Ants find the shortest path

Gareth Hunter

A dissertation submitted in partial fulfilment of the requirements of
Dublin Institute of Technology for the degree of
M.Sc. in Computing (Advanced Software Engineering)

AUGUST 2014

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Advanced Software Engineering), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the test of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Dublin Institute of Technology and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Signed: _____

Date: ***DD Month Year***

ABSTRACT

Ant Colony Optimisations algorithms (ACO) belong to a section of swarm intelligence and are among the best performing meta-heuristics for solving NP-Hard problems. ACO simulates the intelligent behaviour of real ants in nature finding the shortest path between their nest and a food source. The goal of this dissertation is to investigate five different variants of these algorithms for the solving of the Travelling Salesperson Problem (TSP). Ant System the most basic ant algorithm suffers the limitation that once a solution is found and reinforced it becomes difficult for the algorithm to effectively explore other solution spaces. Therefore extended algorithms such as, Elitist Ant System, Best-Worst Ant System, MinMax Ant System and Ant Colony System among many have been created by researchers attempting to address this limitation. The experimental evaluation is based upon tour constructions as proposed since the Second International Contest on Evolutionary Optimisation for the selected instances of TSP, discovering the best performing algorithms both empirically and evaluating the best overall found solution. Furthermore this dissertation introduces Mutated Ant that uses traits from other ACO algorithms combined with mutation from the Genetic Algorithm and evaluation show this algorithm promising. Furthermore all variations of these implementations are extended with local search methods and compare the heuristics that guide them. In addition a Lin-Kernighan heuristic we introduced and applied to the basic algorithms with surprising results. However, local search has the limitation of directing the solution towards to a local optimal, and Random Ant is introduced to effectively demonstrate this limitation. In addition ACO is compared to other approaches such as the Genetic Algorithm and a linear algorithm.

KEYWORDS:

Ant Colony Optimisation, Swarm intelligence, Genetic Algorithm, the travelling salesman problem, Mutated Ant.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to Mark Foley for his patience and his time, his practical advice and discussions on the areas to focus. Furthermore a general thanks to the other master's students, especially Sammeer Handa who kept me motivated when I wasn't. Also Damian Gordon who inspired me with his interesting lectures leading to this dissertation. Thanks to Deirdre Lawless whom gave me the confidence and the tools required to complete documentation without which I would have been lost. Thanks also to. Pierpaolo Dondio for introducing me to functional programming, and Sarah Jane Delany for all her help enrolling me in the MSc programme.

"Nothing is particularly hard if you divide it into small jobs." (Henry Ford)

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF EQUATIONS

TABLE OF CODE LISTINGS

Key Terms

Metaheuristic: consists of a set of algorithmic concepts used to solve a general class of computational problems, which can be applied to different problems with only a few modifications.

A Graph

A graph G = (V, E) consists of a set of objects V = {v_1, v_2, …} called vertices, and another set E = {e_1, e_2, …} whose elements are called edges. Each edge e_k in E is identified with an unordered pair (v_i, v_j) of vertices. The vertices v_i, v_j associated with edge e_k are called the end vertices of e_k. The most common representation of graph is by means of a diagram, in which the vertices are represented as points and each edge as a line segment joining its end vertices. Often this diagram itself is referred to as a graph.

1 INTRODUCTION

In various areas of computer science and robotics, there is always a need to design intelligent systems capable of performing complex tasks with the help of self-organised autonomous components which are distributed and have no central control. Many researchers have been inspired by the collective behaviour of groups of animals like schools of fish, flocks of birds and social insects, including ants, bees, and termites. The ability of swarms to survive and solve complex tasks such as predator invasion and path finding, while being made up of simple creatures with limited capabilities (ants have limited eyesight) is of particular interest to computer algorithm designers. Various algorithms have been designed for distributing problems based on the intelligent behaviour of swarms leading to a paradigm known as swarm intelligence (Reynolds, 1987).

Swarm intelligence can be defined as the collective behaviour of artificial or natural decentralised self-organising systems. Swarm intelligence is generally simpler and more flexible than traditional algorithms at solving very complex problems and can solve many real-world problems. Among the most popular swarm intelligence algorithms is ant colony optimisation (ACO) these algorithms are known as meta-heuristic algorithms (Beni and Wang, 1993).

Comment [LP1]: Think you need to refer to your source here

Ant colony optimisation is a section of swarm intelligence in which researchers study the behaviour of ants. *"Ants are able to discover the shortest path to a food source and share this information with other ants"* (Sim and Sun, 2003a).

These algorithms haven been well researched and variations developed such as Ant System, Elitist Ant System, Ranked Ant System, Best-Worst Ant System, Min-Max Ant System and Ant colony System (Dorigo and Socha, 2006a).

This dissertation investigates these five different variants of the Ant Colony Optimisation algorithms, the genetic algorithm, brute force algorithm and introduce the mutated ant and random ant and present a framework written in C++ to demonstrate.

1.1 Research problem

As legend has it, the creator of the game of chess showed his invention to the king, and the king was so pleased he gave the inventor the right to name his own prize. The inventor who was very clever replied to the king "*I'm not a greedy man, can you place on the first square of my chessboard one grain of wheat and keep doubling that, for each sequential square*". The king was unaware of the exponential nature of the arithmetic and quickly accepted and was even offended by the inventor. However when the grain is placed out on square 64 of the chessboard, this arithmetically accounts for more grain than probably has ever been harvested by humans: (n-1)! grains.

During the past 60 years, modern computers have been used to solve problems; most problems can be solved in a reasonable amount of time, and other problems cannot be solved quickly. For example, sorting a list of strings in the computer can be accomplished very quickly, proportional to the size; these kinds of problems belong to a class called ***P***. However, other problems such as finding the shortest distance between three nodes in a directed graph is very computationally expensive as each permutation has to be calculated. That is given a set{1,2,3} of vertices all (1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1) have to be evaluated.

When optimisation problems are difficult to solve and cannot be solved efficiently or quickly are said to be NP-hard. The most notorious and common NP-hard problem is the Travelling Salesperson Problem (TSP). Informally a salesperson intends to visit a number of cities and can only visit a city once, and returning to the start city, travelling the shortest distance This NP-hard problem is easy to understand but increases in complexity exponentially as each new city is added (Karp, 1977). As explained in more formal terms by Donald, (2011, p. 1) *"Given n is the number of cities to be visited, the total number of possible routes covering all cities can be given as a set of feasible solutions of the TSP and is given as (n-1)!/2"*.

TSP is proven to be NP-hard, meaning that the time to solve the problem increases very quickly at the same rate as the grain on the chessboard. If one imagines there are 300 squares on the chessboard and puts in the context of TSP that is given a 300-city

problem is *"A salesperson intends to visit 300 cities and can only visit a city once, and returning to the start point minimising the distance they have to travel"* The calculations (300-1)! approx. 10^{611} – an even greater result than the number of atoms in the observable universe[1]. In fact, even executing a brute force search on Travelling Salesperson Problem with 50 cities requires approx. 30 vigintillion steps 10^{64}, and increasing the size by just one city is greater than each consequential step added together for 51 cities approx. 1 unvigintillion 10^{65}. A brute force calculation is even impossible for the simplest of TSP problems as ***n*** get bigger ***(n-1)!*** expands faster than exponential. Solving these problems is of great interest to computer scientists as it has many real world applications.

There are alternative methods, among which are methods known as approximation algorithms, but the price for using these algorithms is that they trade accuracy for efficiency, in a sense, guessing the optimal solution.

1.2 Intellectual challenge

The artefacts in this dissertation will be among the few projects to carry out the empirical analysis of the various algorithms side-by-side. In fact, there have been no research papers found on any of the Ant algorithms implemented in C++, and any frameworks written in other languages that have been implemented generally don't compare with other algorithms within the framework. The framework was designed in such a way that plugging in different algorithms can be simple.

Although all ant algorithms have been well researched, swarm intelligence can be a very complicated field and implementation of the algorithms is complex. In addition, as the chosen algorithms are approximation algorithms they can be difficult to analyse, (that is if they are functioning within the correct bounds) and can be extremely hard to implement and test.

[1] the number of atoms in the entire observable universe is estimated to be within the range of 10^{78} to 10^{82}

The size of the framework is a medium-sized project and a big undertaking for just three months of software development combined with research, and dissertation writing. Considering that, the development is the biggest part of this dissertation and has to be well designed to accommodate loose coupling between each objects.
The other main intellectual challenge is how to compare each algorithm for the ant systems as in *O notation* (Chapter 2) they are all the same $O(m.n^2)$ and they are all approximation algorithms. The main key for these algorithms is randomness and this can be hard to compare as the algorithms will not solve the same problem the same way next time they are executed.

This project was a big challenge as the author initially had a very limited knowledge in the field of swarm intelligence and never encountered the Travelling Salesperson Problem. As a result, this framework was developed in C++, as this is the language the author felt most comfortable with in completing a project of this size.

Therefore, for successful completion of this dissertation, the literature review is not limited to the fields of swarm intelligence, ant colony optimisation and the genetic algorithm, but also includes computational complexity, analysis of algorithms, software design, graph theory, C++ and local search methods.

1.3 Research objectives

The aim of this research is to analyse the performance of each variation of Ant Colony Optimisation (ACO) algorithms, using the well-researched and benchmarked Travelling Salesperson Library (TSPLIB) dataset as input (Reinelt, 1991). To demonstrate, a framework called AntSolver is presented, developed in C++ using OO paradigms including design patterns, the ‘don’t repeat yourself’ (DRY) principle and refactoring to completely decouple each part of the application, to enable expandability and an open architecture. The framework is developed using test driven development methodology, keeping in context with the MSC in advanced software engineering being pursued (Fowler *et al*., 1999).

To compare these algorithms with other existing approaches the genetic algorithm and a brute force approach and introducing the Random Ant System have been

implemented within the framework. These implementations have been extended using local search methods, comparing the heuristics that guide the Ant Colony Optimisation algorithms and experimenting with different pheromone updating strategies.

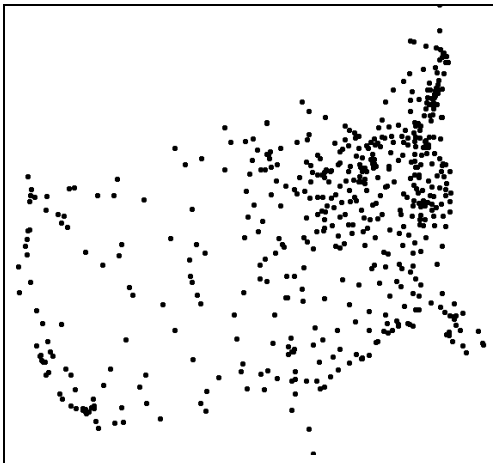
Furthermore, a local search procedure is presented and applied to the basic ant system and Elitist ant system as part of the research. This is an entirely new research procedure, the results of which are surprising.

A new algorithm called Mutated Ant is presented and results show it is an extremely powerful and faster performing ant algorithm. In fact experiments show it is one of the best performing ant algorithms available, especially when working with large Travelling Salesperson Problems. Mutated Ant combines various traits from other ant algorithms, mutation from the genetic algorithm, and a local search.

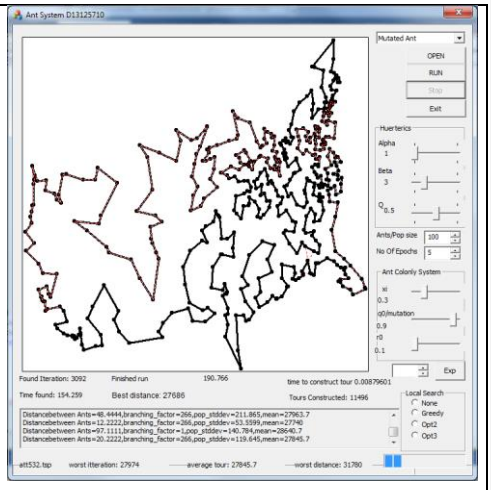
The Random Ant algorithm is also introduced and explored. In this, the ant uses random paths, so it is easy to implement. Its primary function is to explain the main shortcomings of local search algorithms.

In addition, the framework will solve the Travelling Salesperson Problem for Irish cities/towns using Ant Colony Optimisation.

Computational results for selected instances of the TSP library show that the Ant Colony Optimisation meta-heuristic gives results comparable to those of other methods.



TSP instance att532 from TSPLIB; it comprises 532 cities in the USA.



AntSolver framework using introduced Mutated Ant to solve att532 to a tour length of 27686, which in fact is an optimal tour, in 155 seconds

Figure 1.1 Comparison of TSP from TSPLIB with AntSolver framework

1.4 Research methodology

The experiments conducted on the implemented algorithms will be based upon quantitative evaluation. In the first instance, the problem to be solved is discussed, and in this case, the dataset for the framework will implement the Travelling Salesperson Problems from TSPLIB (Reinelt, 1991) These problems are well researched and solutions are clearly defined, making it possible to compare the results obtained from the framework with other researcher's results. The fact that reliable peer-researched results are available to test this framework makes it easy to compare with other meta-heuristics.

Secondly, solution constructions will be used to benchmark the implemented algorithms. This is a *de facto* standard used by many researchers. Solution constructions are obtained after a fixed number of steps (as proposed since the Second International Conference on Evolutionary Optimisation) as explained by (Stutzle and Hoos, 1997, p. 312) Informally, this suggests the number of complete solutions that

each algorithm constructs[2]. The number of complete solutions is limited to 10,000. Each solution in this case has a complexity $O(n^2)$ and $O(m.n^2)$ is the total complexity of the algorithms. Where n = number of vertices and m the population size. Empirical analysis is also used to determine the time taken to construct a complete solution, and the time taken to find the best solution so far. When evaluating, each test will execute 10 times and calculate the algorithm's best solution, average solution and average best solution, the average time for the best solution and the maximum time allowed (Li *et al*., 2011) (Stützle and Hoos, 2000).

The high performance profile counter will be utilised to accurately evaluate each of these measurements and each algorithm will be spawned in its own thread therefore minimising the effects of the operating system processing other messages. In fact the error is within 15 milliseconds and, as our problems requires a few minutes to solve, 15ms is an acceptable error ("Analyzing Application Performance by Using Profiling Tools," n.d.).

1.5 Resources

For completion of the artefacts implemented in the course of this dissertation the following resources are required.

- A desktop computer for writing all artefacts
- Microsoft Visual C 10 professional
- An account with GitHub, for a depository of the source code
- Access to the DIT library, for easy access to various academic resources
- Microsoft Excel to help with the analysis of each algorithm.
- Simple *SimpleXlsxWriter* ("SimpleXlsxWriter," n.d.) an open source to enable writing directly to an Excel file helping to automate the lengthy and time consuming experiments.

[2] a solution in the genetic algorithm (GA) and ACO as applied to TSP, is a complete round tour of the ant or chromosome.

1.6 Scope and limitations

Since swarm intelligence, computational complexity can all be complex theoretical subjects. There are some state-of-the-art paradigms applied, especially to computational complexity. They use complex mathematics, partitioning and advanced data structures to solve NP-hard problems. For the interested reader, the Concorde library houses one such example ("Concorde Home," n.d.) and (Applegate et al., 1998) Therefore these Branch and bound advanced algorithms will not be covered in any detail.

In addition, Chapter 8 assumes that the reader understands object orientated programming languages and all the paradigms used. Therefore this dissertation will not discuss Multi-threading, STL, MFC, design patterns, security aspects and best practices of the C++ library, refactoring, Test-driven development, requirement analysis, development cycles, user testing, acceptance testing, etc.

As the framework is split into two separate modules (that is a graphical user interface (GUI) implementation written in C++ using the MFC library, and the algorithms themselves written in C++ 10 that can be compiled under any C++ 10 compiler), the design and implementation of the GUI will not be discussed, as this falls outside the limits of this dissertation. However, for the interested reader a good start is Petzold (1998) although quite old but still a good reference.

In addition, proving the generic algorithm and ant colony optimisation mathematical convergence proof, is beyond the scope of this dissertation. Further information on convergence of ACO can be found in Huang *et al*. (2009).

Finally, this dissertation assumes that the reader has a good understanding of graph theory.

1.7 Organisation of the dissertation

Chapter 2 presents computational complexity and explains how computer scientists describe the complexity of problems and show the most common types of complex problems and how they are applied to the real world.

Chapter 3 presents the self-organisation principles of social insects and explains the how the ants find the shortest path.

Chapter 4 presents the Ant Colony algorithm and concludes with an example of the Travelling Salesperson problem to demonstrate how the cooperative behaviour of ants can be used in algorithms of combinatorial optimisation.

Chapter 5 discusses methods for improving ant algorithms, including how to apply the Ant algorithms to the Travelling Salesperson Problem, and introduces the Mutated Ant. Different strategies for updating the pheromone matrix are presented, as well as measures to determine algorithm stagnation in order to improve the solutions constructed by the ants.

Chapter 6 presents the Genetic Algorithm and various aspects of the algorithm such as mutation, crossover and selection.

Chapter 7 presents local search methods that can be used to improve both the ant algorithms and the Genetic Algorithm.

Chapter 8 documents all implementation artefacts that were created in the course of this dissertation. There is a focus on the AntSolver and all the classes that were designed in C++.

Chapter 9 presents a basic users' guide for the presented application.

Chapter 10 presents the computational results that were obtained by applying experiments to the implementation taken from the popular (TSPLIB) benchmark libraries.

Finally, Chapter 11 concludes and describes future work.

2 COMPUTATIONAL COMPLEXITY

2.1 Introduction

A Problem is defined as *P* if it can be solved in polynomial time (meaning quickly) on a deterministic[3] Turing machine. A problem is defined *NP* "nondeterministic polynomial time" if it can be solved in polynomial time using a nondeterministic[4] Turing machine. Therefore the solution to a given P or NP problem can be verified in polynomial time on a deterministic or non-deterministic machine respectively.

When optimisation problems are difficult to solve and cannot be solved in polynomial time they are said to be NP-Complete. The theory of NP-Complete classifies problems based upon difficulty. NP-hard is defined as, given the worst possible input ***n*** 'the worst case', the effort to find a solution increases exponentially. Informally a problem is defined as NP-hard if the only way to solve is by permutations of all possible answers and checking each one.

2.2 O-notation

Algorithms are designed to work with an input ***n*** and usually the efficiency is defined as a function of ***n, f(n)*** in either time complexity, space complexity or both. Efficiency analysis is mostly interested in the average case, the expected amount of resources an algorithm takes on typical input ***n***, and in the worst case, the amount of resources an algorithm would use based upon the worst possible input ***n***. Most algorithms are well researched, to the point that accurate mathematical models are available to define the average case and the worst possible case. The worst-case efficiency is defined as computational complexity, i.e., we can say given an input ***n*** the efficiency of the algorithm is proportional to ***n***. It does not matter what kind of computer the algorithm is executed on whether it is a supercomputer or laptop. The performance is proportional to ***n*** this is called ***O-notation***. Defined as:

[3] Deterministic Turing machine, a standard machine

[4] Non-Deterministic a Turing machine with infinite parallelism.

A function ***g(n)*** *is said to be* ***O(f(n))*** *if there exist Constants* c_o *and* n_o *such that* ***g(n)*** *is less than* $c_o f(n)$ *for all* $n > n_o$.

Most algorithms can be sensitive, based on the input ***n*** and the average case can vary in real-world implementations and the worst case scenario may never appear (Sim and Sun, 2003b) (Sedgewick, 1998).

2.3 NP-hard

Informally, NP-hard is mathematical jargon meaning that no known algorithm is guaranteed to solve all instances to optimality within a reasonable time. NP-hard means that the time required in solving an algorithm increases very quickly with the problem size. As Sedgewick (1998) states *"An exponential time algorithm cannot be guaranteed to work for all problems of size of 200 (say) are greater because no one can wait for an algorithm to take* 2^{200} *steps to complete, regardless of the speed of the computer exponentially growth the dwarfs technology changes".*

2.4 Travelling Salesperson Problem

The Travelling Salesperson Problem (TSP) is the most famous kind of NP-hard problem (Cai, 2008a). A salesperson intends to visit a number of cities, visiting a city only once, and returning to the start point minimising the distance they have to travel. In more formal terms, TSP explained by Donald, (2011, p. 1) *"Given n is the number of cities to be visited, the total number of possible routes covering all cities can be given as a set of feasible solutions of the TSP and is given as (n-1)!/2"*. It is not known if TSP belongs in ***P***, there is no polynomial time solution but there is also no proof that such a solution doesn't exist (Karp, 1977).

The Travelling Salesperson Problem (TSP) was first formulated as a mathematical problem in 1920s. Some people credit Karl Menger with popularising the problem to the mathematical community (StreetChicago, n.d.). The Travelling Salesperson Problem is one of the most studied problems in combinational optimisation and is often used for testing optimisation algorithms. TSP can be applied to many fields from transport to genetics ("TSP Gallery," n.d.). TSP has several applications: *"planning,*

logistics, and the manufacture of microchips, scheduling of service calls at cable firms, the delivery of meals to home bound persons, the scheduling of stacker cranes in warehouses, the routing of trucks for parcel post pickup, and a host of others." (Geetha *et al*., (2009, p. 356). So solving the TSP problem is of academic interest and importance.

In the case of the Travelling Salesperson Problem, adding one more city means the calculation grows exponentially. As demonstrated in Table 1, a TSP for four cities to add an additional city would require five-fold more steps, or a 50-city TSP would require a thousand-fold more steps to add an extra 10 cities. It is clear that a brute force calculation becomes impossible as a number of cities increases, so this problem is said to be NP-hard (Karp, 1977) in that no algorithm exists that can find the solution quickly, and an *"exhaustive search"* brute-force algorithm comparing each node must be implemented to find a solution. If you want to use an *exhaustive search* to solve a 20-city TSP problem, then you'll need to generate 121,645,100,408,832,000 different permutations (Sedgewick, 1998). For symmetric TSP, the complexity is (n-1)!/2. and a symmetric TSP (n-1)! (Donald, 2011)

City Number	Number of steps to compute
1	1
1	1
4	2
5	6
10	*362880*
20	121645100408832000
50	10^{61} Approx.
60	10^{79} Approx.

Table 1TSP calculation rates

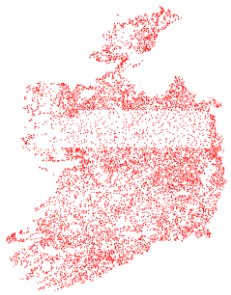


Figure 2.1 TSP 8,246 towns/cities in Ireland ("8,246 populated locations in Ireland," n.d.)

2.5 The Network Routing Problem

When data is sent over a computer network such as the Internet or wirelessly, information gets transmitted over nodes, the information in a sense hops (the cost) from the transmission node to the destination node via a number of nodes in between (the network). Each node in the chain has to decide to which node to route the data next (the protocol). A protocol like that of TCP/IP defines different methods that are used to exchange node information about the structure of the network at various nodes. Due to this nature the network topology is usually very dynamic, that is both the costs and the nodes can change dynamically. The goal is to find the shortest number of hops across the network from start to destination. If the costs are fixed, they can be easily solved with an exact algorithm. However, if the costs associated with transmitting from one node to another vary with time, (e.g. the node could be over a 3G network) the problem becomes NP-hard (Sim and Sun, 2003b).

2.6 The Dynamic Traveling Salesperson Problem

The dynamic Travelling Salesperson Problem is the same as a normal TSP, but the nodes (cities) of the problem can change when the algorithms are finding a solution (i.e. they change dynamically) such that in one iteration there could be 51 nodes, while in the second iteration there could be 47 nodes. The algorithms should be able to adapt to these changes are not get stuck in a local optimum from a previous iteration. The goal is also defined as the new shortest tour after each iteration. Dynamic TSP can change in a number of ways: nodes can be added or removed and the lengths of the

edges (vertices) can change. Problems arise with this when too many changes are added in an iteration and it is not possible to use a solution already found as a search space has been modified too much (Dorigo *et al*., 2006a).

2.7 The Multiple Rucksack Problems

The multiple rucksack can be described as follows: given a set of items, each with a weight and value, calculate the number of items that can fit into a rucksack so that the total weight is less than or equal to a limit and the total value is as large as possible. As stated by Sedgewick (1998)*"A thief robbing a safe finds it full with N types of items of variants size and value, but has only a small knapsack of capacity M to carry the goods, what goods should he carry to maximise the total value",* again this problem is easily solved if M is not large by exact methods (Sedgewick, 1998).

Comment [LP2]: You need a page reference

2.8 Other optimisation problems

Other optimisation problems include the vehicle routing problem, where a fleet of vehicles at an initial depot has to deliver goods to a certain number of customers such that the total travel time distance is minimised (Narasimha *et al*., 2012). Another example is the Job Scheduling problem, where N jobs of various sizes need to be scheduled to M employees. The job scheduling problem seeks to find the best schedule to complete the jobs and in most cases by the shortest time (Ku-Mahamud and Nasir, 2010).

2.9 Exact Methods

Many exact algorithms have been applied to NP-hard problems. Exact methods suffer from high computational costs but they do find the exact solution as they are proven to deliver optimal solutions theoretically. However, in practice, they usually run out of memory, or take an unacceptable time due to the enormous size of the search space. The high memory consumption is due to the fact that the algorithm needs to keep track of the partial solutions already generated (Martello *et al*., 2000),

2.10 Branch and bound algorithms

This algorithm is a general method that solves optimisation problems exactly while reducing the search space by utilising various so-called 'pruning' techniques. The branch part of the algorithm divides the set of all feasible solutions into smaller subsets; the branching is performed recursively on all the subsets. Resulting in a tree, this tree is then pruned. Starting at the root of the tree a lower and upper bound are calculated. A minimum of all upper bounds is stored in a dedicated data structure. If the algorithm approaches a vertex whose lower bound is greater than the dedicated data structures bounds, then all solutions in this branch can be discarded. A branch can also be discarded if the algorithm reaches a vertex having lower bounds equal to its upper bound as it already found the optimal solution within its branch. At the end, all vertices either have been pruned or their lower bounds equal the upper bounds, meaning the optimal solution has been found (Martello *et al*., 2000).

2.11 Concorde

In March 2005, the Travelling Salesperson Problem of visiting all 33,810 points in a circuit board was effectively solved by the Concorde TSP solver to a length of 65,048, 945 units. It was proven that no shorter tour exists. The computation took approximately 15.7 CPU years. In April 2006, an instance with 85,900 points was solved using Concorde TSP Solver, taking over 136 CPU years. Concorde TSP solver uses very advanced methods of branching and bounding and different heuristics to exactly calculate NP hard problems. However, the mathematics and data structures involved are beyond the scope of this thesis. There are over 700 functions to solve TSP problems inside the Concorde library. Concorde library can solve a thousand city problems in approximately three hours on an Intel Pentium III 600Mhz processor. However a problem of 10,000 cities, like usa13509 takes in four years on the same machine. Mona Lisa[5] is a problem created by Robert Bosch containing 100,000 vertices and is still an on-going TSP problem and the last, best solution was found in 2012

[5]see http://www.math.uwaterloo.ca/tsp/data/ml/monalisa.html for details

2.12 Conclusions

NP-hard problems are important for everyday real world problems, from the manufacture of microchips, planning of schedules, and from logistics to a whole range of others. Due to their exponential nature NP-hard problems can take an incredible amount of computational complexity to solve using exact methods and even the state of the art solvers can take years to solve 10,000 nodes. Clearly a better solution is needed and that is where approximation algorithms come in. These algorithms provide an optimisation approach to "best guess" solutions. These approximation algorithms trade optimality for efficiency although they have the advantage of finding good solutions quickly. However, the problem with optimisation algorithms is to be stuck in locally optimal loop discussed later (Dorigo and Socha, 2006a).

3 SWARM INTELLIGENCE

Swarm intelligence (Beni and Wang, 1993) can be defined as the collective behaviour of natural or artificial decentralised self-organising systems. Researchers study the behaviour of social insects, including ants, bees, and termites. The ability that swarm insects have to survive and solve complex tasks, while each insect cannot survive on its own is of particular interest to computer algorithm designers. Sim and Sun, (2003a, p. 560) state *"Ants are able to discover the shortest path to a food source and share this information with other ants"*. Swarm intelligence is of particular interest in the robotics domain, some notable research applications include Sensorfly, a swarm of autonomous helicopters that could be used to search for humans during disasters, the basic principle being that the swarm is better than one (Purohit *et al*., 2011). Swarm-based computer systems are highly fault-tolerant because the failure of one component in a swarm does not affect the overall system. This makes them particularly suitable for hazardous or remote environments, and the military is also interested in swarm research, developing a swarm of Spybot's known as MAST ("Micro Autonomous System Technologies (MAST)," n.d.). In swarm intelligence self-organisation is the key concept. Goss et al., state "*It is important to note that the selection of the shortest branch is not the result of individual ants comparing the different lengths of each branch, but is instead a collective and self-organizing process"* (Goss et al., 1989, p. 667)

3.1 Real Ants

In nature, ants live in a colony in which the number may vary from 30 to tens of millions to super colonies (Browne, n.d.). There is no one ant in charge and each individual ant has a particular function within the colony; even the Queen's function is only to lay eggs. All ants work for the success of the colony - they are a truly social insect. So successful are they that they have survived 100 million years, and in the process have colonised the earth (except for Antarctica and a few small islands). The efficiency of ants in finding and transporting food, overcoming obstacles, building anthills, and other operations is truly optimal. Success does not arise from leadership or the intelligence of a few ants, but emerges from the trial and error of many non-intelligent individuals. A single ant cannot survive outside of the colony for long;

however the ant colony as a system itself is extremely robust: a reduction of up to 40% of insects has practically no effect on the functioning of the whole society. A mass destruction of ants (for example, resulting from a chemical treatment of their habitat) leads to the consolidation of insects from the neighbouring anthills into one family to save the society (Goss *et al.*, 1989) (Hölldobler, 1990). Ants are said to be self-organisational as stated by Prabhakar et al., (2012, p. 2) *"Social insect colonies operate without any central control. Their collective behavior arises from local interactions among individuals"*. This behaviour is known as emergent behaviour which in the context of computer science is defined as the complex and unexpected outcomes arising from the interaction of simple individual entities (Bonabeau, 1999).

