the tour itself in form of a list.

```
repeat
  ('UpdateEnvironment,length,list) ->> {
    do housekeeping
    if updateCounter == numberOfAnts then {
      remember best tour of iteration if it is an improvement
      ...
      reset all variables that are local to the iteration
      iterationCounter +:= 1;
      do pheromone updates
      if iterationCounter > maxIteration then {
        for antHandle in handleList do
          'DieAnt >> antHandle;
        ;
      } else {
        tell all ants to find another tour and inform them of
                    changes to the environment
      };
    }
  } |
  'StopEnvironment ->> {
    display and log the best solution found
    'sio_end_Env >> self();
  } |
  ('listOfAntHandles,X) ->> {
    upon receipt of their handles, start all ants
  }
until 'sio_end_Env;
```

The environment records the result from this ant, increments the *updateCounter* variable and waits for the next message. After receiving one message from each ant (they should all take roughly the same time to find a tour), it updates the pheromone levels on all the trails in the environment according to the procedure described in Section 2.3.2.

[1] Arrays to not exist in April as such and needed to be implemented by the author using theta environments. See Appendix C for more details on the implementation language.

If the required number of iterations has been exceeded, it then tells all ant threads to terminate. Otherwise it initiates the next iteration by telling the ant threads to find another tour. In this initiating message, the environment also informs each ant about the changes to the environment.

This has been found to improve efficiency over the 'real world way', where each ant would request the local pheromone level each time it wants to 'smell' something.

It remains to show how each ant finds a tour. According to their parameterisation, they either tend to exploit a given food source or explore the environment for other food sources. This attitude is controlled by parameter q_0 and β as described in Section 3.1. These parameters are more closely analysed in the forthcoming Chapter 5.

The ant threads have message receive loops, too. When an ant receives a message from the environment thread asking it to find a tour, it starts in the start city which is unique for each ant. Then it continues to pick a next city from those not yet in the tour and move to this next city until the tour is complete.

The decision where to move next is made according to the formulae introduced in Section 2.3.2 and is implemented like so:

```
pickNextCity(city,numOfCities,tabuList,distances,tau,beta,q0) => valof {
  initial housekeeping
  for x in 1 .. numOfCities do {
    if !(x in tabuList) then {
      trail = tau.get(city,x);
      visibility = 1 / distances.get(city,x);
      score = trail * pow(visibility,beta);
      keep track of maximum score
      ...
      build up list with options and probabilities
      theOptions := [(denom,denom+score,x),..theOptions];
      denom +:= score;
    };
  };
  use q0 to either exploit or explore
  randomQZero = rand(1);
  if randomQZero < qZero then {
    theResult := maxScoreCity;
  } else {
    probabilistically pick a city from the options
  };
  valis theResult;
};
```

From a given *city*, all other cities that are not in the *tabuList* (i.e. already visited) are assigned a *score* depending on the distance (between the current city and the city in question) and pheromone level (on the connecting path).

Then with a probability of q_0 the highest scoring city is chosen as the next city. Otherwise, a city is randomly selected from all *theOptions* where better scoring cities are more likely to be picked.

3.3 Parallel algorithm on multiple processors

Having described how the basic algorithm is implemented in April, several continuative experiments are outlined below. The first part of this project concerned itself with ways of exploiting the inherent parallelism in the algorithm. Since April is very suitable for programming in a distributed environment, the following experiments have been conducted. Note, that quantitative and qualitative evaluation of these experiments are presented in Chapter 4.

Initially, it was decided to move the environment thread to another machine. This promises a speed-up, since the ant processes would not preempt the CPU while the environment is updating. During this period they cannot do anything anyway until the environment is finished and has sent them the 'go ahead' messages.

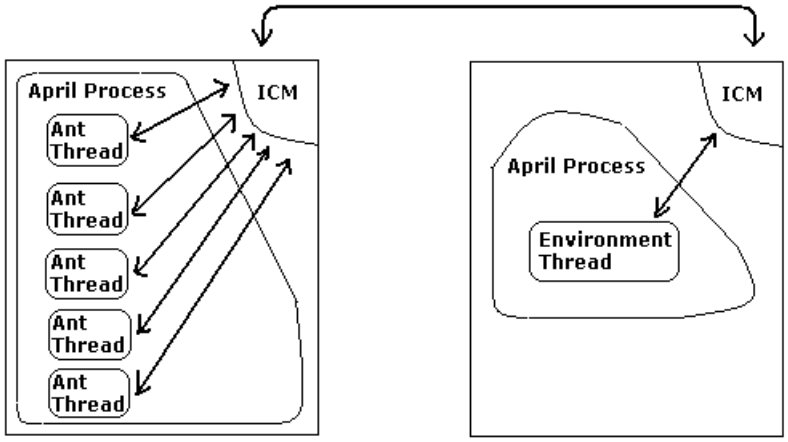


Figure 3.2: Splitting up ant threads and environment thread onto different April processes running on different machines

The experimental setup involved two machines represented by the two big boxes above, each running an invocation of April. The left one contains an April process with five ant threads and the one on the right contains another April process with the environment thread. In order for a thread on one machine to send a message to a thread on the other machine, they have to use the ICM[2].

[2]described in Appendix C

The message sent by a thread is forwarded by the April process to the ICM on the sender's machine and from there to the ICM on the receiver's machine as depicted in Figure 3.2. This ICM forwards the message to the April process of the receiving thread which then delivers it to the corresponding April process. This April process places the message in the message queue of the receiving thread as specified by the sender.

This setup can be extended to include more machines. The five ant threads in the previous two diagrams are used symbolically for all fifty ants used in the parallelisation experiments. In theory, one can use fifty machines each running an April process containing a single ant thread.

However, communication cost considerations detailed in Chapter 4 limit the number of machines that yield an speed improvement. The following figure illustrates how the approach is extended to three machines (one hosting the environment and two that host the ants).

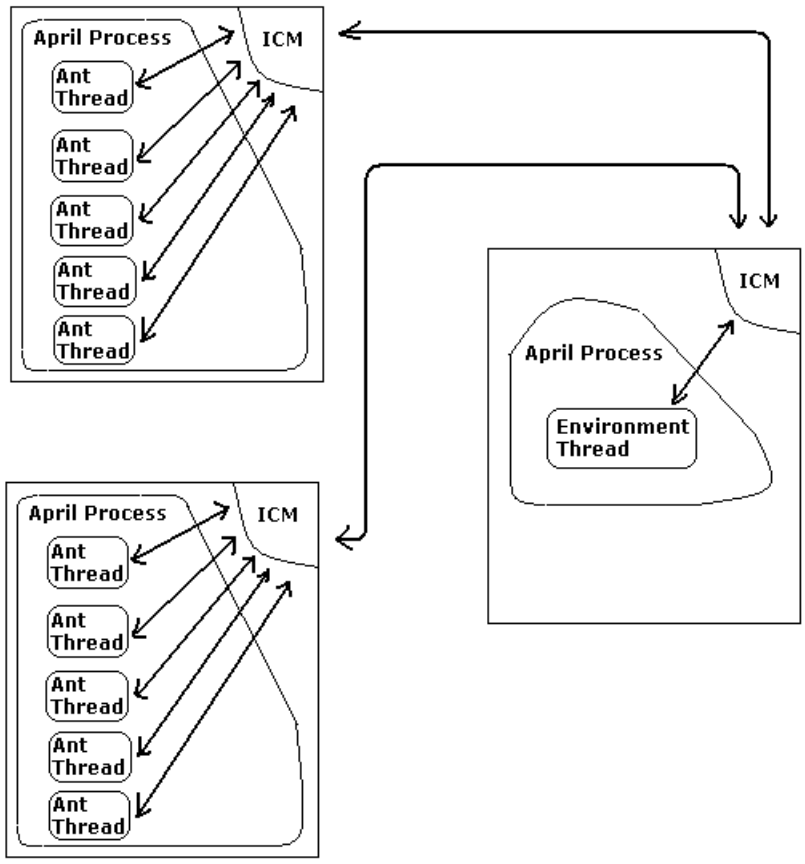


Figure 3.3: Using three machines to parallelise the ant system algorithm

This was found to improve the time it takes to complete a given number of iterations of the algorithm significantly. Further experiments have been conducted and detailed results will be presented in Chapter 4. Another concept that improves the performance of the algorithm with respect to the time it takes to complete 100 iterations is that of *sub-colonies*.

Figures 3.2 and 3.3 indicated how each ant thread communicates with the environment thread. Every iteration involved 50 messages (one from each ant) that went over the network to arrive at the environment process. Using a *sub-colony thread* reduces the message flow over the network dramatically.

Each ant sends a message to the local sub-colony thread which is done very fast without involvement of the ICM. This sub-colony thread then sends a cumulative message over the network.

Similarly, when the environment thread informs the ants about changes to the pheromone levels on the paths, it only sends one message to the sub-colony thread on each machine, which then forwards the information locally to the ant threads.

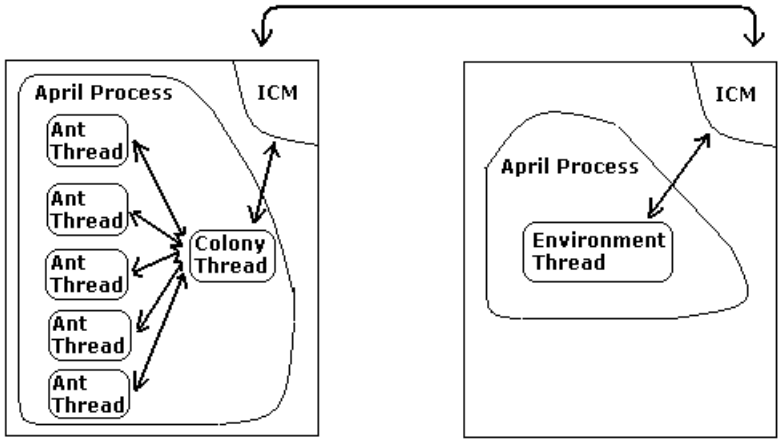


Figure 3.4: Using sub-colonies to reduce overall traffic across the network

This should just give a flavour of the experiments done to utilise the inherent parallelism found in the ant system algorithms. A total of nine different parallel variants have been investigated, both lossless and lossy ones with respect to solution quality.

Reductions of more than 85% of the execution time have been found and a complete description of these experiments and evaluation of the accompanying results are presented in Chapter 4.

3.4 Asymmetric variant

The standard algorithm used for the parallelisation experiments was applied to Euclidean problem instances of the TSP. These instances are all symmetric since the distance between two cities is just the geographic distance between them. So the distance between A and B is exactly the same as the distance between B and A.

However, there are applications where this Euclidean distances do not apply. For example, the 'distance' between two cities could represent the time it takes to travel from one to the other. This time can be different depending on the direction in which the salesman travels due to traffic jams on the motorway.

This has an influence on the optimal tour length as illustrated below:

Figure 3.5: Symmetric vs. asymmetric problem instance

For the asymmetric version on the right only the shortest distance is displayed. This means that the distance between A and C is 5 (as depicted) but the distance between C and A is more than 5. If one ignores the direction on the arrows, the total tour length is 12 as it is for the symmetric example on the left.

However, to complete a tour without ignoring the direction of the arrows is not possible with a length of 12. It either involves CA (which is longer than 5) or CB and BA which are longer than BC and AB.

This flavour of the TSP required some changes to implementation of the ant system algorithm. Most notably the finding of a tour by the ants and to the updating of the environment needs to take the concept of direction into account.

```
for (lengthOfTour,pairListOfCoordinates) in resultsFromAnts do {
    for (i,j) in pairListOfCoordinates do {
        deltaTau.add(i,j,... some amount);
        deltaTau.add(j,i,... some amount); /* remove for ATSP */
    };
};
```

This fragment of code is used by the ant system for the Euclidean version of the TSP. It drops pheromone on each road that was travelled for each of the ants.

Note, that it drops the same amount of pheromone on the edge between i and j as it does for the edge between j and i. In fact, there is only one edge connecting i and j, but implemented in such a way that it can be accessed from both ends.

In the implementation for the asymmetric TSP variant, the second of the two lines in the inner loop is removed, since pheromone is only dropped on one of the edges (only in the direction of travel).

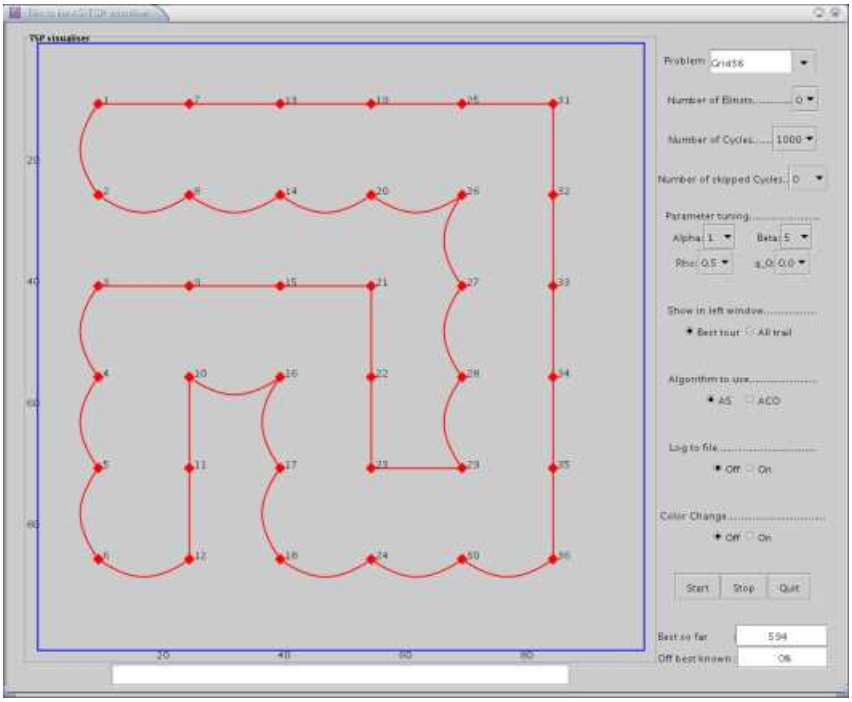


Figure 3.6: Screenshot of the ant application for the Asymmetric TSP

The screenshot in Figure 3.6 shows the ant application ATSP-Ant[3] running for the asymmetric problem instance of a 6 by 6 grid. The straight lines represent the distance in one direction (normal euclidean distance) and the bend lines represent the distance in the other direction which is taken to be 1.2 times the euclidean distance between the two cities.

This ratio is a parameter to the application and chosen to be 1.2 in order to discourage the ants from using diagonals for this screenshot.

[3]implemented by the author

3.5 Different kinds of dynamic behaviour

Another interesting variation is to apply the ant system to *dynamic problem instances*. Dorigo in [DMC96] predicted that the real strength of his algorithm will only come to the fore when it is applied to dynamic instances.

The reason why the ant system should be better suited to dynamic problems is that pheromone information can be re-used without restarting the computation from scratch. On the other hand, this might bias the ants too much in their behaviour, so that no good solution is found after the problem instance changed. A compromise can be found by smoothing the pheromone levels somewhat as proposed in [CS02].

Taking the TSP as the problem of choice, one can think of two different notions of *dynamic*. One is to change the distances between cities on the fly. This could be used to model the traffic on motorways as in the asymmetric example from above. *Traffic jams* would dynamically appear and disperse.

Variations of the ant system have been proposed in [GM01]. During the course of this project a modification of this ant system for the dynamic TSP has been implemented that allows users to change the distances interactively by moving cities around during execution time.

This changes the inter-city distances and might result in changes to the optimal tour. However, after several hundred iterations the pheromone levels are too pronounced. The change in the distance-based heuristic *visibility* will not have any effect.

Therefore, the pheromone levels should be normalised. Resetting them to zero would result in the loss of domain knowledge accumulated so far. In contrast, subtly smoothing the pheromone in the area where the change took place results in a setting where good solutions can be found fast.

A second dynamic variant allows the user to dynamically add or remove cities from the TSP instance. This could represent the metaphor of a salesman who is already travelling but changes his mind and adds or removes some of his destinations.

This has been implemented, too. Several experiments have been conducted, indicating that adding or removing a city should mostly affect the optimal tour in the local area where the change happened. Other parts of the tour remain unchanged.

Unfortunately, due to the lack of benchmarks no reasonable comparison with other heuristics was possible.

The application which visualises the behaviour of the ant system in dynamic settings, can be found as an April executable in the project directory of the author. Source code, both in April and XML is also available from this resource.

3.6 Summary

This chapter introduced April, the ICM and DialoX by means of examples. The general software architecture and the methodology used for most of the experiments were described. A mixture of April and pseudo code was employed to illustrate the working of the main components of the system.

Block diagrams gave an indication of some of the parallelisation strategies (like distributing the ants, reducing the communication or using sub-colony threads) that will be looked at in the next chapter. The main result, a speed-up of more than 85% was hinted at.

Moreover, the asymmetric variant of the TSP has been explained and the changes to the ant system application have been highlighted.

The chapter concluded with a brief digression on the dynamic version of the TSP and the issues that the ant system faces when applied to this dynamic TSP.

Chapter 4

On the parallelisation of the ant algorithm

Parallelisation - the rewriting of an existing program or algorithm to run on a system with multiple processing units - can result in a speed-up of an existing algorithm as long as communication cost between processing units are not prohibitive.

This chapter will introduce several ways of exploiting the inherent parallelism in the *standard ant system* and propose variations of the algorithm that allow for even more parallelism. The ant system operates many individual agents that find solutions to a given problem via *indirect* communication.

The structure of the algorithm lends itself to a parallel implementation using separate processes for each ant rather then 'sending' each ant on a tour sequentially. However, running each process on a different machine would result in highly increased communication between machines and hence, a communication cost that makes this kind of parallelisation infeasible.

In a technical report, Bullnheimer et al. [BKS97] describe two parallelisation strategies for the ant system. Firstly, a straight-forward synchronous algorithm is presented and then a partially asynchronous version is suggested. This second approach reduces the need for communication which provides a further speed-up.

This part of the project first looked at what degree of improvement can be achieved by separating the ant processes and the environment process. Then synchronous as well as several asynchronous variants of the algorithm are analysed and compared.

The trade-off between parallelism and communication cost is then investigated and a compromise between them is sought. We conclude with the presentation of two versions of the notion of *sub-colonies* and a summary of all the results, both quantitative and qualitative, presented in this chapter.

4.1 Experimental setting

For the quantitative experiments undertaken by the author, six identical high performance computers have been used. It needed to be ensured, that the experiments had sole control over the CPU, which limited the number of machines that could be used. However, an inconsistent share of the processor due to other users' demands would render any result useless.

Identical machines were employed to avoid influencing the results by differences in hardware. Each experiment was run several times and the findings were averaged in order to obtain statistically significant results.