3.2 Stigmergy

When searching for food, ants initially explore areas around their colony randomly. Most ants communicate with one another via a chemical substance called a pheromone which other ants can smell. As an ant moves around, it deposits a constant amount of this pheromone that other ants can follow. Each ant moves in a random fashion. As soon as a particular ant finds food it carries some food on its back and returns to its colony (nest) releasing stronger concentrations of the pheromone in proportion to the quality and quantity of the food source found. When another ant smells this pheromone the other ant simply decides to follow the previous path depending on the strength of the pheromone. If it chooses to do so it will reinforce the existing trail, so that the next ant that stumbles upon this trail will almost certainly follow it. Therefore, the more ants that travel along the path, the more attractive the path will become for the next ants. This is known as stigmergy and was discovered by French entomologist Pierre-Paul Grassé within ant colonies in 1959 (Bonabeau, 1999) (Goss *et al.*, 1989). With time, these pheromones evaporate, which allows the ants to adapt to their environment. Therefore an ant using the shortest path will return to the nest quicker therefore reinforcing the path more, over time more ants are able to complete the shorter route and more pheromone accumulates on the shorter path. This continuous random movement of each ant helps the colony find different routes. This also ensures that the ant can find its way around any obstacles that may become present in its path. As Clark (1997, p.186) defines stigmergy *''the use of environmental conditions as instigators of action and the overall ability of the group to perform problem-solving*

activity that exceeds the knowledge and the computational scope of each individual member'' this is a good analogy when applied to computer science

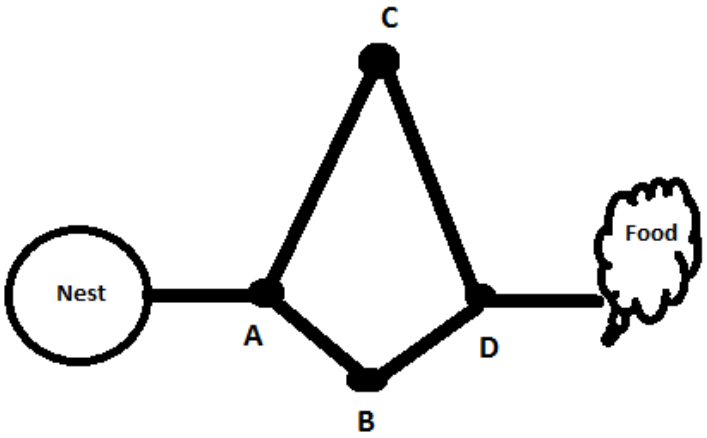


Figure 3.1Asymmetric bridge experiment

The asymmetric bridge in Figure 3.1 was an experiment carried out in a laboratory using a colony of Argentine ants that use pheromones to communicate. Proposed by (Goss *et al*., 1989) it demonstrates how the social operation of ants allows them to find the shortest path. A bridge *ABCD* was constructed. Point *A* is a gate, and is opened. The number of ants travelling along route *ABD* and the route *ACD* to the food and back to the nest was counted. At early stages of the experiment both routes were selected evenly by the ants. But after some time, the ants preferred the shorter route *ABD* as the quantity of pheromone on the shorter path is greater than that of the longer path.

3.3 Conclusions

Swarm intelligence can be defined as the collective behaviour of decentralised self-organising system that shows emergent behaviour. Ants are an example of how simple individuals can self-organise to show emergent behaviour. When travelling from their nests to a food source, an ant deposits a chemical known as a pheromone, the intensity of which depends upon the quality of the food source found. The ant then uses this pheromone trail to find its way back to the nest. Other ants in the colony can sense this pheromone and decide if they want to follow the trail or not, they in turn deposit

pheromone enforcing the trail and making it more attractive to the next ant. Over time, the colony of ants reinforces good trails and the bad trails evaporate. The ants never need to interact directly, as all this information is stored in the strength of the pheromone trail. This communication method is called stigmergy. These traits make swarm intelligence of great interest to computer algorithm designers.

4 ANT COLONY OPTIMISATION

4.1 Introduction

Marco Dorigo founded the field of Ant Colony Optimisation (ACO) in 1991 with his PhD thesis on positive feedback as search strategy (Dorigo *et al.*, 2006b). ACO is a section of swarm intelligence that introduces a meta-heuristic algorithm that uses artificial ants to find solutions to problems. Researchers study the behaviour of real ants in nature in which an individual ant is unable to hunt for food or survive by itself. As part of the collective, the ants together can solve complicated problems and can successfully operate their colony. The concepts of stigmergy and randomness are key elements in the implementation of ant colony optimisation algorithms. Ant colony optimisation are among the best algorithms available for combinational optimisation problems (Manjurul Islam *et al.*, 2006) (Dorigo and Gambardella, 1997a) (Dorigo *et al.*, 1996).

Year	Algorithm	Authors
1991	Ant System (AS)	(Dorigo *et al.*, 1996)
1992	Elitist ant system (EAS)	(Dorigo and Di Caro, 1999)
1996	Ant Colony System (ACS)	(Dorigo and Gambardella, 1997b)
1997	Rank-Based AS (RAS)	(Bullnheimer *et al.*, 1997a)
1998	Min Max AS (MMAS)	(Stützle and Hoos, 2000)
2000	Best Worst AS BWAS	(Cordón *et al.*, 2000)

Table 2 A list of ant algorithms discussed

4.2 From real ants to virtual ants

The ant system (AS) was the first algorithm of the ant colony optimisation (ACO) algorithms proposed by Marco Dorigo. It is inferior to all other ACO algorithms according to most research and Marco himself. As the first of its kind, AS was used simply to prove the concept and is the basis for all other ACO algorithms

Firstly all ant colony algorithms are iterative; at each iteration, a colony of virtual ants solves the particular problem at hand.

4.3 The Ants System meta-heuristic

All Ant Colony Optimisation (ACO) algorithms are to be executed on graphs, a graph representation of the particular problem be solved. This graph is called the construction graph and is created in a way that the ants can walk from the vertex ***i*** to vertex ***j*** over the edges $\boldsymbol{\eta_{ij}}$ to find a solution. The graph is usually a fully connected graph that has no edges to move to, to prevent ants from becoming stuck. Several computational complexity problems as discussed in Chapter 2 can be condensed to graphs and more are discussed later in this chapter "Other problems solved by Ants".

4.3.1 Ant System Solution construction

This step is part of the algorithm in which the ants create and construct their solution. To avoid having to calculate actual probabilities, the next partial solution of the complete solution can be chosen using roulette-wheel selection. Virtual ant ***k*** starts at a vertex ***i*** and chooses the next (state) vertex ***j*** to visit based upon the following probabilistic proportional $\boldsymbol{\rho_{ij}^{k}}$ rule shown in Equation 4.1 (below) to construct a set of states that combine to form a complete solution to the problem to be solved.

Equation 4.1 AS Solution construction rule

$$\boldsymbol{\rho_{ij}^{k}} = \frac{\left[\boldsymbol{\tau_{jl}}\right]^{\alpha}\left[\boldsymbol{\eta_{ij}}\right]^{\beta}}{\sum_{l\in N_i^k}\left[\boldsymbol{\tau_{ij}}\right]^{\alpha}\left[\boldsymbol{\eta_{ij}}\right]^{\beta}} \quad \textbf{if } j \in N_i^k,$$

Where:

$\boldsymbol{\rho_{ij}^{k}}$,- Probability

$\mathbf{N_i^k}$,- Set of all paths from vertex i to all adjacent vertices still not visited by ant k

ij,- Path from the vertex i to the vertex j

$\boldsymbol{\tau_{il}}$,- Amount of pheromone on the edge ij

$\boldsymbol{\eta_{ij}}$,- Amount of pheromone on the edge ij

***α*,-** Adjustable parameter

***β*,-** Adjustable parameter

The probability rule depends upon the combination of two values.

1) The attractiveness of the move η_{ij}^{β} , that is computed by a heuristic value usually the inverse of the distance between vertex ***i*** and ***j*** on the edge η_{ij} and β is heuristic parameter to control the influence of this attractiveness that is $\beta \geq 0$.

2) τ_{il}^{α} The pheromone level indicates how successful that particular move has been in the past and α an heuristic parameter to control the attractiveness of the pheromone level $\alpha \geq 0$.

α and β are adjustable parameters, if ($\alpha = 0$) nearest edge is chosen, like the nearest neighbour algorithm described in Chapter 7. However on the other hand if $(\beta = 0)$ only the pheromone trail is taken into account, which implies that the ants will select suboptimum routes. The selection of these parameters is very important so that the ants can find the best solution to the problem at hand. These parameters are discussed in detail in Chapter 5.

The probabilistic proportional rule is sometimes referred to the roulette wheel principle. Intervals are then placed end to end and a random point picked from this set of intervals. The wider the interval, the more likely that interval is to be hit and the corresponding vertex is chosen as a partial solution. Imagine a pie chart; each vertex of the graph has a certain percentage proportional to the probability. The next vertex is then chosen, like throwing a roulette ball and whenever the ball stops that is the next vertex in the solution. This is discussed further in Chapter 6 “Roulette Wheel Selection”.

4.3.2 Pheromone Update

When a complete solution is found by the virtual ant ***k,*** pheromones are deposited onto all the edges that the ant has visited; the amount the pheromone deposited depends upon the quality of the solution like real ants as defined by Equation 4.2.

Equation 4.2 AS pheromone update rule

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

Where:

τ_{ij}^{k}, - Pheromone deposited for ant ***k*** on edges ***i,j***

Q - Heuristic value where ***(Q > 0)*** usually defined as 1

L_k ,- Ant k complete solution quality.

The better the solution, the greater the amount of pheromone is deposited and the more attractive that particular solution will be for the next generation. This is different to real ants that will deposit a set amount of pheromones regardless of the move.

The pheromone then evaporates based upon the rules of the algorithm, which allows the virtual ants over time to forget previous moves so this area of the search space can be rediscovered (Dorigo *et al*., 2006).

Equation 4.3 AS evaporation rule

$$\tau_{ij} = (1 - \rho).\tau_{ij}$$

τ_{ij} Is the pheromone deposited on edges i,j

ρ is another heuristic that defines the evaporation rate $\rho \in [0, 1]$

4.4 Initial pheromone levels

In the initial ant system Dorigo *et al*. (1996), did not specify values for the initial pheromone levels $\tau 0$. However, they are assumed to be a small random value. In later research papers the initial pheromone $\tau 0$ trails are initialised with

Equation 4.4 Initial pheromones level rule

$$\tau_{ij} = \tau 0 = \frac{m}{c^{nn}}$$

Where m is the number of ants c^{nn} is the total cost of the solution applying the nearest neighbour algorithm (for more detail see Chapter 7).

4.5 Number of ants

As the total number of ants in the colony remains constant, in very large colony having a large number of ants leads to the pheromone matrix growing very quickly to sub-optimal solutions. In contrast, a very small number of ants may result in the breakdown of the pheromone matrix, based upon quick evaporation, and reduced updates of the pheromones. Normally the number of ants is equal to the number of vertices in the problem to be solved. Both Milanovic *et al*. (2013a) and Dorigo and Gambardella (1997a) state that these give good values. However, in experiments of the better performing algorithms discussed later, Dorigo *et al*. (2006b), noted for solving small problems, a small number of vertices (i.e. under 500), the number of ants did not matter very much. In fact a smaller number of ants (between two and 10) gave good solutions and performance. However, in bigger problems they noted that a large population of ants gives better solutions in terms of quality and performance, and the authors go on to state "*the worst computational results were always obtained when using only one ant and the second worst results when using two ants".* (Dorigo and Stützle, 2004a, p. 97)

4.6 Heuristic information

Ant Colony Optimisation (ACO), in common with other meta-heuristics based on natural systems, is quite sensitive to the choice of global parameters alpha, beta and so on as shown in Equation 4.1. Although there has been quite a bit of research on ACO parameters, the general consensus is that one must experiment a bit with free parameters to get the best combination of performance and solution quality see Table 4 for known optimal values of heuristics researched (Birattari *et al*., 2007) (Lissovoi and Witt, n.d.) (Dorigo *et al*., 2006a).

	α	β	e	m	p	T0
AS	1	5	n/a	n	0.5	m/C^{nn}
EAS	1	2	5	n	0.5	$(E+M)/pC^{nn}$
BWAS	1	2	n/a	n	0.5	$0.5r(r-1)/pC^{nn}$
MMAS	1	5	n/a	n	0.5	$1/pC^{nn}$
ACS	1	5	n/a	10	0.05	$1/nC^{nn}$

Here n is the number of cities in the TSP problem file.
C^{nn} is the nearest neighbour distance.
Elitist ant system parameter e=5
MMAS, τ_{min} and τ_{max}.
ant colony system , proportional random choice rule q0 = 0.9

Table 3 "suggested" parameter settings for heuristics

4.7 Exploration and exploitation

Meta-heuristics algorithms have to achieve the appropriate balance between the exploitation of the search space gathered by the algorithm and the solutions not yet discovered, as the ant colony algorithms update a pheromone matrix due to the probabilistic proportional rule. The problem is that this matrix becomes biased in that some solutions eventually get a probability of zero and others a probability of one and then search becomes stagnated. The improved ant system algorithms discussed in Chapter 5 (Parsons, 2005) (Song *et al.*, 2006) (Pour *et al.*, 2008) try to address this problem.

4.8 Travelling Salesperson Problem

TSP was the first problem solved by ant colony optimisation (Dorigo and Gambardella, 1997a). Ants are well suited to finding the shortest path, as discussed in Chapter 2, and can generate good solutions to NP-hard problems. To recap the Travelling Salesperson Problem is defined by Donald, (2011, p. 1) "*Given n is the number of cities to be visited, the total number of possible routes covering all cities can be given as a set of feasible solutions of the TSP"* (Karp, 1977) (Lissovoi and Witt, 2013).

TSP also has the advantage that it is used as a traditional benchmarking tool for combinatorial optimisation algorithms and this makes it easy to compare algorithms with each other (Dorigo *et al*., 2006a).

4.8.1 Solution construction

The TSP problem can easily be adapted to ants as the movement of the travelling salesperson is similar to that of ants. In more formal terms the construction graph is identical to the problem graph. Virtual ant agents are considered to be autonomous in the same sense as an independent travelling salesperson is, in solving his own problem while being unable to see the overall problem; in the travelling Salesperson the constraint is that he can only visit a city once (Dorigo *et al*., 2006a).

When these equations are transcribed to artificial ants the double bridge experiment described in Chapter 3 can be modelled as a graph. The artificial ants travel from one location (vertex) to another searching for the overall solution. At the end of each tour the artificial ants then deposit pheromones at each location edge proportional to the locations (vertices) quality. When the ant chooses the next location to visit it considers the amount of pheromone present on the edge and uses an heuristic to choose which vertex to travel to next (Fejzagic and Oputic, 2013).

4.8.2 Example 4.1

Imagine an ant's fictitious tour around Kevin Street DIT campus, assuming the ant can travel in a straight line to all locations.

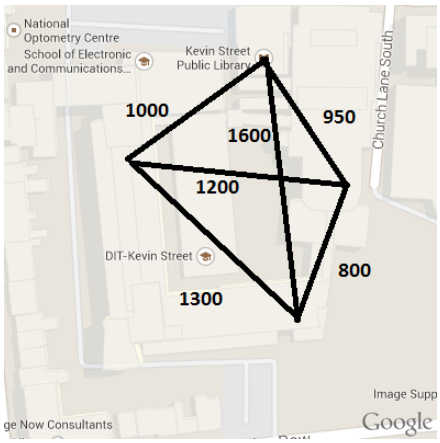


Figure 4.1 Locational distances – DIT Kevin Street

The ant wants to visit all locations, but just once. The distance between each location is as shown in Figure 4.2. It does not matter at which location the ant starts, as starting in

a random location will not affect the overall tour length. The ant wants to find the shortest path possible; this describes the travelling Salesperson problem. Firstly, there are no pheromones deposited on the edges and they have to be initialised with some initial value, so applying Equation 4.4 using the nearest neighbour algorithm and setting m = 5 ants, $\tau 0$ is defined as:

$$\boldsymbol{\tau 0 = \frac{m}{c^{nn}} = \frac{5\ (ants)}{800 + 950 + 1000 + 1300} = \frac{5}{4050}}$$

Next the ant ***k*** starts out in the library ***i*** and has to decide which vertex ***j*** to move to next, so by Equation 4.1

$$\boldsymbol{\rho_{ij}^{k} = \frac{[\tau_{il}]^{\alpha}\,[\eta_{il}]^{\beta}}{\sum_{l \in N_i^k}[\tau_{il}]^{\alpha}\,[\eta_{il}]^{\beta}}}$$

given $\alpha = 1$ and $\beta = 2$

$\eta_{\text{library}-\text{gym}} = \eta_{\text{gym}-\text{libary}} = 1/800$

$\eta_{\text{library}-\text{lecture1}} = \eta_{\text{lecture1}-\text{libary}} = 1/1000$

$\eta_{\text{library}-\text{lecture2}} = \eta_{\text{lecture2}-\text{libary}} = 1/1600$

$\sum_{l \in N_i^k}[\tau_{il}]^{\alpha}\,[\eta_{il}]^{\beta} = \frac{5}{4050} \cdot \frac{1}{800} + \frac{5}{4050} \cdot \frac{1}{1000} + \frac{5}{4050} \cdot \frac{1}{1600} = approx\ 0.000008$

$\rho_{libary-gym}^{k} = \frac{\frac{5}{4050} \cdot \frac{1}{800}}{0.00008} =$ 0.019 Approx.

$\rho_{libary-\text{lecture1}}^{k} = \frac{\frac{5}{4050} \cdot \frac{1}{1000}}{0.00008} =$ 0.015 Approx.

$\rho_{libary-\text{lecture2}}^{k} = \frac{\frac{5}{4050} \cdot \frac{1}{1600}}{0.00008} =$ 0.01 Approx.

Equation 4.5 Example Ant tour DIT Kevin Street

As can be seen $\rho_{libary-\text{lecture2}}^{k}$ has the lowest probability of getting selected as this path has the longest distance, as this is the first iteration and all pheromones have just been initialised, probability is only selected on the initial pheromones levels at this stage but this will change on the next iteration.

The probability decision is based upon the pheromone levels and an heuristic at the beginning of the algorithm. So the higher the heuristic value the more likely the ant will move to that location. This is important as if the heuristic value is low the overall optimal solution may never be found. Thus the ants will keep searching within their initial sub-optimal tour. At each location edge a pheromone matrix is updated, that is the library location will contain pheromones to all other points. It is the same, for all other points so the matrix size is $4^2 = 16$. It is important that the pheromones matrix be initialised by some value that is not too low or too high. If too low, this will cause the ants never to search other solutions, or if too high a couple of solutions will be required to have any real impact. All researchers of ant colony optimisation report that a good value of the average pheromone is the reciprocal of the distance of the nearest neighbour tour (Chapter 7).

Pseudo-code for the Ant System	
1	Create construction graph
2	Initialise pheromone values
3	**while** not stop-condition **do**
4	Construct ants solutions
5	Perform local search (optional)
6	Update pheromone values
7	**end while**

Table 4 Pseudo-code for the Ant System

After the ant finishes its tour the pheromone matrix is updated depending upon the algorithm implemented, and a certain amount of evaporation is applied. This allows the ants eventually to forget previous tours. Pseudo code for the ant system algorithm is shown in Table 3 and Figure 4.2 (Dorigo and Socha, 2006b).

Algorithm AS-TSP
initialize all edges to (small) initial pheromone level τ0;
place each ant on a randomly chosen city;
for t := 1 to t_max do
for k := 1 to m do
build a tour T(k,t) for ant k by applying the probabilistic transition rule;
end;
if (best T(k,t) better than current solution) update current solution to T(k,t);
for every edge (i,j) do
apply pheromone update;

```
end;
apply additional pheromone increment with Q/length(T(b,t));
end
end.
```

Figure 4.2 Ant colony system pseudo code as applied to TSP

4.9 Other problems solved by ant systems

By applying simple modifications to the various ant algorithms, (reducing the problem to a graph; finding the pheromone matrix initialisation and update; assigning heuristic rules for the solution construction (Zhang *et al*., 2014)). Ant Colony Optimisation can be adapted to solve other combinational problems such as hyper-spectral imaging with good results (Zhang *et al*., 2013); applied to the n-Queen problem (Khan *et al*., 2009); and the Vehicle Routing Problem (Narasimha *et al*., 2012). ACO, combined with neural networks, can be used to solve continuous optimisation problems, i.e. where the inputs constantly change (Pour *et al*., 2008). This continuous approach can also be applied to network routing as discussed by (Sim and Sun, 2003a). The Travelling Salesperson Problem can be applied to many fields from transport to genetics. Ant systems have also been applied to computer virus detection (Banerjee *et al.*, 2011).

A comprehensive set of problems is outlined in Chapter 5 of *Ant Colony Optimization* (Dorigo and Stützle, 2004a).

4.10 Conclusions

Ant colony optimisation (ACO) is a meta-heuristic algorithm based upon real ants, and is among the best algorithms available for combinational optimisation problems. The key to ant colony optimisation is a mix between a greedy search, and a learned search (see Chapter 7.) Setting α or β to 0 will show this respectively. But by using these two parameters together the real power of the algorithm can be shown. Suppose Equation 4.1 is just set with $\beta=1$, ignoring the pheromone trail altogether - the ants will just go for whatever vertex is the nearest (greedy search). While this can be a good method for small problems, it is typically not good on larger problems as the ants cannot look deeper into the search space. If the other hand if $\alpha=1$ and $\beta= 0$, (the ants choose the

next vertex to visit by the pheromone matrix alone ignoring the distance to the next vertex), the algorithm will stagnate and become stuck in sub-optimal solutions. This happens as the pheromone matrix is so high on some edges and virtually zero on other edges that the algorithm gets stuck in a positive feedback loop, and cannot break out to find other solutions. Fortunately, the exact values of α and β are not that critical and most researchers state values between one and five provide good results.

The roulette wheel selection is used to calculate the actual probabilities for the next vertex to be chosen in the solution. For each vertex destination a value is calculated that represents, as such, an interval on a pie chart and a random point is picked out. The wider this value is, the more likely the ants will choose this vertex. However, the lower the value the less likely it is that this vertex will be selected.

After the ant has completed its solution, it deposits a quantity of pheromone along each vertex edge that is inversely proportional to the value of the solution. In other words, the better the solution, the more pheromone is deposited and the more attractive that route will be. This is different from real ants as real ants deposit a constant amount of pheromones.

Evaporation is then applied, so that over time, the vertices that are barely used will see their pheromone levels drop toward zero, thereby making them less attractive for the future. On the other hand vertices that are heavily used will continue to have pheromone deposited, resulting in virtual ants forgetting previous solutions with time.

5 IMPROVING THE ANT ALGORITHMS

The ant system can solve many NP-hard problems of low dimensions (n<100) with good accuracy. It compares well with other approaches such as the genetic algorithm (Dorigo *et al*., 1996) (see Chapter 6). For problems of high dimensions (n>100), the simple ant algorithm as Dorigo *et al*. (1996 p. 2), state "*the Ant System can be outperformed by more specialized algorithms. This is a problem shared by other popular approaches like simulated annealing"* these problems can be summarised as follows:

- The global optimal can be lost due to the probabilistic roulette wheel ρ_{ij}^{k} , this happens when the length of the best and worst solutions have the same probability on this wheel, that is when the ant's solutions are almost at the optimal solutions. Informally if you imagine the roulette wheel as a pie chart as described earlier, best and worst solutions near the optimal have almost the same length and will, therefore, have the same probability of selection.
- The convergence of the algorithm near an optimal solution is low due to the best and worst solutions having equal power in updating the pheromone matrix. This happens, again, when the ants' solutions are almost at the optimal solutions, where the best worst and worst solutions near the optimal have almost the same length. Therefore, the pheromones are updated to the same extent, thus stopping the algorithm from finding the optimal solution.
- As the system traverses both the worst and best solutions, for larger dimensions of N, this increases the size of the search area. (As the ants are finding solutions from both the best and worst solutions)

The quality of the solutions produced by the ant system can greatly be improved by utilising different ways of handling pheromones and reducing the search space.
The only difference in the preceding improved algorithms and the ant system is the rate of evaporation, how they handle pheromones, and how the pheromones $\boldsymbol{\tau 0}$ is initialised, in the hope of having the effect of reducing the search space and guiding the ants to a more optimal solution. (Bullnheimer et al., 1997b) (Dorigo et al., 1996). (Stützle and Hoos, 2000) In this sense, training the ants to a more optimal solution as Hardygóra et al., (2004, p. 134) state *"While they differ mainly in specific aspects of*

the search control, all these ACO algorithms are based on a stronger exploitation of the search history to direct the ants' search process"

In addition, the algorithms can be greatly improved using local search methods discussed in Chapter 7 (Dorigo and Stützle, 2004b).

5.1 Elitist Ant Algorithm

The Elitist Ant System (EAS) was introduced by Dorigo, the idea behind which is to give extra pheromone levels to the best solution found so far $\boldsymbol{\tau^{bs}}$ - called the elitist ants. It is likely that some edges of the elite solution are part of the optimal solution; the aim is to guide the search space in succeeding iterations towards these vertices. The concept of elitism can also be found in genetic algorithms (see Chapter 6). The additional pheromone deposited by the elitist is weighted with a parameter w.

Equivalent to the ant system but in addition the elite ant deposits extra pheromones at each iterations as shown in Equation 5-1.

Equation 5.1 EAS pheromone update rule

$$\boldsymbol{\tau_{ij}} \leftarrow \boldsymbol{\tau_{ij}} + \sum_{\boldsymbol{k=1}}^{\boldsymbol{k-m}} \Delta\, \boldsymbol{\tau_{ij}^{k}} + \boldsymbol{w}\Delta\, \boldsymbol{\tau_{ij}^{bs}}\,, \qquad \forall(\boldsymbol{i},\boldsymbol{j}) \in \boldsymbol{\tau}$$

Where;

$\boldsymbol{\tau_{ij}^{bs}} = \frac{\boldsymbol{1}}{\boldsymbol{c^{bs}}}$ $\boldsymbol{c^{bs}}$ Is the best solution, $\boldsymbol{\tau^{bs}}$ is the best solution to date,

$\boldsymbol{w}$ -is an heuristic parameter $\boldsymbol{(\, w \geq 1)}$

Dorigo reported that an appropriate value for $\boldsymbol{w}$ is the number of vertices in the problem being solved. This improvement allows the Elitist Ant System to find better solutions in fewer iterations than in the Ant System. (Dorigo and Gambardella, 1997a) (Cordon *et al*., 2002)

However, allowing the best ants to update pheromones has the problem of stagnating too early, sometimes making finding the optimal tour impossible Stützle and Hoos (2000 p. 7) state *"if too many elitist ants are used, the search concentrates early*

around suboptimal solutions leading to a premature stagnation of the search. Search stagnation as the situation where all ants follow the same path and construct the same solution over and over again, such that better solutions cannot be found anymore."

5.2 Rank-Based Ant System

The rank-based ant system introduced by Bullnheimer *et al*., (1997b) is a modification on the elitist ant system. The best so far solution and the iteration best and best ***(w – 1)*** tour may also update pheromones. Consequently, the amount of pheromone deposited is proportional to the rank of the ant. This additional pheromone update is multiplied by parameter ***(w –r)*** where ***r*** is the ant's rank. Hence for the third best solution in an iteration, pheromones will be multiplied by ***(w - 3)*** the second best ***(w – 2)***, and so on. If ***(w –r)*** is less than zero, then ***(w –r) = 0***.

Equation 5.2 RBAS update rule

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w-1} (w - r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{bs}, \qquad \forall (i,j) \in \tau$$

Where:

$\tau_{ij}^{r} = \frac{1}{L^{r}}$ Lr = the cost of the ants r^{th} solution, w is the additional multiplier.

Yet again, the best results are at $w \geq 1$ the member $w\Delta\tau_{ij}^{bs}$ is similar to EAS algorithm.

Equation 5.3 RBAS evaporation rule

$$\tau_{ij} = (1 - \rho).\tau_{ij} + \sum_{k=1}^{M} \Delta\tau_{ij}^{k}, \qquad \forall (i,j) \in L,$$

Bullnheimer suggests that ***w*** is usually set to 25% of the number of ants. The results presented by Bullnheimer *et al*. (1997b) suggest that rank-based ant system perform slightly better than EAS and significantly better than Ant System.
(Bullnheimer *et al*., 1997b) (Dorigo and Socha, 2006b)

5.3 Best-Worst Ant Algorithm

Introduced by solution (Cordón *et al*., 2000) the Best-Worst Ant System (BWAS) uses mutations from evolutionary computation (Chapter 6) but employing the pheromones

update strategy from the elitist ant system and exploiting the restart and resets of pheromones strategies from MinMax (described in more detail at 5.4 below). The algorithm also implements a form of negative and positive feedback, as shown in Equation 4.3 from the ant colony system.

Equation 5.4 BWAS pheromone update rule

$$\boldsymbol{\tau_{ij} \leftarrow (1 - \rho).\tau_{ij} + \rho\Delta\tau_{ij}^{bs}}$$

$\boldsymbol{\tau_{ij}^{bs} = \frac{1}{C^{bs}}}$ If the path $\mathbf{ij \in T^{bs}}$, $\mathbf{T_{bs}}$ is the best to date round tour.

$\boldsymbol{C^{bs}}$ - Length of the solution.

In addition, the edges from the solution of the worst ant for the current iteration that are not in the best to date solution get evaporated at a greater influence.

Equation 5.5 BWAS evaporation rule

$$\boldsymbol{\tau_{rs} \leftarrow \tau_{rs} . (1 - \rho_w)}$$

Here ρ_w is additional evaporation all $C_{rs} \in T_w \; and \; C_{rs} \notin T_W \cap T_{bs}$, T_W is the worst solution for the given iteration, and $T_{bs}$the best to date.

5.3.1 Example 5.1

Informally, this evaluates differences in solutions of the best to date and the worst of the iteration so if $T_{bs} = \{1,2,3,4,5\}$ and $T_W = \{1,2,4,3,5\}$ then the edges $\{3,5\}$ in T_W are subject to further evaporation.

The Best-Worst Ant System also utilises a novel re-initialisation strategy that is similar to the MinMax ant system. When the algorithm conveys too much, that is the difference between the best to date solution and the worst of iteration solution becomes less than a threshold value the pheromone matrix is reset to τ0 (Zhang *et al*., 2011). In addition to these changes, best worst ant system also uses mutations. There exists a probability for an edge to be mutated by the following formula.

Equation 5.6 BWAS mutation rule

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

Where:

a- a uniform random number between *[0-1].*

k is the current iteration.

τ_{thres}- is the average value of the edges in the best so far solution.

(Cordón et al., 2002)

Equation 5.7 Mutation threshold rule

$$\tau_{thres} = \frac{\sum(i.j) \in T_{bs}\, \tau_{ij}}{|T_{bs}|}$$

The main drawback with this algorithm is there is a lot of heuristics, and either overstating or understating any of parameters will result in an inefficient algorithm (Dorigo and Socha, 2006b). It has been shown that the best worst ant system is best at solving quadratic equations (Cordón *et al*., 2002).

5.4 MinMax Ant System

Introduced by (Stützle and Hoos, 2000) the MinMax Ant System is similar to the Elitist Ants System in that only the best to date ants are allowed to update pheromones. Minimum and maximum pheromone limits are introduced to limit the quantity of pheromone on the solutions between vertices, τ_{min} and τ_{max}. The limits enforced on the pheromones τ_{min}, τ_{max} make it possible for the algorithm to change the search space thus avoiding stagnation as Stützle and Hoos (1997) state *"The trail limits alleviate the problem associated with the early stagnation of search especially for long runs, thus leading to a higher degree of exploration"* (Stutzle and Hoos, 1997 p. 310) Some researchers initialise τ(0) pheromones to τ_{max} (Dorigo and Socha, 2006b). However there are variants in the selection of which ants are allowed to update pheromones: the best to date ant, or the best for current iteration. MinMax also introduces a pheromone smoothing technique such that when stagnation is detected using the branching factor (see Branching Factor equation 5.18 below). τ_{max} and τ_{min}

are reinitialised, according to a proportional rule. The advantage of this system is that it enables the ants to re-explore the search space see

Equation 5.8 MMAS proportional rule

$$\textbf{increase} \sim \tau_{max} - \tau_{ij}^{bs}$$

So, evaporation for (MMAS) is

Equation 5.9 MMAS evaporation rule

$$\tau_{ij} \leftarrow \max((1-\rho)).\tau_{ij}, \tau_{min}),$$

and updating the pheromones:

Equation 5.10 MMAS update rule

$$\tau_{ij} \leftarrow \min(\tau_{ij} + \Delta\tau_{ij}^{bs}, \tau_{max}),$$

$\tau_{ij}^{bs} = \frac{1}{C^{sel}}$ if the path $ij \in T^{sel}$, T^{sel} is the selected ants round tour, C^{sel} is the length of the selected solution.

The MinMax ant system is one of the best performing ant colony optimisation algorithms confirmed by many researchers (Stützle and Hoos, 2000) (Dorigo and Socha, 2006b).

5.5 Ant Colony System

Ant Colony System (ACS) was introduced by Dorigo and Gambardella (1997a). This algorithm is significantly different to the others. Whereas with previous algorithms pheromones are updated when the ants finished their complete solution, in ACS pheromones are updated when the ant chooses the next vertex - in a sense, eating the pheromones as it goes along. Therefore the construction step of the algorithm is substantially different to the other algorithms. In addition, ACS updates the edges of the best so far solution T^{bs}at the end of each iteration. This has the advantage that the complexity of the pheromone update is reduced from $O(n^2)$ to $O(n)$.

So, evaporation in ACS is defined as

Equation 5.11 ACS evaporation

$$\tau_{ij} \leftarrow \max((1-\rho)).\tau_{ij},\ \tau_{min}), \quad \forall(i,j) \in T^{bs}$$

And the pheromone update for each ant

Equation 5.12 ACS update rule

$$\tau_{ij} \leftarrow \min(\tau_{ij} + \Delta\tau_{ij}^{bs},\ \tau_{max}),$$

$\tau_{ij}^{bs} = \frac{1}{C^{sel}}$ if the path $ij \in T^{sel}$, T^{sel} is the selected ant's solution C^{sel} is the length of the selected solution.

Also, the selection mechanism of the vertex to choose next is altered in ACS; the ants will choose a more travelled route rather than using the roulette wheel selection. Ensuring that the ant's path is more exploitation than exploration, this new selection mechanism is defined by the following equation.

Equation 5.13 ACS solution construct rule

$$j = arg \quad max_{l \epsilon \tau_i^k} \{\tau_{il} . \eta_{il}^{\beta}\} \text{ if } q \leq q_0$$

i- is the current vertex.

τ_i^k is a set of vertices adjacent to the current vertex i, at iteration k.

τ_{il} - is amount of pheromones on the partial solution C_{il}.

η_{il}^{β} is $\frac{1}{d_{il}}$ and d_{il} is the distance between vertex il.

Equation 5.13 is selected by a probability of q falling in-between q_o else the original roulette wheel selection of the ant system is used Equation 4.1. q is defined as a uniform distributed random number between $(0 \leq q \leq 1)$ and q_0 is a parameter between zero and one $q_0 \in [0, 1]$

(Dorigo and Gambardella, 1997a).

5.6 Mutated Ant System

The Mutated Ant system (MA) is an algorithm that uses various parts from all other algorithms, including the genetic algorithm, to produce a very effective and fast operating algorithm for all values of *n.* In this algorithm there are three ants, the ant,

the mutated ant and the elite ant. The ant gets mutated by a mutation factor *mf* to create the mutated ant, based upon the genetic algorithm. The solution construction is the same as in the Ant Colony System. Combined, then, is the elitist strategy from the Elitist Ant System that is pheromones are only updated with the best ant to date:

Equation 5.14 MA update rule

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{k-m} \Delta \tau_{ij}^{k} + w\Delta \tau_{ij}^{bs}, \qquad \forall (i,j) \in \tau$$

Where:

$\tau_{ij}^{bs} = \frac{1}{c^{bs}}$ c^{bs} Is the best solution, τ^{bs} is the best solution to date,

w -is an heuristic parameter $(w \geq 1)$

The edges of the solution of the worst ant for the current iteration get evaporated

Equation 5.15 MA evaporation rule

$$\tau_{ij} = (1 - \rho).\tau_{w}$$

τ_{ij} Is the pheromone deposited on edges i,j

ρ is an heuristic that defines the evaporation rate $\rho \in [0, 1]$

τ_w is the worst solution for the given iteration that is if the either the ant or the mutated ant

Like the other ant systems initial pheromone $\tau 0$ trails are initialised with

Equation 5.16 MA initialisation rule

$$\tau_{ij} = \tau 0 = \frac{m}{c^{nn}}$$

Where m is the number of ants, c^{nn} is the total cost of the solution applying the nearest neighbour algorithm (see Chapter 7)

From the MinMax Ant System, a pheromone smoothing technique is used such that when stagnation is detected, the pheromone matrix gets reset to $\tau 0$.

In addition, from the genetic algorithm and the Best Worst Ant System, a vertex is selected randomly for mutation in uniform probability. Then each of these vertices is

swapped, that is a vertex from the Ant solution is selected randomly and swapped with its counterpart. A method that reduces the frequency of the mutation, as time increases, is also added. As the algorithm runs, the nearer it gets to the optimal solution (perhaps only 1 or 2 vertices within optimal) so varying the mutation factor as time increases greatly increases the chances of finding the optimal solution. Fogarty mutation frequency (Ochoa *et al*., 2000) was chosen because it is easy to implement and provides good quality. Fogarty who "*found that varying the mutation rate over time and across the bit representation of individuals or both significantly improved the performance of the GA*". (Ochoa et al., 2000 p. 8)

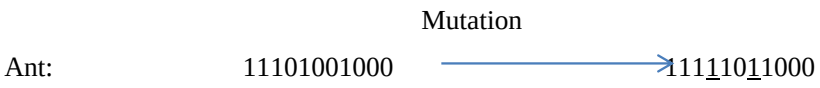


Figure 5.1 – Mutation example, as illustrated bit 4 and 6 are swapped

The main function of the operator is to inject diversity into the algorithm allowing for new solutions, thereby lowering the risk of being trapped in a local optimal. Defined by

Equation 5.17 MA mutation rule

$$\mathbf{c^{nm}} = \begin{cases} \mathbf{C_{ij}} + \boldsymbol{mut}(\mathbf{k}, \boldsymbol{\tau}_{\mathbf{thres}}), \end{cases}$$

The selection process is taken from the Ant Colony System as per Equation 5.13

5.7 Benchmarks

This section discusses benchmarks of each algorithm as researched by the various top researchers in the field. Most researchers agree that the best way to compare the performance of the algorithms is to measure construction of solutions. Researchers generally use notation; (KN 10,000) N is a number of nodes in the instance, in this benchmark.

	α	β	e	m	p	T0
AS	1	5	n/a	n	0.5	m/C^{nn}
EAS	1	2	n	n	0.5	$(E+M)pC^{nn}$
ASrank	1	2	6	n	0.5	$0.5r(r-1)/pC^{nn}$
MMAS	1	5	n/a	n	0.5	$1/pC^{nn}$
ACS	1	5	n/a	10	0.05	$1/nC^{nn}$

Here n is the number of cities in the TSP problem file; C^{nn} is the nearest neighbour distance.

Table 5 Parameter settings used for the benchmark in Table 5.2

TSP Problem	optimal	MMAS	ACS	ASRANK	EAS	AS
kroA100	21282	21320.3	21420.0	21746.0	***20152.8***	22471.4
eil50	426	***427.6***	428.1	434.5	428.3	437.3
d198	15780	***15972.5***	16054	16199.1	16205	16702.1
ry48p	14422	14553.2	14565.4	***14511.4***	14685.2	15296.4
ft70	38673	***39040.2***	39099.0	39410.1	39261.8	39596.3
krol24p	36230	***36773.5***	36857.0	36973.5	37510.2	38733.1
Ftv170	2755	2828.8	***2826.5***	2854.2	2952.4	3154.5

For each instance the average solution quality is shown, best results are shown in the italic bold. Each algorithm used parameters as shown in table 5.1. All use the same number of solution constructs (10,000).

Table 6 Benchmarks on various AS algorithms

The benchmarks shown in Table 5.2 have been taken from Stützle and Hoos (2000) but also compare well to other researcher's results, namely Bullnheimer *et al*. (1997b) Parsons (2005) *Coleman et al.* (2004) Cordón *et al*. (2002). However, in this table the ant colony system was executed in Dorigo and Gambardella (1997a) and some suggest that the ant colony system performs better. After carefully examining the two papers, it was unclear where the actual results for the ant colony system were coming from, as in Dorigo and Gambardella (1997a) it only has a benchmark for eil50, without local search. However, it may be assumed that both the MinMax Ant System and the Ant Colony System are the best performing ant colony algorithms (Lissovoi and Witt, 2013)

Looking at these results it can be seen that Min Max ant system and Ant Colony system generally have the best performance; this also corresponds to findings from most other researchers. Other statistics that are not shown in this table are that the average solution for MinMax Ant System and the Ant Colony System was better than the best solution for the ant system, and, as Stützle and Hoos (2000) state *"Regarding the performance of AS it can be clearly seen that AS performs very poorly compared to the other algorithms"*

5.8 Stagnation Measures

When the pheromones on the ant trails become too unbalanced (some edges have large quantities of pheromones, while others have none, making the search space biased) all ants will follow more or less the same path/to more or less the same tours. In order for the algorithm to detect this case, Dorigo and Stützle proposed countermeasures.

5.9 Variation Coefficient

The Variation Coefficient in probability theory is a ratio of standard deviation to the mean. Applied to Ant colony optimisation it is the standard deviation of the constructed pheromones and their average pheromones. The disadvantage of this method is that some ants can generate totally different solutions but the variation coefficient can be the same. However it is efficient and easy to implement.

5.10 $\overline{\lambda}$ Branching Factor

The Branching Factor evaluates the pheromone matrix looking at every node and determines the value that fulfils Equation 5.18. The $\overline{\lambda}$ Branching Factor is defined as the average of all branching factors, informally representing the average number of routes that the ant has a high probability of selecting.

Equation 5.18 $\overline{\lambda}$ Branching Factor

$$\tau_{ij} \geq \tau_{min}^{i} + \lambda\left(\tau_{max}^{i} - \tau_{min}^{i}\right)$$

τ_{min}^{i} Is the minimum value of pheromones and τ_{max}^{i} is the maximum. Dorigo and Stützle use λ with a value of 0.05.

5.11 Pheromone reset

Pheromone reset is utilised mainly by the Best Worst Ant System, the MinMax Ant System and Mutated Ant. As discussed earlier, if the ant gets stuck in a local loop, pheromones can be reset to the initial start values: $\tau(0)$. Doing this enables the search to be restarted. However, previous history is not destroyed. The process is as follows:

firstly the system is reset; if the best tour to date is greater than the best to date tour after reset has been initialised, the 'best to date' tour is changed to the restart best. But the other algorithms could benefit from this procedure as it has been proven to be effective against stagnation (Dorigo and Socha, 2006b).

5.12 Parallel implementation

All virtual ants are autonomous, so each virtual ant constructs its tour individually without needing help from other virtual ants. This is the same for all ant algorithms. This makes parallelisation of the algorithms simple; most implementations for parallelisation can be categorised as having either a couple of ants executing on a single processor, or having separate colonies executing on different processors. In the latter case, there can be a large communication overhead communicating between the separate colonies. However, it should be noted that the only information that the ants need to share is the pheromone matrix. But imagine a 64-city TSP instance that corresponds to a 64^2 with each ant in each colony trying to update the matrix. This will result in a considerable amount of communication. Any locks on the matrix (e.g. mutexs or semaphores) will result in significant instances of locking, requiring the ants to wait, thereby slowing the algorithms. So researchers searched for a faster way to speed up communications.

One such solution (Chen *et al*., 2008) suggested that limited information should be exchanged amongst the colonies. The main principle was to divide the ants between equally-sized sub-colonies, with each sub-colony operating on a different processor searching for an optimal local solution. This experiment showed that is better to exchange between processors only the best solution found, and not to update the complete pheromone matrix. The authors demonstrate that this approach worked well, hence improving overall efficiency.

5.12 Convergence

As with most meta-heuristics, ACO experiments guide the research efforts, but due to the randomness of the algorithms, it can become difficult to prove convergence (i.e.

will the algorithm find the optimal solution? If so how long will it take?) The Ant Colony and MinMax ant systems have been proven eventually to converge at the optimal solution, but there is no way of defining the time it will take. More information can be found in Chapter 4 of (Dorigo and Stützle, 2004a) and in (Huang et al., 2009)

5.13 Conclusion

The extensions of the basic ant system introduced in this chapter try to address the shortcomings of the original algorithm the Ant System (AS). The elitist ant system deposits an extra pheromone based upon the best solution so far. This has the effect of guiding the ants to a more trodden path thus producing a better quality solution, however this also has its limitations in that every ant in the colony, including the worst ant get to update the pheromone matrix.

The rank-based ant system improves upon this by giving extra weight to the best so far solution, the second best and so on, thus minimising the level of pheromones updated by the worst solutions. Therefore in the neighbourhood of the optimal solution (when the best solution and worst solution are almost the same) the ranking ant system significantly guides the ants quicker to the optimal solution.
The Best-Worst Ant System is similar to the rank-based ant system, but tries to break out of local optima by using the operation of mutation, as in the genetic algorithm, to try and extend the search spaces. In addition, when approaching stagnation (when the best and worst solutions differ all only by fewer edges) the differences between the edges is subject to further evaporation.

The algorithms described above still suffer from the probabilistic proportional rule update strategy that effectively gives the probability of less trodden paths that may be part of the optimal solution a near zero chance of selection.

The MinMax Ant System addresses this by imposing restrictions on the pheromone matrix and re-initialising the matrix if stagnation is detected. This enables the ants to re-search the search space without losing any of the previously learnt information, a very effective method. Research shows that the MinMax ant system almost always

finds the optimal solution. The $\overline{\lambda}$ Branching Factor and the Variation coefficient can detect when an algorithm has stagnated - a strategy that, in a sense, enables the MinMax ant algorithm to switch between exploitation and exploratory stages during its execution.

The Ant Colony System uses a different probabilistic rule altogether, the ant only selects the next edge based on the probabilistic rule very rarely and chooses a more greedy solution most of the time, controlled by q_0. The adapted probabilistic wheel selection enables the algorithm to adapt and introduce new search spaces so it remains always in an exploitation and exploratory stage. It also uses the elitist strategy update but with a twist: the ant updates pheromones when it hops from edge to edge, thus implementing a powerful algorithm. However, because of the new probabilistic wheel selection and the elitist strategy update, the algorithm can get stuck in a local optimal, that can take quite a while to break from (it if ever manages to do so), as the best so far edges may never be on the global optimal solution.

The Mutated Ant system introduced uses only two ants, combined with the elitist updating strategy and the selection strategy from ACS. The pheromone matrix is updated by the 'best ant' in the iteration and the 'best ant so far'. Evaporation is then controlled by the 'worst ant' in the iteration. However, this can lead to the problem of the algorithm getting stuck at the local optimal much more quickly than in the other algorithms. This can be turned into an advantage, by combining the probabilistic proportional rule from ACS and mutation from the genetic algorithm to inject diversity into the solution thus enabling the algorithm to break and explore the search space. In addition if stagnation is detected, Mutated Ant implements the MMAS reset strategy to reinitialise the pheromone matrix, allowing the algorithm to jump into a new search space. The Mutated Ant combines the best strategies from various algorithms and with only two ants for which to construct solutions, produces a very efficient and effective algorithm for all values of n.

Ant algorithms can be greatly enhanced by applying parallelism, as each ant is considered a simple autonomous agent and therefore can be easily adapted to parallelism. But the drawback is that the pheromone matrix has to be accessed by each

individual agent thus creating a bottleneck. However, there are a few different methods described earlier, on how to best approach this problem. Successful parallelisation can greatly improve the efficiency of all the Ant algorithms significantly.

6 THE GENETIC ALGORITHM

The Genetic Algorithm, like Ant Colony Optimisation (ACO), also belongs to the family of Meta-heuristics, that is, they solve the conflict between speed and accuracy. Such an algorithm tries to find improved solutions to a given optimisation problem, generally NP-hard, helped by the guidance of an underlying problem-specific heuristic. Meta-heuristics generally run in iterations and, in each new iteration, more solutions are generated using knowledge from the previous iteration (Chaiyaratana and Zalzala, 1997).

6.1 Introduction

The genetic algorithm (GA) was developed by John Holmes, his colleagues and students of the University of Michigan in the 1970s, inspired by Darwin's theory of evolution. There has been a lot of research since then. However despite the GA age, these algorithms are still actively studied, widening the number of problems that can be solved using these algorithms. Based on Darwin's principle of evolution selection, crossover and mutation, the GA utilises these to traverse large search spaces. GA algorithms are widely used in a wide range of real-world problems like ACO (Chaiyaratana and Zalzala, 1997):

1. resource allocation, shop scheduling, timetable planning, and TSP
2. network routing satellite orbit selection
3. machine learning

The algorithm operates on a population of individuals called chromosomes, containing a representation of potential solutions to the problem that is being analysed. Chromosomes are represented by a fixed length data array (strings, integers, etc.). Each data bit in the array is called a gene. Initially, the population is initialised with a set of candidate solutions (the population) either with random chromosomes or by a heuristic, and the fitness is calculated. At each iteration, a new generation (population) is created via selection and crossover from the old population. This produces better solutions, as better solutions are more likely to be combined with each other to produce even better solutions. The algorithm then manipulates a percentage of the

new population's chromosomes via mutation enabling exploration of the search space (Geetha *et al*., 2009).

Steps are as follows:

1. Initialisation - a population of random solutions (chromosomes) is created. At this point it is very important that the population is diverse as possible (randomness);
2. Fitness - each chromosome is rated on how well the chromosome solves the given problem;
3. Selection - the fittest chromosomes are selected to create new generation;
4. Crossover and mutation - a percentage of high fitness chromosomes from the new generation are selected to undergo crossover and mutation, with the new modified chromosomes inserted back into the population.
5. Repeat from step 2 until terminated.

6.2 Selection

Not all chromosomes make it into the next population; only a select few chromosomes are selected. The main idea is to select fitter individuals for the next generation, thus improving the algorithm. The choice of selection operators can greatly influence the solutions generated by the algorithm. There are many different forms of selection operators, most common are Roulette Wheel Selection, tournament selection and elitism (Srinivas and Patnaik, 1994).

6.2.1 Roulette Wheel Selection

A selection technique for choosing a chromosome, proportional to its fitness, as defined by: $\rho_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$, where ρ_i is the probability of choosing chromosome i, f_i is the fitness of chromosome i, over the sum of all chromosome fitness f_j. As Buckland states, "*Imagine that the population's total fitness score is represented by a pie chart or roulette wheel. Now you assign a slice of the wheel to each member of the population. The size of the slice is proportional to that chromosome's fitness score: the fitter a member is, the bigger the slice of pie it gets. Now, to choose a chromosome, all*

you have to do is spin the wheel, toss in the ball and grab the chromosome that the ball stops on." (Srinivas and Patnaik, 1994).

6.2.2 Elitism

In elitism, the complete population of chromosomes is sorted by its fitness. The probability that an individual is selected is inversely proportional to its position in the sorted list; the individual chromosome at the head of the list is more likely to be selected (Srinivas and Patnaik, 1994).

6.2.3 Tournament selection

This is a selection technique that chooses a chromosome via a tournament. Individual chromosomes are selected from the population randomly (uniform probability), and the winning chromosome (the best fitness) is selected. Tournament size can be adjusted (selection pressure) to allow a greater probability of only the fitter chromosomes being selected. This method has several benefits, it is easy to implement and can be run in parallel. However, in large populations, the tournament size has to be equally large to ensure that the higher fitness chromosomes have greater chance of selection (Srinivas and Patnaik, 1994).

6.3 Genetic operators

6.3.1 Crossover

There are many different methods of crossover, the simplest being One Point Crossover. In this method a random point (gene) of two parent's chromosomes is selected and these genes are exchanged with their consecutive counterparts. To create two new children chromosomes as illustrated in Table 6.1. The procedure for creating crossover is as follows creating crossover is as follows:

1. Select two parent chromosomes (selection)
2. Select random positions of one or more genes to create crossover points
3. Copy the genes outside of the crossover point of parents one's chromosome into child one.
4. Copy the genes outside of the crossover point of parent two's chromosome into child one.

5. New child is created
6. Repeat the same process with parent two, creating child two

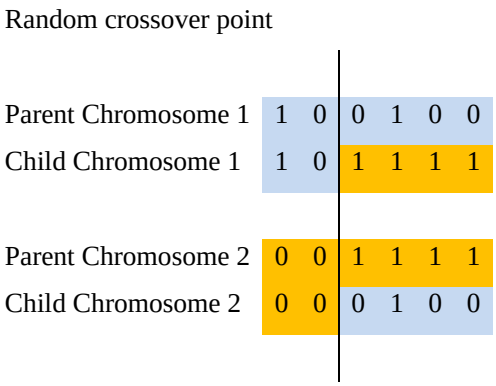


Table 7 One Point Crossover Method

6.3.2 Multi point crossover

This method is similar to one point crossover except two or more random points are selected parents (Genes) as illustrated in Table 6.2:

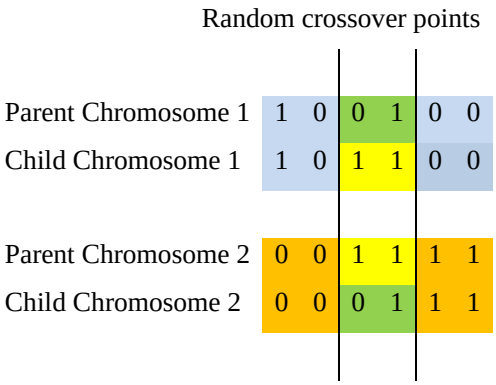


Table 8 Multi Point Crossover Method

These two variants of crossover are the most popular, but there are many different variants of this crossover operation that depend on the problem being analysed. Later a special form of crossover to solve the problem of the travelling Salesperson will be analysed (Chapter 9) (Srinivas and Patnaik, 1994).

6.3.3 Mutation

Here the crossover and mutation operators are discussed as they are the core of the genetic algorithm. However, there are many different operators (Ochoa *et al*., 2000). A certain proportion μ (a low value usually less than 1%) of newly selected chromosomes created by the selection process outlined earlier is selected randomly for mutation in uniform probability. Then each of these chromosomes is mutated, a gene from the chromosome is selected randomly and swapped with its counterpart. As illustrated:

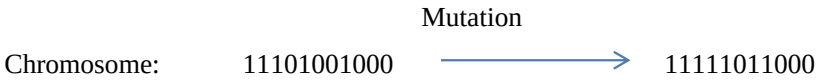


Figure 6.1 Chromosome mutation

The main function of the operators is to inject diversity into the new population allowing the genetic algorithm for new solutions lowering the risk of being trapped in a local optimal.

6.4 Limitations

As the Genetic Algorithm can converge to find an optimum solution it can also converge to find a bad solution. Bad solutions can be caused by many factors - premature convergence, poor parameter settings or simply down to luck with randomness. The methods of crossover and mutation try to address these problems. However, as most problems analysed by the generic algorithms are NP-complete we can properly never know the exact answer to the problem being solved.

On the other hand, generic algorithms can be great in solving problems when it is not possible to compute all possible solutions or we do not know the ways to reasonably solve the problem space, as with the travelling Salesperson problem.

6.5 Conclusions

TO DO

7 LOCAL SEARCH

7.1 Introduction

A local search begins with an initial solution and tries to improve this solution by local changes in the neighbourhood, defining the possible solutions in which the local search algorithm can move. If an improving solution is found, the current solution is replaced with the improved solution, this is called local optimum The most common local search heuristics are 2-Opt and 3-Opt, but a potential problem arises as the algorithm could stop at a very poor local optimum. There are also meta-heuristic approaches to local search, such as Tabu search (Johnson and McGeoch, 1997) (Dorigo and Stützle, 2004b, p. 93).

Local search has been applied to a large number of hard computational problems in computer science. Local search is implemented by changing certain positions on the vertices of the problem being analysed or in the case of the genetic algorithm the genes to find a more local solution.

7.2 Local search applied to meta-heuristic

Once an initial solution is found to the problem space, top researchers on meta-heuristics tell us that the best approach to obtaining a high quality solution is to combine with a local search. All meta-heuristic algorithms defined earlier can be extended with local search methods; this can greatly improve the solutions constructed by the algorithms, even guiding the algorithm to the optimal solution more efficiently. As Stützle and Hoos (2000) states *"The ants' solutions are not guaranteed to be optimal with respect to local changes and hence may be further improved using local search methods. Based on this observation, the best performing ACO algorithms for many NP-hard static combinatorial problems are in fact hybrid algorithms combining probabilistic"*. (Stützle and Hoos, 2000)

Comment [LP3]: Page ref

7.3 Nearest neighbour

The nearest neighbour algorithm is a very simple algorithm that looks at one vertex on a graph and finds which vertex is the nearest, then moves to this vertex and repeats the process, until all vertices are visited. However, due to the greedy nature of this algorithm, it can sometimes miss better solutions (Beyer *et al*., 1999). The starting position is usually selected randomly as outlined in the table below.

1.Select a random city.
2.Find the nearest unvisited city and go there.
3.Are there any unvisitied cities left? If yes, repeat step 2.
4.Return to the first city.

Table 9 Nearest Neighbour process (Dorigo and Stützle, 2004b, p. 101)

This algorithm generally keeps its tour within 20% of the optimal solution (Johnson and McGeoch, 1997).

7.4 Greedy heuristic

The greedy heuristic constructs a solution by repeatedly selecting the short edge, if there is no circular cycle; the shortest edge is added to the overall solution.

1. Sort all edges
2. Select the shortest edge and added to the tour if it is not already there.
3. Do we have *n* edges in the final solution? If no repeat step two

Table 10 Greedy Heuristic process

This algorithm normally keeps solutions within 10 to 20% of the optimal solution (Johnson and McGeoch, 1997).

7.5 2-opt

2-opt is a simple local search algorithm proposed by Croes in 1958 and is as follows: given a set of edges from the initial solution, delete any two edges, thus breaking the tour into two paths; then reconnect those paths in the other possible way so that the route is still valid (Johnson and McGeoch, 1997). In the case of the Travelling Salesperson Problem (TSP), the reinsertion is done only if the new route is shorter.