Also, different runs used different machines wherever possible in order to cancel out differences in network routing delays.

The machines named media01 up to media06 in the project lab were used in their idle times on weekends with permission from other project students who had been allocated these machines. They each have a Pentium IV chip which clocks at 2.8GHz and 1GB of RAM. In all the following tables, the machine names will be abbreviated (m04 represents media04.doc.ic.ac.uk).

All experimental runs involved 100 iterations of the algorithm augmented with a variation of the local search procedure *2-OPT* introduced in Section 2.2.4. This was done to lengthen the time each iteration of the algorithm takes and spread out the results. Local optimisation is performed by each ant before it reports back to the environment and is therefore done in parallel.

The variation of the *2-OPT* search procedure used here, takes the tour found by the ant and attempts to improve it by swapping over any pair of adjacent cities. The best tour thus obtained is used as the tour for the ant in question at this iteration.

All parameters to the standard algorithm (to be introduced in Chapter 5) were fixed for all experiments at:

Parameter	Value
α	1
β	7
γ	0
ρ	0.4
q_0	0
τ_0	0.0001
number of ants	number of cities
number of iterations	100

Table 4.1: Parameters used for the parallelisation experiments

Note, the last parameter changed slightly in the experiments on sub-colonies where each colony ran some iterations independently. However, the overall number of iterations was fixed at 100 throughout.

Running the environment process and the ant agents on the same machine, the next table shows the impact of the local search procedure on the running time of the algorithm. The problem instance Eilon75 was used in this experiment, details of which can be found in Appendix D.

The four columns represent whether or not local optimisation *2-OPT* was employed, the time taken in seconds, the tour length found and the percentage off the best known solution for this problem instance. More details can be found in Appendix E.1.

2-OPT	Time	Length	Percent off best known tour
No	**1628.80**	555.226	2.38
Yes	**1981.26**	550.096	1.434

Table 4.2: Influence of local search on execution time

The mean running time without the local search procedure *2-OPT* was 1628.8 seconds compared to 1981.26 seconds when the local search was enabled. So, on average, each run took 21.64% longer to complete when local search was added.

The quality of the tour found by the algorithm using the extra local search is slightly better than without *2-OPT*. However, the quality of solution is of no concern for the analysis in this chapter. We are only concerned with the time it takes to perform 100 iterations of the algorithm.

All experiments that follow use this local search procedure to spread out the resulting times.

4.2 Separating environment and ant processes

A first step to improve the execution speed of the algorithm, is to separate the housekeeping operations of the environment from the active searching of the ants. Each ant has to find a tour by picking n cities (for a n-city problem instance), calculating the length of this tour and then running the local optimiser *2-OPT*, checking n swaps of adjacent cities.

After all ants have reported their best tour together with the best length to the environment process, this process has to update the pheromone levels on each road between two cities for each ant according to the equations described in Section 2.3.2.

While this updating is taking place, the environment process is the only process ready to run, since all the ant processes are waiting for a message telling them to start their next iteration. This message will only arrive after the environment process has finished updating the pheromone levels. The ant processes

need this message to continue execution, since they need to be informed of the changes to the environment.

But when only one machine is used for all processes, control is switched to the ant processes for them to check their message queue, even when the environment is still not finished updating. This does not happen, if the environment process is the only process running on the machine and is proved by the following experiment.

Firstly, all ant processes and the environment process ran on the same machine. Several runs were made using different machines depending on available idle machines and to allow for a good average.

Secondly, media04 was chosen to be the machine on which the environment process will run. This choice was arbitrary from a computational point of view, but based on the fact, that it was allocated to the author for the duration of this project. The 50 processes representing the ants, were then run on a different machine in order to avoid the superfluous switching of processor control mentioned above.

Environment	Time
on same machine as ants	**607.5**
on separate machine	**347.1**

Table 4.3: Separating ants and environment processes

The Table 4.3 shows average results for this experiment detailed results can be found in Appendix E.2. It can be seen, that using two machines instead of one already gives a very significant improvement. Separating the environment process from the ant processes gives a speed-up of more than 40%. It is important to understand, that this experiment still ran all *ant processes* on the same machine.

So, the real parallelism inherent in the algorithm has not been utilised at all so far.

4.3 Distributing ants over multiple machines

This parallelism can be made use of by distributing the ants across several machines. For the experiments in this section, again the machine media04 was used to run the environment process and the local search *2-OPT* was employed.

The last section showed that the time taken for 100 iterations can be reduced from 607.5 seconds to 347.1 seconds by using a separate machine to run the ant processes on.

This section illustrated how this time can be further improved to 261.2 seconds by using *four* machines to run the ant processes on. It is interesting to see, that even the use of just *two* extra machines gives a significant improvement on the time taken by the algorithm.

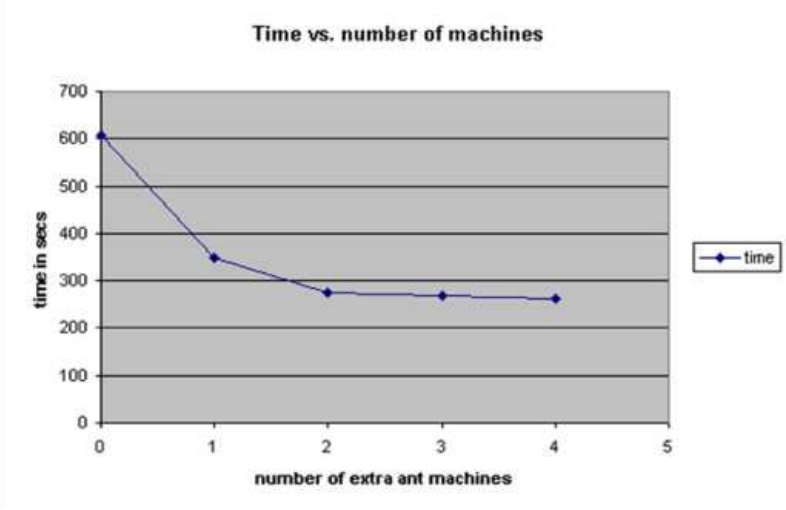


Figure 4.1: Time taken for 100 iterations given a number of processors

It is expected that using five or six extra machines would speed up the execution even further, although the extra speed-up gained will be minimal. However, due to the lack of more idle machines, this could not be tested.

It is comforting to know that the curve flattened out dramatically for even a small number of machines and so the need to use more than two or three extra machines is not given. This can be seen in the plotted diagram in Figure 4.1.

In fact, two extra machines seem to give sufficient speed-up and any further parallelisation will only result in a minimal improvement. However, this result is dependent on the problem instance chosen. Any other problem instance with a similar number of cities will yield the same result. But if the number of cities are increased by an order of magnitude, using more machines will clearly be desirable.

Over all the 25 runs for the problem instance Eilon50 from this and the previous section, the mean length was 434.44 which is just 1.51% off the optimal solution. This result was achieved in 100 iterations with every ant informing the environment after each iteration. Detailed results can be found in Appendix E.3.

The next section will demonstrate how to improve the time taken for these 100 iterations even further by reducing the communication between the ants

and the environment. This will lead to a loss of quality of the solution found whereas the parallelisation described so far was lossless in this respect.

4.4 Reduced communication

The costly communication between processes can be reduced by limiting the number of times an ant is allowed to update the environment. Since the ants find their tours probabilistically, it is likely that the same ant will find different tours on repeated attempts in the same environment.

Only the best of these tours needs to be reported to the environment process and the next experiment is trying to establish an optimal balance between the number of repeated local attempts and updates send to the environment process.

Clearly, if there is too little communication between the ants and the environment, then the ants cannot learn from each others attempts and the positive feedback, *stigmergy*, cannot take effect. Too much communication on the other hand will lead to sub-optimal performance with respect to execution time.

This experiment used the problem instance *Eilon50*. For all runs, the environment process executed on machine media04 and the ant processes were distributed over two extra machines, since the last section revealed that this is optimal for this particular problem instance. The local optimiser *2-OPT* was used again.

It is worth remembering a result from the last section, namely that doing just one local attempt (i.e. reporting each tour to the environment) with the ant processes distributed over two extra machines took 274.8 seconds for 100 iterations.

Local Attempts	Time	Length	Percent off best known tour
2	**143.4**	431.32	0.786
5	**70.08**	433.78	1.362
10	**62.28**	434.06	1.432
14	**64.06**	434.32	1.498
20	**63.68**	436.92	2.094

Table 4.4: Results of reduced communication experiments

This table shows that this time can almost be halved to 143.4 seconds by allowing each ant to find two tours probabilistically and only reporting back to the environment the better of these two. Experimental runs have been conducted with x set to 2, 5, 10, 14 and 20 where x is the number of tours each ant has to find before reporting the best to the environment (detailed results in Appendix E.4).

Total iterations were 100/x so that the total number of tours per ant does not change. Note, that for x=14, seven total iterations have taken place, meaning that the number of tours per ant was only 98. The average time of 62.78 seconds has been scaled by a factor of $\frac{100}{98}$ to get the accurate result displayed.

The following graph shows, that for any $x > 10$, the communication overhead is negligible in the system used for these experiments (i.e. the DoC network and machines with Pentium IV processors). The improvement from $x = 1$ to $x = 2$ is largest, but values of five or ten for x are also reasonable.

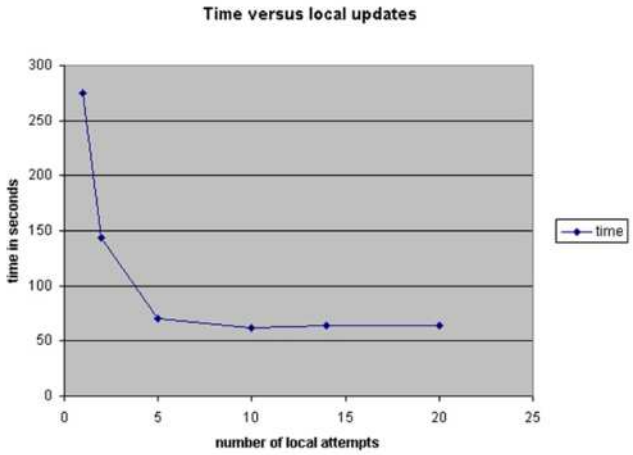


Figure 4.2: Time taken for 100 iterations given a number of local updates

It remains to check what influence the reduced communication has on the quality of the solution returned after each ant has found 100 tours. We would expect the quality for $x = 20$ to be rather poor, since 20 local attempts mean, that the environment only got updated five times.

This does not allow for much collaboration between the ants and the stigmergy effect described in Section 2.3.2 cannot take place.

Indeed, the graph in Figure 4.3 shows that reducing the communication too much, results in solutions of poor quality. Interestingly, the graph reveals an improvement in quality when two attempts are undertaken locally by each ant instead of one.

This could be explained by the fact that fewer updates initially prevent disadvantageous paths to be marked with a lot of pheromone. Also, the quality of the tours reported to the environment is better, since every ant has two attempts and only the better one is sent to the environment.

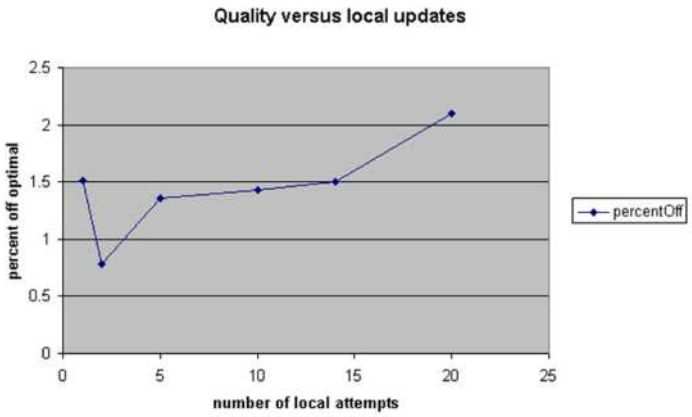


Figure 4.3: Quality of solutions given a number of local updates

Looking at both of the last two graphs, it is obvious, that using $x = 2$ instead of $x = 1$ is preferable. It improves both quality and time by almost 50%. However, using $x = 5$ or $x = 10$ is also feasible. While these last two values result in a small loss of quality, they improve the time it takes to execute 100 iterations of the algorithm.

A compromise has to be made between the desired quality and the time one is willing to wait for the completion of the 100 iterations. If solution quality is most important, then $x = 2$ should be chosen. Otherwise, if execution speed is more important, values of five or ten for x are more appropriate. Figure 4.4 shows that the optimal number of local updates lies between 2 and 5.

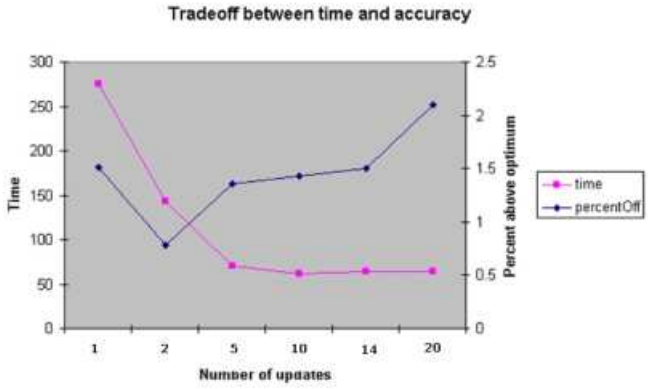


Figure 4.4: Tradeoff between execution time and accuracy

4.5 Sub-colonies

This section will introduce the notion of *sub-colonies* for the ant system algorithm. The reduction in communication described in the previous section still left room for improvement. After all the ant processes reported their best tour to the environment process, this process updated the pheromone levels on all edges before sending a message *to each ant process*, informing them about the changes.

Now, these messages are identical for each ant, since they contain the new pheromone levels of the environment. Section 4.2 described, why it is desirable to run the environment process on a different machine than the ant processes. The number of messages passed between these machines, should be minimal.

A first step in this direction is to run a *sub-colony thread* on each machine that host ant processes. The environment process then only has to send one message across the network to each machine's sub-colony thread which forwards it locally to all ant processes running on the same machine.

This reduces the number of messages that have to be sent across the network. Local messages can be transmitted much faster resulting in a significant speed-up of the execution time of the algorithm.

This sub-colony metaphor has no impact on the quality of the solution, since the same number of iterations and the same number of pheromone trail updates are performed. The only difference is in the amount of communication that takes place over the network.

In a similar experiment without sub-colony threads described in Section 4.3 the execution time was 274.8 seconds on average. Using the idea of limited cross-network traffic reduces the time the algorithm takes to complete 100 iterations to 189.2 seconds, which is an **improvement of 31%**.

But even now, each ant tells the environment directly about its findings. The sub-colony thread is only used for communication in one direction. This leaves still room for improvement.

The proposed idea is to use the sub-colony thread for communication in both directions. This thread will collect the tours found by all ants in this sub-colony and then only send the best ant from this colony to the environment.

# of Machines	Time without sub-col	Time with sub-col	Improvement
2	274.8	**169.62**	38.0%
3	267.7	**118.3**	55.8%
4	216.2	**86.12**	67.0%

Table 4.5: Two-way sub-colony communication and ants on two, three and four machines

It can be argued that this is somewhat flawed, since the environment does pheromone level updates for each ant. Remember, the ants 'use' some of the pheromone while walking along a path as described in Section 2.3.2.

A possible solution is to use local environments. However, updating and synchronising between these local environments was found to be really messy. So just the best tour from each sub-colony is sent to the environment and we change the metaphor of the ant system. *The best ant is allowed to walk its walk again in the global environment.*

The thus modified environment is then sent to the sub-colony threads which forward it to every ant and a new global iteration can begin.

Table 4.5 shows the averaged results of experiments with two-way sub-colony communication. Note, that he average running time of 169.62 seconds using two machines is now an improvement of 38% over the algorithm without sub-colony threads and an improvement of more than 10% over the algorithm with one-way communication via the sub-colony threads.

The graph in Figure 4.5 compares two parallel variants of the ant algorithm. One, where no sub-colony threads were used and the ant processes were distributed over two, three and four machines. The second algorithm, the one with superior performance, uses two-way communication via sub-colony threads as explained earlier.

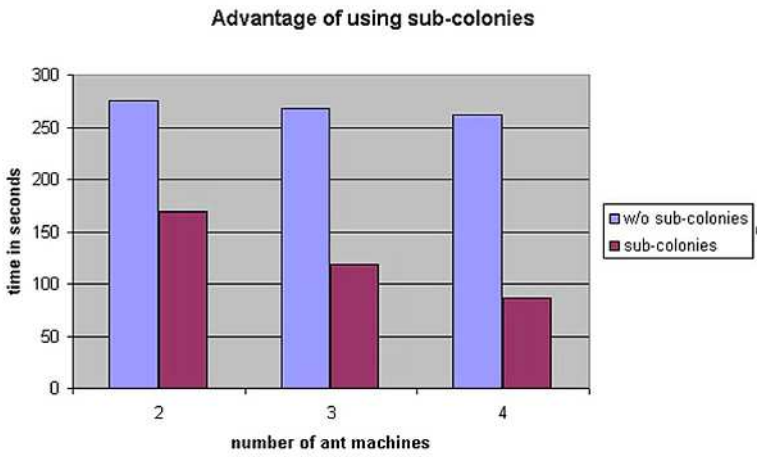


Figure 4.5: Comparison of performance with and without sub-colony threads

Again, detailed results for these experiments can be found in Appendix E.5.

4.6 Aggregated results

This chapter investigated several approaches and proposed new ones to parallelise the *ant system* algorithm. Throughout most of this chapter, the Eilon50 problem described in Appendix D has been used to compare the different approaches.

Some of the parallelisation strategies that are lossless with respect to the quality of the solution are summarised in table 4.6 together with the time it takes each of them to execute 100 iterations. These 100 iterations equal 5000 total tours (50 per iteration) which is an infinitesimal fraction of the more than $3 * 10^{62}$ possible tours for this problem.

One of these possible tours has the optimal length of 427.96 and can only be found by exhaustive search. However, all the experimental runs that were mainly concerned with speeding up the *ant system* algorithm, found tours within 4.07% of this optimal tour. Most of them were even within 1% of the length of the optimal tour.

These high quality solutions were found after 100 iterations of the algorithm which takes roughly ten minutes if no parallelisation is used. However, comparing the time it took the first experiment (607.5 seconds in Table 4.6) with the time it took the fastest strategy (86.1 seconds for experiment nine in the same table) to complete these 100 iterations, a total speed-up of 85.8% was achieved.