This is continued until no further optimisation is valid. The final solution called 2 optimal (2-opt) as shown:

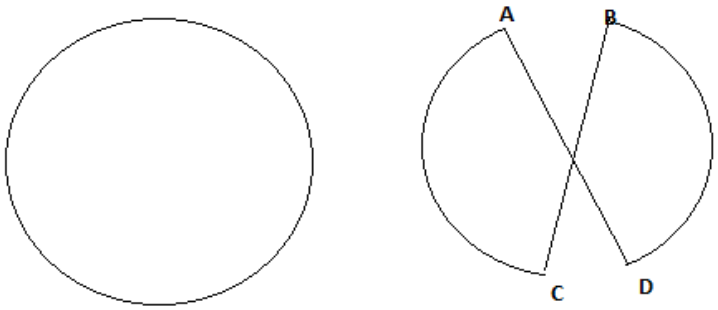


Figure 7.1 2-Opt

7.6 3-Opt

The principle of 3-opt is the same as 2-opt except instead of two edges being removed, three are removed until there are no three possible moves left (Johnson and McGeoch, 1997)(Johnson, 1990)

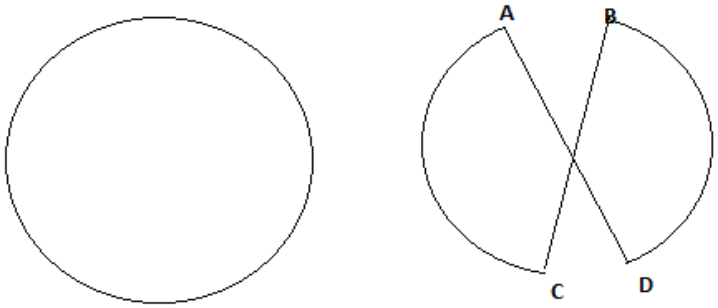


Figure 7.2 3-Opt

7.7 Tabu search

Tabu search, proposed by Glover (1989), uses both 2-opt and 3-opt searches together and the best solution is selected. However, the algorithm maintains a certain number of previous moves and adds them to the so-called tabu-list. Therefore a previous move will not be repeated, thus preventing the algorithm from moving in a circular pattern.

7.8 Lin-Kernighan (LK) heuristic

Lin-Kernighan created an algorithm that can get within 2% of the local optimal, briefly it is an extended version of 2-Opt and 3-Opt that can swap many vertices, that is the heuristic is adaptive and decides how many vertices need to be swapped at each stage as stated by Stützle and Hoos (2000) *"The best performing local search algorithm with respect to solution quality for symmetric TSPs is the LK heuristic which considers a variable number of arcs to be exchanged. Yet, the LK heuristic is much more difficult to implement than 2-opt or 3-opt, and careful fine-tuning is necessary to get it to run very fast and produce high quality solutions"* (Stützle and Hoos, 2000). In AntSolver we use the Lin-Kernighan heuristic as developed by Stützle (Stützle and Hoos, 2000) and which has a complexity of approximately $O(n^{2.2})$ (Donald, 2011)(Lin and Kernighan, 1973)

Comment [LP4]: Add page ref

7.9 Local and Global Optimum

The global optimal is the best possible solution within a search area, whereas the local optimum is a solution to the problem within a set of candidate solutions. It is very easy for an optimisation algorithm to get stuck in the search area of a local optimal. The optimisation algorithm starts in its initial configuration and repeatedly moves to an improving configuration. A search space is created, but if this search space is not in the global optimal neighbourhood the algorithm will get stuck in the local optimum. One remedy for this problem is called basins of traction; the search space is divided into subparts, each consisting of the initial configuration that led to local optimal. The search is then restarted from these configurations. For further information see the paper "How easy is local search?" (Johnson *et al*., 1988) (Mladenović and Hansen, 1997).

7.10 Local search Applied to Ant Colony Optimisation (ACO) and Genetic Algorithm (GA)

Local search meta-heuristics suffer from the problem of converging at local minima if used by themselves. However the solutions for a local search in this context are generated via meta-heuristic algorithms. Informally local searches are bad at creating initial solutions and the quality of the final solution depends upon the initial solution. "*The task of algorithms like ACO algorithms is to guide the search towards regions of the search space containing high quality solutions and, possibly, the global optimum. Clearly, the notion of search space region is tightly coupled to the notion of distance, denoted by an appropriate distance measure, between solutions*" (Stützle and Dorigo, 1999). Ant colony optimisation and the genetic algorithm are excellent at creating initial solutions; combined these with local search can greatly enhance their performance (Stützle and Hoos, 1999). In the case of the algorithms described in Chapters 5 and 6 there are different methods of applying a local search - it can be applied to all solutions in an iteration, or only the iteration best solution, or to the global best solution (the best so far solution). There is also the question of Lamarckian versus Darwinian updates. This can be described as follows: a set of solutions *Sp*, applied to a local search procedure becomes *Slp*. The question then is which solution (*Sp* or *Slp*) get introduced into the next iteration of the algorithm. This can be explained easily in the case of ACO there are two ways of updating the pheromones after a local search, either with *Sp* (Darwinian the original) or *Slp* (Lamarckian local search applied). There are arguments for and against each (Dorigo and Stützle, 2004b). Lamarck seems to have the better argument in that if there is a better solution then, as Diagro states *"it would be stupid to use the worst tour"*. The Darwinian view is that meta-heuristic algorithms with local search with Lamarckian updates just learn how to generate good solutions for the local search procedure. The research consensus is that Lamarckian outperforms Darwinian (Dorigo and Stützle, 2004b, p. 97).

Comment [LP5]: Page ref

Comment [LP6]: Do you need to name them?

Comment [LP7]: Check – is this correct?

Comment [LP8]: This sentence is a little confusing

Comment [LP9]: What is the page ref referring to?

7.11 Random Ant

This algorithm is used to demonstrate the limitations of the local search on the ant systems. It shows how a randomly generated solution to a problem can generate excellent results with no complexity (i.e. there is no pheromone matrix, evaporation or initialisation), but typically fails miserably when the problem size is large.

Random Ant works simply by generating a random solution to the problem being defined. That is, the ant just walks to each vertex randomly. This random solution is then brought to its local optimal via a local search, implemented using the Lin-Kernighan heuristic. Mutation is then applied to the local optimal solution, to try to prevent stagnation of the algorithm. Mutation uses the same equation as the mutated ant with the same frequency. The process is then repeated so that the mutated solution is then fed back into the local search on the next iteration.

However if mutation or local search are not used the algorithm is effectively useless, as the search space will converge on the local optima, and will suffer the same limitations as other local search methods.

Pseudo-code for random ant

Create Local neighbourhood
S0 <-Create Random Tour
while not stop-condition **do**
S2 <- Perform local search
S3 <- mutate S2
If s2 is better than S1
S1 = s2
If stagnation detected reset to s0
End while
end while

Code listing 7.1 Pseudo code for random ant

In addition, if stagnation is detected, the algorithm is reset to the first random solution.

```
Set stagnations = 0
Build nearest neighbour tour
For i:= 1 to max irritation

for t := 1 to no of city's do
        build a tour T(k,t) for ant place each ant k on a uniform randomly chosen city
end;

for t := 1 to mutation rate do
        Ran1 = Get Random number from 0 to max no of city's
        Mutate the ants tour
        T(k,t) = T(Ran1)
end;

Current solution = bring tour to local optimum T(k,t)
if (best T(k,t) better than current solution) update current solution to T(k,t);
else
        Increment stagnations
                End ifelse
        if (stagnations more than threshold)
                Reset T(k,t) to best T(k,t)
end;
end irritation
```

Figure 7.3 Random Ant pseudo code

7.12 Conclusion

This chapter presented local search as applied to ant colony optimisation and the genetic algorithm. All researchers agree that applying local search to the solutions generated by approximation algorithms generated better solutions. All researchers also agree that Lamarckian updates produce better solutions. Local searches such as the nearest neighbour algorithm and greedy searches can generate results within 20% of the optimal solution relatively easily, but suffer the problem of getting stuck in local optima. It has been outlined that initial solutions are generally generated randomly but when combined with an approximation algorithm can produce excellent results. There are various types of local search, and the AntSolver framework has implemented the nearest neighbour, greedy heuristic, 2-Opt, 3-Opt and converted Thomas Stützle/Lin Kernighan implementation to work within the framework. This chapter also introduced the random ant algorithm that basically generates random solutions and feeds them to the Lin Kernighan heuristic. In the case of ACO with local search, the use of heuristic values is not that important.

8 DESIGN AND ENGINEERING OF ANTSOLVER

8.1 Introduction

In this chapter the most important points of the implementation of AntSolver are introduced and described. AntSolver implements all the algorithms and local search described in Chapters 5, 6 and 7 (excluding Tabu search), and all artefacts have been created during the course of this dissertation. The artefacts are available under the GNU license for download and modification at https://github.com/d13125710

AntSolver is implemented in C++ and the main algorithm classes have been implemented using ISO standards, therefore enabling these classes to be imported to any other C++ application on any operating system.
Five distinct AntSolver modules are introduced:

1. The main algorithm classes, the entire logic of all the ant algorithms, and genetic algorithm have been encapsulated into these classes. All are accessed via an interface and created via a factory;
2. A Local search module with the methods of local search is discussed and can be applied to each solution of each algorithm (except the brute force algorithm);
3. The Problem module implements the Travelling Salesperson Problem to demonstrate the flexibility of these algorithms;
4. A GUI interface module was created to visualise the algorithms and a MS Windows 64bit and 32 bit GUI interface to the algorithms classes was built with MFC compiled using visual C++ 10;
5. A testing suite for automating the experiments is defined in Chapter 10.

In creating AntSolver the following code was used to provide access to MS Excel ("SimpleXlsxWriter," n.d.) to save time executing experiments. A Thomas Stützle/Lin Kernighan local search was also added. In addition, the simple Ant System algorithm written by Andrew Colin ("Dr. Dobb's | Good stuff for serious developers," n.d.) helped immensely in the understanding of the basic Ant system algorithm.

AntSolver was tested on a 64-bit Windows 7 machine, using the command prompt and the GUI TheWindows 7 machine is an AMD K7 running that 3000 MHz and 8 GB of RAM. Visual C++ 10 professional with service Pack 2 installed was used to compile all versions. There are no extra libraries required compiling main modules. However, the further experiment libraries used to execute the lengthy experiments (Chapter 10) outputs to an Excel file and this library was provided by ("SimpleXlsxWriter," n.d.).

8.2 Why C++?

C++ was selected for the building of the framework because the author is very confident with this language. Microsoft Visual C++ 10 professional will be the compiler used ("Visual C," 2014). AntSolver is designed using the methodology of well-researched design patterns and the Object Orientated paradigm to enforce the advanced software engineering part of the MSc being studied. Well-structured C++ also has the advantage of outperforming almost all other programming languages; infact only C and assembler can be faster[6].

8.3 C++ methodology

It must be said that C++ does come with its problems. It is considered to be an insecure language due to its complex nature and can cause memory, stack and buffer overflow problems. To minimise these problems, it was decided to use the standard template library (STL) container classes to manipulate matrixes and arrays. This approach can slow the system down due to the extra overhead of initialising these containers, however, these containers can provide safety against these problems somewhat, and can also be made 'thread safe' easily in parallel implementation.

In AntSolver it is preferred to create each object on the stack, rather than on the heap. This ensures that the compiler will handle memory management. To boost performance AntSolver almost always prefers passing by reference to passing by value

[6] For a quick benchmark see http://benchmarksgame.alioth.debian.org/u64/which-programs-are-fastest.php

or passing by pointer. Passing by reference ensures that the compiler at compile time knows exactly what kind of structure is being passed to a function so there is no need for complex pointer casting, thereby improving the quality of the software. In addition, to ensure any functions do not modify the contents of these structures, the C++ *const* keyword is used where appropriate. The standard template library allows containers to be dynamically allocated then removed, to speed up the process. Old containers are recycled so they don't have to be reinitialised. Exceptions, an important feature of the C++ standard, were excluded from AntSolver because the algorithm classes will only fire exceptions if the containers are overflowed or under flowed or accessed inappropriately. If this happened at any stage it would mean that there was a severe problem within the implementation (Meyers, 1995) (Louden and Lambert, 2011). For further information on the C++ language refer to (The C++ Programming Language 4th Edition [Opsylum], n.d.).

8.4 Design Patterns

Furthermore, in this chapter we show that the design of the AntSolver algorithms uses advanced patterns and methodologies, implemented with test driven development (TDD) (Beck, 2002) and re-factoring (Fowler *et al*., 1999). This approach is proven to be very effective and the end result is a highly modular AntSolver design, in that each class module is completely decoupled from each other and all classes can communicate with the same local search procedure and problem instance The use of software design patterns gives AntSolver an open ended architecture by its ability to execute different algorithms and different input problems at run-time, with no side effects on the empirical run-time analysis for each algorithm(Zhang et al., 2008).

8.5 Software quality

There are many design methodologies for estimating the quality of software of which design patterns is one. When implemented correctly within software, they can completely enhance the reusability, modularity, abstract, understandability and maintainability within source code. Design patterns guarantee reusable objects by

creating abstract objects avoiding dependencies on any part of the system and thus avoiding tight coupling between these objects (Gamma, 1995). *"…it is useful to design programs with design patterns even if the actual design problem is simpler than that solved by the design patterns"* (Prechelt *et al*., 2001). The factory pattern is a good example, whereby each object can be called the same way but the object construction depends on the run time of the system. New objects can be added that are completely decoupled from the client code. So, in AntSolver a factory can be used to implement different algorithms on the same system using the same client but executing a completely different algorithm that is detected at run-time (Gamma, 1995) (Huaxin and Shuai, 2011). Various researchers analyse software quality by the level of design patterns implemented in source code (Khaer *et al.*, 2007) (Zhang and Budgen, 2012). By using design patterns, the client can be completely decoupled from the actual implementation. This will also help in the future if any expansion is required (Milanovic *et al*., 2013b).

Comment [LP10]: Need page ref

8.6 Design and implementation

Figure 8.1 shows the basic class diagram of the algorithms; the basic client code of AntSolver works as shown in Code Listing 8.1. Firstly, a Parameter class is created; this defines which algorithm to create and all other algorithm parameters including local search methods. This then constructs an abstract interface to the algorithm via a static factory method, returning the particular algorithm required at run time. This interface is the same for all algorithms. This code implements the bridge pattern and factory pattern. Decoupling concrete objects from their implementation.

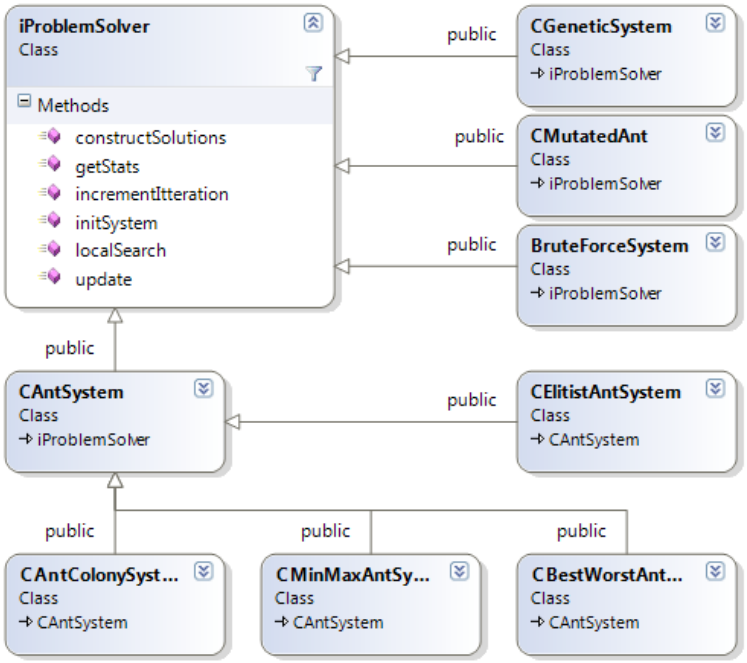


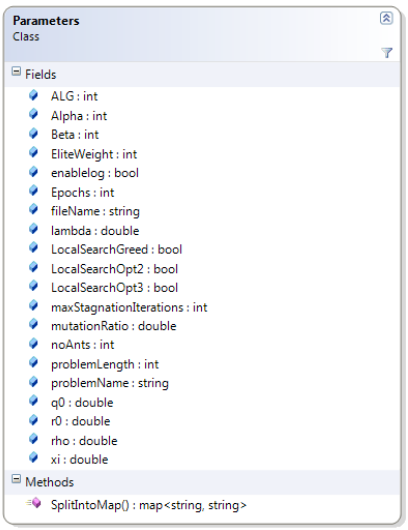
Figure 8.1 Class Diagram AntSolver

```
1 Parameter par;
2 iProblemSolver *pMM = TSPSolverFactory::create(par, &m_pmatrix_);
3 pMM->initSystem();
4 while(pMM->getTourNo() < par.Epochs){
5       pMM->constructSolutions();
6       pMM->localSearch();
7       pMM->update();
8       pMM->incrementItteration();
9       int dist= (int)pMM->getBestPathLengthSofar();
10      if(dist < d ){
11            d = dist;
12      }
13 }
14 delete pMM;
```

Code listing 8.1 AntSolver Client code

8.6 Parameters

A Parameter class was created so that the construction of each algorithm is uniform. *ALG* defines the algorithm required and can be the values as shown in Figure 8.2



ALG can be set to the following

MMAS = MinMax Ant Algorithm
ACS = Ant Colony Algorithm
AS= Ant Algorithm
GS = Genetic Algorithm
EAS = Elitist Ant Algorithm
BWAS = Best-Worst Ant Algorithm
MA = Mutated Ant Algorithm
RA = Random Ant
BF = Brute Force Algorithm

Figure 8.2 AntSolver Parameters Class Diagram

Other parameters (such as no. of ants, alpha, beta, mutation ratio etc.) can also be defined in this class. This class has default constructors that will instantiate all parameters to their researched optimal values, or where any options like local search can be turned on/off via *Parameters.LocalSearchOpt3 = true;*

8.7 Tour

A Tour class is populated at the end of each iteration storing all statistics about the particular algorithm executed at that particular point; this can be then sent to a log file or the GUI for processing. This saves us having to query each class for results, as all algorithms including the genetic system store the same statistics. It stores such details as the best solution so far, the average solution in the iteration the time taken to construct a solution, for a complete list please see the class diagram Figure 8.3.

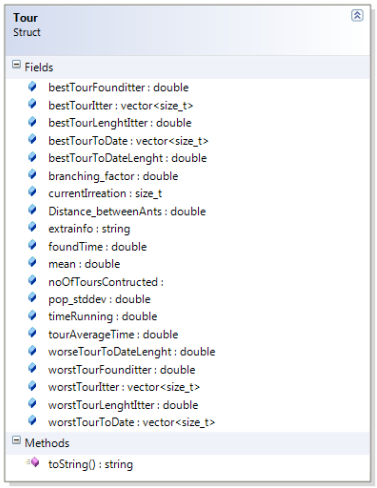


Figure 8.3 AntSolver Tour Class Diagram

8.8 Travelling Salesperson Problem Files

The dataset for AntSolver will implement the Travelling Salesperson Problems (TSP) from TSPLIB (Reinelt, 1991). As these problems are well researched and solutions defined, it is possible to compare the results obtained from AntSolver framework with other researchers' results. Therefore, given the advantage that reliable peer-researched results are available to test the framework, it is relatively easy to compare with other meta-heuristics such as particle swarm optimiser or genetic algorithm. In the words of Dorigo *et al*. (1996) *"In order to make the comparison with other heuristic approaches easier"*. TSPLIB data is presented in many formats including XML, and does not just deal with TSP problems but also many related problems such as the sequence ordering problem, vehicle routing problem etc.

Comment [LP11]: Cite including page ref

Examining a TSPLIB problem and solution file att48.tsp the contents are as follows

AME : att48 *COMMENT : 48 capitals of the US (Padberg/Rinaldi)* *TYPE : TSP* *DIMENSION : 48* *EDGE_WEIGHT_TYPE : ATT* *NODE_COORD_SECTION* *1 6734 1453* *2 2233 10* *3 5530 1424* *4 401 841* *5 3082 164…* *…* *48 401 841* *EOF*	*TYPE : TOUR* *DIMENSION : 48* *TOUR_SECTION* *1* *2* *242* *243* *...* *279* *3* *280* *-1*
TSP file format, att48.tsp	TSP solution format att48.tsp

Table 11 TSPLIB problem and solution file att48.tsp contents

The library starts numbering at 1; under NODE_COORD_SECTION, type TSP means Travelling Salesperson Problem; dimension is the number of nodes (48). Each node has three numbers - node number and (x,y). These are generally Cartesian coordinates, as the coordinates don't have to be an integer. Historically, computers were inefficient at evaluating 'floating point; numbers hence why this library is in integers. However, there are different coordinates in the library, the distances of these coordinates can be then calculated based upon there type which can be Manhattan, Maximum, Geographical, Pseudo-Euclidean, and Euclidean. See (Gerhard Reinelt, n.d.).
A tour can be started upon any node and the file is terminated with EOF string.

Comment [LP12]: This is not clear. Not sure if what I suggest is what you mean

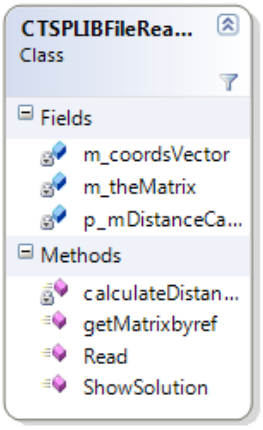
The solution files come in the following format, consisting of a list of tours or sub-tours with each optimal tour terminating with -1. For a particular problem there could be many optimal tours.

8.8.1 Implementing TSPLIB in C++

Now that we have defined how TSPLIB presents data and solutions, there we move to the implementation of TSPLIB in C++. A symmetrical Travelling Salesperson Problem instance is given as the coordinates of a number of *n* points. We can hold these points in a matrix and calculate them on the fly. However, this method would be inefficient and it is better to calculate them in one go and store the results in a matrix which we will call the distance matrix. We can assume that the data required is the

vertex and the distance to every other vertex and as TSP stores distances in integers it will produce a matrix of size n^2. Based on this data, Test driven development methods were created as shown in table 8.4 and then the code was implemented, creating *CTSPLIBFileReader* class. This class contains standard parsing of files and reading them into a matrix. It also has various functions to help the plotting of points in the GUI.

Comment [LP13]: Not sure is this right?



Client Code:

CTSPLIBFileReader fileReader("berlin52.tsp");

Bool success = fileReader.Read();

vector<vector<int>>&matrix=fileReader.getMatrixbyref();

Figure 8.4 AntSolver CTSPLibFileReader Class diagram

In this class one notable feature that has been implemented is a function factory to calculate the distances of the coordinates at runtime. As shown in the following class diagram (Figure 8.5), this virtual base class *IDistance* contains a virtual function *calculate*, and the corresponding concrete implementation is executed dynamically at run time, separating concerns of the matrix to the calculation of coordinates, providing loose coupling thereby improving our design.

Comment [LP14]: Check the sense of this sentence. It was unclear in its original format

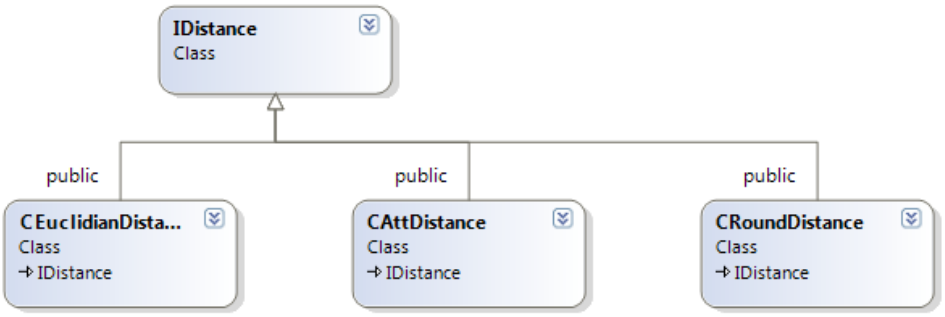


Figure 8.5 AntSolver IDistance Class diagram

Further expansion of this class is possible to enable reading point files. Any expansion can be added easily because of the modular design. Please see *CTSPLibFileReader.cpp* class in the artefacts for more details.

8.9 Memory Requirements

The main data structures in the ant algorithms are mainly the distance matrix the pheromone matrix and the heuristics matrix. All are of size n^2 and defined in this section. The genetic system and mutated ant have their own unique implementations, which are discussed later.

- Distance matrix: a symmetrical TSP instance is given as coordinates of a number of *n* points
- Pheromone matrix: at each of the vertices of the graph, pheromone level *[i,j]* is stored and is manipulated by the ants. This pheromone matrix controls updates, evaporation and all other aspect of the pheromone levels. This matrix was extracted into its own class.
- Heuristics matrix: when constructing a tour, the ant on city *i* chooses the next city *j* based on the probabilistic proportional rule. The ants in the same iteration use the very same values and need to be computed by each ant. However, computing this matrix after the pheromone updates significantly reduces the workload by each ant.

One other final detail to note is the fact that these matrices are symmetrical, as defined by the symmetrical TSP problems that are being analysed. That is if vertex x_i is

connected to a vertex x_j then *t[i][j]* is equal to *t[j][i]*. Therefore in symmetrical TSP we only need to *store n(n-1)/2.* However, this provides complex matrix access and is more beneficial use an n^2 matrix. However to expand to an asymmetrical matrix that is *t[i][j]* is not equal to *t[j][i],* this requires no code change in the allocation of the matrices, but only the updating as discussed in Chapter 2.

The ant is also required to store a list of cities it has visited, and the solution itself, defined in the ant class as a collection of Booleans, and a collection of cities (integers) each of size n. The number of ants in the colony is also one of the big data structures, storing an ant class *m* times where *m* is the size of the colony. The more advanced local search procedures include a neighbour matrix of *n*nn,* where nn is the number of nearest neighbours. Therefore the memory required for the AntSolver algorithms is low ($3.n^2 + m.2n$) and ($3.n^2 + m.2n + n.nn$) if local search is used where m is the number of ants, *n* is the number of nodes in the problem in the colony for the main algorithms. Some additional memory is required for the GUI and for housekeeping but these amounts are also very low.

8.10 The ant system classes

As the ant is defined as an agent that constructs solutions to problems autonomously, it knows very little about the overall problem therefore only needs to store an n+1[7] vector containing the cities the ant visits, and another list of the same size to state which city has been visited or eliminated from the search. In addition for ease of computation, the tour length is also stored. This is done by summing all the lengths in the solution edges. The class diagram is shown in Figure 8.6 and the file is called *CAnt.h* located in the artefacts.

Comment [LP15]: In the image, it is shown as CAnt (without .h)

[7] +1 to include the complete round trip

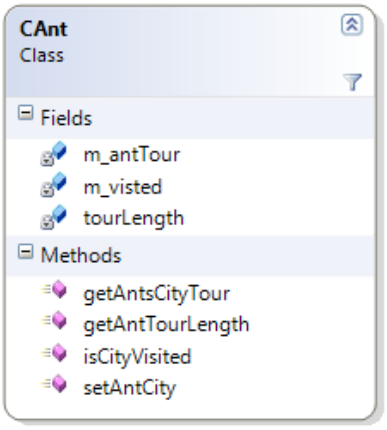


Figure 8.6 AntSolver Ant class diagram

The pheromone matrix class as shown in Figure 8.7 controls all operations applied to the matrix for all the ant algorithms, controlling updating and evaporation, calculating the Branching factor outlined in Chapter 5. In turn this class encapsulates an n^2 matrix that contains the pheromone levels over all edges. In Chapter 10 a visual representation of this matrix diagram XXXX is provided. See *PheroMatix.h* in the artefacts for further information.

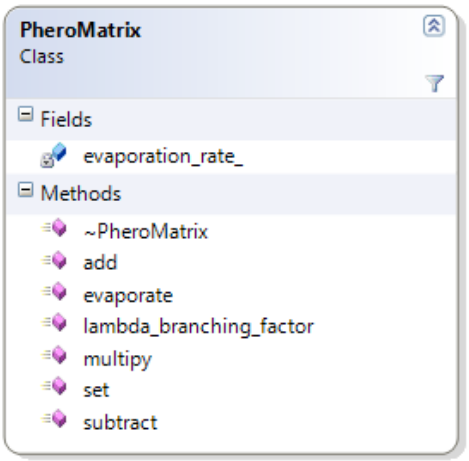


Figure 8.7 AntSolver Pheromone Matrix class diagram

```
1 void CAntSystem::constructSolutions()
2 {
3       m_tourTime.startTimer();
4       initAnts();
5       for (size_t k = 0; k < m_Ants.size(); k++ )
6       {
7               m_Ants[k].setAntCity(0,m_randomPath[k]);
8               m_Ants[k].setCityVisited(m_randomPath[k]);
9       }
10      normal_distribution<double> rndSelTrsh(0.0,1.0);
11      rndSelTrsh(g_rndTravel);
12      for(size_t step = 1 ; step < m_noNodes; step++)
13      {
14              for(size_t k = 0; k < m_Ants.size(); k++)
15                      decisionRule(k,step, rndSelTrsh);
16      }
17      //add in the last city to the first city tour ...
18      for(size_t k = 0; k < m_Ants.size(); k++)
19      {
20              size_t tourstart=m_Ants[k].getCity(0);
21              m_Ants[k].setAntCity(m_noNodes,tourstart);
22              m_Ants[k].setAntTourLength((int)
                        this->calculatePathLength2(m_Ants[k].getAntsCityTour()));
23      }
24      m_tourStats.tourAverageTime = (m_tourTime.getElapsedTime())/  (m_Ants.size()-1);
25      m_tourTime.stopTimer();
26      updateBestSoFarPath();
27 }
```

Code listing 8.2 AntSolver Ant System Solution Construction

The main part of all algorithms is the solution construction as shown in figure 15.

1. First each ant's memory is emptied. This is done in line 4;
2. Each ant in the colony has to be assigned an initial city; here this is accomplished by creating a random tour stored in *m_ramdomPath* and assigning the start city to this value. Lines 5-9;
3. Next the ant constructs a complete tour; at each step the ant uses the *desisionRule line 15* to get the next city to visit (based upon Equation 1 Chapter 4). Passed to this function is the step of the tour and the ant;
4. In lines 18-24, the ant's end city is set to its start city as to complete the full tour defined by the Travelling Salesperson Problem. The solution length is also calculated so as to save the program doing so later.
5. Line 26 tests if any of the solutions are better than the current best tour and populates the Tour class that maintains the current statistics.

8.10.1 Designing the Probabilistic proportional rule

The technique for an ant in city j moving to city i is defined by: $\rho_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$ where ρ_i is the probability of choosing city i, f_i is the distance of city *i*, over the sum of all city's distances f_j, as defined in detail in Chapter 4. In the code, *k* is the current ant and *step* the construction stage. Effectively here we place each city on a roulette wheel. Based on the value stored in the matrix, the better the value of the heuristic at the vertices, the better chance the ant has of moving to that city.

```
1 void CAntSystem::decisionRule(size_t k, size_t step)
2 {
3       size_t c = m_Ants[k].getCity(step-1);
4       for (size_t i=0; i < m_noNodes; i++)
5       {
6               t_prob[i] = 0.0;
7               m_strength[i] = 0;
8               if (m_Ants[k].isCityVisited(i) == false && c!=i)
9               {
10                      t_prob[i]= (double)(*m_heuristicMatrix)[c][i];
11              }
12      }
13      for (size_t z =0; z < m_noNodes; z++)
14              m_strength[z+1] = t_prob[z] + m_strength[z];
15
16      double x = fRand(0, 1.0) * m_strength[m_noNodes];
17      size_t y = 0;
18      while (!((m_strength[y] <= x) && (x <= m_strength[y+1])) )
19              y++;
20      m_Ants[k].setAntCity(step, y);
21      m_Ants[k].setCityVisited(y);
22 }
```

Code listing 8.3 AntSolver Ant System Decision rule

1) The current city of the ant k is found and placed into value c (line 3);
2) Each city *i* not visited by ant k is added to a slice of the proportional wheel stored in *t_prob[i]*, the size of the slice is proportional to the strength of Equation 1 Chapter 4 and is pre-calculated and stored in *m_ heuristicMatrix* as to avoid additional calculation (lines 4-12);
3) Next, the wheel is spun by the function *fRand* that creates a random number (line 16) in uniform probability; wherever this number ends up is the ant's next city; that city is then set to false so it is not visited again (lines 19-20).

This function is defined in the system as a virtual function and only the ant colony system and mutated ant actually override this method according to the algorithm selection feature (see Chapter 4).

8.10.2 The Ant System

The ant system class diagram is shown in Figure 8.8. This class is the implementation of the basic ant system algorithm defined in Chapter 4. It is also the base class that all other ant algorithms derive from and it itself derives from *IproblemSolver* interface. It comprises various virtual functions that can be overridden in derived classes. It contains the collection of ants in the colony and their tours, all information the ants require to solve the particular problem at hand, the pheromone matrix and all other information common to the other algorithms. The virtual functions mostly deal with the manipulation of the pheromone matrix, as that is how the algorithms defined in Chapter 5 differ. It is also the access point for the local search if implemented for all ant algorithms.

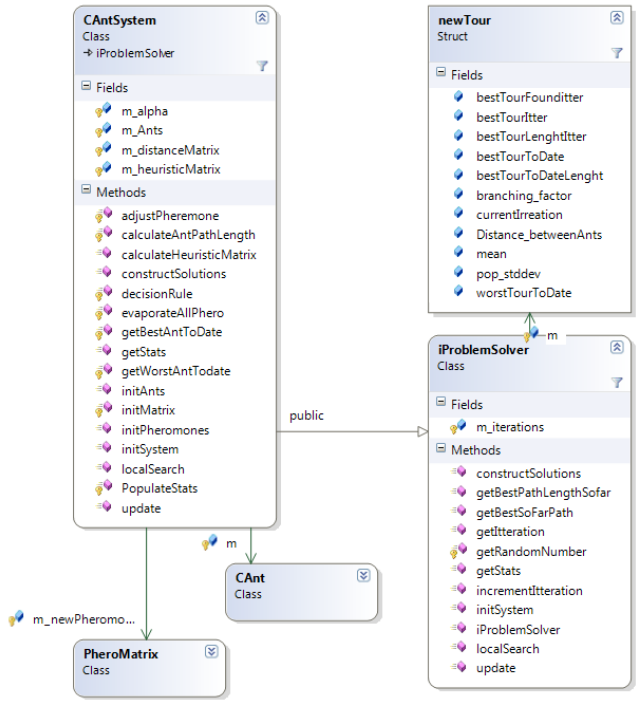


Figure 8.8 AntSolver Ant System Class Diagram

8.10.3 Elitist Ant System

The Elite Ant System algorithm inherits all functionality from its parent classes that is *CAntSystem* and *IproblemSolver*, this class is the exact same as the ant system except for Equation 5 Chapter 5.

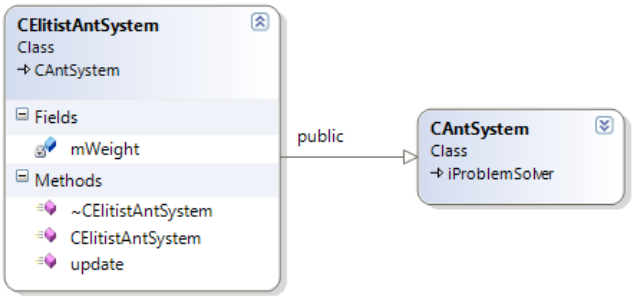


Figure 8.9 AntSolver Elitist Ant class diagram

As per the equation the virtual function update is the only function overridden as shown as the best to date ant gets to deposit extra pheromones.

```
1 void CElitistAntSystem::update()
2 {
3       evaporateAllPhero();
4       double d_tau = (double)mWeight/ m_BestAntToDate.getAntTourLength();
5       for(size_t city = 1; city <m_BestAntToDate.getNoNodes(); city++)
6       {
7               size_t from = m_BestAntToDate.getCity(city-1);
8               size_t to = m_BestAntToDate.getCity(city);
9               // eq 5
10              this->m_newPheromoneMatrix->add(from , to , d_tau);
11      }
12      for(size_t  k = 0; k <m_Ants.size(); k++)
13              adjustPheremone(k);
14      calculateHeuristicMatrix();
15 }
```

Code listing 8.4 AntSolver Elite Ant update

1 The best to date ants deposit extra pheromones along the edges based upon the elite weight as described in lines 4-11.

8.10.4 MinMax Ant System

The MinMax Ant System also derives from *CAntSystem* but adds 3 different Equations (12, 13 and 14) as outlined in Chapter 5, thereby overriding the virtual initialisation function, resulting in setting the initial pheromones to the nearest neighbour heuristic. MinMax also overrides the update function to add the pheromone smoothing technique and the reset method. In addition extra functions are added to control the minimum and maximum values for the pheromone matrix and to calculate the Branching factor is used to detect stagnation. Please see the file *CMinMaxAntSystem.cpp* in the framework for more information.

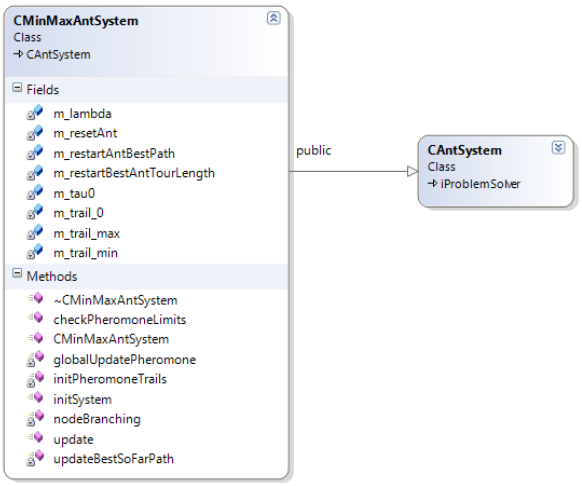


Figure 8.10 AntSolver MinMax Ant System Class Diagram

8.10.5 Best Worst Ant System

The best worst ant also derives from the *CAntSystem* class. It differs mostly in the updating of pheromones, as defined in Chapter 5; a further feature is the inclusion of mutation. Figure 8.11 shows the best worst ant class diagram. And Code Listing 8.5 shows the main update method.

Comment [LP16]: This had been called a Figure, but I changed it to code listing

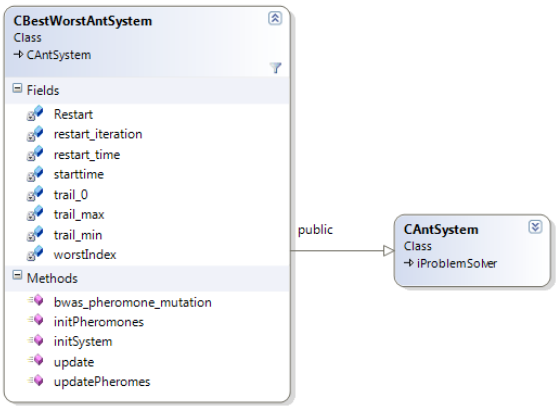


Figure 8.11 AntSolver Best-Worst Ant Class Diagram

```
1 void CBestWorstAntSystem::updatePheromes()
2 {
3      ASSERT(m_BestAntToDate.getNoNodes() == m_noNodes +1 );
4      double delta = 1.0 / ((double)(m_BestAntToDate.getAntTourLength()));
5      for(size_t i=0;i<m_BestAntToDate.getAntsCityTour().size();i++)
6      {
7       if(i==0)
8        m_newPheromoneMatrix->add(m_noNodes-1, m_BestAntToDate.getCity(i) ,(delta));
9       else
10     m_newPheromoneMatrix->add(m_BestAntToDate.getCity(i-1),
                    m_BestAntToDate.getCity(i) , (delta));
11     }
12     BWUpdate(*m_pWorstAntItt , m_BestAntToDate );
13     long distance_best_worst = distanceBetweenAnts(m_BestAntToDate , *m_pWorstAntItt);
14     long reset =  (long) (0.05 * m_noNodes);
15     if ( distance_best_worst < reset )
16     {
17            initPheromoneTrails(trail_0);
18            restart_iteration = m_iterations;
19            restart_time = GetTickCount();
20     }
21     else
22     {
23            Mutation();
24     }
25     calculateHeuristicMatrix();
26 }
```

Code listing 8.5 AntSolver Best Worst Ant update method

1) Lines 4-11 are the same as the Elitist Ant system such that the elite ant gets to deposit extra pheromones along its solution edges;
2) Line 12 defines Equation 8 from Chapter 5: any edges in the worst ant that are not in the best to date ant are subjected to further evaporation;
3) Line 13 calculates the distance between the best and worst ant. This can be used to detect stagnation as if stagnated the solutions for the worst and best tour can almost be the same (as discussed in Chapter 5);
4) Line 14 detects if the algorithm is stagnated, note the *p*-value 0.05[8]. If this is the case the pheromone matrix is reset line 15-20 (as discussed in Chapter 5);
5) Mutation is then called as defined by Equation 10.

8.10.6Ant Colony System

As will be recalled from Chapter 5, the Ant colony system also derives from the *CAntSystem* class. ACS pheromones are updated when the ant chooses the next vertex (in a sense, eating the pheromones as it goes along). Therefore the construction step of the algorithm is substantially different to the other algorithms. The class diagram is

[8] See http://en.wikipedia.org/wiki/P-value for definition of the P-Value

shown in Figure 8.12, and the Decision rule is shown in Code Listing 8.6 expanding on the *CAntSytem* decision rule.

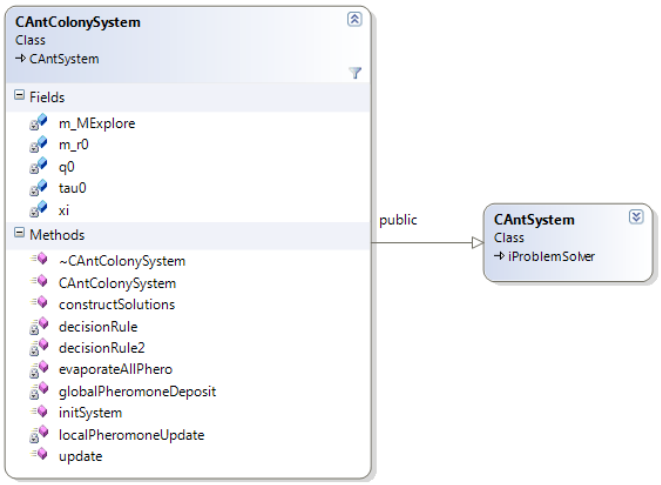


Figure 8.12 AntSolver ACS Class diagram

```
1 void CAntColonySystem::decisionRule(size_t k, size_t step)
2 {     //roulette wheel selection
3       size_t c = m_Ants[k].getCity(step-1);
4       double sum_prob =0;
5
6       for (size_t i=0; i<m_noNodes; i++)
7       {
8              t_prob[i] = 0.0;
9              m_strength[i] = 0;
10             if (m_Ants[k].isCityVisited(i) == false )
11             {
12                    t_prob[i]= (*m_heuristicMatrix)[c][i];
13                    sum_prob +=t_prob[i];
14             }
15      }
16      for (size_t z =0; z < m_noNodes; z++)
17             m_strength[z+1] = t_prob[z] + m_strength[z];
18      double x = fRand(0,1.0)  * m_strength[m_noNodes];
19      int j = 0;
20      while (!((m_strength[j] <= x) && (x <= m_strength[j+1])))
21        j++;
22      size_t randomDecision =j;
23      //find best neg
24      double maxHeuristic = -1;
25      int maxHeuristicIdx = -1;
26      for(j = 0; j < m_noNodes; j++)
27      {
28              double choice = (*m_heuristicMatrix)[c][j];
29              if(maxHeuristic < choice && !(m_Ants[k].isCityVisited(j)))
30              {
31                    maxHeuristic = choice;
32                    maxHeuristicIdx = j;
33             }
34       }
35       x=fRand(0,1.0);
36       if(x < q0)
37             {
38              m_Ants[k].setAntCity(step, maxHeuristicIdx);
39              m_Ants[k].setCityVisited(maxHeuristicIdx);
40       }
41       else //if exploitation
42       {
43            m_Ants[k].setAntCity(step, randomDecision);
44            m_Ants[k].setCityVisited(randomDecision);
45       }
46
47 }
```

Code listing 8.6 AntSolver ACS decision rule

1. Lines 2-21 are the same as the other ant systems' roulette wheel selection.
2. Lines 24-35 calculate the ant next city via the city that is nearest to it via a greedy heuristic.

3. Line 35 calculates the probability of selecting the greedy heuristic exploitation or if to go with the roulette wheel selection, the ant then moves to the selected city.

8.10.7Mutated Ant

Mutated ant also derives from *CAntSystem*, using the updating strategy from the Elitist ant system, evaporation from the Best-Worst Ant system, and the decision rule from ACS defined in Code Listing 8.6 ACS decision rule. The class diagram is shown in Figure 8.13 and the code snippet for mutation as defined in Equation 21 is shown in Code Listing 8.7.

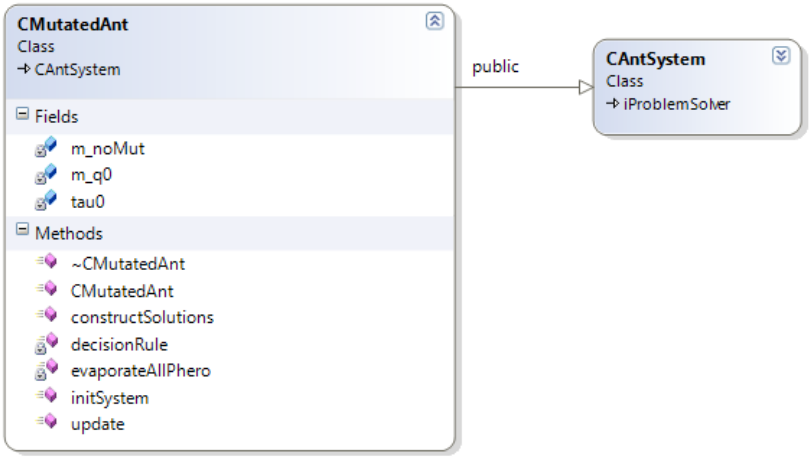


Figure 8.13 AntSolverMutated Ant Class Diagram

```
1 uniform_int_distribution<size_t> rndSelTrsh2(0,m_noNodes-1);
2 for(size_t k = 0; k < m_Ants.size()-1; k++)
3 {
4       //caculate the mutation rate
5       double nm = (1. /  m_Ants.size() +
                .5/(2*m_noAnts *this->m_iterations+1));
6       m_noMut= (nm*10) == 0 ? 1 : (nm*10) ;
7
8       for(size_t i = 0; i < m_noMut; i++)
9       {
10              size_t idx1 = rndSelTrsh2(g_rndTravel);
11              size_t idx2 = rndSelTrsh2(g_rndTravel);
12              size_t swap = m_Ants[k].getCity(idx1);
13              size_t swap2 = m_Ants[k].getCity(idx2);
14              m_Ants[k].setAntCity(idx2,swap);
15              m_Ants[k].setAntCity(idx1,swap2);// = swap;
16      }
17}
```

Code listing 8.7 AntSolver MA Mutation snippet

1. As there are only 3 ants in this colony, one ant is selected line 2;
2. Line 5 selects the number of mutations to apply, decreasing as the algorithm runs, using Fogarty equation defined in Chapter 5;
3. Lines 8-16 then implement the mutation based upon a uniform random number and swaps the cities.

8.11 Subscriber observer

One other class of note is the Subscriber observer pattern. Client code can subscribe to the algorithms and be notified of any changes like a new best tour etc. This is useful for writing logs and updating the display as in the MFC module, the algorithms execute in different threads. The class diagram is shown in Figure 8.14; for further information see source code *observer.h* or (Gamma, 1995).

Comment [LP17]: What is this. Use full term

Comment [LP18]: Is this correct?

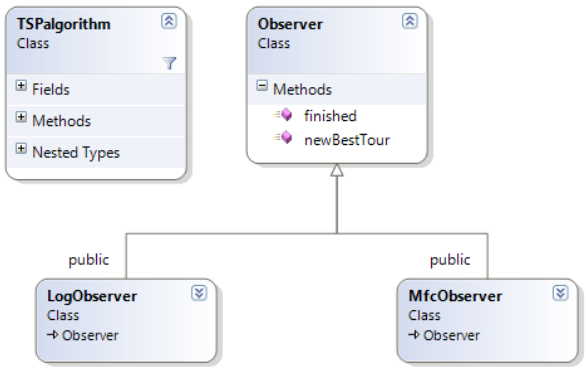


Figure 8.14 AntSolver Subscriber pattern

8.12 The Genetic Algorithm

The Genetic Algorithm implementation is very similar to the ant system also derived from the interface *iProblemSolver*. As it is used as a comparison case within the framework and is a complex class, it will not be disused in detail. For further information on the genetic algorithm, the source code is located in files CGeneticSystem.cpp and CChromo.cpp within the framework arefacts.

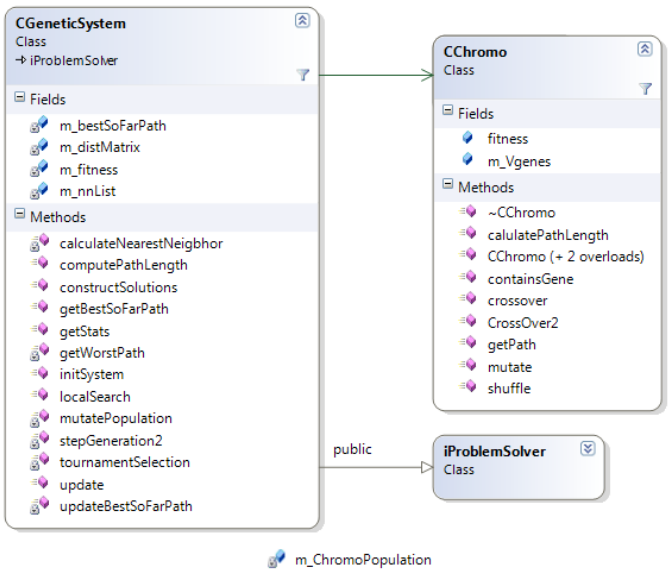


Figure 8.15 AntSolver Genetic Algorithm class diagram

8.11.1 Genetic algorithm configuration

The population size of the genetic algorithm is 10 individuals. The initial population was generated with the nearest neighbour heuristic, beginning with a randomly chosen city for each individual. An elitist strategy is used, and individuals are selected through rank selection. In cases of mutation (pmut = 0:85)

8.13 Implementing local search

Local search is also implemented; this class is also complex and will not be discussed in any detail in this dissertation. All the methods from Chapter 7 are implemented (except Tabu search). The methods can be called with a complete tour and these methods will try to improve the tour using the local search. As shown in the class diagram (Figure 8.16) the LK heuristic takes a nearest neighbour list defined by *nn_list;* all tours are passed by reference, hence it will be modified if a better solution is found. For further information please see *CLocalSearch.h* in the framework artefacts.

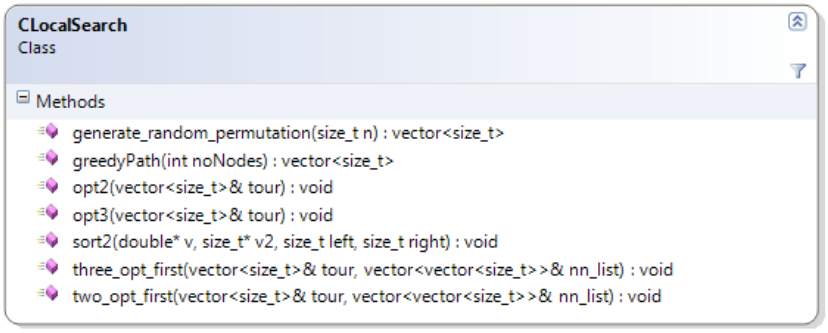


Figure 8.16 AntSolver Local Search class diagram

8.14 Conclusion

The source code for this project is large. The algorithms have 31 classes alone and not every aspect has been covered. This chapter is designed to give an introduction to the implementation of the ant algorithms and some of the basic design of these classes. Interacting, the application, the graphical user interface, the experiment classes, random ant, the brute force algorithm have been left out of this chapter, and Test Driven Development (TDD) and re-factoring techniques used have not been discussed at all.

8.15 Notes

When implementing the ant colony system algorithm, the equations were applied as shown in Chapter 4. The local pheromone update was executed during the decision when the ant moved from one city to another, as per the equations, however, this local pheromone update did not produce optimal results, as the ant colony system does not apply global evaporation. The system did not work as expected and an optimal solution never found, even though the literature stated it was one of the best performing ant colony algorithms. The equations where checked on numerous occasions, but the algorithm did not behave as expected. After a considerable period looking into a problem, the ant colony system built by Thomas Stützle was downloaded onto a Linux machine and compiled. His system showed the ant colony algorithm converging extremely fast on the same test TSP instances. Surprised by this, the C code developed by Stützle was examined in detail and was seen to be updating the pheromone matrix

using the same equations. This was extremely surprising, as it suggested a serious error within in the AntSolver implementation. After closer examination and further coding, Stützle's implementation was re-examined, and one surprising aspect - that a local search was being used – was noted. This local search was disabled and the code recompiled, and the results were the same as the ants of AntSolvers implementation.

Confident now that AntSolver implementation was correct as applied to this algorithm, the author applied the local pheromone update after the ant had completed its full tour. This greatly improved the algorithm from finding no optimal solutions to being one of the most effective ant colony algorithms as the literature states.

However, this implementation may not be exactly how the author intended. Thomas Stützle's code also illustrated the same limitations. The changes where implemented as shown in the class CAntColony System Appendix XXXX.

Comment [LP19]: You or Stützle?

9 USER MANUAL

9.1 Introduction

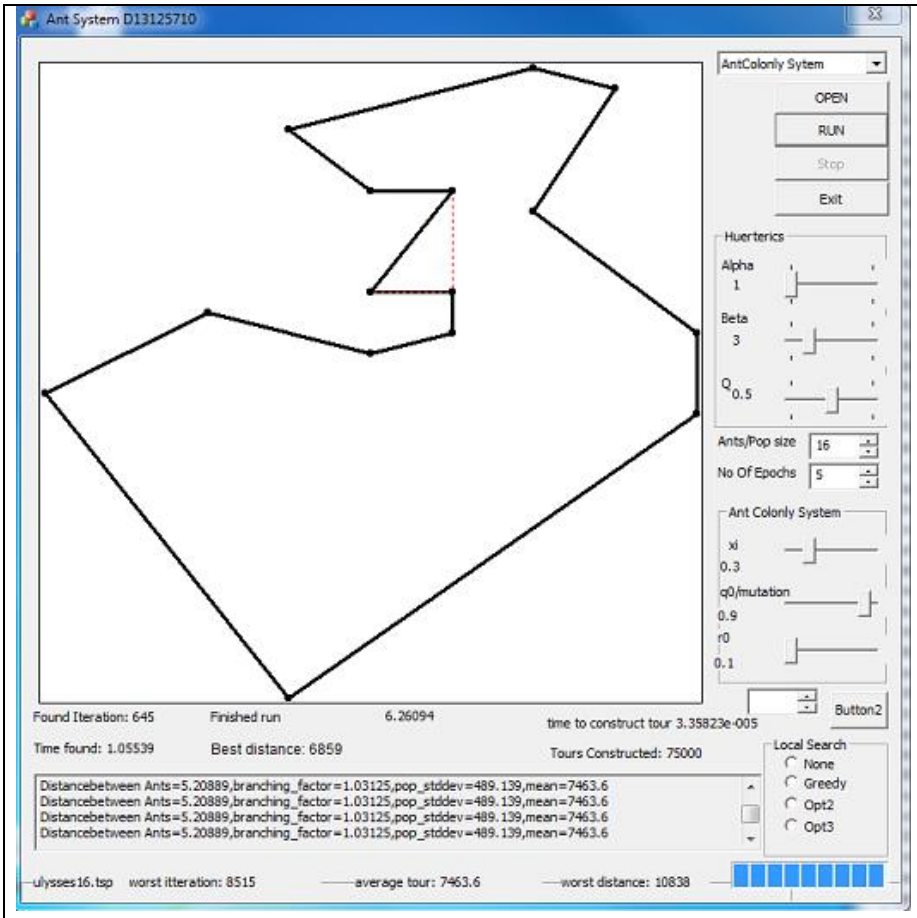


Figure 9.1 AntSolver Application GUI

The GUI Application can be executed on Windows 64 bit operating system. The seven algorithms used to solve the Travelling Salesperson Problem can be selected from the drop down combo box. Each button is self-explanatory. The open button opens a TSP file or cords file. The Start button is used to start the simulation and the Stop button is used to stop it and exit to close the application. The first slider controls the "heuristics" of the ant algorithms (alpha, beta and evaporation factor Q).The spin edit boxes control the number of ants in the simulation or population size for the genetic algorithm and

the number of Epochs (iterations * 1000) and xi, r0 for the ACS q0/mutation, selects q0 for ACS or mutation factor for mutated ant or the genetic algorithm.
Local search can be applied to each Algorithm by selecting either of the local search radio buttons. “opt2”, “opt3”, “LK” , “greedy” or “none”.

The Experiment button automates the experiments as discussed in Chapter 9.

9.2 Visual feedback

A view of the cords window is presented to the user with a view of the open problem file, with the best tour so far plotted in black and the changes since for the last iteration plotted in red.

- Found iteration: the iteration the best solution was found so far.
- Elapsed time: the algorithm has been running (does not take into account the amount of time updating the display that is costly, as the algorithms run in their own dedicated thread that is timed independently).
- Time taken to find the best solution so far in seconds
- The Current iteration of the simulation
- Average time to construct a single ant tour (solution)
- Best distance: the best distance found so far
- Tours constructed: the number of tours solutions constructed so far
- Ulysses16.tsp: the problem open
- Worst iteration: the worst distance in the current iteration of the simulation
- Average tour: the average of all tours in the simulation.
- Worst distance: the worst distance found to date

In the edit box, additional information is provided about the current iteration. The mean, standard deviation, branching factor, and the distance between the ants.(that’s how much the best tour and the worst tour in the current iteration are different by).

A file that is optional can be dumped in excel format or cvs format to capture all data. Please look at these files for further information.

Comment [LP20]: These don’t correspond to the figure exactly. Where is Finished run? What is 6.26094

Comment [LP21]: Can’t see this heading. Is it 6.26094?

Comment [LP22]: What does this correspond to on the figure?

Comment [LP23]: What does this correspond to on the figure?

Comment [LP24]: Where is this on the figure?

Comment [LP25]: What/where are they?

10 EXPERIMENTATION & EVULUATION

10.1 Introduction

Within this chapter we document our experiments with all algorithms described in Chapters 5 and 6, and local search methods described in Chapter 7. Based on the results of these experiments we determine the best algorithm for different instances of the Travelling Salesperson Problem. The exact version of the software used in the experiments chapter can also be downloaded from https://github.com/d13125710. The problems we will use will all have extremely large number of cities (over 50); most of the literature only reviews problems involving fewer than 50 cities. However, some small instances are found (i.e. eil51.tsp and Berlin52.tsp) allowing for the comparison of results with those of other researchers. Optimal solutions for TSP problems have been evaluated exactly by the Concorde library (see Chapter 2). All experiments were performed with TSP problem as shown in Table 12. A series of experiments have been planned with a particular research goal in mind. The genetic system was initialised with parameters suggested by (Ochoa et al., 2000)

- Identification of good heuristic values for each algorithm.
- Evaluation and optimisation of each algorithm.
- Comparison of all algorithms identifying the best performing.
- Comparison of the algorithms with local search options.
- Comparison of local searches with nearest neighbour lists.
- Comparison of the newly introduced mutated ant, with all variants of ant algorithms with local search enabled.