# of Experiment	# of Machines	Strategy	Time
1	1	no parallelisation	607.5 secs
2	2	Split env. from ants	347.1 secs
3	3	Distribute ants	274.8 secs
4	4	Distribute ants	267.7 secs
5	5	Distribute ants	261.2 secs
6	3	Sub-colony one-way	189.2 secs
7	3	Sub-colony both ways	169.6 secs
8	4	Sub-colony both ways	118.3 secs
9	5	Sub-colony both ways	86.1 secs

Table 4.6: Summary of different parallelisation strategies

Note, that this improvement of 85.8% can be achieved without the solution quality being affected. If one is willing to sacrifice a small amount of quality then the strategy to reduce communication between the environment process and the ant processes gives a further improvement of 77.3% as described in section 4.4. However, this does not directly translate to the other experiments.

The problem tackled by the strategy in section 4.4 is the high number of messages flowing between the machine hosting the environment process and the

machines hosting the ant processes. The same problem has been tackled by the introduction of the notion of sub-colonies. However, a further improvement can be found by combining both strategies and reducing the communication between the environment process and the sub-colony threads.

This improvement was found to be statistically irrelevant. This can be explained by the fact that the *sub-colony strategy* sends only one message per iteration which clearly cannot be the bottleneck. Reducing this to one message for every two, five or ten iterations saves only fractions of a second.

4.7 Summary

This chapter investigated several parallelisation strategies. First, the experimental setting was described and tests conducted without parallel execution to obtain a benchmark for the ant system implemented in April.

The environment was then separated from the ant processes yielding an improvement of more than 40%. This was extended to 57% by using multiple machines to run the ant processes. It was found that a number of machines of two or three is satisfactory for a thirty-city problem instance of the TSP.

A notion of reduced communication has that been subject of this study and was found to improve the execution time even further. However, reducing the communication too much led to a loss of solution quality since without sufficient communication no stigmergy, the positive feedback via the environment, can take place.

The concept of sub-colonies was then established and presented as a one-way or two-way communication variant. It was found to provide an improvement over the ant system without parallelisation of more than 85%. This speed-up is achieve lossless in that the solution quality is not affected by the parallelisation.

Finally, the results are aggregated and put into perspective.

Chapter 5

On optimal parameters

In the previous chapter, the way how execution speed is affected by different parallelisation strategies has been analysed. Now, the choice of parameters will be subject of the research described.

Using good parameter values for the algorithm described in Section 2.3.2 is important both with the respect to efficiency and effectiveness. Inappropriate values will certainly slow down the discovery process of the best solution to a given problem instance but may even prevent the algorithm from finding the best solution at all. This has been shown since using optimal parameters a new best tour was found for the Eilon50 problem instance.

Dorigo and Stützle in [DS03] suggested that the exact values of the parameters are often problem dependent. Throughout the literature one can find many different values used in many variations of the original *ant system* algorithm. However, the author of this report predicts, that even within one particular problem (the *Travelling Salesman Problem* is chosen here) optimal parameters may vary for different instances of the problem.

This chapter will introduce the parameters that one has to specify to instantiate the ant colony meta-heuristic together with a justification for the range of sensible values for each of them. A classification for instances of the Travelling Salesman Problem is then proposed.

One particular problem instance is then analysed in detail, before the best parameters found for this instance are tested on other instances. A speed-up of execution time of more than 90% was found over the use of parameter settings from the literature. The generalised results are reported and the need for an automatic method to set parameters is justified.

5.1 Description of parameters

$$p_{ij}(t) = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha} * [\eta_{ij}]^{\beta}}{\sum_{k \in allowed_k} [\tau_{ik}(t)]^{\alpha} * [\eta_{ik}]^{\beta}} & if j \in allowed_k \\ 0 & otherwise \end{cases} \tag{5.1}$$

$$\tau_{ij}(new) = \rho * \tau_{ij}(old) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{5.2}$$

where m is the number of ants. These equations are repeated here from section 2.3.2 for ease of reference.

α - controls the relative importance of pheromone. Since it is used in equation 2.1 as an exponent for the trail (pheromone) level, it should have a value of at least 1. A value of zero would turn the algorithm into a standard greedy one, since no pheromone information would be used.Further inspection of the equation however tells us, that one can ignore either alpha or beta.

This is possible, since the ratio $\frac{\alpha}{\beta}$ can still be controlled by fixing one of the two parameters at 1 and varying the other parameter. It was decided to fix alpha and vary beta (though this choice is arbitrary) and hence this parameter was set to 1 for all other experiments.

β - controls the ratio between importance of pheromone and importance of distance between cities. In equation 5.1, beta is the exponent of visibility η_{ij}, which is defined as 1/distance. So visibility ranges between 0 (very big distance) and 1 (small distance).

The higher the value beta will adopt, the smaller will be the component that takes distances into account. So a high beta value means, a lot of exploration will happen during the execution of the algorithm. A beta value of zero means, that distance is completely irrelevant and only the (initial) trail level τ_0 is used to choose which path to take.

Since τ_0 is the same for every edge, this does not appear to be sensible. Hence a range of $0 < \beta < 15$ is proposed. The upper limit is solely there to facilitate computation. In fact, the highest sensible value for beta in any combination of parameters was found to be 13.

If beta is small, we get more of a greedy algorithm, since we put more emphasis on distance then pheromone. Ants in the real world probably uses a high beta value, since they can't really measure the distance but only smell pheromone around them. In order to find the optimal tour, we have to 'help' the ants a bit by providing distance information, but still employ their exploration behaviour by setting beta to some reasonably small number.

γ - controls the importance of the best solution from the current iteration (*iteration best*) compared to the best solution found over all solutions so far (*global best*). As described in section 2.3.2, all ants 'use up' some of the pheromone on the trails they travel on. But only certain ants can 'drop' pheromone to reinforce the tour they took.

Gamma represents the number of times the *iteration best* ant is allowed to 'drop' pheromone for every time the *global best* ant is used in the update procedure. A first constraint on this parameter is $0 \leq \gamma \leq numAnts$ since we do not update the environment more than once for each ant.

Early experiments showed, that after the initial three or four iterations, the *global best* and *iteration best* solutions coincide. The sample execution trace in Appendix F can be consulted to check this. Hence, this parameter was set to zero for all other experiments. Only the globally best solution is used to reinforce the pheromone levels.

ρ - controls the evaporation of pheromone in the environment. It is intuitively limited by $0 \leq \rho \leq 1$. Equation 5.2 from the original algorithm shows, that $\rho = 0$ means we 'wipe clean' the environment after every iteration.

A value of 1 would lead to continually increased pheromone levels since pheromone would never evaporate. Clearly this is undesirable and a sensible range of $0 < \rho < 1$ is proposed.

q_0 - is a parameter introduced by the ant colony optimisation meta-heuristic. It controls explicitly the ratio between exploitation and exploration. Its range is $0 \leq q_0 \leq 1$ where a value of 1 means that always the highest scoring city from equation 5.1 is chosen next.

A value of zero forces the algorithm to choose a city according to the probability distribution defined by equation 5.1. Values between these two extremes enable the system to probabilistically do one or the other where q_0 is the probability of choosing the highest scoring city next.

τ_0 - specifies the initial pheromone level on all edges. Intuitively, one would start with no pheromone and $\tau_0 = 0$. However, for computational purposes, a small but non-zero value is preferable. Bonabeau, Dorigo and Theraulaz in [BDT99] suggest a value of 1/(n*best_greedy_length) where n is the number of cities in the problem instance.

numAnts - represents the number of ants used by the algorithm. Clearly, the more ants we use, the slower the algorithm will be. However, using too few ants means, that no meaningful pheromone matrix is created and the search will take more iterations to find the best solution.

With the possibilities of parallelising the algorithms described in Chapter 4, speed is no longer such an issue. Ants can be distributed over several machines and hence the upper limit is not fixed.

Initial experimentation showed that the product of ants and iterations needed to find the optimal solution is fairly constant and so this parameter is irrelevant. For computational purposes, the number of ants used in the algorithm is set equal to the number of cities in the problem instance and each ant starts in a different city to achieve diversity with respect of tours generated.

max_cycles - controls the number of iterations of the algorithm. A range of $10 \leq max\_cycles \leq 1000$ is suggested. Fewer iterations will not be enough to produce decent solutions and after 1000 iterations improvements will rarely be found.

To summarise this section, we propose that half of the parameters are kept constant:

Parameter	Constant Value
α	1
γ	0
τ_0	1/(n*best_greedy_length)
$numAnts$	number of cities

Table 5.1: The constant parameters

The remaining four parameters vary in the ranges displayed in the following table:

Parameter	Value Range
β	$0 < \beta < 15$
ρ	$0 < \rho < 1$
q_0	$0 \leq q_0 \leq 1$
max_cycles	$10 \leq max\_cycles \leq 1000$

Table 5.2: The variable parameters

The last parameter in this table has no influence over how many iterations the algorithm takes to find the optimal solution. However, it is a hard bound limiting the time the algorithm runs. It can be softened by modifying the algorithm to terminate whenever no improvement to the currently best solution has been found for a certain number of iterations.

5.2 Classification for TSP instances

Instances of the Travelling Salesman Problem come in many different forms. There are different types such as *symmetric-euclidean, symmetric-non-euclidean, asymmetric, dynamic, special (BWTSP, RATSP and others)* as described in Section 2.2.5.

But even within the class of euclidean instances differences can be found. Most obviously, these instances vary in the numbers of cities involved. But even if two problem instances have the same number of cities, one can distinguish between them in many different ways.

Before we look at the characteristics used to make these distinctions, we have to make two restrictions:

1. the distance between two cities is taken to be the *geometric distance* between them. This ensures that the problem is an instance of the Euclidean TSP described earlier.

2. the graph representing the problem instance must be *fully connected*. That is, there is a direct road from any given city to any other city.

The usual statistics like *mean*, *median* or *modes* are of little use when classifying problem instances. Scaling an instance will change all these three metrics but should not affect any of the metrics in the proposed classification.

The first property indicating the complexity of a problem instance is the **number of cities** involved. The bigger that number, the less feasible exhaustive search of the space of all tours will be. For twenty cities, the number of possible tours exceeds 60,822 trillion.

Applying the ant algorithm to problems of different sizes makes a difference in terms of speed. Generally, the bigger the problem, the longer it takes to find good, very good or optimal solutions. However, a seven by seven grid with 49 cities may be solved faster than a 40-city problem where the cities are placed randomly.

Note, that throughout this section, *'speed'* refers to the number of iterations the algorithm takes to find a solution of a specified quality.

A second property that takes the uniformity of the problem instance into account is termed *normalised nearest-neighbour distances (nNNd)* by the author of this report. The **standard deviation of the nNNd's** is taken as an indication of how uniform the problem is.

In the above mentioned seven by seven grid, the distances between each of the 49 cities and their nearest neighbours are equal. In a randomly generated problem this will not be the case and the nNNd's will follow a normal distribution.

The standard deviation of the nNNd's is calculated as follows:

1. find the nearest neighbour for each city and calculate the distance between city and nearest neighbour

2. divide them by the mean distance of all edges in the graph to take into account scaled instances of the same problem; call these numbers (one for each city) the normalised nearest-neighbour distances (nNNd)

3. calculate standard deviation of the nNNd's

A final property that looks beyond the nearest neighbours is the **coefficient of variation** (also called variation coefficient in some books on statistics). It

takes all distances into account. This is important, since for a problem instance with n cities, there will be $(n*(n-1))/2$ inter-city distances but only n nNNd's.

The coefficient of variation of a set of numbers X is defined as their standard deviation divided by their mean. It gives an indication of the relative magnitude of variation in relation to the mean of set X.

Some authors equate the coefficient of variation with the absolute relative deviation w.r.t. the mean expressed in percentage ($VariationCoefficient = 100 * |Std.dev./mean|percent$).

It is independent of the actual distances involved as can easily be seen in the example below:

Figure 5.1: Two identical problem instances of the TSP modulo scaling

Now clearly, our classification should not distinguish between them. Problem instances that can be made identical by scaling one of them, should be treated as being identical. The next table shows that indeed all metrics have the same values for the two problem instances that differ only by a scaling factor:

Metric	left-hand problem	right-hand problem
Number of cities	3	3
nNNd's	$(\frac{3}{4},\frac{3}{4},1)$	$(\frac{6}{8},\frac{6}{8},1)$
std. dev. of nNNd's	0.144338	0.144338
mean	4	8
std. dev.	1	2
variation coefficient	0.25	0.25

Table 5.3: Metrics for two scaled problem instances

However, the following two problem instances should be distinguished by the classification. The one displayed on the right hand side has a longer distance between A and C while all other distances are equal.

Figure 5.2: Two different problem instances of the TSP

If this triangle would be a subpart of a bigger problem instance, then in the example on the left, the algorithm is more likely to choose 'road' AC then in

the example on the right. A length of 6 on this 'road' makes it advantageous to use AB and BC instead. While this is marginally longer than using AC, it visits an extra city.

In the example on the left however, the advantage is smaller, since AB and BC combined are significantly longer than AC.

Both problem instances have the same number of cities, but the other two metrics distinguish between them. The table below shows the differences:

Metric	left-hand problem	right-hand problem
Number of cities	3	3
nNNd's	$(\frac{3}{4},\frac{3}{4},1)$	$(\frac{9}{13},\frac{9}{13},\frac{12}{13})$
std. dev. of nNNd's	0.144338	0.133235
mean	4	4.333333
std. dev.	1	1.527525
variation coefficient	0.25	0.352506

Table 5.4: Metrics for two different problem instances

This classification seems to distinguish problem instances in the desired way. The table below displays the values of all three metrics for several well-known, random and uniform problems. For visual presentations of the problem instances in question, please consult the Appendix D.

Problem	num. of cities	std. dev. of nNNd's	variation coefficient
Oliver30	30	0.1684230131	0.4800819555
Eilon50	50	0.0448722591	0.4625115236
Eilon75	75	0.0540639830	0.4723591767
Random	40	0.0832759984	0.4709384165
Grid	16	0	0.4048511937
Grid	36	0	0.4424179773

Table 5.5: Metrics for a variety of standard problem instances

5.3 Optimal parameters for instance Oliver30

One of the results of the research undertaken in the course of this project was that finding better parameters will reduce the time taken by 90%.

The well-known problem instance Oliver30 was chosen as the standard instance, since many benchmarks exist for it. It was first defined in [SWF89] and

heavily analysed over the past 15 years. Exact definition and visualisation can be found in Appendix D.

Dorigo et al. in [DMC96] found the best known solution to this problem to be 423.741 for real-valued tour lengths (and 420 if distances are rounded to the nearest integer). They instantiated the algorithm with $\alpha = 1$, $\beta = 5$, $\rho = 0.5$ and $q_0 = 0$ and obtained this optimal solution after 342 iterations of the algorithm.

Note, that they used the *ant system algorithm* which did not feature the parameter q_0. This parameter was later introduced by Dorigo and Gambardella in [DG97b]. Setting q_0 to zero in the improved algorithm would simulate the original *ant system*.

The shortest tour visiting each city exactly once was measured to be 423.7405630. This agrees with Dorigo's findings but is slightly more precise. However, using optimal parameters, it was found after 32 iterations on average, an improvement of 90.6%.

Section 5.1 concluded with the realisation, that four out of eight parameters can be fixed for a given problem instance. These are given in the following table for the problem instance *Oliver30*:

Parameter	Constant Value
α	1
γ	0
τ_0	0.0001
$numAnts$	30

Table 5.6: Constant parameters for the Oliver30 instance

Furthermore, we noted that the remaining four parameters had ranges of feasible values associated with them. The following experimental set-up was chosen to find optimal parameters.

Parameter	Value Range	Step Size
β	(2,13)	0.5
ρ	(0,1)	0.1
q_0	[0,1)	0.1

Table 5.7: Variable parameters and their ranges for Oliver30

Note, that all ranges are specified exclusively, with the exception of q_0 where the range includes the value zero.

The fourth parameter, the number of iterations, was said to range from ten to a thousand. This parameter was used as a measure of the quality of each

parameter combination. The fewer iterations it took to find the optimal path for a given combination of parameters, the better this combination was ranked.

There are 21 distinct values for *beta*, nine for ρ and ten for q_0 giving a total of 1890 different combinations of parameters. Each of these combinations was employed in ten runs of the algorithm resulting in 18900 runs.

Each run was terminated if the best known solution was not found within 1000 iterations. Total running time on cpu3, a shared server with two dual 2.2GHz processors and 2GB RAM was more than eight weeks.

The average time per run was below ten minutes, which is pretty good seeing that a lot of runs used 1000 iterations, due to infeasible parameter combinations.

The next table will show the ten best combinations of parameters ρ, β and q_0 together with the average number of iterations it took to find the *optimal* solution.

ρ	β	q_0	number of iterations
0.9	6.0	0.1	32.2
0.6	6.0	0.2	34.2
0.6	7.5	0.7	37.4
0.6	4.5	0.1	39.899
0.4	7.0	0.4	41.899
0.3	5.5	0.4	42.9
0.8	6.0	0.2	43.3
0.6	7.0	0.6	44.0
0.6	7.0	0.5	44.199
0.9	5.5	0.0	44.7
0.63	6.2	0.32	N/A
0.1946	0.9189	0.2347	N/A

Table 5.8: Parameters for optimal solutions for Oliver30

The last line two lines are mean and standard deviation of the parameter values for the ten best combinations. Both of these are meaningless for the last column.

Note, that only 60% of the parameter combinations found the optimal tour on all ten runs. This made 40% of the 18900 runs ineligible for the mean calculation. The mean number of iterations over all eligible runs is 118.82.

The parameter combination suggested and used by Dorigo ranked 298th out of all possible combinations and averaged 89.6 iterations over ten runs. It ranged from 50 iterations to 165 iterations and it is unclear why Dorigo's implementation took 342 iterations.

This analysis was performed for parameter combinations that allow the algorithm to find the *best known solution*. However, in many situations sub-optimal solutions of a high quality are sufficient.

We will now look at the ten best parameter combinations if solutions are required to be within 1% and 5% respectively, of the best known solution.

Clearly, the algorithm will take fewer iterations to achieve this. The next table list the combinations that found solutions within 1% of the best one in the fewest number of iterations.

ρ	β	q_0	number of iterations
0.4	11.0	0.4	4.799
0.3	6.0	0.7	5.599
0.8	10.0	0.6	6.299
0.6	10.0	0.3	6.5
0.8	11.0	0.0	6.7
0.4	9.0	0.4	6.9
0.5	12.0	0.3	6.9
0.8	9.5	0.1	7
0.5	11.0	0.1	7
0.6	10.0	0.0	7.099
0.57	9.95	0.29	N/A
0.1829	1.6406	0.2424	N/A

Table 5.9: Parameters for very good solutions for Oliver30

Finally, the Table 5.10 shows parameter combinations that enable the algorithm to find good solutions (within 5% of best solution) fastest.