	Instance	Optimum solution	Number of vertices notes
1	Ulysses16.tsp	6859	16 GEO
2	Ulysses22.tsp	7013	22 GEO
3	Eil51.tsp	426	51 EUC_2D
4	Berlin52.tsp	7542	52 EUC_2D
5	Eil101.tsp	629	101 EUC_2D
6	Bier127.tsp	118282	127 EUC_2D
7	Ch150.tsp	6528	150 EUC_2D
8	D198.tsp	15780	198 EUC_2D
9	Tsp225.tsp	3916	198 EUC_2D

10	Gil262.tsp	2378	plots TSP, 225 EUC_2D
11	att532.tsp	27686	532 EUC_3D
12	Ei8246.tsp	unknown	8246 towns in Ireland EUC_2D
13	usa13509.tsp	19982859	13509 settlements in the USA

Table 12 Experiment problem instances

Chapter	Algorithm	Abbreviation
4	Ant System	AS
5	Elitist Ant System	EAS
5	Best Worst Ant System	BWAS
5	MinMax Ant System	MMAS
5	Ant Colony System	ACS
5	Mutated Ant System	MA
6	Genetic Algorithm	GA
7	Greedy Heuristic	GH
7	2-Optium	2OPT
7	3-Optium	3OPT
7	Lin-Kernighan	LK

Table 13 Algorithm Abbreviation

10.2 Experimentation

Solution constructions will be used to benchmark the implemented algorithms with each other; this is a *de facto* standard used by many researchers. Solution constructions are obtained after a fixed number of steps (as proposed since the Second International Conference on Evolutionary Optimisation). Informally, this suggests the number of complete solutions each algorithm constructs[9]. The number of complete solutions is limited to 10,000. Each solution in the case of the current research has a complexity $O(n^2)$; $O(m.n^2)$ is the total complexity of the algorithms, where n = number of vertices and m the population size. Empirical analysis is also used to determine the time taken to construct a complete solution, and the time taken to find the best solution so far. When evaluating, each experiment will execute ten times and calculate the algorithm's best solution, average solution, average best solution, the average time for the best solution the maximum time allowed and the error percentage. (Li *et al*., 2011) (Stützle

[9] A solution in the genetic algorithm (GA) and ACO as applied to TSP, is a complete round tour of the ant or chromosome.

and Hoos, 2000). However some experiments use the iteration count, as this is a more direct way to analyse values (so that is x number of iterations over an average of 5 or 10 runs depending upon the experiment).

The high performance profile counter will be utilised to accurately evaluate each of these measurements and each algorithm will be spawned in its own thread therefore minimising the effects of the operating system processing other messages. In fact the error is within 15 milliseconds; and as the problem requires a few minutes to solve 15ms is an acceptable error ("Analyzing Application Performance by Using Profiling Tools," n.d.).

10.3 Heuristic values

All Ant Colony algorithms have the same base heuristics α, β and p, and the extensions all have their own specific parameters:

No experiments have been run on $\tau 0$ (see Heuristic information Chapter 4), as according to Dorigo and Stützle (2004a) the values presented are good to initialise the pheromone matrix. The values use a greedy heuristic that calculates a good initial value of the matrix (see Chapter 7). The value is just slightly higher than the amounts of pheromone deposited by an ant at each iteration. To recap, if $\tau 0$ is too high, a lot of iterations are required to adjust the pheromone evaporation levels, resulting in the ant exploring a non-optimal search space. However, if too low the first solutions generated by the ants biased the search space too much in sequential iterations (see Chapter 4 for further details).

The second parameter relates to the pheromone evaporation rate p, which controls the amount of pheromones removed after each iteration. If this value is too low, the algorithm is left with too much memory of previous solutions causing stagnation of the search space. If too high the algorithm will not be directed enough towards a specific search space (see Chapter 4). Again, Dorigo and Stützle (2004a) suggest a good value for p of 0.1 for the Travelling Salesperson Problem.

10.3.1 α, β parameters

For finding the ideal *α, β* parameters, a simple experiment was designed on the first six instances of the Travelling Salesperson Problem as defined in Table 12, without local search applied. The algorithms Ant System (AS), Elite Ant System (EAS), MinMax Ant System (MMAS), Ant Colony System (ACS), Mutated Ant (MA) and Best Worst Ant System (BWAS) have been evaluated.

α, β = [0,0]	**[0,1]**	**[0,2]**	**[0,3]**	**[0,4]**	**[0,5]**	**[0,6]**	**[0,7]**	**[0,8]**	**[0,9]**	**[**
1,0]	**[1,1]**	**[1,2]**	**[1,3]**	**[1,4]**	**[1,5]**	**[1,6]**	**[1,7]**	**[1,8]**	**[1,9]**	**[2,0]**	**[**
2,1]	**[2,2]**	**[2,3]**	**[2,4]**	**[2,5]**	**[2,6]**	**[2,7]**	**[2,8]**	**[2,9]**	**[3,0]**	**[3,1]**	**[**
3,2]	**[3,3]**	**[3,4]**	**[3,5]**	**[3,6]**	**[3,7]**	**[3,8]**	**[3,9]**	**[4,0]**	**[4,1]**	**[4,2]**	**[**
4,3]	**[4,4]**	**[4,5]**	**[4,6]**	**[4,7]**	**[4,8]**	**[4,9]**					

Table 14 α, β experiment values

The experiment was executed with each parameter as defined in Table 14, set with a limit of 5000[10] complete solution constructions executed over an average of 5 runs. The average solution and the error percentage were calculated; all other parameters that will be investigated later have been set to values as suggested by their inventors (as defined in Heuristic information Chapter 4). The average error percentage was calculated for each *α, β* value. It was found empirically that all algorithms are highly dependent upon these values and the results are plotted in Figure 10.1.

[10] 5,000 was selected instead of 10,000 as would produce a lesser error margin

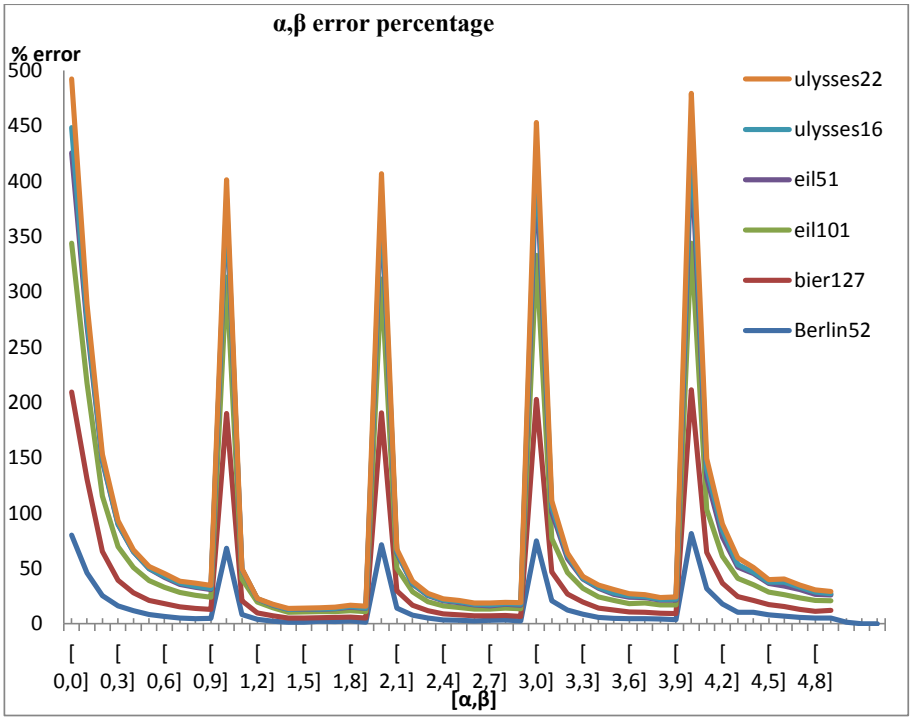


Figure 10.1 α,β error percentage

10.3.1.1 Results

As shown in Figure 10.1 we can see the lowest error % is in between the *range [**α, β**] = [1,3] -> [1,8]* for all problem instances. This is in line with Dorigo and Gambardella, (1997a) that say *[1,5]* are good values but as calculated by AntSolver in this experiment and illustrated in figure 10.2 *[1,4]* gives an average error percentage of 2.29% whereas *[1,5]* gives an average error percentage of 2.36%. It is also shown that when *[α = 0 and β <3]*, when *β* is greater than 3 the greedy heuristic as defined by *τ0* takes over and the error falls to between 20-40%. Other high error percentages are when *β =0*, this mean that the attractiveness of the pheromones is all that are guiding the ants (this is also discussed in the conclusion to Chapter 4). So when *α* grows bigger than *β*, the results are biased by the attractiveness of the pheromones, producing a bad error percentage. If *β* is greater than *α* this will make the greedy heuristic more biased.

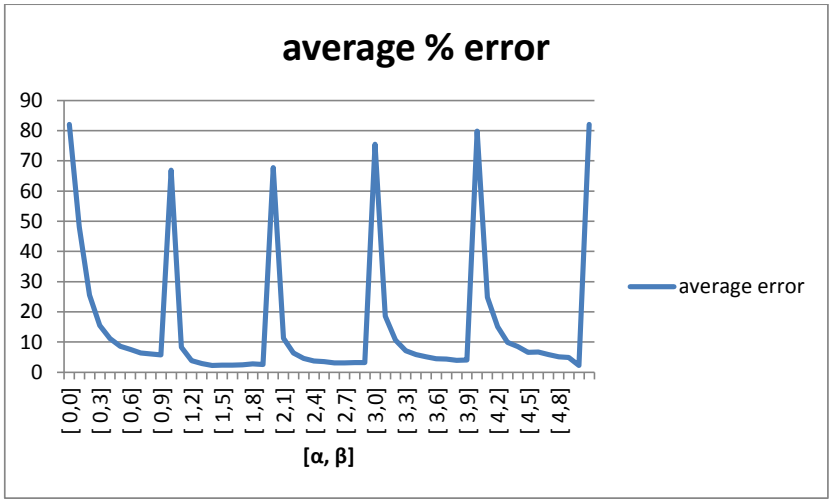


Figure 10.2 Average percentage error of figure 31

The maximum average error percentage is found when $\alpha = 0$ and $\beta = 0$ at 90%. And when $\beta = 0$ i.e. at *[4,0]* the error is 75%, the minimum error % is at *[1,4] = 2.29%* . Therefore we can conclude that *[1, 4]* are the best values of α and β for AntSolver with these problem instances, and values of zero for either *[α, β]* resulting in very high error percentages.

10.3.2 Elitist ant system

The elitist ant system deposits additional pheromones on the edges of the 'best so far' solution. This pheromone is added by elitist weight *w*, as described in Chapter 5. In this experiment the weight *w* is adjusted from *[1..10]* and a simple experiment was designed on the first six instances of the Travelling Salesperson Problems as defined in Table 10.1, without local search applied. This test uses a different method as the optimal value for the number of ants has not been evaluated yet and research shows that for the elitist ant system the number of ants should equal the number of nodes in the problem. Therefore this experiment will use iterations at 500 and considering the algorithm is only being compared to itself this will provide better accuracy in the calculation of *w*. Therefore 500 iterations were run over an average of five runs and the average error percentage from the optimal solution was then calculated and plotted as shown in Figure 10.3.

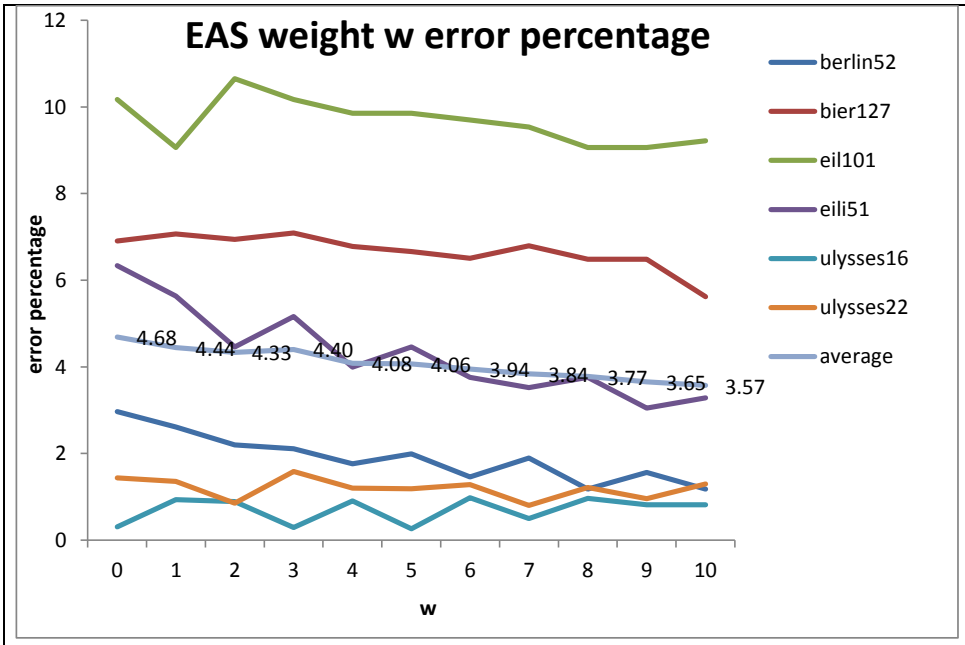


Figure 10.3 Elitist weight

10.3.2.1 Results

This experiment was mostly inconclusive, in that is all values of $w \geq 1$ did indeed find the optimal solution, but $w=10$ had the least error rate for the samples at 3.57%. Dorigo's own research has stated 5 is a good value for *w*, but as can be seen in the experiment, the error rate only drifts by 1% so any values 1-10 are optimal values. If w *is* increased i.e. $w=[20\text{-}40]$ it does not make much difference in this experiment, but would do if fewer iterations are sampled. The higher the value of *w*, the less the elite ant updates the matrix and hence the less effect the 'best so far' tour has on the overall outcome. So based upon this experiment the optimal values are $w=10$.

10.3.3 The Number of Ants

The number of ants greatly influences the result of the algorithms as described in Chapter 4. The more ants there are, the more constructed complete tours per iteration and more complete the pheromone matrix will be for ants in the next iteration, hence the better the tours constructed. For example, if there is only one ant in the colony, only one solution is completed per iteration. Therefore this solution is the only tour

used in the pheromone matrix and any edges between edges not used in this tour will not receive pheromones. This will have the effect that sequential iterations will not consider any of these edges, hence a shortening of the search space. However, if there are many ants in the colony per iteration the solution search space increases sequentially. The author experimented with are 5, 10, 25, 50, 100 and (n) number of cities in the TSP problem, utilising all the ant colony algorithms except Mutated Ant (as only 3 ants are in the colony) and Ant Colony system (which was used in Dorigo own research with 25 ants). The experiment was averaged over five runs with 2500 iterations, using the first 6 instances of TSP problems from Table 12.

	5	10	25	50	100	n
AS	8162	7984	7782	7779	n/a	7762
Berlin52	[0]	[0]	[0]	[0]	n/a	[0]
AS	133661	131224	128647	127337	126167	126551
Bier127	[0]	[0]	[0]	[0]	n/a	[0]
AS	462	459	453	447	n/a	447
Eil51	[0]	[0]	[0]	[0]	n/a	[0]
AS	7091	6921	n/a	n/a	n/a	6926
Ulyssess16	[0]	[0]	n/a	n/a	n/a	[5]
AS	7404	7240	n/a	n/a	n/a	7121
Ulyssess22	[0]	[0]	n/a	n/a	n/a	[5]
AS	731	716	706	698	690	686
Eil101	[0]	[0]	[0]	[0]	[0]	[0]
EAS	8210	7903	7640	7641	n/a	7649
Berlin52	[0]	[0]	[5]	[5]	n/a	[4]
EAS	130977	128144	127045	126344	126176	125592
Bier127	[0]	[0]	[0]	[0]	[0]	[0]
EAS	467	449	437	434	n/a	434
Eil51	[0]	[0]	[0]	[0]	n/a	[0]
EAS	6939	6879	n/a	n/a	n/a	6872
Ulyssess16	[0]	[5]	n/a	n/a	n/a	[5]
EAS	7621	7128	n/a	n/a	n/a	7112
Ulyssess22	[0]	[0]	n/a	n/a	n/a	[5]
EAS	699	675	682	683	684	686
EIL101	[0]	[0]	[0]	[0]	[0]	[0]
BWAS	8126	7903	7640	7644	n/a	7649
Berlin52	[0]	[2]	[5]	[5]	n/a	[5]
BWAS	129608	127232	125832	124859	124272	123716
Bier127	[0]	[2]	[5]	[5]	n/a	[5]
BWAS	711	694	670	664	659	654
Eil51	[0]	[0]	[0]	[0]	[0]	[0]
BWAS	446	444	440	437	n/a	434
Eil101	[0]	[0]	[0]	[0]	n/a	[0]
BWAS	6921	6868	n/a	n/a	n/a	6872
Ulyssess16	[5]	[5]	n/a	n/a	n/a	[5]
BWAS	7132	7044	n/a	n/a	n/a	7034
Ulyssess22	[0]	[5]	n/a	n/a	n/a	[5]
MMAS	7799	7809	7678	7601	n/a	7593
Berlin52	[4]	[5]	[5]	[5]	n/a	[5]
MMAS	126322	122949	122450	121521	120945	120645
Bier127	[0]	[0]	[0]	[0]	[0]	[0]

MMAS Eil51	452 [0]	436 [4]	433 [1]	432 [0]	n/a n/a	433 [0]
MMAS Eil101	689 [0]	673 [0]	661 [0]	651 [0]	649 [0]	647 [0]
MMAS Ulyssess16	6876 [5]	6867 [5]	n/a n/a	n/a n/a	n/a n/a	6867 [5]
MMAS Ulyssess22	7101 [5]	7133 [5]	n/a n/a	n/a n/a	n/a n/a	7033 [5]

Average distance of the 5 runs
[] denotes the times the optimal solution was found
n/a AntSolver framework is only designed to work with the maxium number of ants= number of nodes in the problem.

Table 15 Number of ants

10.3.3.1 Results

The results from Table 15 suggest that the AS and EAS prefer bigger colonies. As in both of these algorithms all ants get to deposit pheromones after each iteration, the algorithms have rarely found the optimal solution where the number of ants is much less than the number of nodes in the problem. Therefore, more ants are required to generate a more useful pheromone matrix. However, MMAS and BWAS are shown to find optimal solutions with a small number of ants, the solution average does improve with more ants. Based upon the experiment the values for the number of ants for both MMAS, and BWAS of around 25, and for AS and EAS the number of nodes in the problem instance. These results compare exactly the same other researchers

Comment [LP26]: Couldn't make sense of this. Sorry

Algorithm	No Of Ants
EAS	*n*
AS	*n*
BWAS	25
MMAS	25
ACS	25
MA	3

n is the number of nodes in the problem

Table 16 No Of Ants – Results

For the rest of the experiments, the size of the colonies will be set to that of Table 16, as these colony sizes have been proven here and by other researchers to be the most optimal. (Dorigo and Stützle, 2004b) (Stützle and Hoos, 2000).

10.4 Ant colony system

The Ant colony system introduces two extra parameters *q0* and *xi*. A simple experiment was designed on the first six instances of the Travelling Salesperson

Problem as defined in Table 10.2, without local search applied. The values experimented with are shown in Table 17: using 5,000 tour constructs run over an average of five runs the average error from the optimal solution was then calculated and plotted as shown Figure 10.4.

[xi,q0] [0.1,0.1] ->[0.6,1,0] in increments of 0.1
[xi,q0]=[0.1,0.9] -> [0.1,0.99] in increments of 0.01
[xi,q0]=[0.3,0.9] -> [0.3,0.99] in increments of 0.01

Table 17 ACS xi and q0 experiment parameters

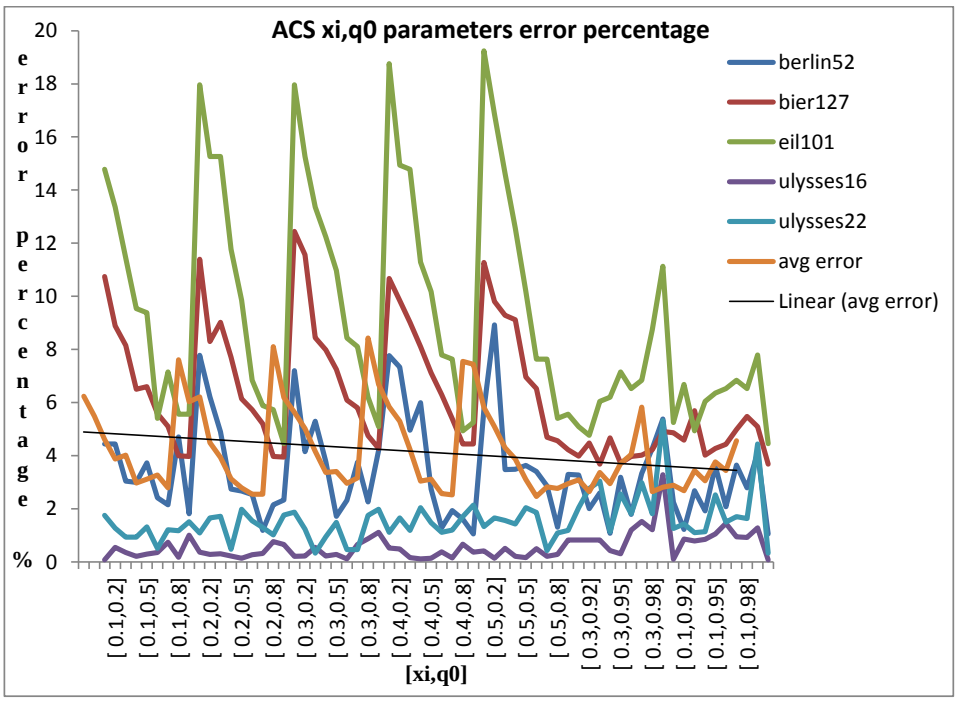


Figure 10.4 ACS parameters error percentage

10.4.1 Results

As can be seen in Figure 10.4 the least error percentage can be seen when *xi = 0.1 or xi = 0.3* and *q0 ≥ 0.9* and *q0 < 1*. So further testing was added to *compare [xi,q0]=[0.1,0.9] -> [0.1,0.99]* in increments of *0.01 and [xi,q0]=[0.3,0.9] -> [0.3,0.99]* in increments of *0.01*. It is shown that any of these values produce good results, but for bigger problems, the higher values of *xi* and *q0* are preferred and for small problems, lower values. Sub-optimal values can be shown when *x0 > q0*. This is in line with

Marco own research that states *xi=0.1* and *q0=0.90* give good all-round results. Also *xi=0.2* and *xi=0.5* also give good results (Dorigo and Gambardella, 1997b).

10.5 Comparison of Ant Colony Algorithm

Now that we have defined optimal parameters for each ant colony algorithm from our problem instances, they can now be compared to one another and with the genetic algorithm. The parameters are shown in Table 18.

Algorithm	**α,**	**β**	**ρ**	No Ants (m)	w	Q0	xi	*τ0*
AS	1	4	0.3	n	n/a	n/a	n/a	$1/C^{nn}$
EAS	1	4	0.3	n	10	n/a	n/a	$1/C^{nn}$
BWAS	1	4	0.3	n	n/a	n/a	n/a	$1/pC^{nn}$
MMAS	1	4	0.3	25	n/a	n/a	n/a	$1/pC^{nn}$
ACS	1	4	0.1	15	n/a	0.98	0.1	$1/C^{nn}$
MA	1	4	0.1	n/a	20	0.98	0.1	$1/C^{nn}$

n is a number of cities in the TSP instance. C^{nn} is the nearest neighbour distance.
m is the number of ants GA set to elite rank, roulette wheel selection, population at 10, mutation ratio 90%,

Table 18 Optimal heuristic values

The experiment is executed over 10,000 tour constructs with 500 second time limit[11], averaged over 10 runs. If the optimal solution is found multiple times this shows up in the error percentage. The error percentage is the average best solution from the optimal. The average iteration to find the best solution, and the average time for the best solution were also calculated.

Problem	AS	BWAS	MMAS	ACS	EAS	GS	Optimal
Ulysses16	**6859**	**6859**	**6859**	**6859**	**6859**	**6859**	6859
	(6895)	(6893.8)	(6860.18)	(6859)	(6860.1)	(6862.6)	
	[9]	[10]	[10]	[10]	[23]	[3285]	
	0.001sec	0.0014sec	0.0014sec	0.002 sec	0.014 sec	0.45 sec	
	0.5%	0.5%	0.016 %	0 %	0.016%	0.05%	
Ulysses22	7046	**7013**	**7013**	**7013**	**7013**	**7013**	7013
	(7097.4)	(7071.3)	(7023.2)	(7013.6)	(7067.5)	(7077.9)	
	[41]	[485]	[70]	[18]	[2]	[444]	
	0.008sec	0.9sec	0.16sec	0.0042sec	0.002sec	0.06sec	
	1.2%	0.83%	0.14%	0.009%	0.78%	0.93%	
Eil51	447	**426**	428	427	428	437	426
	(452.1)	(434.4)	(433)	(429.9)	(431.2)	(446.2)	
	[200]	[97]	[203]	[207]	[207]	[4519]	
	0.26 sec	0.12 sec	0.25 sec	0.8 sec	0.8 sec	2.6 sec	
	6.12%	1.97%	1.64%	0.91%	1.22%	4.74%	
Berlin52	7662	**7542**	**7542**	**7542**	**7542**	**7542**	7542

[11] None of these instances required over 500 seconds to build 10000 solution constructions.

	(7707.3) [42] 0.28 sec 2.19%	(7666.5) [21] 0.1 sec 1.65%	(7742.9) [101] 0.23 sec 2.66%	(7560) [26] 0.23 sec 0.23%	(7651.2) [45] 0.64 sec 1.45%	(7971.7) [836] 2.8 sec 5.70%	
Bier127	124663 (125706.9) [212] 1.65 sec 6.3%	122166 (125392) [150] 1.15sec 6.0%	122606 (123877) [365] 2.79 sec 4.73%	**118944** (120672.3) [304] 2.35 sec 2.02%	119032 (120827.2) [295] 4.3 sec 2.15%	126639 (131807) [8911] 13.31 sec 11.43%	118282
Ch150	6684 (6767.1) [270] 3.16 sec 3.66%	6769 (6933.3) [133] 1.47 sec 6.2%	6619 (6700.7) [366] 4.07 sec 2.6%	6575 (6665.4) [296] 3.34 sec 2.1%	**6562** (6630.1) [313] 6.1 sec 1.56%	7423 (7614.6) [8942] 16.72 sec 16.64%	6528
D198	17024 (17386.6) [152] 2.7 sec 10.18%	16985 (17300.5) [253] 4.4 sec 9.63%	16832 (17386.8) [340] 5.86 sec 10.18%	16409 (16647.9) [326] 5.7 sec 5.5%	**16216** (16562) [332] 9.6 sec 4.95%	17750 (18383.6) [9672] 27.7 sec 16.50%	15780
Tsp225	4271 (4364.5) [242] 5.59 sec 11.45%	4221 (4354.2) [224] 5.19 sec 11.19%	4274 (4491.2) [377] 8.67 sec 14.69%	**4003** (4146.1) [299] 6.91 sec 5.88%	4016 (4095) [385] 13.65 sec 4.57%	6074 (6506.7) [1968] 7.11 sec 66.12%	3916
Gil262	2595 (2658.3) [291] 9.82 sec 11.79%	2591 (2643.9) [157] 5.38 sec 11.18%	2621 (2785.2) [353] 11.67 sec 17.12%	**2422** (2505.7) [361] 12.03 sec 5.37%	2440 (2508.4) [352] 16.90 sec 5.48%	3982 (4326.3) [1981] 9.48 sec 81.93%	2378
Att532	33054 (33607.2) [222] 30.14 sec 21.38	31126 (32666.5) [283] 38.32 sec 17.99%	34828 (35142.9) [195] 25.96 sec 26.93%	30286 (30811.4) [363] 48.63 sec 11.29%	**29872** (30569.2) [363] 62.80 sec 10.41%	77130 (80589.2) [1991] 34.69 sec 191%	27686

Best found solutions are shown in bold. Optimal is the best proven tour
[] brackets denotes the iteration the best solution was found for ACO , for GA denotes tour construction
() brackets denote the average best tour.
sec denotes the average time in seconds to find the average solution.
% is the average error percentage of all best distances.

Table 19 Comparison of algorithms

10.5.1 Results

As the results suggest it can be seen (Table 19) that all algorithms found the optimal solution, for the small instances up to (Berlin52) all with good averages and low error margins MMAS and ACS performed the best, with almost identical error margins and found the optimal solutions in most runs. However EAS found all optimal solutions up to Berlin52, impressive however the average error percentage is higher than MMAS, ACS and approximately the same as BWAS All algorithms for the small instances compared almost identical empirically except for the genetic algorithm, as this

algorithm is more complex than the Ant Systems and requires more CPU time (Dorigo and Stützle, 2004) .

On the second set 100-200 cities (Eil101, Bier127, Ch150, D198) this is where the basic ant systems EAS, AS, begin to break down; the reasons are discussed in (Chapter 5). However they are both still producing good results with an average error percentage between 4 and 10%, and managing to find their average solutions empirically at approximately 1 second for (Eil101) to 5 seconds for (D198). Both MMAS and ACS found better solutions than all other algorithms, in fact even their average solution is better than EAS, BWAS, AS and the genetic algorithm best solutions. It is also noted here that MMAS and ACS algorithms converge quickly to the optimal solution, usually during the first couple of iterations. This implies that in the pheromone matrix $\tau 0$ initialisation plays an important role in allowing the algorithm finding the optimal solution quickly (Zhang et al., 2011). However empirically they take longer, this is because these systems keep producing better solutions as they prevent stagnation as discussed in Chapter 5.

These results compares with literature namely (Stützle and Hoos, 2000) (Bullnheimer et al., 1997) (Parsons, 2005) (Coleman et al., 2004) The BWAS is in third place as the error margins suggest this is also confirmed by literature, BWAS has been proven to work better with quadratic equations than with TSP (Cordón et al., 2002).

The Genetic Algorithm is the worst performing of all, although some researchers suggest that the GA compares well with the basic AS, However these papers only benchmark with up to 50 cities and as the results here show it is on par with AS up to 50 cities. (Parsons, 2005) (Li et al., 2008). However with problems over 100 cities it starts to break down and with 500 cities it is completely off at 191% (This is mostly to do with the random generation of the initial chromosomes). However in later experiments we retrofit the basic GA with improved techniques as this experiment is only to compare the basic algorithms side by side not there more advanced variants.

In the really large instance (att532) we can clearly see that MMAS and ACO stand out from all other algorithms producing error rates between (10 – 11%).

If each algorithm in the experiment was allowed greater solution constructions, the results of all algorithms would improve and both MMAS and ACS will almost always find the optimal solution for all instances even for (att532). As stated earlier most researcher only test up to 50 cities, and report error margins of around 1% for ACS and

MMAS, and 2.5% for EAS and 6% for AS, this compares with the results gathered with AntSolver (Shtovba, 2005)

10.6 Hybridisation combining ACO and GA with local search method

In this section Ant Colony Optimisation and Genetic Algorithm are combined with 2-Opt, 3-Opt, and the LK local search heuristics. Also introduced is the LK heuristic to AS and EAS this technique was not discovered in any literature. Introduced also in this experiment is the Mutated Ant System, as it is a hybridisation algorithm (described in Chapter 5). It can only be compared to other Ant algorithms when local search is applied. However these experiments are using Darwinian updates as discussed in (Chapter 7) as it is proven to provide better solutions. To recap local search is applied after each ant construct its tour and before the pheromones are updated. In addition all tours will be applied to the local search heuristics, not just the best so far tour. This increases the algorithms chances for success.

10.6.1 2-Opt

Here is an experiment with 2-Opt using the same experiment as defined previously "Comparison of Ant colony algorithm". The only difference here is that 2-Opt local search method is enabled. Again 10000 solution constructions over an average of ten runs, the best distance, average best distance, the average iteration for the best solution and the average time to find the best solution, with the percentage error.

Problem	AS	BWAS	MMAS	ACS	EAS	GS	Optimal
Ulysses16	**6859** 6863) [277] 0.09sec **0.06%** {0.50%}	**6859** 6874) [134] 0.06 sec **0.22%** {0.50%}	**6859** 6862) [53] 0.06 sec **0.05%** {0.02%}	**6859** 6859) [32] 0.04 sec **0%** {0%}	**6859** 6859) [85] 0.12 sec **0%** {0.02%}	**6859** 6859) [467] 0.16 sec **0%** {0.05%}	6859
Ulysses22	**7013** (7077) [269] 0.06 sec **0.92%** {1.2%}	**7013** (7048) [170] 0.04 sec **0.50%** {0.83%}	**7013** (7013) [78] 0.02 sec **0.00%** {0.14%}	**7013** (7013) [78] 0.02 sec **0.00%** {0.01%}	**7013** (7082) [200] 0.14 sec 0.99% **{0.78%}**	**7013** (7013) [449] 0.07 sec **0.00%** {0.93%}	7013
Eil51	430 (438) [190] 0.48 sec	431 (431) [136] 0.33 sec	430 (435) [159] 0.44 sec	**426** (428) [96] 0.30 sec	427 (429) [145] 0.67 sec	430 (437) [1153] 0.91 sec	426

	3.00%	**1.38%**	**2.14%**	**0.59%**	**0.70%**	**2.70%**	
	{6.12%}	{1.97%}	{1.64%}	{0.91%}	{1.22%}	{4.74%}	
Berlin52	**7542**	**7542**	**7542**	**7542**	**7542**	**7542**	7542
	7581)	(7667)	7738)	7553)	7649)	7979)	
	[116]	[78]	[107]	154	[97]	[1290]	
	0.23 sec	0.15 sec	0.21 sec	0.31 sec	0.36 sec	0.85 sec	
	0.53%	1.66%	**2.61%**	**0.15%**	**1.43%**	5.80%	
	{2.19%}	**{1.65%}**	{2.66%}	{0.23%}	{1.45%}	**{5.70%}**	
Eil101	681	643	635	635	**631**	680	629
	(690.80)	(650)	(652)	(642)	(641)	(707)	
	[153]	[104]	[256]	[213]	[187]	[1617]	
	1.06 sec	0.70 sec	1.70 sec	1.39 sec	1.79 sec	2.06 sec	
	9.83%	**3.34%**	3.72%	**2.08%**	**2.03%**	12.48%	
	{10.5%}	{6.09%}	**{3.05%}**	{2.75%}	{2.88%}	**{9.84%}**	
Bier127	124468	119481	120565	118698	118647	127286	118282
	126061	122255	122861	120035	120001	136823)	
	[258]	[210]	[334]	[285]	[233]	[1883]	
	2.98 sec	2.49 sec	3.81 sec	3.45 sec	3.39 sec	3.42 sec	
	6.58%	**3.36%**	3.87%	**1.48%**	**1.45%**	15.68%	
	{6.30%}	{6.00%}	**{3.05%}**	{2.75%}	{2.88%}	**{11.43%}**	
Ch150	6727	6626	6563	6575	**6550**	7933	6528
	(6813)	(6756)	(6687)	(6618)	(6596)	(8287)	
	[253.60]	[159.40]	[323.80]	[335.80]	[207]	[1922]	
	4.10 sec	2.51 sec	5.37 sec	5.28 sec	3.80 sec	4.52 sec	
	4.38%	**3.49%**	**2.44%**	**1.39%**	**1.04%**	26.96%	
	{3.66%}	{6.2%}	{2.6%}	{2.1%}	{1.56%}	**{16.64%}**	
D198	16875	16137	16631	16148	**16007**	20717	15780
	(17252)	(16773)	(17075)	(16340)	(16447)	(22814)	
	[180]	[229]	[316]	[262]	[306]	[1958]	
	4.46 sec	5.76 sec	7.83 sec	6.52 sec	8.93 sec	6.76 sec	
	9.33%	**6.30%**	**8.21%**	**3.55%**	**4.23%**	44.58%	
	{10.18%}	{9.63%}	{10.18%}	{5.5%}	{4.95%}	**{16.50%}**	
Tsp225	4215	4036	4254	4035	**3961**	5657	3916
	(4308)	(4197)	(4330)	(4091)	(4067)	(6167)	
	[244]	[193]	[367]	[242]	[335]	[1954]	
	9.43 sec	7.43 sec	13.86 sec	8.96 sec	13.79 sec	10.16 sec	
	10.03%	**7.20%**	**10.59%**	**4.49%**	**3.87%**	**57.48%**	
	{11.45%}	{11.19%}	{14.69%	{5.88%}	{4.57%}	{66.12%}	
Gil262	2567	2503	2534	2448	**2437**	3994	2378
	(2643)	(2574)	(2711)	(2489.20	(2483)	(4315)	
	[248]	[180]	[359]	[318]	[343]	[1972]	
	13.93 sec	10.29 sec	21.35 sec	17.86 sec	17.84 sec	12.45 sec	
	11.16%	**8.28%**	**14.02%**	**4.68%**	**4.43%**	**81.47%**	
	{11.8%}	{11.18%}	{17.12%}	{5.37%}	{5.48%}	{81.93%}	
Att532	32960	30436	33093	**29077**	29246	81138	27686
	(33767)	(31681)	(33999)	(29916)	(29894)	(87228)	
	[151]	[252]	[345]	[362]	[372]	[1989]	
	29.64 sec	51.75 sec	70.74 sec	73.71 sec	67.16 sec	38.77 sec	
	21.97%	**14.43%**	**22.81%**	**8.06%**	**7.98%**	215%	
	{21.38%}	{17.99%}	{26.93%}	{11.29%}	{10.41%}	**{191%}**	

Best solutions are shown in bold. Optimal is the best proven tour.
() brackets denote the average best tour.
[] brackets denote the iteration best solution was found for ACO, for GA denotes tour construction.
Sec denotes the average time in seconds to find the average solution.
% is the average error percentage of all best distances.
{} denotes the average percentage error from the optimal solution without local search.

Table 20 Comparison of algorithms with 2-Opt local search

10.6.1.1 Results

As illustrated in Table 20, enabling 2-Opt local search methods makes a slight improvement reducing the average error percentage by a few points, but at an expense increasing the runtime by $O(n^{2.2})$, as illustrated by the (ulysses16) instance, AS finds the optimal solution on an average of [9] irritation without local search (Table 19), with 2-Opt enabled this increases to iteration [277]. In addition to empirical performance, 2-Opt can also have side effects, in that it can steer an algorithm into a local optimal. Therefore applying 2-Opt to some problems can actually be detrimental, this can be seen in some cases notably AS and GA where the large problem instances increases the error percentage instead of decreasing it i.e. (att532) Table 16 the error rate for GA increases from 191% to 215%. This can be explained, when AS, EAS or GA are initialised, their initial values are generated randomly. These values could either be good or bad. As Balaprakash et al., (2006) states *"the centre of gravity is far from the global optimal"*. If the initial random route is far from the centre of gravity the local search method improves and the results are used as feedback to the algorithm, the algorithm at this stage has not "learnt" optimal routes and is stuck in a local optimal (Balaprakash et al., 2006) (Whitley et al., 2004). Stützle and Dorigo (1999) also discovered this problem and state *"local search gets stuck at suboptimal solutions in all the four instances tested"* All other algorithms do not suffer this effect as much, as their initial routes are created by the nearest neighbour heuristic as discussed in Chapter 5. However 2-Opt is a simple algorithm to implement but not worth the extra complexity, as if each algorithm is let run without 2-Opt and add the extra runtime expense the algorithms will improve the tours themselves.

10.6.2 3-Opt

This is an experiment with 3-Opt (Chapter 7) using the same experiment as defined previously "Comparison of Ant colony algorithm", the only difference here is that 3-Opt local search method is enabled. Again 10000 solution constructions over an average of ten runs, the best distance, average best distance, the average iteration for the best solution and the average time to find the best solution, with the percentage error.

Problem	AS	EAS	BWAS	MMAS	ACS	GA	Optimal
Ulysses16	**6859** (6859.00) [84] 0.07 sec **0.00%** {0.5%}	**6859** (6859.00) [49] 0.04 sec **0.00%** {0.5%}	**6859** (6859.00) [30] 0.08 sec **0.00%** {0.016 %}	**6859** (6859.00) [30] 0.09 sec **0.00%** {0 %}	**6859** (6859.00) [46] 0.11 sec **0.00%** {0.016%}	**6859** (6859.00) [198] 0.16 sec **0.00%** {0.05%}	6859
Ulysses22	**7013** (7060.20) [44] 0.01 sec **0.67%** {1.2%}	**7013** (7013.00) [175] 0.04 sec **0.00%** {0.83%}	**7013** (7034.00) [41] 0.01 sec 0.30% **{0.14%}**	**7013** (7013.00) [48] 0.02 sec **0.00%** {0.009%}	**7013** (7037.10) [94] 0.07 sec **0.34%** {0.78%}	**7013** (7013.00) [802] 0.14 sec **0.00%** {0.93%}	7013
Eil51	427 (432.10) [136] 0.51 sec **1.43%** {6.12%}	**426** (430.00) [49] 0.24 sec **0.94%** {1.97%}	427.00 (430.40) [194] 0.64 sec **1.03%** {1.64%}	**426** (426.80) [215] 0.74 sec **0.19%** {0.91%}	**426** (427.80) [129] 0.79 sec **0.42%** {1.22%}	434.00 (438.30) [1236] 1.27 sec **2.89%** {4.74%}	426
Berlin52	**7542** (7587.30) [173] 0.35 sec **0.60%** {2.19%}	**7542** (7621.60) [17] 0.04 sec **1.06%** {1.65%}	**7542** (7736.60) [107] 0.21 sec **2.58%** {2.66%}	**7542** (7542.00) [128] 0.26 sec **0.00%** {0.23%}	**7542** (7634.60) [109] 0.46 sec **1.23%** {1.45%}	**7542** (7740.40) [1088] 0.82 sec **2.63%** {5.70%}	7542
Eili101	670 (685.20) [188] 1.28 sec **8.93%** {10.48%}	639 (647.40) [153] 1.12 sec **2.93%** {6.09%}	641 (653.80) [249] 1.84 sec 3.94% **{3.05%}**	**631** (639.30) [263] 2.08 sec **1.64%** {2.75%}	642 (645.90) [171] 1.66 sec **2.69%** {2.88%}	675 (686.60) [1669] 2.65 sec **9.16%** {9.84%}	629
Bier127	124834 (126197.3) [225] 2.90 sec 6.69% **{6.3%}**	119908 (121004.7) [234] 3.55 sec **2.30%** {6.0%}	120378 (122441.2) [306] 4.19 sec **3.52%** {4.73%}	118773 (119487.6) [320] 4.22 sec **1.02%** {2.02%}	**118822** (119720.5) [263] 4.09 sec **1.22%** {2.15%}	125929 (132205.4) [1817] 3.75 sec 11.77% **{11.43%}**	118282
Ch150	6670 (6761.70) [241] 4.11 sec **3.58%** {3.66%}	6644 (6726.50) [141] 2.51 sec **3.04%** {6.2%}	6592 (6668.80) [316] 5.96 sec **2.16%** {2.6%}	6562 (6615.60) [289] 5.43 sec **1.34%** {2.1%}	**6548** (6588.70) [145] 2.98 sec **0.93%** {1.56%}	7286 (7635.60) [1875] 5.42 sec 16.97% **{16.64%}**	6528
D198	16988 (17213.70) [146] 3.99 sec **9.09%** {10.18%}	15989 (16469.20) [168] 4.31 sec **4.37%** {9.63%}	16551 (16919.30) [302] 7.80 sec **7.22%** {10.18%}	**15983** (16220.0) [254] 6.83 sec **2.79%** {5.5%}	16108 (16345.50) [262] 8.42 sec **3.58%** {4.95%}	18579 (20551.10) [1958] 7.33 sec 30.24% **{16.50%}**	15780
Tsp225	4234 (4344.60) [206] 6.87 sec **10.94%** {11.45%}	4057 (4152.70) [203] 6.88 sec **6.04%** {11.19%}	4120 (4241.20) [373] 12.51 sec **8.30%** {14.69%}	3993 (4036) [318] 10.78 sec **3.06%** {5.88%}	**3962** (4007.70) [275] 10.16 sec **2.34%** {4.57%}	5339 (5699.30) [1975] 9.21 sec **45.54%** {66.12%}	3916
Gil262	2573 (2658.00) [242] 11.51 sec	2448 (2512.10) [145] 7.07 sec	2521 (2631.30) [361] 17.11 sec	2415 (2461.80) [322] 15.34 sec	**2406** (2442.50) [321] 15.32 sec	3570 (3926.00) [1965] 11.48 sec	2378

	11.77%	**5.64%**	**10.65%**	**3.52%**	**2.71%**	**65.10%**	
	{11.79%}	{11.18%}	17.12%}	{5.37%}	{5.48%}	{81.93%}	
Att532	33416	29699	32542	**28650**	29171	65290	27686
	(33905.50)	(30612.80)	(33634.60)	(29535.00)	(29775.00)	(77092.20)	
	[118]	[295]	335.00	370.20	[346]	[1974]	
	23.68 sec	58.10 sec	65.17 sec	72.48 sec	63.99 sec	41.08 sec	
	22.46%	**10.57%**	**21.49%**	**6.68%**	**7.55%**	**178.45%**	
	{21.38%}	{17.99%}	{26.93%}	{11.29%}	{10.41%}	{191%}	

Best solutions are shown in bold.
Optimal is the best proven tour
() brackets denotes the average best tour
[] brackets denote the iteration best solution was found for ACO, for GA denotes tour construction.
Sec denotes the average time in seconds to find the average solution.
% is the average error percentage of all best distances.
{} denotes the average percentage error from the optimal solution without local search.

Table 21 Comparison of Algorithms with 3-Opt local search

10.6.2.1 Results

As illustrated by table 21, running the algorithms with 3-Opt enabled provides a moderate improvement over no local search, however like 2-Opt adding additional complexity this time $O(n^3)$, to all algorithms i.e. AS on (berlin52) without local search finds the average solution at iteration [42] , with 3-Opt at iteration [177]. 3-Opt improves most problems average solutions by a few points. Good results for a simple algorithm that is easy to implement. In addition 3-Opt like 2-Opt still has a detrimental effect on AS and GA with some problems as discussed in the previous experiment of 2-Opt. MMAS and ACO are the clear winners with the improvements made by 3-Opt, with their average solutions much better than the best solutions of the rest.

10.6.3LK heuristic

Here the experiment is with LK using the same experiment as defined previously "Comparison of Ant colony algorithm". The only difference here is that LK local search method is enabled. Again 10000 solution constructions over an average of ten runs, the best distance, average best distance, the average iteration for the best solution and the average time to find the best solution, with the percentage error. Included in this experiment is Mutated Ant (as described in Chapter 4).

Problem	AS	EAS	BWAS	MMAS	ACS	GA	MA
Ulysses16	**6859**	**6859**	**6859**	**6859**	**6859**	**6859**	**6859**
Opt 6859	(6859)	(6859)	(6859)	(6859)	(6859)	(6859)	6859
	[0]	[0]	[0]	[0]	[0]	[0]	[0]
	0.00 sec	0.00 sec	0.00 sec	0.00 sec	0.60 sec	0.00 sec	0.00 sec
	0.00%	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**

	{0.5%}	{0.5%}	{0.02%}	{0%}	{0.016%}	{0.05%}	n/a
Ulysses22	**7013**	**7013**	**7013**	**7013**	**7013**	**7013**	**7013**
Opt 7013	(7013)	(7013)	(7013)	(7013)	(7013)	(7013)	(7013)
	[0]	[0]	[0]	[0]	[0]	[0]	[0]
	0.00 sec	0.00 sec	0.00 sec	0.00 sec	0.00 sec	0.00 sec	0.00 sec
	0.00%	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
	{1.2%}	{0.83%}	{0.14%}	{0.01%}	{0.78%}	0.93%}	n/a
Eil51	**426**	**426**	**426**	**426**	**426**	**426**	**426**
Opt 426	(426)	(426)	(426)	(426)	(426)	(426)	(426)
	[2]	[1]	[2]	[3]	[5]	[2]	[9]
	0.01 sec	0.01 sec	0.02 sec	0.02 sec	0.03 sec	0.01 sec	0.01 sec
	0.00%	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
	{6.12%}	{1.97%}	{1.64%}	{0.91%}	{1.22%}	{4.74%}	n/a
Berlin52	**7542**	**7542**	**7542**	**7542**	**7542**	**7542**	**7542**
Opt 7542	(7542)	(7542)	(7542)	(7542)	(7542)	(7542)	(7542)
	[0]	[0]	[0]	[0]	[0]	[0]	[0]
	0.00 sec	0.00 sec	0.00 sec	0.00 sec	0.00 sec	0.00 sec	0.00 sec
	0.00%	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
	{2.19%}	{1.65%}	{2.66%}	{0.23%}	{1.45%}	{5.7%}	n/a
Eili101	**629**	**629**	**629**	**629**	**629**	**629**	**629**
Opt 629	(629)	(629)	(629)	(629)	(629)	(629)	(629)
	[7]	[5]	[10]	[10]	[10]	[5]	[12]
	0.07 sec	0.05 sec	0.10 sec	0.09 sec	0.11 sec	0.02 sec	0.02 sec
	0.00%	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
	{10.48%}	{6.09%}	{3.05%}	{2.75%}	{2.88%}	{9.84%}	n/a
Bier127	**118282**	**118282**	**118282**	**118282**	**118282**	**118282**	**118282**
Opt	(118282)	(118282)	(118282)	(118282)	(118282)	(118282)	(118282)
118282	[34]	[9]	[16]	[8]	[6]	[8]	[37]
	0.78 sec	0.28 sec	0.51 sec	0.27 sec	0.21sec	0.09 sec	0.22 sec
	0.00%	**0.00%**	**0.00%**	**0.00%**	**0.00 %**	**0.00%**	**0.00%**
	{6.3%}	6.0%}	{4.73%}	{2.02%}	{2.15%}	{11.43%}	n/a
Ch150	**6528**	**6528**	**6528**	**6528**	**6528**	**6528**	**6528**
Opt 6528	(6528)	(6530.10)	(6528)	(6528)	(6528)	(6528)	(6528)
	[18]	[6]	[10]	[7]	[18]	[9]	[56]
	0.99 sec	0.37 sec	0.64 sec	0.44 sec	0.94 sec	0.12 sec	0.60
	0.00%	**0.03%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
	{3.66%}	{6.2%}	{2.6%}	{2.1%}	{1.56%}	16.64%}	n/a
D198	**15780**	**15780**	**15780**	**15780**	**15780**	**15780**	**15780**
Opt	(15786.2)	(15780.1)	(15780.4)	(15780.2)	(15780.8)	(15780)	(15780)
15780	[177]	[68]	[101]	[94]	[76]	[73]	[528]
	0.11sec	[5] sec	[6] sec	[6] sec	[7] sec	[1] sec	[8] sec
	0.04%	**0.01%**	**0.01%**	**0.0%**	**0.01%**	**0.00%**	**0.00%**
	{10.18%}	{9.63%}	{10.18%}	{5.5%}	{4.95%}	{16.50%}	n/a
Tsp225	**3916**	**3916**	**3916**	**3916**	**3916**	**3916**	**3916**
Opt 3916	(3919)	(3916)	(3916)	(3916)	(3916)	(3916)	(3916)
	[151]	[10]	[31]	[13]	[8]	[31]	[90]
	5.75 sec	0.45 sec	1.21 sec	0.55 sec	0.36 sec	0.26 sec	0.76 sec
	0.08%	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
	{11.45%}	{11.19%}	{14.69%}	{5.88%}	{4.57%}	{66.12%}	n/a
Gil262	2380	**2378**	**2378**	**2378**	**2378**	**2378**	**2378**
Opt 2378	(2382.70)	(2379.10)	(2378.20)	(2378.50)	(2378)	(2378)	(2378)
	[198]	[24]	[111]	[162]	[44]	[100]	[565/4]
	14.38 sec	1.79 sec	7.71 sec	8.62 sec	2.36 sec	1.48 sec	8.30 sec
	0.20%	**0.05%**	**0.01%**	**0.02%**	**0.00%**	**0.00%**	**0.00 %**
	{11.79%}	{11.18%}	{17.12%}	{5.37%}	{5.48%}	{81.93%}	n/a
Att532	27869.00	**27686**.00	27693.00	27705.00	27693.00	**27686**.00	**27686**.00
Opt	(27951.2)	(27720.70)	(27722.60)	(27710.2)	(27716.8)	(27701.50)	(27707.70)
27686	[282]	[59]	[317]	[302]	[174]	[567]	[371]
	60.02 sec	12.58 sec	66.65 sec	60.48 sec	30.80 sec	16.74 sec	66.45 sec

	0.96% {21.38%}	**0.13%** {17.99%}	**0.13%** {26.93%}	**0.09%** {11.29%}	**0.11%** {10.41%}	**0.06%** {191%}	**0.08%** n/a

Best solutions are shown in bold.
Opt - optimal is the best proven tour
() brackets denotes the average best tour
[] brackets denote the iteration best solution was found for ACO, for GA and MA denote tour construction.
Sec denotes the average time in seconds to find the average solution.
% is the average error percentage of all best distances.
{} denotes the average percentage error from the optimal solution without local search.

Table 22 Comparison of Algorithms with LK local search

10.6.3.1 Results

As illustrated by table 22 applying the LK heuristic greatly improves all algorithms, especially the AS and EAS. They are now on par with MMAS ACS and MA for up to 225 node problems producing 0% error rates and finding the optimal solution in less than a second on average. However some instances are being solved in 0 irritation's suggesting that the LK heuristic is solving the problems, not the algorithms. The AS is shown empirically as the worst performing and the EAS empirically being one of the best performing that is the fastest for all algorithms but not the most accurate, followed closely by MA, ACS and MMAS.

The genetic algorithm (GA) also sees a massive increase in accuracy with the LK heuristic applied, making it one of the best algorithms in fact this experiment suggest the top ranked as illustrated by Table 23. GA was the worst performing in previous experiments now improving enormously on all TSP problems i.e. (tsp225) from 66.12% to 0% and empirically finding the average solution in less than 1000 solution constructions resulting in less than a second on average to solve this problem in fact empirically the second fastest algorithm. The reason why this algorithm improved so much is the same reason as why it failed sometimes with 2-Opt and 3-Opt. To recap the chromosome in GA uses crossover from the best solutions that are provided by the LK heuristic in turn then GA creates offspring based upon these best solutions and as the LK provides good solutions for the GA thus the search space is very focused on the search space from the LK heuristic for further information see (Chapter 6).
On the problems over 250 nodes the AS and the EAS start to break down, but are still very accurate, MA, MMAS ACS and the introduced performing much better that is, they have a lower error margin overall. For problem (att532) Stützle and Dorigo

(1999) report an average of 27707.9 over 25 runs for MMAS with LK enabled this framework AntSolver reports 27710.2 over 10 runs. For (d198) they report 15780.2 for MMAS identical to AntSolver at 15780.2. And for ACS Dorigo and Gambardella (1997b) report 27718.2 and AntSolver reports 27716.8 for (att532) and they report 15781.7 for (d198) and AntSolver 15780.8. Therefore proving AntSolver is producing accurate results.

Rank	Algorithm	Total average percentage error
1	Genetic Algorithm (GA)	0.0055%
2	Mutated Ant (MA)	0.0073%
3	MinMax Ant System (MMAS)	0.010%
4	Ant Colony System (ACS)	0.012%
5	Best-Worst Ant System (BWAS)	0.014%
6	Elitist Ant System (EAS)	0.02%
7	Ant System (AS)	0.12%
Total average error % from the selected problems of table 22 that where Averaged over 10 runs.		

Table 23 Total error percentages all algorithms with LK heuristic

However by far the best algorithm for the best average solution with LK heuristic is the Genetic Algorithm (GA) with an overall error margin error margin of 0.0055%, for the Ant algorithms it is Mutated Ant (MA) with 0.0073%, MinMax Ant System at 0.010%, Ant Colony System at 0.012%. Literature states that ACS and MMAS are the best preforming ant algorithms followed by BWAS, EAS and AS and this is verified with this experiment as illustrated by table 23 (Dorigo et al., 2006a).

Empirically the fastest algorithms for finding the average best solution is the Elitist Ant System, although not the most accurate ranked 6, followed closely by the genetic algorithm ranked 1 , then ACS followed by BWAS , MA , MMAS and the slowest being AS.

10.6.3.2 Notes

However, as illustrated by Table 22, there are virtually zero percentage error margins, and the LK heuristic is proven here to be a very powerful heuristic. So the question is and are the ants actually learning useful solutions? Later an experiment with Random

ant will analyse this very question, in which the experiment feeds the LK heuristic with randomly constructed solutions. See this chapter “Random Ant Experiments”.

10.6.4 Disadvantages of local search

As illustrated by previous experiments, applying local search can greatly improve most of the time, the algorithms performance in finding the optimal solution. (See Opt-2 experiment for times that local search is a hindrance). However the disadvantage of applying local search is at the expense of adding additional runtime complexity to each algorithm, the complexity added for each type of local search (as discussed in Chapter 7). The additional complexity is illustrated by each experiment in Table 20, 21 and 22; it can be shown empirically that when compared to Table 19 without local search the average run time of each algorithm increases. For example AS without local search finds its average best solution for (bier127) problem at an average iteration of 212 which evaluates to approximately 1.65 seconds on the experimental machine. However with 2-Opt local search enabled AS find its average best average solution at iteration 258, taking approximately 3 seconds empirically, thus taking approximately 1½ times longer empirically, the same can be seen for the other local search methods (3-Opt, LK) as defined by their complexity in Chapter 7 and proven experimentally in this section.

So in conclusion, it can be stated that applying local search can greatly improve the algorithms optimal solution when applied to TSP, but at the expense of additional run-time complexity. The best local search to implement although the most complex is the LK heuristic followed by 3-Opt and 2-Opt being the least complex and providing the least improvement.

10.7 Miscellaneous Experiments

The miscellaneous experiments have been designed to illustrate to the reader various parts of the algorithms and experiments have been run on λ Branching Factor and the pheromone matrix. In addition Random Ant and the brute force algorithm are evaluated, to convey to the reader stagnation and computational complexity respectively.

10.7.1 $\overline{\lambda}$ Branching Factor

The average branching factor is discussed in Chapter 5; now experiments will be executed on each algorithm to show this value, too verify the parameters defined in previous experiments, as all th1e algorithms do not execute excessive exploration. The higher this value the more the ants explore the search space. It represents the average number of vertices that have a high probability of being chosen by an ant. Again, this test was executed with the parameters as shown in 18, and this time only one problem instance (bier127.tsp) was selected, and again, the results were averaged over five runs and 5,000 tours.