ρ	β	q_0	number of iterations
0.8	12.0	0.2	1.6
0.9	8.5	0.2	1.9
0.8	9.0	0.0	1.9
0.5	12.0	0.4	2
0.1	10.0	0.3	2
0.6	10.5	0.4	2.1
0.5	9.5	0.1	2.1
0.1	9.0	0.2	2.1
0.5	12.0	0.3	2.2
0.9	12.0	0.0	2.2
0.57	10.45	0.21	N/A
0.2946	1.4424	0.1449	N/A

Table 5.10: Parameters for good solutions for Oliver30

It is worth noting, that none of the parameter combinations featured in more than one table with the exception of (0.5,12,0.3) which was ranked 7th in the second table and 9th in the third table. This exception is due to the randomness of the algorithm and can be ignored.

It is also interesting to point out that none of the 18900 runs failed to find a solution that is within 5% of the best known one. The worst possible parameter combination for this task was (0.1,3.0,0.0) which averaged 218.5 iterations.

The same parameter combination was ranked last when it came to the task of finding a solution within 1% when it failed to do so in seven of its ten runs (in no more than 1000 iterations).

Ten percent of parameter combinations failed this task in at least one of their ten runs. In comparison 40% of the parameter combinations failed to find the optimal solution in at least one run (as mentioned above).

Looking at the best value for ρ in Table 5.8 for optimal parameters, one can find the value 0.6 six times amongst the best nine combinations. Also, the average for this value is 0.63 with a low deviation. Even in the other two tables an average of 0.57 indicates that ρ is pretty robust at 0.6.

This is not much different from Dorigo's intuition, who proposed $\rho = 0.5$ as a good parameter value in [DMC96]. Parameter β however, varies a lot depending on how accurate results one requires. The more accurate a result is wanted, the smaller value for beta should be chosen. Optimal solutions are found fastest with β set to 6 or 7 (seven of the nine best combinations have one of these values for β). If one only requires solutions within 1% or 5% off the best known solution, than a value between 10 and 12 will be more appropriate for β.

All these results for β are higher than the value 2, 'guessed' by Dorigo in [DG97a]. In the same paper, he uses a value of 0.9 for the parameter q_0, limiting the exploration dramatically. Remember, q_0 ranges between zero and one and a high value favours exploitation over exploration.

Such a high value for q_0 was not found to be optimal in any of the experiments undertaken. The optimal value varies between 0.2 and 0.3 with quite a high deviation for optimal solutions. So summarising the results, the author picks $(\beta = 6, \rho = 0.6, q_0 = 0.2)$ as optimal parameters for this problem instance. Other combinations are also possible and may even be preferable if one is willing to compromise on accuracy.

5.4 New optimal solution for instance Eilon50

The previous section described an experiment that took eight weeks of execution time in total. A similar experiment was to run for different problem instances to see whether the optimal parameters found for the *Oliver30* instance are optimal for other instances, too.

Another famous instance is called *Eilon50* which contains 50 cities. The fact that the optimal tour will involve 66% more cities than the instcance used in the previous section hinted that it would run for about 12 weeks. This was unfeasible given the time constraints of the project and the number of runs needed to be reduced.

Using the experience gained from the Oliver30 instance, the ranges of all parameters where reduced to exclude the extreme values that never featured in any good parameter combination before. Also, the number of trial runs per parameter combination was reduced from ten to five and the number of iterations per run was reduced from 1000 to 500. Knowing that this might stop the algorithm from finding the optimal solution in some case, this sacrifice was excepted to obtain results before the project ran out of time.

The best known solution from Dorigo for the Eilon50 problem instance was 427.965. Using this intelligent, semi-exhaustive search through the parameter space, the author of this report improved this solution to 427.855. This new best known solution was found twice for the parameter combinations ($\beta = 5, \rho = 0.6, q_0 = 0.1$) and ($\beta = 5, \rho = 0.6, q_0 = 0.2$).

The problem instance and the new best solution are depicted in Figure 5.3. The former best known tour is indicated by the numbering on the cities which are represented by the red dots. Unsurprisingly, the new best tour does not differ much from the old best tour. The change is non-trivial which is expected since otherwise local optimisation such as *2-OPT* would have found it in Dorigo's experiments.

The change is limited to cities from 37 to 45 inclusive. Instead of 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46 one ant decided to explore the following sub-path 36, 38, 39, 37, 42, 43, 41, 40, 45, 44, 46 and found this to be shorter.

5.5 Generalised results

The optimal values for the parameters β and q_0 are *problem dependent* since they vary substantially between problem instances (the well-known Oliver30 instance and a random one have been used here).

They are also *goal dependent* in that different parameter values are optimal for different accuracies. Remember, that Tables 5.8 and 5.9 showed that even for the same problem instance, optimal parameters vary depending on the required accuracy.

The parameter ρ on the other hand seemed to be very stable at a value of 0.6.

This was found by running a similar experiment for a random problem with 30 cities, the best parameter combination was ($\beta = 11, \rho = 0.6, q_0 = 0.4$). This combination was best both for the task of finding the optimal solution and for the task of finding a solution within 1% off the best known one. For the task of finding a solution within 5% off the best known one, it was ranked 11^{th}. It is also close to the average of the ten best combinations, which is ($\beta = 11, \rho = 0.58, q_0 = 0.44$)

The worst possible combination of parameters was ($\beta = 5, \rho = 0.9, q_0 = 0.6$) which was the only combination which failed to find the best solution to

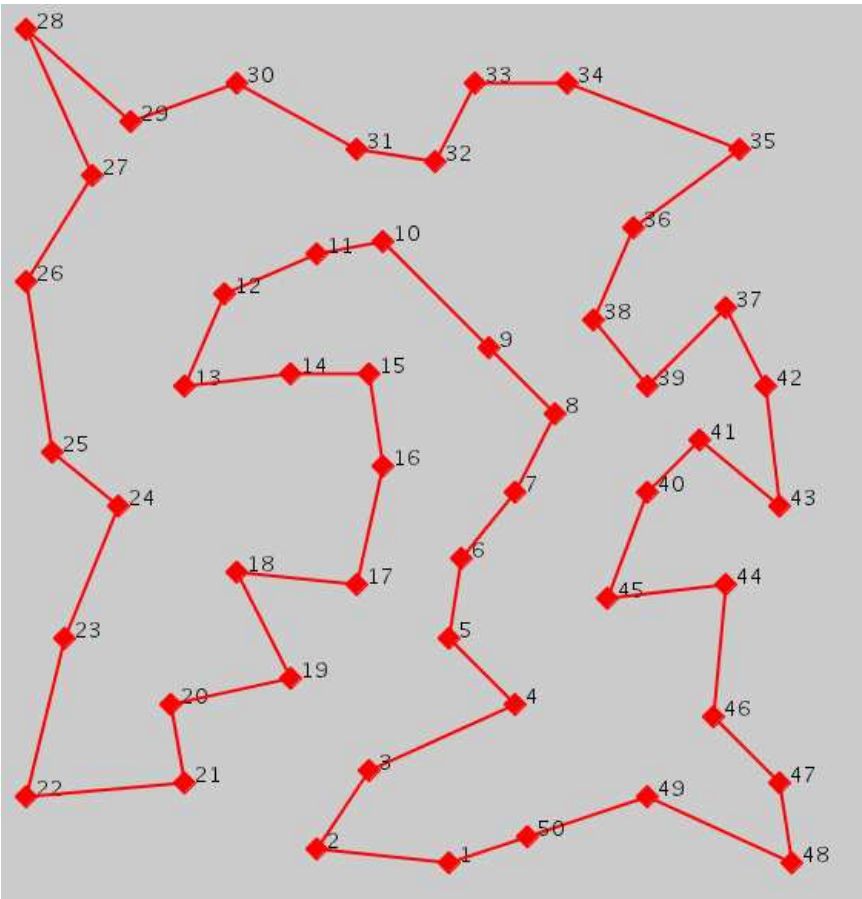


Figure 5.3: Improved solution to the Eilon50 problem instance of the TSP

the random problem. This best solution has been established previously by running the standard algorithm for 5000 iterations. None of the test runs in this experiment improved this best solution.

Comparing the best and worst parameter combinations for this random problem with their equivalent combinations for Oliver30, an interesting discovery is made. Remember, that 1890 combinations were tested for Oliver30. The best combination to the random problem is ranked 837^{th} for the Oliver30 problem and the worst solution to the random problem is ranked 606^{th}.

Both between problem instances and within the same problem for different accuracies, the optimal parameter value for ρ never deviated much from 0.6. The averages of the best ten values for ρ for optimal, very good and good solutions for Oliver30 and the random 30 city problem are 0.63, 0.57, 0.57, 0.58, 0.54 and 0.65 respectively.

5.6 Summary

This chapter described all the parameters needed to instantiate Dorigo's ant system. A classification of TSP instances was proposed using three metrics: the number of cities, the variation coefficient and a novel metric proposed by the other.

This novel metric - the standard deviation of the normalised nearest-neighbour distances - gives an indication of how uniform or regular the cities are spread out. The smaller this metric, the more ordered the city landscape (a grid of cities has a value of zero for this metric).

Then the optimal parameters for the Oliver30 instance are found by exhaustive search. These optimal parameters were tested for optimality on other problem instances.

While doing so, a new shortest tour for the Eilon50 instance was found.

The problem and goal dependency of some of the parameters of the algorithm justify the need for an automatic way of setting these parameters. Dorigo's claim, that *"intuition, experience and luck are required to find the right parameters for a problem"* supports our results.

Hence, the author of this report tried to combine ideas from genetic algorithms to *evolve* the parameters of the ant algorithm. Since both of the parameters that exhibited the dependencies are related to the ants themselves (remember, ρ is the parameter controlling the evaporation of pheromones in the environment), the novel algorithm that is proposed here is termed *Genetically Modified Ant System*.

Chapter 6

On genetically modified ants

6.1 Motivation

The previous chapter investigated the optimal parameters for the ant system algorithm, finding that some of them are dependent on the problem instance and the required solution quality.

Since no strong correlation between the classification attributes from Section 5.2 and the optimal parameter values were found, a novel approach was needed. The concept of genetic algorithms introduced in Section 2.4 is advantageous for the evolution of solutions.

In this new algorithm, this concept is not utilised to evolve solutions, but to *evolve the ants that find the solutions*. Each ant is characterised by a number of parameters, two of which vary highly for different problem instances. These parameters, β and q_0, will be evolved over a number of generations to create *fitter ants*.

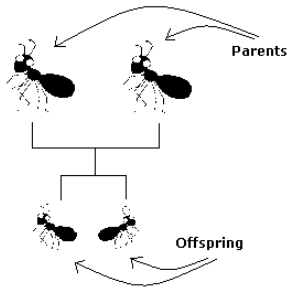


Figure 6.1: Family tree of ants

The metaphor from the animal kingdom could be modified thus:

> A population of ants with different characteristics forage for food. They drop and smell pheromone wherever they go. Some ants are *explorers*, ignoring most of the pheromone signals while other ants are *exploiters* who follow the pheromone trails to the known food sources. Other ants do both, exploring and exploiting.
>
> Every so often, the ants also mate and generate offspring which inherits the characteristics from the parents. Some young ants are subject to mutation by environmental influences and after a certain number of days, old ants finally die.
>
> The size of the ant population stays pretty constant with offspring being created and older, less fit ants dying. However, the average fitness of the whole population increases from generation to generation, since only the successful, well nurtured ants will proliferate and their characteristics are spread in the gene pool of the next generation.

This is taken as a paradigm for a system which combines the ant system algorithm and genetic algorithms. It is termed *Genetically Modified Ant System* or GMAS for short.

6.2 Implementation

This section will present the issues that arose and the decisions that were made during the implementation process. The main elements of a genetic algorithms are the encoding of a solution, a selection method for the mating process, a form of mating or recombination and the mutation that individuals are subjected to.

There are several different options for each of these elements and the design decisions are crucial for the performance of the system.

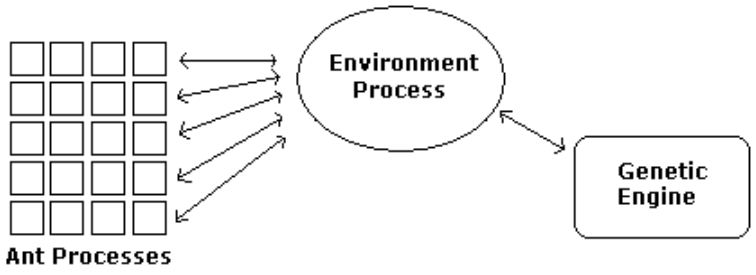


Figure 6.2: Simplified design of the Genetically Modified Ant System

The general design of the system is simplified in Figure 6.2. Many ant processes communicate with the environment process as before but the environment process now uses a *genetic engine* to breed ants and replace the old generation of individuals with a new one.

Conceptually, the genetic engine operates on the population of ants. However, for efficiency reasons, the environment process mediates between the ant threads and the genetic engine.

6.2.1 Encoding

Encoding is the design task in which one decides how to represent a solution to the problem. This is the most difficult part because the way one encodes a solution implicitly defines the solution space. Therefore it also defines how a solution looks like. Note, that the term *solutions* is used here for the individuals in the population of a genetic algorithm.

Since the GMAS evolves ants and more specifically, the parameters of the ants, an individual could be represented by the pair (β, q_0).

However, the favourite encoding used in most genetic algorithms is the representation of solutions as bitstrings similar to the way DNA is encoded. This is of little use for the evolution of our parameter pair. One could concatenate the bit representations of both parameters to obtain a bitstring, but crossover would not work very well.

Imagine, each parameter is represented by four bits allowing for 16 different parameter values. Concatenating both parameters one gets eight bit parents, say 00011000 and 00000111. This means, that both parents have average values for q_0 (namely, 1000 and 0111 representing very close real parameter values). Crossover after bit location five would yield offspring 000000000 and 00011111.

Now the two offspring have values of 0000 and 1111 for their respective q_0 parameter, which does not resemble either of the parents. Also, both siblings are as different as only possible, which does not agree with the metaphor we used. We therefore propose to use a real-valued encoding, representing the parameter pair. There is little information on real-valued encodings as mentioned in Section 2.4.2 but it seems the most natural encoding for the problem at hand.

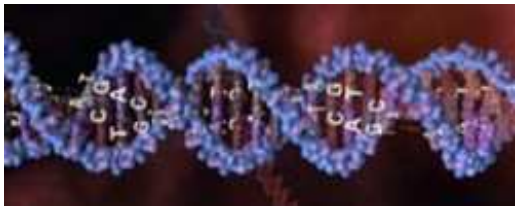


Figure 6.3: DNA double helix encoding instructions for components and processes of all living things

6.2.2 Selection

The general principle underlying fitness-proportionate selection is that fit individuals should be selected for reproduction more often than individuals who are less fit. This is widely used and chosen for the GMAS, too.

In order to decide just how fit an individual is, one needs to calculate its fitness $f(i)$ according to the following formula:

$$f(i) = \frac{g(i)}{\overline{g}}$$

where

$$\overline{g} = \sum_{k=1}^{n} \frac{g(k)}{n}$$

So, $g(i)$ is the *worth* of an individual and $\overline{g}$ is the average worth in the whole population. Clearly, $g(i)$ should be higher than $g(j)$ if i represents a better solution than j. Defining function g is the trickiest aspect of selection and will be discussed in detail later on. For now, we take the worth of a solution (TSP tour) $g(i)$ to be

best greedy length - tour described by i

This ensures, that shorter tours are rewarded with a higher function value for g and therefore with a higher fitness than longer tours. It is also sufficiently fast calculated which is important since this calculation takes place fairly often. One characteristic that this definition of fitness does not possess is a broad spread of individual fitness values.

All ants will find tours that are in the same order of magnitude. Initial experiments even showed that the worst tour found is not more than twice as long as the best tour. So individual values for the worth g are all pretty close and the fitness values f range only between 0.5 and 1.5. It is desirable for this range to be as big as possible and the influence of scaling this range will be investigated in Section 6.3.

Having calculated the fitness value for every individual in the population, we can build an intermediate population (IP) from which parents are chosen for reproduction. This intermediate population should contain proportionally more copies of the fitter individuals to increase their chances of mating.

If an individual is exactly as fit as the average of the population, then a copy of this individual is placed in the IP. If the individual is twice as fit as the average, then two copies of it will go into the IP. And if $f(i)$ is has a fractional part, then this part will give the probability of another copy of i being put into

the IP (so for $f(i) = 2.7$ we have two copies of i in the IP for sure and with a probability of 70% we also have a third copy).

This intermediate population represents the pool of individuals from which parents are chosen for the recombination process. Note, that some unfit individuals may not be in this IP and therefore not eligible to proliferate. On the other hand, very fit individuals will feature several times in the IP increasing their chances of reproduction. After two parents are chosen and recombined in a way described shortly, they are replace into the IP as is standard in all GAs.

```
/* sort population by tour length & calc. mean value for g */
intermPopu : [];
popuList = sort(popuList);
numOfAnts = listlen(popuList);
(maxLengthIn,_,_) = popuList#numOfAnts;
maxLength = maxLengthIn + 1;
mean = maxLength - averageLengthIn;

for x in popuList do
    (Length,Beta,QZero) = x;
    gValue = maxLength-Length;
    fitness = gValue / mean;
    (fitnessInt,fitnessFrac) = modf(fitness);

    /* add the copies that go into IP for sure */
    for y in 1 .. fitnessInt do
        intermPopu := [(Length,Beta,QZero,fitness),..intermPopu];
    ;

    /* probabilistically add an extra copy */
    if fitnessFrac > rand(1) then
        intermPopu := [(Length,Beta,QZero,fitness),..intermPopu];
    ;
;
```

The code snippet above shows how the intermediate population *intermPopu* is created from a list of individuals that make up a generation. The variable *maxLengthIn* holds the longest tour found by any member of this generation. Empirically, this is always shorter than the length of the best greedy tour and therefore sufficient to ensure that g is always positive.

Figure 6.4: Selection and recombination for the GMAS

6.2.3 Recombination

After the intermediate population has been put together, recombination takes place. For this we use a tournament method where two individuals are picked and the fitter one is chosen as a parent, then two more individuals are picked and the fitter one of those is chosen as the second parent.

The two parents then mate using uniform crossover. This is the only sensible way to recombine non-bitstring encoded individuals which have very short chromosomes. Each gene of the single offspring is selected randomly from the corresponding genes of the parents. Since there are only two genes in the chosen encoding, any two parents can only produce one of four possible offspring.

Experiments have also taken place with a variant of uniform crossover. The average value from both parents for a gene is taken for the offspring's gene. This proved unsuccessful, since characteristics that control the fitness of an individual were smoothed. Hence values around the midpoint of the allowed ranges dominated the population after few iterations.

Figure 6.5: *Acromyrmex versicolor* looking for mating partners

6.2.4 Mutation

Each gene of the offspring is then subjected to mutation with a small probability p_m. This was set to 0.1 which is much higher than the standard 0.001 but justified by the short chromosomes used for the encoding of the parameter pair (β, q_0).

If a gene was chosen to be mutated, then the parameter value that is represented by the gene was increased or decreased by about 5%.

6.2.5 Generation gap

Another decision that crucially affects the quality of the GMAS is the transition from one generation to the next. Initially, the generational approach was taken, replacing the whole generation by an equal number of generated offspring. This is an area for ample experimentation.

Replacing the whole generation at once can be negative, if the fittest offspring are less fit than the best individuals from the parent generation. Therefore, a steady-state approach was investigated, too. This involved replacing only some percentage r of the least fit individuals by offspring and various tests have been conducted for r set to 10%, 20% and 30%.

It promises to keep *selective pressure* on the population to continuously improve. However, the problem of early stagnation described in the next section still persisted. Attempts to counter that stagnation were made, but failed to show an improvement initially. One such attempt was to generate the next generation in three stages. One third was kept from the parent generation (the best third of parental individuals), one third was generated offspring recombined as explained before and the last third was a set of randomly generated individuals.

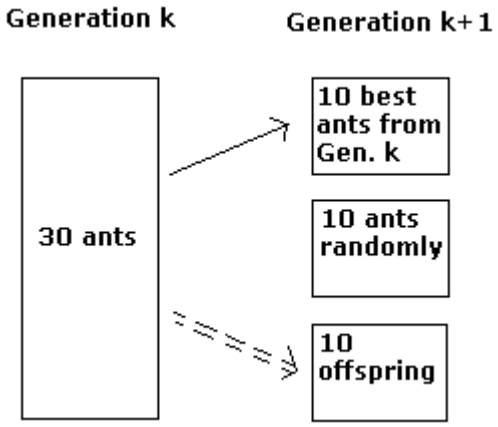


Figure 6.6: The three stages in a variant of the steady-state approach

It was hoped, that this would lead to more variety and counter the stagnation behaviour, but failed to prove its effectiveness.

On the other hand, scaling the fitness values reduced the problem of premature convergence. Different scaling methods will be presented shortly after the stagnation problem is described in more detail.

6.3 Stagnation problems

The following stagnation problem was observed when running the GMAS on the well-known problem instance Oliver30. The individuals with the lowest values for β and the highest values for q_0 dominated the population very quickly.

This is not unsurprising, since these individuals avoid exploration and focus only on exploitation. A low β value ranks distance higher than pheromone information. Also, a high value for q_0 means that the locally best option is chosen rather than any option probabilistically.

This leads to relatively short tours, since no exploration takes place. Exploration generally results in longer tours and therefore individuals who favour exploration will be seen as less fit and their genes will vanish from the gene pool. In rare cases exploration successfully finds a shorter tour, but generally 'exploration favouring' gene values will be eradicated before they get a chance to prove their worth.

'Exploitation favouring' genes on the other hand come to dominate the population due to this premature convergence. One would really like to *defer* the selection for a few iterations in order to give the 'exploring' ants a chance to find an improved tour.

β	q_0	shortest tour	iterations
5.0	0.95	431.99	17
3.4	0.95	433.73	25
9.0	0.95	448.95	8
6.0	0.95	438.29	31
3.0	0.90	430.85	7
Averages:		**436.76**	**17.6**

Table 6.1: Five runs of the GMAS with breeding after each iteration

The table above shows the results of five runs of the GMAS for 400 iterations each. Interestingly, in its original form it performed worse than the standard ant system (remember, in Chapter 4 the average tour length was 434.44 after 100 iterations for the standard ant system).

The first two columns display the dominant parameter values which was almost constant at the indicated value after a few iterations. The parameter q_0 was very high for all runs as expected and β varied a lot. Remember, the valid ranges for these two parameters are $0 \leq q_0 \leq 1$ and $2 \leq \beta \leq 13$.

Stagnation set in after 17.6 iterations on average leading to the low quality solutions and the variation in values for β. It shows that some individuals that are fit early on in the execution dominate the population and reduce selection pressure.

A first attempt to counteract this premature convergence was done by running the normal ant system five times before using the genetic engine. The fitness of each ant was dependent on the best of these five runs, eliminating the randomness somewhat. An ant that randomly chose a bad path in one iteration of the algorithm can subsequently perform better and hence increase its chances of reproduction.

β	q_0	shortest tour	iterations
3.0	0.90	429.86	32*5
5.0	0.80	430.54	24*5
6.2	0.65	428.10	39*5
3.0	0.95	433.68	54*5
3.8	0.75	432.71	65*5
Averages:		**430.97**	**42.8*5**

Table 6.2: Five runs of the GMAS with breeding after every five iterations

The experiment in Table 6.2 was run for 80 iterations, since each iteration involved five tours generated per ant and a total number of 400 iterations was needed to be able to compare the results with those in Table 6.1.

Firstly, the table reveals, that the solution quality has improved by roughly 1.5% on average. Also, the best tour was found after more than 200 iterations on average, indicating that there is less stagnation (remember, the previous experiment stagnated after 17.6 iterations on average). This was confirmed by inspection of the actual individuals of the population.

While the previous experiment saw all members of the population to be almost equal after 20 iterations, this did not happen for the second experiment. Even after 200 iterations there was some variation in the population. The values displayed for β and q_0 in Table 6.2 represent rough averages of all individuals.

The literature describes other alternatives to the stagnation problem. All these alternatives increase the selection pressure on the population. This pressure ensures that the individuals keep evolving. The most famous ones are linear scaling, sigma scaling and power law scaling where each of them scale the fitness values of all individuals in the population in order to spread them out more.

As mentioned before, all ants will find tours that are in the same order of magnitude and the fitness values f range only between 0.5 and 1.5. This means, that all ants are almost equally likely to proliferate (or at least to go into the intermediate population). Scaling their fitness values will mean that the least fit individuals have an infinitesimal chance of reproducing whereas the fittest ones are almost certain to mate.

Three scaling methods are described presently.

6.3.1 Scaling methods

Scaling ensures, depending on the scaling function, that certain individuals or groups of individuals increase or decrease their reproduction probability.

Having calculated the fitness value $f(i)$ for each individual i and the average fitness of the population $\overline{f}$ one can use one of the three methods below to obtain f' as follows:

Linear scaling

$f' = a \cdot f + b$

where a and b are chosen to make $\overline{f'} = \overline{f}$. The second constraint on a and b defines the actual range. The highest value for f' is set to be a multiple of $\overline{f}$.

Sigma scaling

$f' = f - (\overline{f} - k \cdot \sigma)$

where σ is the standard deviation of the population and $1 \leq k \leq 3$.

Power law scaling

$f' = f^p$

where p is application dependent and may need to be changed dynamically during the execution.

Out of these three, power law scaling is the least popular. Linear scaling has a flaw in that some scaled fitness values can be negative for very bad individuals. While one can theoretically just assign zero to all negative scaled fitness values, a better solution is to use sigma scaling.

Sigma scaling has been implemented and tested with parameter k set to 1, 2 and 3 in different test runs. Note, that sigma scaling can in some cases lead to negative fitness values, too. This is the case especially for small values of k.

The graph in Figure 6.7 shows how sigma scaling effects the fitness of the population. A typical generation from a run of the algorithm on the Eilon50 problem instance has been taken and sorted according to fitness values. It is obvious, that scaling does not affect the relative order of the individuals in a population, but their reproduction likelihood.

For example, using the original fitness values, all but the worst individual can theoretically produce offspring. Individuals 41 to 49 as depicted in Figure 6.7 have small but non-zero probabilities of selection (remember, reproduction probability is calculated as *fitness of individual* divided by *average fitness of population*).

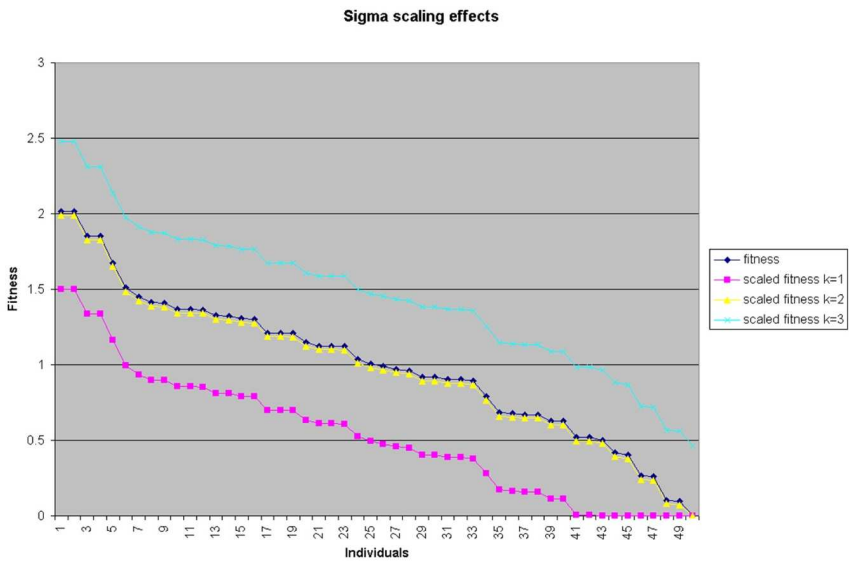


Figure 6.7: Effects of sigma scaling on the fitness values of an ant population in the GMAS

Using sigma scaling with parameter k set to 1, the same individuals have no chance of creating offspring. Sigma scaling with k set to 3 even assigns a small probability to the worst individual. This leads to the conclusion, that the smaller a value for k is chosen, the more selection pressure is put on the population.

An improvement to the algorithm could be achieved by dynamically changing the selection pressure. Initially, there should not be too much selection pressure because this would lead to premature convergence. However, after half of the scheduled iterations have been executed, the value of k could be lowered to increase the pressure on the individuals to perform.

In Table 6.3 the results of experiments with varying parameter k are displayed. Its influence on the performance of the GMAS is significant. In the literature on genetic algorithms and sigma scaling in particular, the range for k is given by one to three.

Smaller values of k would lead to the exclusion of too many individuals from the recombination process. Choosing too high values for k would eliminate the desired scaling effect.

All 15 runs have been executed for 200 iterations and all but one converged to the highest possible value for parameter q_0. Values for β on the other hand remained varied throughout all runs. This may be due to the stronger influence of q_0 on the exploration/exploitation ratio. It seems as if the scaling attempts successfully avoided premature convergence for parameter β.

k	β	q_0	shortest tour	iterations
1	5.6	0.9	434.82	140
1	3.0	0.95	432.95	163
1	2.8	0.95	433.70	131
1	3.4	0.95	437.04	123
1	7.6	0.95	436.66	185
Averages:			**435.03**	**148.4**
2	7.2	0.95	430.96	177
2	3.0	0.95	433.68	174
2	5.0	0.95	432.28	95
2	5.2	0.95	435.29	71
2	3.6	0.95	435.45	176
Averages:			**433.53**	**138.6**
3	5.6	0.95	430.83	127
3	7.2	0.9	432.92	14
3	4.0	0.95	429.59	87
3	9.4	0.4	429.59	78
3	2.6	0.95	433.32	191
Averages:			**431.25**	**99.4**

Table 6.3: Results of the GMAS with sigma scaling for different k

However, q_0 resisted the effects of scaling since a high value forces all ants to exploit rather than explore and this is advantageous for the ants when it comes to selection for the reproduction process.

The previous experiment allowed each ant to find five tours and only the best one was used in the fitness calculation. This approach led to some variability even of q_0 and hence it is proposed to combine this approach with the sigma scaling presented here.

To establish the justification of this claim an experiment was designed executing five runs of 40 iterations each. In each iteration, each ant found 5 tours, resulting in a total number of 200 iterations to allow for comparison with the experiment from Table 6.3. The parameter k that controls the sigma scaling of the fitness values was set to 3 since this seemed most promising given the previous experiment.

It outperformed the previous experiments were either sigma scaling or less frequent breeding was used. Combining these two procedures, the solution quality was 430.8 on average and was found after about 100 iterations (with 20 breeding operations).

This indicates that stagnation did not set in as early as before (remember, in the experiment without modification, stagnation set in after 17.6 iterations). Table 6.4 shows the results of five runs of this new algorithm.

run number	shortest tour	iterations
1	431.48	24*5
2	429.33	13*5
3	430.59	24*5
4	433.46	21*5
5	429.33	22*5
Average:	**430.838**	**20.8*5**

Table 6.4: Results of the GMAS with sigma scaling and less frequent breeding

6.4 Comparison with the standard ant system

Comparing this novel *Genetically Modified Ant System* with the standard ant system introduced by Dorigo in [DMC96], one can conclude that both systems have a raison d'être.

Both systems achieve roughly the same accuracy or quality of solution. However, this assumes that feasible and appropriate parameters are supplied to the standard ant system. Given sub-optimal parameters, the ant system will not find the best solution as the author demonstrated in Section 5.4.

The GMAS is preferable in so far as it does not require the manual setting of crucial parameters. Starting from random parameters, a genetic engine evolves them to values that enable the system to find good solutions to the TSP instance provided.

If on the other hand the optimal parameters are known then the standard ant system should be used since in this case, it performs faster than the GMAS. However, optimal parameters are only known for the few problem instances investigated

Below one can find a comparison between the standard ant system and the improved algorithm proposed by the author. The average tour length for the Eilon50 problem instance after about[1] 100 iterations is displayed.

Algorithm	Average tour length after 100 iterations
Dorigo's ant system	434.44
GMAS	430.84

The runs for Dorigo's standard ant system used the parameters that he suggested. Remember, in Section 5.3 we showed that these parameters are sub-optimal.

[1] the result from Table 6.4 is used here

6.5 Summary

This chapter introduced the novel *Genetically Modified Ant System*. Firstly, the motivation and changed metaphor from the animal kingdom are described.

This is followed by a walkthrough of the proposed implementation explaining which encoding, selection method and recombination procedure is chosen. The way mutation and the transfer from one generation to the next are handled is the laid out.

Furthermore, the stagnation problem is described and different solutions to this problem have been detailed. One of them, the so-called sigma scaling is investigated more closely and found to counteract stagnation sufficiently.

The new system was then compared to and evaluated with respect to the standard ant system. Its main advantage is that it eliminates the need of expert domain knowledge and guess work to set the parameters of the ant system algorithm. The system will automatically evolve parameters to help finding the shortest tour for a given TSP instance.

Chapter 7

Conclusions

7.1 Contributions

Ant algorithms have first been proposed a decade ago by Marco Dorigo. Ant like agents use indirect, positive feedback in a process called stigmergy to collectively find good solutions to a given problem.

Since his PhD thesis[Dor92] was published, many researchers applied these algorithms to a variety of combinatorial optimisation problems. However, the algorithm itself did not undergo any major changes even though Dorigo was aware of some of its shortcomings.

This project has demonstrated that the research field of *ant algorithms* is far from matured. While this year marks the fourth repetition of the biennial International Workshop on Ant Colony Optimisation and Swarm Intelligence, many questions remain unanswered.

Research undertaken as part of this project succeeded in answering some of them. The following points are the major achievements:

- Optimal parameters for the ant algorithm applied to a famous TSP instance have been found.
- These optimal parameters led to a new shortest tour previously unknown.
- A classification for instances of the TSP has been defined.
- Problem dependency of optimal parameters has been shown by comparing the optimal parameters for a number of problem instances. Furthermore, goal dependency of optimal parameters has been shown in that differences in required solution quality lead to differences in optimal parameters.
- A novel algorithm has been developed combining the ant algorithms with ideas from evolutionary computing. This algorithm, the *Genetically Modified Ant System* has been compared to the standard ant system both in terms of efficiency and quality.

- Several parallelisation strategies have been proposed and tested and led to an improvement in execution time of 85% over the standard version of the ant system without loss of solution quality.
- An even higher speed-up has been found when a slight loss of solution quality was accepted.
- Applications have been developed in April and DialoX to visualise and experiment with different versions of the ant algorithm.

7.2 Future work

This project answered some of the open questions about ant algorithms. However, there is considerable potential for further research, both in new areas and in the areas touched by the author.

More parallelisation This project was restricted to the use of six identical computers from the DoC labs. Future work could attempt to employ one of the most powerful computing resources in the UK, namely the *Imperial College Parallel Computing Centre*.

Their systems provide a peak performance of nearly 50 GFlops/s allowing for larger TSP instances to be tackled with the parallel versions of the ant system presented in this report.

More TSP variants The Black-and-White TSP and the TSP with replenishment arcs presented in Section 2.2.5 could be tackled with modified ant algorithms. The author did promising initial experiments for the TSP with replenishment arcs, but due to time constraints no results can be reported yet.

Other optimisation problems The ant system and its variants have been applied to a variety of discrete and continuous optimisation problems ranging from Quadratic Assignment Problems and Multiple Knapsack Problem to Set Covering and Partitioning.

However, some famous optimisation problems have not been tackled with ant algorithms yet. This is most often due to difficulties in finding a representation of the solution space that can be travelled by ants. Researchers continue to find solutions to these difficulties and apply the ant system to an ever increasing range of problems.

Further analysis of the GMAS The analysis of the *Genetically Modified Ant System* has been restricted to its application to the Travelling Salesman Problem. It is believed, that the GMAS will be equally successful on other optimisation problems. A first step would be to apply it to the Quadratic Assignment Problem since there exist quite a few benchmarks for it.

While adopting the GMAS to other combinatorial optimisation problems one could experiment with different aspects of the genetic engine used in the GMAS. Concepts like encoding, scaling and selection offer various options not explored during the course of this project.

Teaching tools While all applications that have been developed for this project focused on functionality rather than beautiful graphical user interfaces, it should be straightforward to implement a program that can be used to teach students the concepts and subtleties of ant algorithms. The applications presented should suffice as prototypes for this matter.

Appendix A

Game of Life

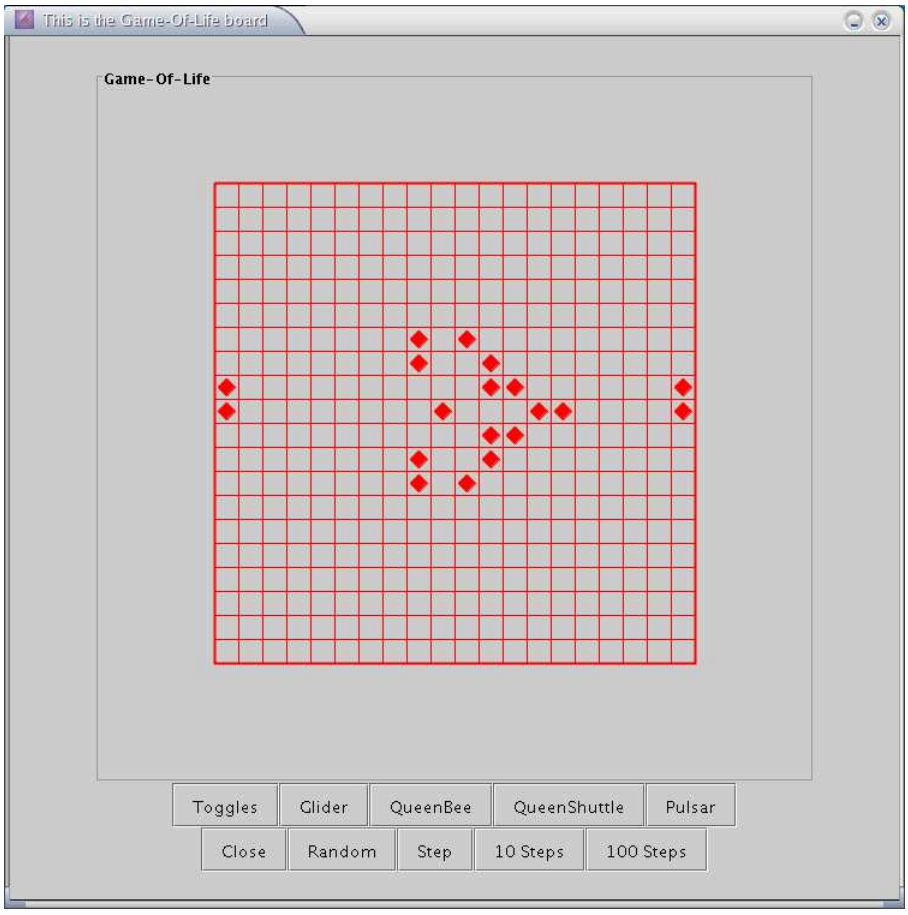


Figure A.1: GameOfLife application depicting the fifth iteration of the pattern known as Queen Bee Shuttle

As a preparation for this project, this implementation of the *Game of Life* was used to experiment with thread creation and management. 400 threads have been created to represent every cell on the game board.

It was found that this number of threads leads to a very slow execution of the program. It was therefore decided to use TSP instances of no more than 100 cities for the ant system experiments.

Appendix B

Swarm platforms and modelling environments

The recent research into swarm intelligent systems led to the development of tools that facilitate the modelling of such systems. Particularly the domains of social and economic studies lend themselves well to such a simulation. Here, model building tools and swarm-alike modelling environments are presented.

StarLogo, NetLogo and Agentsheets are relatively simple products designed to facilitate the understanding of decentralised systems especially for students. They allow the creation of models through graphical user interfaces and do not require extensive programming knowledge.

Swarm, Ascape and RePast on the other hand are advanced research applications. They are used extensively to explore social, economic and ecologic behaviour in complex adaptive systems.

Only the six most wide-spread applications are introduced in detail. However, there are some less well known platforms and models, that are briefly mentioned below:

Echo is a simulation tool derived from a general model of ecological systems. It was conceived and designed by John Holland (the pioneer in the field of genetic algorithms) in 1994 at the Santa Fe Institute[URLi]. The entities and interactions in this model are highly abstract and Echo makes specific commitments about agent types and interactions. It is highly correlated to nature since an agent in Echo replicates as soon as it as acquired enough resources to copy its 'genome'. Echo also specifies structural features of the environment in which the agents evolve, making it suitable to model complex adaptive systems.

IMT (short for Integrating Modelling Toolkit) is the engine of the Integrated Modelling Architecture (IMA) proposed in 1999 by Ferdinando Villa from the University of Maryland. He observed that modelling requires:

- paradigms for generating hypotheses
- specific semantics for describing scenarios subject to predictions of these hypotheses
- techniques to ground the semantics with validation on real data
- modular tools (ideally designed by domain experts)

Its development seems to be in a somewhat paused state since Dr. Villa left the University of Maryland. He is still conducting foundational research on IMA semantics and is building a web presence[URLg] for the IMA which is now a project of the Eco-informatics collaboratory at the University of Vermont.

MadKit [URLh] is multi-agent platform built in Java. It provides general agent facilities and allows many customisations. However it does not seem to be very suitable for simulations involving thousands of agents that execute in parallel.

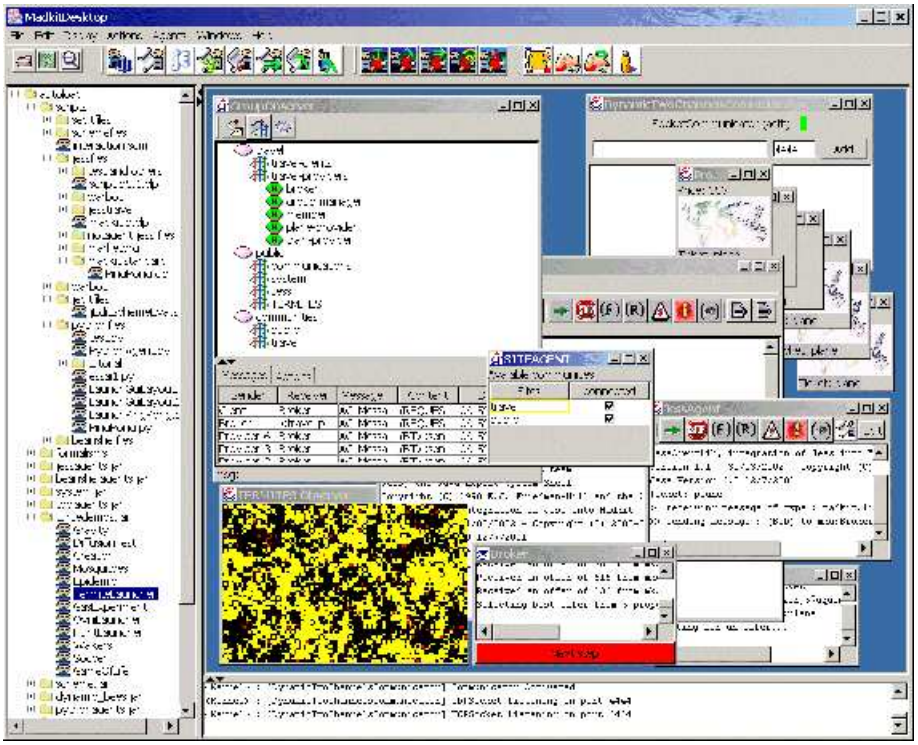


Figure B.1: A very crowded screenshot of the MadKit platform

B.1 StarLogo

StarLogo[URLa], developed at the Media Laboratory, MIT, Cambridge, Massachusetts was designed to help people to build their own models of complex, dynamic systems. It was conceived by Mitchel Resnick and is a specialised version of the Logo programming language.

Logo is both a philosophy of education and an evolving family of programming languages. Developed more than 28 years ago as a dialect of Lisp, it is a tool for learning. It is modular, extensible, interactive and very flexible. Requiring little programming knowledge, it is suitable even for young children, while also allowing sophisticated projects by advanced users.

The Logo environment is based around a 'turtle' that a user can control by issuing commands to it. This 'turtle', originally a robotic creature, soon appeared on graphical computer screens and can move around on a canvas, using a pen to draw shapes and pictures.

StarLogo, a programmable modelling environment for exploring the workings of decentralised systems, extends Logo by allowing thousands of these 'turtles' to run around and draw on the canvas in parallel. They can carry out independent actions and interact with each other. Their environment is made up of 'patches', which are programmable themselves and can be modified by the 'turtles'. The 'turtles' can sense their environment and base their actions upon what is sensed. The third ingredient in StarLogo is the 'observer', which provides a bird's eye view, allowing the user to monitor and analyse the system. It furthermore enables the creation of new 'turtles' and modification of the environment by the user.

'Turtles' can represent anything from ants in a colony to antibodies in an immune system. They are used to draw sophisticated mathematical images like Sirpinski fractals and Gaussian distributions, simulate physical phenomenons like diffusion limited aggregation and sinusoidal motion and analyse social systems like traffic jams or segregation in cities.

In Figure B.2 below, one can see how hundreds of termites gather randomly distributed wood chips and pile them up nicely in clusters.

Figure B.2: Termites gathering wood chips simulated with StarLogo

Mitchel Resnick, StarLogo's inventor, describes how all this is done in [Res94] and demonstrates the applicability of StarLogo to simulations of swarm intelligence.

A main advantage of using StarLogo especially for children is the fact that one does not require advanced mathematical or programming skills. The user employs templates or writes simple rules for individual behaviour and then watches many 'turtles' simultaneously follow these rules and the patterns that arise in the system. It helps students and researchers to understand that birds flock without a leader and ant colonies do not have a coordinator.

It is markedly well suited for ALife projects and many examples from biology have been implemented using StarLogo. These examples range from bees building honey combs and fireflies synchronising their flashing to termites piling up wood chips and predator-prey behaviour.

B.2 NetLogo

The Centre for Connected Learning and Computer-Based Modelling at Northwestern University took StarLogo a step further and developed NetLogo[URLe]. It is a cross-platform agent based parallel modelling and simulation environment and fully programmable.

Similar to StarLogo, it is a Logo dialect extended to support unlimited number of agents and parallelism and is extremely well suited for modelling complex systems. The user gives instructions to thousands of 'agents' (similar to the 'turtles' in StarLogo) that operate in parallel. NetLogo enables exploration of micro-level behaviour of individuals and emerging macro-level behaviour that results from the interactions of the individuals. It is simple enough for students to run and build simulations but also advanced enough for use as a research tool.

NetLogo is written in Java to be platform-independent and runs as a standalone application. However, models generate by this application can be saved as applets to embed them in a web-page. This allows the sharing of individual models within a wide community of researchers and is a step forward from the portability problems that plagued StarLogo.

An interesting novelty in NetLogo is a feature called HubNet, which allows users to run 'participatory simulations'. Each student controls an agent in a simulation through networked computers or handheld devices. A classroom of students could simulate the traffic in a virtual city where each student controls a traffic light with the objective to make the traffic flow smoothly.

NetLogo provides extensive documentation, tutorials and a model library as well as workshops organised by Northwestern University.

B.3 AgentSheets

While most other platforms are available as open source software, AgentSheets is a commercial product[URLf]. You can buy a single-user license for this agent based Web authoring tool on-line for $120.

AgentSheets allows users, from children to professionals, to build applications through a graphical user interface. You can create agents with behaviours and missions and see how they interact. With the help of the unique Visual AgenTalk® tactile and a rule-based language, users can interact with their agents via different modalities such as music, speech recognition and synthesis.

Using this technology, a wide range of applications, from SimCity-like, interactive simulations to games and intelligent web agents can be built. However, AgentSheets does not appear to be well suited for large-scale parallel simulations due to the high maintenance cost of each individual agent.

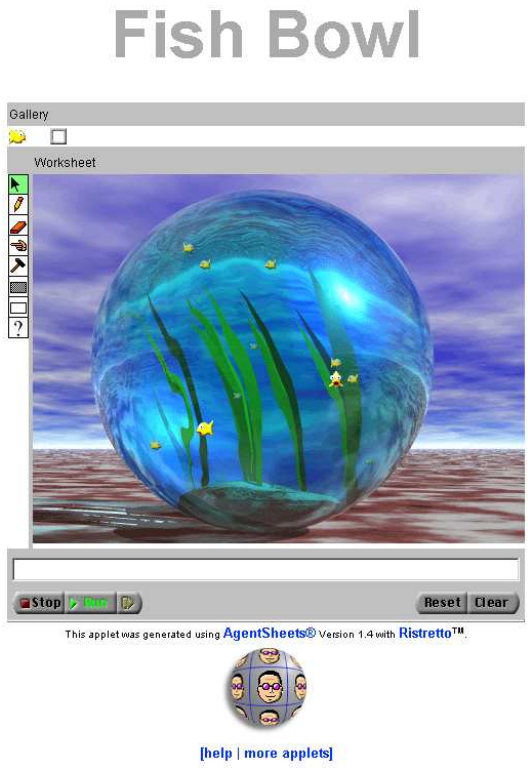


Figure B.3: A 'school of fish' applet implemented using AgentSheets 1.4

B.4 Swarm

Leaving the relatively simple products behind, a presentation of the more sophisticated research applications follows now. The first on the list is Swarm[URLc], originally developed at the Santa Fe Institute in New Mexico in 1994. Chris Langton, Swarm's inventor wanted to create a shared simulation platform for agent based models that allows the study of spatial interaction, adaptive, heterogeneous agents and nested subsystems like economies, firms or markets. Swarm should facilitate the development of these models and promote the study of complex nonlinear systems.

This software package for multi-agent simulation of complex systems is intended to be a useful tool for researchers in a variety of disciplines. According to [SBD03], some scientific domains that use multi-agent based simulations are sociology, biology, physics, chemistry, ecology and economy.

The basic architecture of Swarm is a of collection of concurrently interacting agents. This architecture allows the implementation of a large variety of agent based models. The Swarm project's main contributions are:

- event management via swarms and activity library
- information management via probes
- graphical input and output via GUI objects
- memory management
- support for multiple languages

In the Swarm system, a swarm is a collection of agents together with a schedule of events for these agents. It represents a model and allows construction of hierarchical models whereby an agent is composed of swarms. The task of observing the model is separated from the actual model and is implemented using special 'observer' agents. This allows to change the model and observer independently and makes Swarm stand out among its competitors.

Swarm is a collection of object oriented software libraries which provide support for simulation programming. These libraries are written in *Objective C* (a superset of C) and to write a simulation users needed to be able to program in 'Objective C' and 'Swarm' to incorporate the libraries in their own programs. However, in 1999 with the release of Swarm version 2.0, it became possible to use Java to utilise the libraries, which made Swarm much more accessible for researchers from non-computing backgrounds.

The software is available to the general public under GNU licensing terms for multiple platforms. Note that Swarm is experimental software, so while it is complete enough to be useful, it will always be under development. For this purpose the Swarm Development Group (SDG) was founded in September 1999. It aims to advance the state of the art in multi-agent based simulations, support the Swarm community and promote the free interchange of simulation models between computing specialists and the public.

B.5 Ascape

Ascape[URLb] was developed at Brookings Institute to support agent based research. It is designed to be a powerful (allow model interaction without programming), expressive (enable users to define complete models with small descriptions), flexible, abstract and high-level software framework for developing and analysing complex model designs.

In Ascape, an agent object exists within scapes, which are collections of agents such as arrays and lattices and these scapes are themselves agents. A typical Ascape model is made up of *collections of collections* of agents. The scapes provide a context for agent interaction and sets of rules that govern agent behaviour. The management of graphical views, parameter control for scape models and collection of statistics are some of Ascape's features.

Advanced end user tools make it possible for people with little programming experience to explore many aspects of model dynamics and give Ascape its 'easy-to-use' predicate.

Ascape follows some of the ideas behind Swarm. For example, the fact that agents exist in scapes, which are themselves agents is similar to the hierarchical models in Swarm. However, with the multitude of different tools to gather statistics, create graphs and influence the simulation, Ascape is slightly easier to use. It also provides the user with the ability to create movies of running models.

The framework is written in Java, which makes it platform independent but requires the user to possess some Java knowledge to code the models.

There is little written documentation, but a well maintained mailing list. There are some biological, anthropological and economic models available at *The Centre on Social and Economics Dynamics* at the Brookings Institute which developed Ascape and is also responsible for the well known Sugarscape simulation.

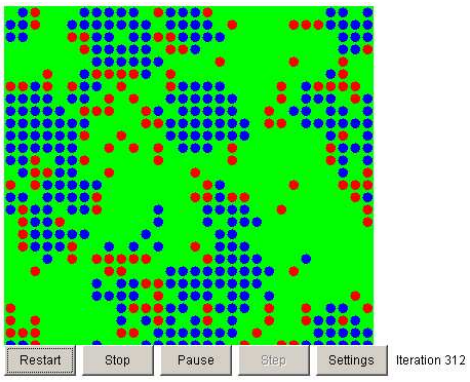


Figure B.4: A simulation of the well known Prisoner's Dilemma as a Java applet implemented using Ascape

B.6 RePast with SimBuilder

The final application under consideration is called RePast[URLd] and was developed at the Social Science Research Computing lab at the University of Chicago. It is a software framework for creating agent based simulations and both in philosophy and appearance very 'Swarm'-like. In addition to borrowing a lot from the Swarm simulation toolkit, it also possesses some of Ascape's features such as run-time model manipulation with the help of GUI widgets.

RePast, which is an acronym for REcursive Porous Agent Simulation Toolkit, provides a library of classes for creating, running, displaying and collecting data from agent based simulations. It is written in Java and hence platform independent.

RePast envisions a simulation as a state machine with two components: infrastructure and representation. The infrastructure is similar to what Swarm calls 'observer'. It is responsible to run the simulation, display and collect data and so forth. The actual simulation model however is termed 'representation'. All changes to the simulation occur through a schedule to provide clarity and extensibility.

Two of the main aims of RePast's developers are:

- to support the modelling of belief systems, agents, organisations and institutions as recursive social constructions
- to allow situated histories to be replayed with altered assumptions

However, neither of these aims has been achieved so far. On the bright side, RePast can take snapshots of running simulations, and create Quicktime movies of simulations.

SimBuilder, a rapid application development environment for RePast simulations shall not be forgotten. It is a visual drag'n'drop editor that allows the user to compose a simulation out of component pieces and specify the behaviour of these pieces using a subset of Phyton, called 'Not Quite Phyton'.

Without achieving its two main goals, there is no advantage in using RePast over Swarm. The fact that RePast uses Java and has a visual simulation building tool is no longer an argument since Swarm extended its list of languages one can use.

Appendix C

Implementation languages and technologies

According to the Network Agent Research group at Fujitsu Labs, the development of a multi-agent application demands the following characteristics of an implementation language. It should:

- facilitate symbolic programming in a distributed environment
- possess the ability to define processes easily
- support easy inter-process communication
- have some macro processing features to extend the language
- support powerful data structuring to store the state of the environment

It is decided to use April, the Agent PRocess Interaction Language, developed by Clark and McCabe[CM94a] since it fulfils all the above listed requirements. April is a strongly typed, higher-order language that is ideal for building multi-agent systems. It is implemented on top of the language C and uses a standard TCP/IP protocol to pass messages between different April invocations. It gives a very fast and portable foundation upon which multi-agent applications can be built.

To allow agents on different machines to converse with each other, April employs the *InterAgent Communication Model (ICM)*. This asynchronous communication model developed by Francis McCabe at Fujitsu Laboratories of America is based on a store-and-forward message architecture that allows messages to be sent (and later delivered) to agents even when they are offline.

April comes along with DialoX, a distributed HCI server, especially designed for symbolic programming languages. DialoX, also created by Francis McCabe at Fujitsu Labs provides window primitives and graphics capabilities to enable the user to create graphical user interfaces easily. Since DialoX is a server that

runs as a separate process, the application that utilises it has to communicate with it through means of message passing.

These three technologies, *April*, *ICM* and *DialoX* are the main components of all applications developed during the course of this project and will now be presented in more detail.

C.1 April

April is a process oriented, symbolic programming language which combines many of the features needed to implement a multi agent application. It is distributed and has strong support for problem solving and knowledge representation. Amongst its syntactic features are a strong type system, certain higher order features and a macro processing sub-language, which can be used to extend April by building layers on top of the basic language.

April was first released in 1994 and is in its fifth version currently. There is ample documentation available as well as sample programs and HowTos describing how certain things can be achieved with April. Nevertheless it is only used in academic and industrial research. It is currently maintained and supported by Francis McCabe at Fujitsu Labs.

April is chosen as an implementation language for this project due to its cheap forking of large numbers of processes. Agents are created by forking a code closure as a processes either from the command line or as part of the execution cycle of another agent.

Agents have their own thread of execution and communicate via message passing. Their state is private and protected in that external sources can only influence the state of an agent via a message and the agent can choose to ignore that message. Some key features of April are presented below:

Publicly named processes - all April processes have names associated with them. These names, also called *handles*, are a quartet of

- the target destination within the process (used for multi-threaded agents)
- the name of the process
- the home location of process
- the list of locations where the process may be found (current node, home,...)

The *name* and *home* fields combined give the agent an unique, global identifier. Both the *home* and the *locations* field can be specified as DNS addresses, IP addresses or URLs. A template for a handle is thus:

Target:Name@Home/[Location1, Location2, ...]

For example:

```
comThread:theAntEnv@sync04.doc.ic.ac.uk/[146.169.50.134,127.0.0.1]
```

This denotes the communication thread within the agent that represents the 'ant environment' which registered with the communication server on machine sync04 which can be found either there or on the local machine (note, sync04's and localhost's IP addresses are in the location field).

These handles can be passed around as arguments to function or procedure calls. Also, to send a message to a process P, one has to send it to the P's handle. Handles assigned by the programmer, such as *comThread* in the example above, are local to the April invocation in which the process runs. However, they can be made public by registering them with April communication servers that operate just like Internet domain name servers. This allows for global April applications.

Communication - is based on message passing. It uses the ICM described in section C.2 to send a message between two processes. If the processes reside on different host machines, then the message gets forwarded from the communication server (CS) of the sender to the communication server of the receiver. This can be depicted as:

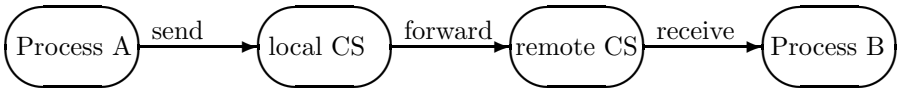


Figure C.1: The message flow between processes on different host machines

April's communication is one-way asynchronous. That means, no event is generated upon receipt of a message. Synchronous communication could be achieved, if the sender blocks after the sending of the message and the receiver immediately acknowledges the receipt.

Each process has a unique message receive buffer, in which a message is put, if the process is identified as the receiver in a *message send* statement. The message itself can be an arbitrarily complex symbolic structure and even executable code (a prerequisite for mobile agents).

```
message >> handle_of_receiver;
```

The complement of the message send is the *message receive*. It comes in two forms. A process can either receive a message from an unnamed process like so:

```
message <<;
```

or from a named process like so:

```
message << handle_of_expected_sender;
```

The previous two statements require the term *message* to be a variable. One can also restrict the types of messages a process can receive by using patterns. Patterns in April are combinations of literals and variables. For example, suppose a process only ever wants to receive pairs of the form ("Age", 23) for varying values of age. Then the message receive would use a pattern like so:

```
("Age",number?theAge) <<;
```

Note, that this requires the string 'Age' to be spelt exactly as defined in the pattern. So a message containing ("AGE", 23) would not be matched and remain in the message buffer.

Messages can also be received repeatedly like so:

```
repeat {
    "FindTour" ->> actions1
  | "DropPheromone" ->> actions2
  | _        ->> "Unknown message received\n" >> stdout;
} until 'quit;
```

Note, the underscore on the third line matches *everything*. If an incoming message is not the symbol *'quit* and does not match either of the strings "FindTour" and "DropPheromone", then the underscore rule matches by default. The action taken in this case in the above example would be to print an error message (by sending another message to the process handle *stdout*).

If a message receive statement finds no message that matches its pattern in the message buffer, then the process will suspend. Every time a new message arrives for this process, it will be reactivated to check whether the new message matches the pattern. If this is not the case, it will suspend again.

Another noteworthy point is, that messages are ordered by time of arrival not by time of transmission except when they were sent from the same thread.

Environment - April is designed to be used in a heterogeneous environment. Applications written in other languages can communicate with an April application using TCP/IP and functions and procedures developed in languages other than April can be linked into April applications.

The other application will see the April system as some kind of intelligent file while the April system treats the other application just as an ordinary process.

Real-time - April has constructs in it which allow for synchronisation with real-time events. Time-outs allow control over the length of time waited for a message to avoid being suspended forever. For example, the repeat loop in the previous example could be extended like so:

```
repeat {
    pattern1 ->> actions1
  | pattern2 ->> actions2
  | _        ->> "Unknown message received\n" >> stdout;
  | timeout 5 secs ->> actions3
} until 'quit;
```

This ensures that whenever the process does not receive any message for five seconds, the actions specified as *actions3* will be executed. This could just be writing a warning on the screen, but it may also contain a *leave* statement, that terminates the repeat loop.

Note however, it is very hard to guarantee exact timely behaviour in a multi-tasking environment (since the garbage collected might use the CPU at exactly the moment of the timeout), but April guarantees that the timeout will take place as soon as possible after the specified time.

Higher-order features

April provides two higher-order features allowing anonymous definitions for funcitons and modules. Lambda abstractions are anonymous functions that are defined and executed in line as opposed to named functions which are pre-defined and called in line.

Theta environments are anonymous (or named) programs or modules and they are referred to as environments because they can contain local variables and local functions and procedures. In the macro layer, the 'program' and 'module' statements are converted into theta environmetns like so:

```
‘ var; fun/proc; fun/proc.fun/proc’
```

where the '.' operator executes the specified function or procedure when the theta environment is elaborated.

Macro language - April's syntax is extensible. It has an operator precedence grammar that enables programmers to define new operators and incorporate them into a program. The semantics of the operators defined in such a way can be specified using macro definitions.

These macro definitions are just rewrite rules that are applied to the *program text*. Their general form is:

```
#macro pattern ==> replacement
```

where *pattern* is very similar to the normal patterns used for matching in April and *replacement* is a template describing the text that will replace the part of the program that matched the *pattern*. Variables in the definition of the replacement will have to be bound by the pattern matching operation that precedes the replacement.

Note, that these rewrite rules have no information on the true type of an expression but operate solely on the *text of the program*.

The order in which macro definitions are applied to the text of a program is important, since several patterns from different macro definitions may overlap. April's macro language uses a *inner-outer-last-first order*. Inner expressions are matched preferably over outer expressions, so that the smallest subexpression that can be rewritten, will be chosen. If more than one macro definition applies to the same subexpression, then the one defined last will be chosen.

The results of the application of all possible rewrite rules can be inspected by running the April compiler with the *-E* option on the command line.

Mobility Since procedures and functions are first class data values (from code fragments to whole agents), April lends itself naturally for mobile agent implementations.

Mobility is achieved by sending these first class data values as closures to other processes which can then either incorporate them (in the case of code fragments) or launch them as a new process (in the case of agents). This increases the interoperability between heterogeneous systems.

The execution stack is not serialised, so the agent starts at the beginning of the code when it is re-launched at another location. Also, the original agent (the sender) remains in place and can still receive messages as before (it could forward them, for example).

Note, that no distinction is made by April between agent stations and mobile agents themselves. Each agent can receive another agent and then travel to some other place if it wishes to do so.

The interested reader is referred to references such as Jonathan Dale's PhD thesis on mobile agents for distributed information management[Dal97].

C.2 InterAgent Communication Model

When talking about how communication works in April, we mentioned the ICM briefly. The InterAgent Communication Model is based on a store-and-forward message architecture which is quite similar to the email protocol. The receiving agent need not be on-line when a message is sent to it, since the message will be stored until the agent goes on-line. This allows for implementations of truly asynchronous applications.

The ICM consists of three components:

1. A low-level encoding mechanism that formats arbitrary messages into a more suitable form for transport across the network.
2. Communication servers (CS) that handle message routing, transportation and delivery (minimises the use of system resources when communicating with a large number of different host computers). CS run on a well-known port on each machine.
3. A library of primitives enabling agents to send and receive messages asynchronously, manipulate messages and converse with the communication server. These primitives are programming language-independent and APIs exist for April, Java, C, Prolog and Perl.

Each process has a handle (as described above) which is globally unique. This allows the ICM to route messages easily between different agent platforms. These messages contain structured content of the form of strings, numbers, records, list or even executable code.

Since the ICM possesses the mechanism that encodes and decodes the messages, applications do not have to use predefined message format information. This makes the ICM extensible since additional message oriented services can be added to it. However, this is also considered a security hole and hence the ICM was removed from April in its latest version.

The ICM specifies how messages are constructed and transported between agent platforms. It sits logically 'below' an agent communication language. KQML-style messages can be built in terms of the ICM's simpler components.

C.3 DialoX

With the invention of DialoX, Francis McCabe tackled the well known user interface problem that states that 70% of the development time of commercial applications is spend solving user interface issues. Multiple windows and mouse-driven interaction require tedious programming and DialoX aims to eliminate this tedium. The goal of DialoX is to reduce the time it takes to program the GUI of an application to 30%, leaving more time for the actual functional issues and the correctness of the program.

For the project undertaken by the author of this report, DialoX was instrumental in the fast prototyping of applications to visualise the effects of the ant algorithms. DialoX facilitate the GUI development enormously since it automates the layout of the user interface. On the other hand, this makes it unsuitable for applications where the GUI needs to be laid out precisely by the programmer.

At Fujitsu Laboratories of America, McCabe created this XML-based distributed HCI server called DialoX which interacts with applications via a special

purpose protocol. The user interface is specified in an external XML file and the application only invokes the top level window and then reacts to user interaction which arrives at the application in form of a message.

In its initial version, DialoX was based on Gtk/Gnome but is now implemented in Java to allow programmers of platform-independent applications to use it. Currently, Twidle and McCabe are working on a new version which will see DialoX extended by a scripting language to shift some of the computation from the application-side over to DialoX and thereby reduce the number of messages flowing between the application and the DialoX server.

However, DialoX is not restricted to window primitives. It also has graphics capabilities in form of the graph-widget, which can be specified in XML just like buttons, labels and scroll boxes. We will now have a look at how a window containing a graph widget and two buttons can be specified:

```
<window title="Blue Square and Two Buttons">
  <col>
    <row>
      <graph var="theGraphVar" height="900" width="900">
        <picture key="theSquare">
          <poly thickness="2" color="blue">
            <pt x="20" y="20"/>
            <pt x="880" y="20"/>
            <pt x="880" y="880"/>
            <pt x="20" y="880"/>
          </poly>
        </picture>
      </graph>
    </row>

    <row>
      <button msg="okClicked">
        OK
      </button>
      <button msg="cancelClicked">
        CANCEL
      </button>
    </row>
  </col>
</window>
```

This window represents a column in which two rows are stacked on top of each other. The top row contains the graph widget and the bottom row contains two buttons, labelled *OK* and *CANCEL*. If the user clicks the *OK* button, a message containing the string *okClicked* is sent to the user. We will look at the protocol in more detail shortly.

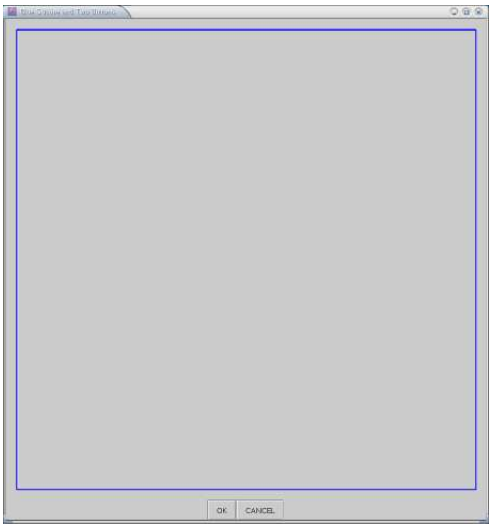


Figure C.2: Example of a DialoX window with graph widget and two button widgets

The graph widget in the example above specifies a drawing canvas of 900 by 900 pixels. It contains only one picture element, which contains a polygon. The *poly* object itself contains four points which represent the corners of a square which is to be drawn in *blue*.

The graph widget has a variable, namely *theGraphVar*, associated with it to allow for future updates to the canvas. One could add other picture elements, or even modify the picture element that contains the blue square. This modification requires the key of the picture element (*theSquare* in the example used here) and instructions on how to change picture.

The interaction protocol mentioned above will be described now using DialoX's interfaces with April. Note however, that a standard TCP connection to the server would be sufficient to send and receive XML tags as ASCII text.

Imagine the example with the graph widget and the two buttons from before is saved in a file called *blueSquare.xml*. Then the following command will show the window:

```
sio_term(1,xmlNode("open",[("url","./blueSquare.xml")],[])) >> DX;
```

The first argument to *sio_term* is the message number to keep track of what messages have been sent and received. The second argument is an XML description of what we want DialoX to do. The whole *sio_term* construct is sent as a message to the handle of the process that represents the DialoX server (denoted *DX* in this example). If, after the window with the blue square is displayed, the user presses the *OK* button, a message of the following form will arrive at the April application that launched the window:

```
sio_term(2,xmlNode("clicked",[],[xmlText("okClicked")]))
```

Using the message receive loop from section C.1 one can distinguish between the buttons pressed and react accordingly:

```
repeat {
  sio_term(_,xmlNode("clicked",[],[xmlText("okClicked")])) ->> act1
  sio_term(_,xmlNode("clicked",[],[xmlText("cancelClicked")])) ->> act2
  X ->> "Unknown message received: " ++ X^0 >> stdout;
until 'quit;
```

Upon receiving a message of the second type, the actions specified in *act2* could be something like:

```
'quit >> self();
```

which would send the *quit* symbol to itself and hence terminate the message receive loop. Receipt of a message of the first type, on the other hand, might result in a message send from the application to the DialoX server asking it to update the graph widget.

Appendix D

Oliver30 and other instance data

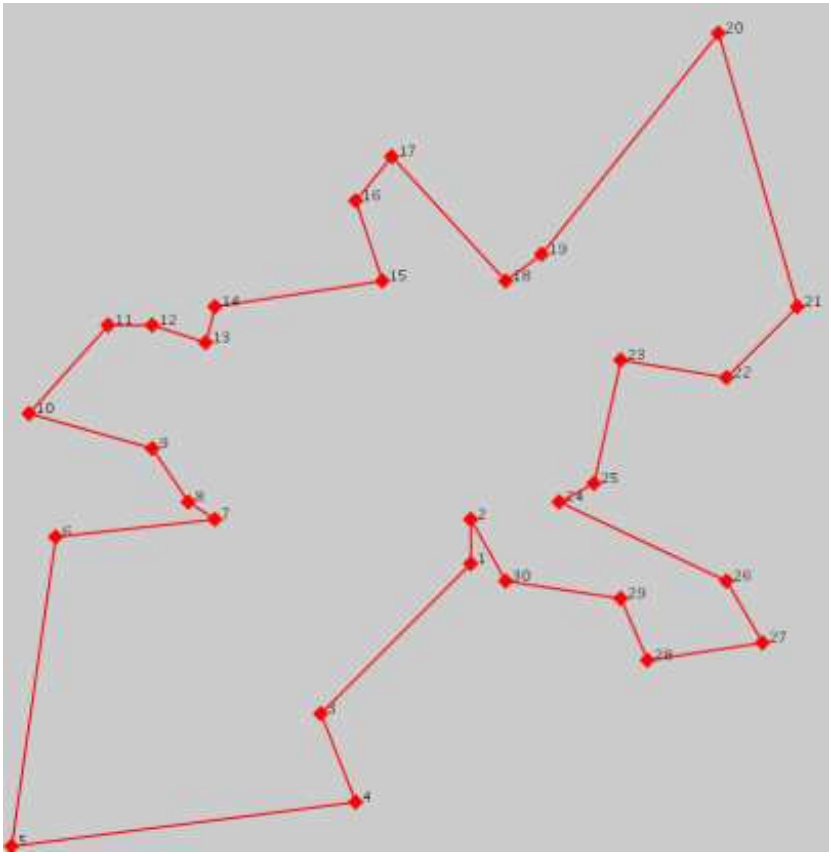


Figure D.1: Best known solution to the Oliver30 problem instance of the TSP

The enumeration of the cities indicates the formerly best known solution to this problem instance from [SWF89]. The ant system found an improvement by swapping over cities 1 and 2 and also cities 24 and 25.

Problem	cities	best length	coordinates of cities
Oliver30	30	423.74	(2,99) (4,50) (7,64) (13,40) (18,54) (18,40) (22,60) (24,42) (25,62) (25,38) (37,84) (41,94) (41,26) (44,35) (45,21) (54,67) (54,62) (58,35) (58,69) (62,32) (64,60) (68,58) (71,44) (71,71) (74,78) (82,7) (83,46) (83,69) (87,76) (91,38)
Eilon50	50	427.96	(5,64) (5,25) (5,6) (7,38) (8,52) (10,17) (12,42) (13,13) (16,57) (17,33) (17,63) (20,26) (21,47) (21,10) (25,32) (25,55) (27,68) (27,23) (30,48) (30,15) (31,62) (31,32) (32,22) (32,39) (36,16) (37,69) (37,52) (38,46) (39,10) (40,30) (42,57) (42,41) (43,67) (45,35) (46,10) (48,28) (49,49) (51,21) (52,33) (52,41) (52,64) (56,37) (57,58) (58,27) (58,48) (59,15) (61,33) (62,42) (62,63) (63,69)
Eilon75	75	542.31	(6,25) (7,43) (9,56) (10,70) (11,28) (12,17) (12,38) (15,5) (15,14) (15,56) (16,19) (17,64) (20,30) (21,48) (21,45) (21,36) (22,53) (22,22) (26,29) (26,13) (26,59) (27,24) (29,39) (30,50) (30,20) (30,60) (31,76) (33,34) (33,44) (35,51) (35,16) (35,60) (36,6) (36,26) (38,33) (40,37) (40,66) (40,60) (40,20) (41,46) (43,26) (44,13) (45,42) (45,35) (47,66) (48,21) (50,30) (50,40) (50,50) (50,70) (50,4) (50,15) (51,42) (52,26) (54,38) (54,10) (55,34) (55,45) (55,50) (55,65) (55,57) (55,20) (57,72) (59,5) (60,15) (62,57) (62,48) (62,35) (62,24) (64,4) (65,27) (66,14) (66,8) (67,41) (70,64)

Table D.1: Coordinates of cities for three famous problem instances

These three problems have been used repeatedly in the past to compare generic and specialised approximation algorithms for the Travelling Salesman Problem.

Appendix E

Parallelisation data

E.1 Local optimiser

Problem	Machine	2-OPT	Time	Length	off-%
Eilon75	m04	No	1655.1	548.73	1.18
Eilon75	m05	No	1632.6	561.08	3.46
Eilon75	m05	No	1517.6	560.51	3.35
Eilon75	m05	No	1634.2	552.12	1.81
Eilon75	m06	No	1704.5	553.69	2.10
		Averages:	**1628.80**	555.226	2.38
Eilon75	m06	Yes	2055.8	556.25	2.57
Eilon75	m06	Yes	1954.3	551.51	1.69
Eilon75	m06	Yes	2147.0	545.62	0.61
Eilon75	m06	Yes	1848.2	551.20	1.64
Eilon75	m06	Yes	1901.0	545.90	0.66
		Averages:	**1981.26**	550.096	1.434

Table E.1: Influence of local search on execution time

E.2 Splitting the environment from the ant processes

Problem	Machine-Ants	Machine-Env	Time	Length	off-%
Eilon50	m06	m06	556.0	439.5	2.69
Eilon50	m04	m04	622.9	432.2	1.01
Eilon50	m04	m04	630.8	445.4	4.07
Eilon50	m04	m04	635.1	445.4	4.07
Eilon50	m05	m05	592.7	430.7	0.64
		Averages:	**607.5**	438.64	2.496

Table E.2: Running ants and environment on the same machine

Problem	Machine-Ants	Machine-Env	Time	Length	off-%
Eilon50	m02	m04	345.1	434.3	1.48
Eilon50	m02	m04	388.0	428.4	0.11
Eilon50	m05	m04	365.0	432.8	1.13
Eilon50	m06	m04	313.2	431.4	0.80
Eilon50	m03	m04	324.2	432.0	0.96
		Averages:	**347.1**	431.78	0.896

Table E.3: Running ants and environment on separate machines

E.3 Distributing the ant processes

Problem	Machines-Ant	2-OPT	Time	Length	off-%
Eilon50	m02	Yes	345.1	434.3	1.48
Eilon50	m02	Yes	388.0	428.4	0.11
Eilon50	m05	Yes	365.0	432.8	1.13
Eilon50	m06	Yes	313.2	431.4	0.80
Eilon50	m03	Yes	324.2	432.0	0.96
		Averages:	**347.1**	431.78	0.896
Eilon50	m02/m06	Yes	274.8	429.4	0.33
Eilon50	m02/m06	Yes	275.4	432.2	1.01
Eilon50	m02/m06	Yes	275.3	431.1	0.73
Eilon50	m03/m06	Yes	270.9	432.2	0.98
Eilon50	m03/m05	Yes	277.6	445.4	4.07
		Averages:	**274.8**		
Eilon50	m02/m05/m06	Yes	258.2	445.4	4.07
Eilon50	m02/m05/m06	Yes	254.8	432.8	1.13
Eilon50	m02/m05/m06	Yes	268.3	431.4	0.80
Eilon50	m02/m05/m06	Yes	327.2?	432.0	0.96
Eilon50	m03/m05/m06	Yes	230.0	432.8	1.13
		Averages:	**267.7**		
Eilon50	m02/m03/m05/m06	Yes	274.3	432.2	1.01
Eilon50	m02/m03/m05/m06	Yes	240.1	428.4	0.11
Eilon50	m02/m03/m05/m06	Yes	271.6	442.2	3.32
Eilon50	m02/m03/m05/m06	Yes	259.6	433.0	1.19
Eilon50	m02/m03/m05/m06	Yes	260.7	428.4	0.11
		Averages:	**261.2**		

Table E.4: Running ant process on several machines

E.4 Reduced communication

Local Attempts	Machines-Ant	2-OPT	Time	Length	off-%
2	m02/m05	Yes	125.2	431.1	0.73
2	m03/m06	Yes	127.6	433.0	1.19
2	m02/m06	Yes	195.7 ?	431.1	0.73
2	m03/m05	Yes	133.0	431.1	0.73
2	m05/m06	Yes	135.5	430.3	0.55
		Averages:	**143.4**	431.32	0.786
5	m02/m06	Yes	67.9	435.0	1.65
5	m02/m05	Yes	71.4	435.2	1.69
5	m02/m06	Yes	68.6	429.8	0.44
5	m05/m06	Yes	71.7	433.7	1.34
5	m03/m06	Yes	70.8	435.2	1.69
		Averages:	**70.08**	433.78	1.362
10	m02/m05	Yes	58.6	435.5	1.78
10	m03/m05	Yes	61.6	433.7	1.34
10	m05/m06	Yes	61.6	432.9	1.17
10	m03/m06	Yes	68.2	431.8	0.91
10	m03/m06	Yes	61.4	436.4	1.96
		Averages:	**62.28**	434.06	1.432
14	m02/m06	Yes	64.9	431.3	0.79
14	m02/m06	Yes	64.8	432.2	1.01
14	m02/m05	Yes	61.6	435.7	1.82
14	m05/m06	Yes	62.8	439.6	2.73
14	m03/m06	Yes	59.8	432.8	1.14
		Averages:	**64.06**	434.32	1.498
20	m03/m06	Yes	67.2	434.3	1.48
20	m03/m06	Yes	64.1	432.7	1.12
20	m03/m05	Yes	64.0	445.4	4.07
20	m02/m05	Yes	60.8	440.8	3.00
20	m02/m06	Yes	62.3	431.4	0.80
		Averages:	**63.68**	436.92	2.094

Table E.5: Results of reduced communication experiments

E.5 Subcolonies

Local attempts	Machines-Ant	2-OPT	Time	Length	off-%
1	m02/m05	Yes	190.6	432.2	0.99
1	m02/m05	Yes	188.4	432.2	0.99
1	m03/m05	Yes	189.0	431.1	0.74
1	m03/m05	Yes	184.3	432.8	1.13
1	m02/m03	Yes	193.8	434.3	1.48
		Averages:	**189.22**		

Table E.6: Using sub-colony communication one-way

Local attempts	Machines-Ant	2-OPT	Time	Length	off-%
1	m02/m05	Yes	167.1	432.7	1.11
1	m02/m05	Yes	171.2	428.4	0.11
1	m03/m06	Yes	165.7	428.4	0.11
1	m03/m06	Yes	176.0	431.1	0.74
1	m02/m03	Yes	168.1	432.8	1.13
		Averages:	**169.62**		

Table E.7: Two-way sub-colony communication and ants on two machines

Local attempts	Machines-Ant	2-OPT	Time	Length	off-%
1	m02/m03/m05	Yes	109.9	441.7	3.21
1	m02/m03/m05	Yes	120.4	433.4	1.29
1	m02/m03/m05	Yes	124.2	433.0	1.17
1	m02/m03/m05	Yes	120.6	430.5	0.59
1	m02/m03/m05	Yes	116.4	432.7	1.11
		Averages:	**118.3**		

Table E.8: Two-way sub-colony communication and ants on three machines

Local attempts	Machines-Ant	2-OPT	Time	Length	off-%
1	m02/m03/m05/m06	Yes	97.1	429.8	0.44
1	m02/m03/m05/m06	Yes	83.9	431.3	0.79
1	m02/m03/m05/m06	Yes	81.5	428.5	0.13
1	m02/m03/m05/m06	Yes	75.4	437.7	2.29
1	m02/m03/m05/m06	Yes	92.7	431.7	0.88
		Averages:	**86.12**		

Table E.9: Two-way sub-colony communication and ants on four machines

Appendix F

Sample execution trace

Output for the Oliver30 Problem with best known result 423.74

0	602.191680952806	481.179537073577	481.179537073577	51.9362556922699
1	539.120789066102	460.504631807133	460.504631807133	46.6464303172493
2	546.827703374946	444.56788548067	444.56788548067	39.1456527958425
3	514.363459582563	441.002547633235	441.002547633235	44.3897895019038
4	523.377283318957	455.394567903649	441.002547633235	39.3268174314845
5	522.499390913142	453.224171598663	441.002547633235	32.7581073836472
6	523.049041994126	437.406557794296	437.406557794296	51.7935059462957
7	531.279551351457	465.310297608873	437.406557794296	37.5386865650413
8	506.465951505182	425.820140589076	425.820140589076	39.8639118203405
9	525.733262064195	454.928575298146	425.820140589076	39.4933670634493
10	508.247941669055	430.949077800447	425.820140589076	39.3203173169933
11	517.811168097665	425.820140589076	425.820140589076	40.2834618737472
12	521.354553136352	447.917022468885	425.820140589076	39.4782091805378
13	516.298260156901	442.124716717941	425.820140589076	35.6938919522941
14	510.440243121162	443.908240791291	425.820140589076	47.0416828714723
15	514.787382109178	444.740688473083	425.820140589076	43.6875091176694
16	522.36592080706	425.820140589076	425.820140589076	44.7595623345933
17	509.760945591483	444.110806898799	425.820140589076	42.2229648730764
18	513.086804604047	452.113899236964	425.820140589076	41.8458013905418
19	508.789460776726	443.861821194275	425.820140589076	39.239197754526
20	516.018224366652	425.820140589076	425.820140589076	42.1331733034413
21	521.622720516464	451.44210273346	425.820140589076	49.8176150956573
22	518.851355616363	446.990640662187	425.820140589076	44.3915505323419
23	508.142544273972	435.275227566801	425.820140589076	41.1320371080417
24	514.360459518876	434.922496816667	425.820140589076	43.8205842833484
25	515.051343673296	435.575387639329	425.820140589076	36.9717250855875
26	518.096345324004	447.046937335453	425.820140589076	36.5152518706797
27	517.538130195142	444.10348967671	425.820140589076	41.5084345554057
28	520.096823828178	425.820140589076	425.820140589076	45.2742021978376
29	514.65219668635	425.820140589076	425.820140589076	41.5441613243873
30	513.241498574045	455.3688640232	425.820140589076	31.1607313537418

Columns represent, in order, iteration number, iteration average, iteration best, global best so far and standard deviation of current iteration. The fact, that the last column shows a number bigger than zero means that no stagnation has occurred yet.

A very good solution (0.49% off the optimal one) is found after 8 iterations.

Bibliography

[AS61] E. L. Arnoff and S. S. Sengupta. The traveling salesman problem. *Progress In Operations Research*, 1, 1961.

[AW83] Rollin S. Armour, Jr. and John Archibald Wheeler. Physicist's version of traveling salesman problem: statistical analysis. *American Journal of Physics*, 51(5):405–406, May 1983.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: from natural to aritificial systems*. Oxford University Press, 1999.

[Ben88] Gerardo Beni. The concept of cellular robotic systems. In *Proceedings 1988 IEEE Int. Symp. On Intelligent Control*, pages 57–62. IEEE Computer Society Press, 1988.

[BHS99] Bernd Bullnheimer, Richard F. Hartl, and Christine Strauss. A new rank based version of the ant system: A computational study. *Central European Journal of Operations Reserach*, 7:25–38, 1999.

[BKS97] B. Bullnheimer, G. Kotsis, and C. Strauss. Parallelization strategies for the ant system. Technical report, University of Vienna, 1997.

[BN68] M. Bellmore and G. L. Nemhauser. The traveling-salesman problem: A survey. *Operations Research*, 16:538–558, 1968.

[CDMT94] A. Colorni, M. Dorigo, V. Maniezzo, and M. Trubian. Ant system for job-shop scheduling. *JORBEL - Belgian Journal of Operations Research, Statistic and Computer Science*, 34(1):39–53, 1994.

[CM94a] Keith L. Clark and Francis G. McCabe. April - Agent PRocess Interaction Language. In *Intelligent Agents - ECAI-94 Workshop on Agent Theories, Architectures, and Languages*, Lecture Notes in Computer Science, pages 324–340. Springer, 1994.

[CM94b] Keith L. Clark and Francis G. McCabe. Distributed and object oriented symbolic programming in April. Technical report, Dept. of Computing, Imperial College, London, 1994.

[Cor99] David Corne. *New ideas in optimisation*. McGraw-Hill, 1999.

[CS02] C.J.Eyckelhof and M. Snoek. Ant systems for a dynamic TSP. In Marco Dorigo, Giani Di Caro, and Michael Sampels, editors, *Ant Algorithms*, LNCS, pages 88–99, 2002.

[Dal97] Jonathan Dale. *A Mobile Agent Architecture for Distributed Information Management*. PhD thesis, University of Southampton, England, 1997.

[Day90] Judith E. Dayhoff. *Neural network architectures: an introduction*. Van Nostrand Reinhold, 1990.

[DC99] Marco Dorigo and Gianni Di Caro. Ant colony optimization: A new meta-heuristic. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1470–1477. IEEE Press, 1999.

[DG97a] M. Dorigo and L. M. Gambardella. Ant colonies for the traveling salesman problem. *BioSystems*, 1997.

[DG97b] 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.

[DMC96] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. The ant system: Optimization by a colony of coorperating agents. *IEEE Trans. on Systems, Man and Cybernetics-Part B*, 26(1):29–41, 1996.

[Dor92] Marco Dorigo. *Ottimizzazione, Apprendimento Automatico, ed Algoritmi Basati su Metafora Naturale*. PhD thesis, Politecnico di Milano, Italy, 1992.

[DS03] Marco Dorigo and Thomas Stützle. The ant colony optimization metaheuristic: Algorithms, applications, and advances. In Fred Glover and Gary A. Kochenberger, editors, *Handbook of Metaheuristics*, International Series in Operations Research and Management Science, pages 251–285. Kluwer Academic Publishers, 2003.

[EKS01] Russell C. Eberhard, James Kennedy, and Yuhui Shi. *Swarm Intelligence*. Morgan Kaufmann, 2001.

[Fid02] Stefka Fidanova. Aco algorithm for MKP using different heuristic information. In 5^{th} *international conference of numerical methods and applications*, Lecture Notes in Computer Science, pages 434–440. Springer, 2002.

[Fog01] David B. Fogel. *Blondie24: Playing at the edge of AI*. Morgan Kaufmanm, 2001.

[GADP89] S. Goss, S. Aron, J. L. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579–581, 1989.

[GD95] L. M. Gambardella and M. Dorigo. Ant-Q: A reinforcement learning approach to the traveling salesman problem. In A. Prieditis and S. Russell, editors, *Proceedings of ML-95, Twelfth International Conference on Machine Learning*, pages 252–260. Morgan Kaufmann, 1995.

[GM01] Michale Guntsch and Martin Middendorf. Pheromone modification strategies for ant algorithms applied to dynamic TSP. In E.J.W. Boers, editor, *Applications of Evolutionary Computing: Proceedings of EvoWorkshop 2001*, volume 2037 of *LNCS*, pages 213–222. Springer-Verlag, 2001.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[GTD98] L. M. Gambardella, E. Taillard, and M. Dorigo. Ant colonies for the QAP. *Journal of the Operational Research Society*, 1998.

[HC02] Theodore W. Hong and Keith L. Clark. Concurrent programming on the web with webstream. Technical report, Dept. of Computing, Imperial College, London, 2002.

[Hol75] John Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975.

[Hub88] Bernardo A. Huberman. *The Ecology of Computation*. North-Holland, 1988.

[Joh01] Steven Johnson. *Emergence: the connected lives of ants, brains, cities and software*. Scribner, 2001.

[Jon75] Kenneth Alan De Jong. *An analysis of the behaviour of a class of genetic adaptive systems*. PhD thesis, University of Michigan, Ann Arbor, 1975.

[KOW03] Simon Kaegi, Terri Oda, and Tony White. Revisiting elitism in ant colony optimization. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2723 of *LNCS*, pages 122–133. Springer-Verlag, 2003.

[Liu01a] Jiming Liu, editor. *Agent Engineering*, volume 43 of *Series in machine perception and artificial intelligence*. World Scientific, 2001.

[Liu01b] Jiming Liu. *Autonomous agents and multi-agent systems: explorations in learning, self-organization, and adaptive computation*. World Scientific, 2001.

[Mak01] Vicky Mak. *On the Asymmetric TSP with Replenishment Arcs*. PhD thesis, University of Melbourne, Australia, 2001.

[MC99] Vittorio Maniezzo and Alberto Colorni. The ant system applied to the quadratic assignment problem. *Knowledge and Data Engineering*, 11(5):769–778, 1999.

[McC95] Francis G. McCabe. *April Reference Manual*, 1995.

[Mer00] Stephan Mertens. Random cost in combinatorial optimization. *Physical Review Letters*, 84(6):1347–1350, 2000.

[Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[NLG00] N.L.Boland, L.W.Clarke, and G.L.Nemhauser. The asymmetric travelling salesman problem with replenishment arcs. *European Journal of Operations Research*, 123:408–427, 2000.

[Pap77] C. H. Papadimitriou. Euclidean TSP is NP-complete. *Theoretical Computer Science*, 4:237–244, 1977.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Res94] Mitchel Resnick. *Turtles, termites and traffic jams: explorations in massively parallel microworlds*. Bradford Books, 1994.

[SBD03] Jaime S. Sichman, Francois Bousquet, and Paul Davidsson, editors. *Multi-Agent-Based Simulation II: Third International Workshop, MABS 2002*. Lecture Notes in Artificial Intelligence. Springer, 2003.

[SH97] Thomas Stützle and Holger Hoos. Improvements on the ant-system: Introducing the MAX-MIN ant system. In *International Conference on Artificial Neural Networks and Genetic Algorithms*. Springer, 1997.

[SKS02] Krzysztof Socha, Joshua Knowles, and Michael Sampels. A $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System for the University Timetabling Problem. In Marco Dorigo, Giani Di Caro, and Michael Sampels, editors, *Proceedings of ANTS 2002 – Third International Workshop on Ant Algorithms*, volume 2463 of *LNCS*, pages 1–13. Springer-Verlag, Berlin, Germany, 2002.

[SWF89] T. Starkweather, D. Whitley, and D. Fuquay. Scheduling problems and travelling salesman: the genetic edge recombination operator. In J. David Schaffer, editor, *Proceedings of the* 3^{rd} *international conference on genetic algorithms*. Morgan Kaufmann, 1989.

[URLa] `http://education.mit.edu/starlogo`.

[URLb] `http://www.brook.edu/dybdocroot/es/dynamics/models/ascape`.

[URLc] `http://www.swarm.org`.

[URLd] `http://repast.sourceforge.net`.

[URLe] http://ccl.sesp.northwestern.edu/netlogo.

[URLf] http://www.agentsheets.com.

[URLg] http://www.integratedmodelling.org.

[URLh] http://www.madkit.org.

[URLi] http://www.santafe.edu/projects/echo/echo.html.

[URLj] http://www.fipa.org/specs/fipa00008/SC00008I.pdf.

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

[URLl] http://www.claymath.org/millennium/P_vs_NP/Official_Problem_Description.pdf.