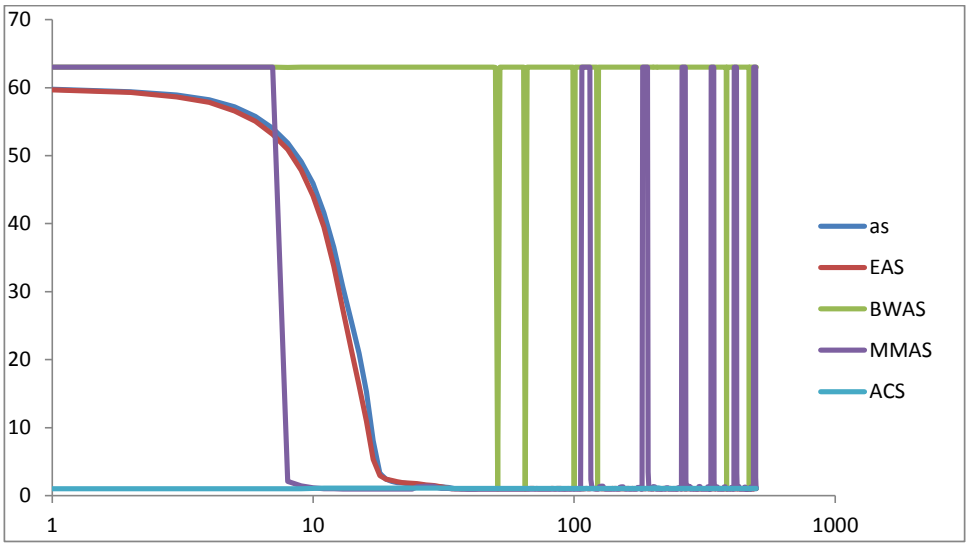


Figure 10.5 λ Branching Factor

10.7.1.1 Results

As illustrated by figure 10.5, EAS and AS, branching factor falls to a range between 1.1 and 1.15 by iteration 20, as literature states meaning they will not search the entire search space, and stagnate. MMAS also stagnates quickly but the pheromone reset is initialised when the algorithm detects stagnation and resets the pheromone matrix hence the rise and fall. BWAS is always in an exploration stage and when the algorithm detects stagnation by its technique (By detecting the change in the number of node from the worst and best solution), Chapter 5. The pheromone matrix is reinitialised. MA and ACS, the average branching factor is always low between (1-1.2)

due to that these systems do not use the probability proportional rule; therefore the branching factor is not so important.

10.7.2 Pheromone matrix

This experiment has been executed on (berlin52) TSP problem instance on both AS and ACS. A snapshot of the pheromone matrix was taken on initialisation, and after both algorithms found the optimal solution and a snapshot of the matrix has been plotted. This experiment has been designed to give the reader a visual impression of the pheromone matrix.

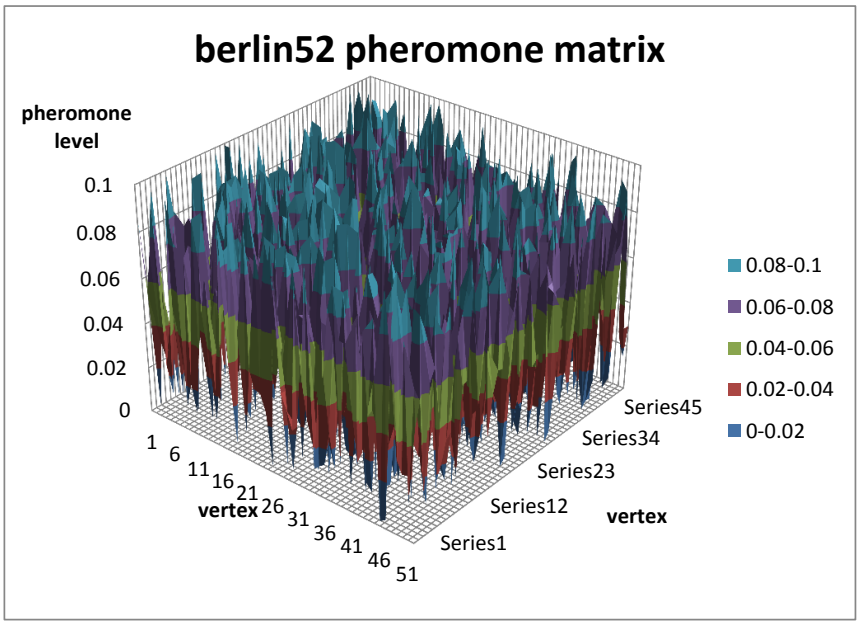


Figure 10.6 Berlin52 AS initial matrix

As illustrated in figure 32, during initialisation the original AS discussed in chapter 4, initialises the pheromone matrix with random values range from 0.0 to 0.1 for (berlin52) as the literature states. After the algorithm AS finds the best solution after 782 iterations, figure 33 plots a visual representation of the matrix at this stage.

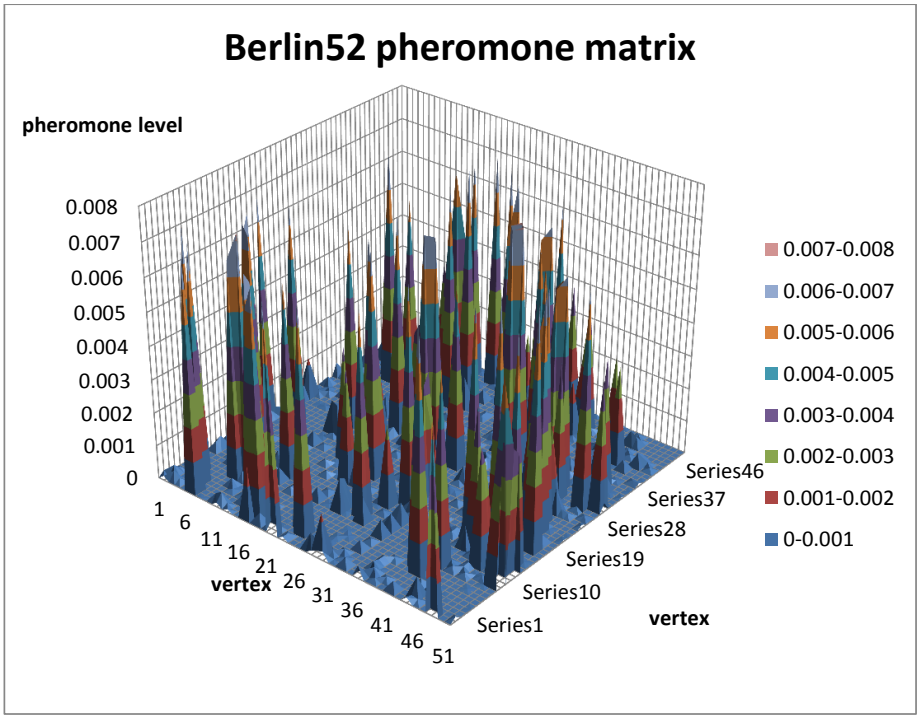


Figure 10.7 AS Berlin52 pheromone matrix after finding optimal solution

As illustrated by Figure 10.7 most vertices are reduced to a probability of almost 0, and other vertices show high probability of being selected, allowing AS finding the optimal solution.

In contrast to AS, ACS uses a completely different pheromone initialisation (as discussed Chapter 5) that is all vertices in the pheromone matrix are initialised with the nearest neighbour algorithm producing a pheromone matrix as shown in Figure 10.8. ACS solved the optimal solution in 22 iterations and the pheromone matrix at this stage is as shown in Figure 10.9. This matrix is much more defined than AS as shown most vertices still have a small probability of being selected, and the preferred edges are only slightly greater than the less preferred edges, allowing the algorithm to search for better solutions.

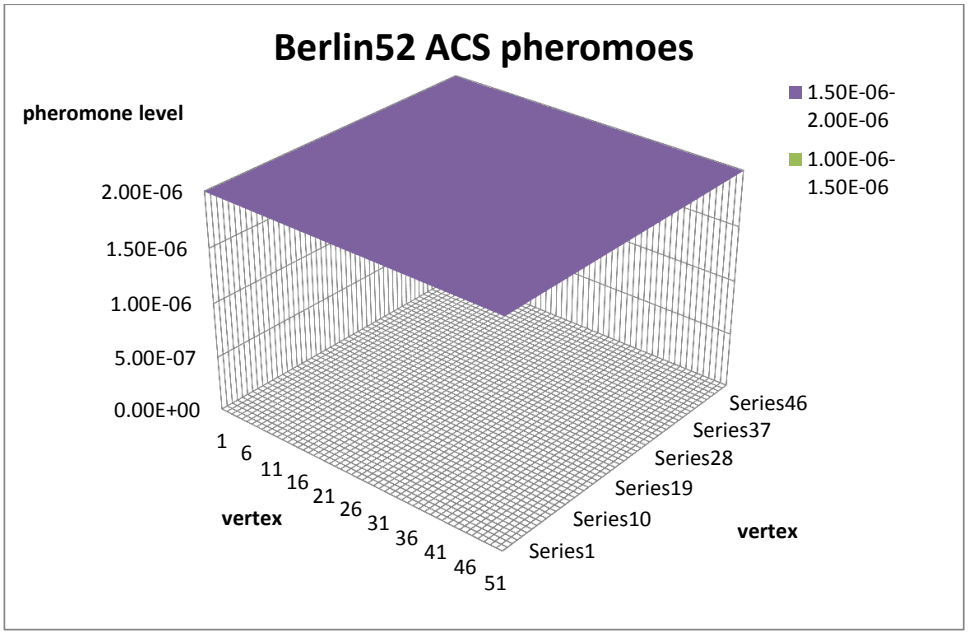


Figure 10.8 ACS berlin52 initialisation pheromone matrix

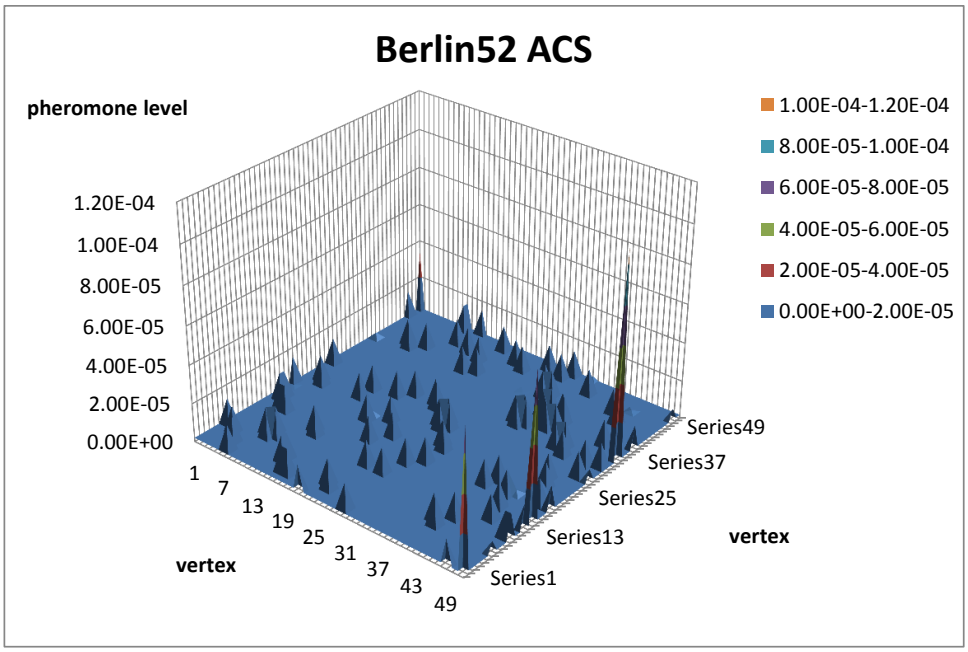


Figure 10.9 ACS berlin52 pheromone matrix after finding the optimal solution

10.7.2 Brute Force experiment

In this experiment, the brute force system (BF) is empirically compared with MMAS on small instances of TSP. To create the small TSP files, nodes were deleted from eli51 that is for the 10 city problem (eil51) was truncated at node 10 e.c.t. This is the same method that researchers use to create dynamic TSP files (Lissovoi and Witt, n.d.). MMAS was set to the optimal values discovered in previous experiments and executed with no local search; the optimal value was calculated by the Concorde TSP solver see (Chapter 2).

Algorithm	MMAS	BF	Optimal
10 City TSP	0.00047 seconds	0.12 seconds	**159**
11 City TSP	0.0049 seconds	0.15 seconds	**167**
12 City TSP	0.00224 seconds	1.77 seconds	**169**
13 city TSP	0.00324 seconds	21.84 seconds	**190**
14 City TSP	0.00217 seconds	292 seconds	**191**
15 City TSP	0.00345 seconds	4340 seconds	**208**
Time Taken in seconds to solve to optimal. Optimal found by Concorde TSP solver			

Table 24 Brute Force and MinMax Ant System

10.7.2.1 Results

Table 24 clearly shows the exponential growth of time in the brute force algorithm in solving NP-Hard problems rendering it impractical for problems bigger than 13 nodes, whereas MMAS is clearly superior as 3.5ms compared to 4340 seconds to solve a 15 city problem to optimal.

10.7.3 Random Ant Experiments

When executing the LK experiments, a question arose such that, if the LK could bring all ant algorithms to their optimal, greatly improving their solutions was there any need for ants at all? To experiment, the Random Ant algorithm was created. As discussed in Chapter 7, Random Ant randomly constructs tours and sends them to the LK heuristic for improvement. This experiment shows that this is very powerful method for low values of n but fails as n gets bigger as the algorithm gets stuck in a local optimum that it cannot break from. This was compared to MMAS with heuristics values set as illustrated by Table 18 with LK local search. Both allowed 10,000 tours over 5 runs.

Instance	MMAS	RA	Optimal
Berlin52.tsp	**7542** [5] 0%	**7542** [5] 0%	7542

Eil51.tsp	**426** [5] 0%	**426** [5] 0%	426
Eil101.tsp	**629** [5] 0%	**629** [5] 0%	629
Att532.tsp	27705 [1]	28087 [0]	202339
Tsp225.tsp	**3916** [5] 0%	3921 [0]	3916
Rat575.tsp	6773 [2]	6925 [0] (2.24%)	6773

Average Best solution over 5 averages.
Optimal solutions are shown in bold.
[] denote how many times the algorithm produced the optimal solution.
% average error percentage from optimal.

Table 25 Comparison between MMAS and RA

10.7.3.1 Results

As shown in Table 25 RA can produce optimal solutions for low values of n, however on the bigger problems it gets stuck in a local optimum that it cannot break from. This can be seen for both att532 and rat575 in that it gets to 2% of the optimal (the values were the same for each run). In comparison, MMAS if given enough time will get to the optimal solution. This was achieved a number of times in this experiment, as MMAS does not get stuck in an optimal. Random Ant was the inspiration on which Mutated Ant was developed.

10.8 Evaluating MMAS ACS and MA

This experiment has been designed to evaluate the best performing ant algorithm for really large problem instance that is to find the best average solution. This experiment will be executed on the 8246 towns and cities of Ireland (ei8246) in which the optimal solution is unknown, plus other large instances of TSP as defined by Table 26, with the best ant algorithms as found by previous experiments MMAS, ACS and MA. Each algorithm for problems (p654, d1291 vm1748) is allowed 10,000 solution constructions or 500 seconds (ei8246) was run for 1000 sec or 1000 solution constructions whichever comes first averaged over 10 runs. However with (usa13509) each algorithm was allowed 2400 seconds or 1000 tours constructs over 5 averages. The best average solution calculated and the error percentage if known was calculated. Again, the parameters evaluated in this Chapter Table 18 were used, and the LK heuristic was enabled.

Instance	MMAS	ACS	MA	Optimal
p654	**34643** (34661.5) [191] 52 sec 0.053%	**34643** **(34647.8)** [137] 30 sec **0.013%**	**34643** (34649.8) [588/3] 46 sec 0.019%	34643
d1291	50803 **(50852.4)** [194] 280 sec **0.101%**	**50801** (50890.1) [171] 192 sec 0.17%	**50801** (50908.2) [600] 225 sec 0.211%	50801
vm1748	337229 (337961.90) [156] 354 sec 0.439%	**336847** (337385.8) [280] 457 sec 0.2465%	**336847** **(337217.9)** [665/3] 382 sec **0.196%**	336847
ei8246	215957 (216936.4) [11] 592 sec 2.45%	212012 (212526.4) [26] 991 sec 0.375%	**211732** **(212305)** [68] 996 sec **0.27%**	Unknown 211732 by MA
usa13509	20894400 (21034860) [14.6] n/a 5.265%	20617100 (20645839.8) [22.8] 2343 sec 3.174%	**20605100** **(20615320)** [63] 2390 sec **3.165%**	19982859

Best solutions found in all averages. Best or optimal is shown in bold
Optimal - is the best proven tour
() brackets denotes the average best tour
[] brackets denote the iteration best solution was found for MMAS and ACS, for MA denotes tour construction.
Sec denotes the average time in seconds to find the average solution.
% is the average error percentage of all best distances.
n/a denotes invalid result

Table 26 Comparison of MA, ACS and MMAS on large instances

10.8.1 Results

As illustrated by Table 26 all algorithms perform well on (p654) problem with all finding the optimal solution and all having a very low error margin, however ACS preforms slightly better at a 0.013% error, MA at close second at 0.019% error and MMAS with an error percentage of 0.054%. Empirically ACS finds its average solution quicker at an average of 30 seconds, MA second at 43 seconds and MMAS at 52 sec.

For problem (d1291) MMAS performed slightly better and found the best average solution at an 0.101% error although it never finds the optimal, unlike MA and ACS

that do although just once each not illustrated by this table. ACS for this problem had a 0.17% error and MA at an error percentage of 0.211%. Empirically ACS was fasted again at 192 seconds, MA at 225 seconds and MMAS at 280 seconds.

For problem (vm1748) MA is slightly better performing finding the best average solution at 0.196% error and ACS second at 0.2465% error and both these algorithms found the optimal solution once. Empirically MMAS found it average solution much faster although not as accurate at 354 seconds, then MA at 382 seconds, then in third empirically ACS at 457 seconds.

However, for the big problems (ei8246) MA is clearly better performing with an average solution better than the best solution recorded by MMAS, and its average solution and best solution much better than ACS. However MMAS empirically performed the fasted at 592 seconds due to it stagnating earlier and both MA and ACS still producing good results right up to the 1000 seconds mark.

On the enormous problem (usa13509) that takes four years to solve optimally (Chapter 2). All algorithms where allowed 2400 seconds each, MA clearly found the best distance with its best distance being 3.165% from the optimal and its average solution being better than MMAS and ACS best solution.

Table 27 Shows experimentally MA performs best with these problem instances with an average error of 0.77%, ACS 2nd with an average error of 0.7957%, However not much between them.

Rank	Algorithm	Average error %
1	Mutated Ant	0.77%
2	Ant Colony System	0.7957%
3	MinMax Ant System	1.5206%

Average error %; sum of all errors from Table 26 divided by the number of problems.
As the optimal solution is not known for ei2462, the best found solution found was used at 211732

Table 27 Best Ranked Ant Algorithms average best solution

10.9. Evaluating AntSolver with ACOLIB

This experiment is designed to run the instances from the previous experiment on a completely different framework as the results from the previous section may look bias towards Mutated Ant and to ensure AntSolver functions correctly. Thomas Stützle inventor of MMAS and one of the main researchers in ACO, C framework ACOTSP was used and can be downloaded from http://www.aco-metaheuristic.org/. However to get this to compare with our framework as it does not count tour constructions, it uses iterations, so the number of iterations was set to 400 to recap at each iteration the program constructs (irreations*noofAnts) tour constructs, and as the number of ants in this framework is set to 25. Therefore 400*25 Ants is equal to 10000 tour constructs. The empirical evaluation has been disabled for this experiment as there is no direct way to test empirically over programming languages and operating systems (as this framework is built GNU C and running under CentOS 6.1 in a virtual machine running Virtualbox 4.3.10). In ACOTSP framework all values are set to the same as the previous experiment that is α=1, β=4, p = 0.3, q0=0.9 and xi=0.3 and LK local search enabled allowed to run 100000 tour constructs or 500 seconds over an average of 10 runs. However problem instance vm1748 took longer than 500 seconds to construct its 10000 tours averaged around 9000 in the allocated time. These evaluations were executed on MMAS and ACS for the larger TSP instances ones that created an average error % in AntSolver. According to

--ants 25 --time 500 -s 400 --tries 1 -a 1 -b 4 -e 0.3 -i att532.tsp –l 3 --ALGORATHM

Table 28 Command line parameters for ACOTSP

Instance	ACS	MMAS
d198	15781 **(15780.5)** {15780.8}	15781 (15781) **{15780.2}**
tsp225	n/a couldn't open file	n/a couldn't open file
gil262	2378 (2379.2) [13.5] **{2378}**	2378 **(2378.1)** [161] {2378.5}
att532	27693 **(27715.3)**	27686 **(27705.8)**

	[47] {27716.8}	[46] {27710.2}
vm1748	337037 (337682.2) [318.8] **{337385.8}**	338833 (339650.1) [104.4] **{338035.2}**
ei8246	n/a ran out of memory	n/a ran out of memory
usa13509	n/a ran out of memory	n/a ran out of memory

Minimum distance found.
Bold represents better averages.
() denotes ACOTSP average best solution over 10 runs.
[] ACOTSP average iteration over 10 runs for best solution.
{} denotes AntSolvers average best solution over 10 runs.
% difference between AntSolver and ACOTSP.
n/a denotes ACOLIB crashed and was unable to process.

Table 29 Evaluation of ACOTSP

10.9.1 Results

As illustrated by Table 29 Thomas Stützle ACOLIB, produces almost identical results for each problem instance as AntSolver, however ACS solutions in AntSolver are slightly better this could be for reasons discussed in Chapter 8 notes. However the limitation discussed shouldn't affect the results that much as LK heuristic is enabled. As illustrated for (d198) both antSolver and ACOLIB produce almost the same averages and this is also the same results by Stützle and Dorigo (1999) they report 15780.2 for their average for (d198) and AntSolver reports the same. AntSolver for the (gil262) problem produces optimal solutions for 10 out of 10 runs giving a 0% error, whereas ACOLIB has an average of (2379.2) producing 9 out of 10 optimal solutions a 0.046% error. However this is a very small difference. For problem (att532) AntSolver gives an average of 0.085% error and ACOLIB 0.081% so we can say both benchmarks are identical. Stützle own results for MMAS for this problem is 0.81% error at an average of 27911 (Stützle and Hoos, 2000). For MMAS on (gil262) AntSolver produces 0.02% error and ACOLIB 0.01% error almost identical again. However for problem (vm1748) AntSolver reported better results overall. As illustrated with these results we can see the frameworks are almost identical and any differences can be accounted for by the random nature of each algorithm.

10.10 Conclusion

Based upon the results of the experiments evaluated in this chapter obtained are the final results of this dissertation for each algorithm. Concluded for these experiments are that the guiding heuristics α, β optimal values between [α, β] = [1, 2] -> [1, 8], ρ=0.3, w = 5, xi=0.1 and q0=0.90 and the number of ants as defined as shown in Table 16. These results compare with literature in fact identically however most literature suggests [α, β] = [1, 2] to [α, β] -> [1, 5].
With these parameters, the experiment continued to prove the best overall algorithm without local search and was found that both the MMAS and ACS performed the best finding better overall average solutions, with BWAS, EAS and AS following behind. In addition it was shown that all the ant algorithms empirical performance was more or less identical for each of the problem instances. However, the genetic algorithm performance both with best average solutions and empirically was far behind the ant systems, especially on problems instances greater than 100 nodes.

In addition presented in this chapter, applying each algorithm with a local search and from the results obtained, that local search improves the best solution found by each algorithm and 2-Opt providing the least improvement, 3-Opt moderate improvement, and the LK heuristic greatly improving. Local search adds additional complexity to each algorithm. In fact the LK heuristic improves the solution greatly and enables AS and EAS to be excellent contenders for problem instances up to 250 nodes; however at this stage they start to breakdown with MMAS, ACS and MA proving to be the most robust of all the ant algorithms and produced excellent results for a large TSP problem instances as illustrated.

In addition, miscellaneous experiments are presented that provide the reader with additional insights into various ant algorithms visually illustrating the branching factor and the pheromone matrix.

Each ant algorithm was evaluated with the genetic algorithm, showing that each ant algorithm outperforms the basic genetic algorithm, both empirically and with best average solutions. However with LK enabled the genetic algorithm ranked first for average best solutions and second empirically.

In this chapter it can be concluded that exact methods cannot even cope with a 16 node TSP instance, whereas the meta-heuristics algorithm provided excellent results.

Further in this chapter was an experiment to discover best overall ant algorithm, using large instances of TSP problems and was found that MA preformed best that is finding the best average solutions then ACS, and slightly behind both was MMAS. Empirically MMAS performed better, due to the algorithm stagnating early thus producing a greater error percentage. ACS and MA were more or less equivalent empirically.

In the last experiment AntSolver was evaluated with ACOLIB a different framework from the inventor of MMAS and it was found AntSolver compares well, proving that AntSolvers ACS and MMAS algorithms preform almost identical to that of other researchers.

11 CONCLUSION

11.1 Introduction

In recent years there has been a significant impact of biosciences on computer technologies, using biological principles to improve technology. The generic algorithm has its origins in bioscience as it is based on the principle of evolution. Ant Colony Optimisation (ACO) belongs to swarm intelligence which within itself is a bioscience. Each algorithm is based on the self-organisational characteristics of the ant, and over 20 years has moved from a novel research subject to an extremely important field in the theory of optimisation.

Comment [LP27]: Is this correct?

11.2 Research Definition & Research Overview

This dissertation has shown how to solve combinatorial optimisation problems with the help of the Travelling Salesperson Problem. TSP was chosen because it is a very difficult NP problem; it has been particularly well studied by researchers, but is easy to understand. It is a traditional benchmarking problem for combinational optimisation methods and a large library of travelling salesperson problems and methods (including solutions) exists making it possible to compare efficiency of the ant algorithms with other approaches.

Basic techniques for improving the ACO algorithms such as the elitist ant, and local search, trail smoothing techniques and mutation, among others, have been discussed. Newer incarnations of the algorithms the Ant Colony System, MinMax Ant System and the Best Worst Ant System have been analysed. This dissertation also introduces a new ant algorithm, called the Mutated Ant that uses various features of each ant system and the genetic algorithm. Both the genetic algorithm and the ant algorithms can be applied to all optimisation problems that can be reduced to searching for the shortest path on a graph with constraints. Routes are selected by the algorithms on the graph by the probability rule.

The Ant algorithm ensures a good balance between the solution accuracy and the optimisation time. As the Ant algorithms, when implemented correctly, never fall into

a local optimal, a variety of solutions can always be investigated even when local search is enabled. To demonstrate this Random Ant has been introduced such that the solutions are generated by randomness and brought to their local optimal by local search. While this can be good for low values of n, it is very poor for large as experiments have shown.

11.3 Contributions to the Body of Knowledge

In this dissertation, five ant algorithms, the genetic algorithm, meta-heuristics from literature, and two introduced algorithms have been applied to the Travelling Salesperson Problem demonstrated by a framework called AntSolver written in C++.

Based upon these algorithms, experiments were executed comparing the heuristics that guide them, and comparing each algorithm side by side. Showing that ACS, MMAS were the best performing followed closely by BWAS, EAS and AS being the worst performing. These algorithms were then compared to other approaches. That is the generic algorithm and a brute force algorithm, the genetic algorithm compared well with the basic Ant algorithm (AS) up to 100 nodes after this, AS was proven to be superior. The brute force algorithm is only practical for very small problem instances as proven.

Furthermore hybrid approaches were analysed by applying local search methods. The research also applied the Lin-Kernighan heuristic (LK) to the elitist ant system and ant system, a technique not discovered in any literature; these hybrid algorithms produced excellent results and greatly improved them, allowing them to find the optimal solution for all problems. However, they were out-performed both empirically and by solutions constructions by the other algorithms described next when they were also enabled with the LK heuristic.

It was discovered that the LK heuristic was by far the best local search, and asked the question were the ants doing anything or was it all the LK contribution? Random Ant was introduced to answer this question with local search applied to random solutions and they get stuck in a local optimal over 200 nodes. However Random Ant provides

excellent solutions solving small TSP instances up to 100 cities. It was noted that the heuristic values make little difference when the LK heuristic is enabled. and the inclusion of opt-2 and opt-3 did not make very much difference sometimes proven even detrimental.

The overall experiments suggest that the introduced Mutated Ant System (MA), Ant Colony System (ACS), MinMax Ant System (MMAS), and when combined with LK produce the best results for the Travelling Salesperson Problem respectively.

However it is the view of the author that adding other heuristics takes away from the ants themselves and it was observed that applying the LK heuristic was like forcing the "*ants down a pipe*" as my supervisor Mark Foley stated. This is an excellent analogy, as all that is happening is that the ants are learning good solutions for the heuristic, therefore making it more difficult to adapt the algorithms to other problems as the algorithms become more tightly coupled with the problem at hand.

In addition, the Genetic Algorithm is much more complex to implement especially crossover than any of the ant algorithms, providing little benefit on its own as proved experimentally in this dissertation. However when combined with the Lin-Kernighan heuristic improves this algorithm to the best performing experimentally and ranked first. However no experiments were run by GA on the really large problems.

In addition the newly introduced Mutated Ant experimentally was proven to be the best ant algorithm and to counter act any bias AntSolver was compared with ACOLIB confirming that the frameworks produced almost identical results for MMAS and ACS. However ACOLIB crashed for a few of the problems we tested.

In addition we proved a problem in ACOLIB, discussed in Chapter 8 notes, and when tested with AntSolver ACS algorithm worked better with slightly less error percentage than ACOLIB as illustrated by table XXX.

In conclusion ant algorithms combined with local search provide excellent results solving NP-hard problems, providing fast performance, optimal solutions, and are easy to implement.

11.4 Experimentation, Evaluation and Limitation

The experimental results of this dissertation have only been evaluated on a small subset of TSP problems instances. TSP was chosen because of the following characteristics, it is a very difficult NP problem, and it has been particularly well studied by researchers. In addition TSP is easy to understand. It is a traditional benchmarking problem for combinational optimisation methods and there is a big library of Travelling Salesperson Problems and methods including solutions which make it possible to compare efficiency of the ant's algorithms with other approaches.
In this dissertation experiments were executed on large instances of TSP compared with most literature that mostly evaluates up to 50 nodes so we couldn't compare every problem with other researchers only some instances as discussed. In addition, there are lots of other NP-Hard problems the framework has not been adapted to solve. Most notably are those presented in Chapter 2 and comparing the algorithms with other NP-Hard problems would evaluate the algorithms in better detail. Literature shows that the ant algorithms provide good solutions to these other NP-Hard problems see Chapter 2 "Other problems solved by Ant Systems" for further information.

The experiments were run over 10 averages; however this is the average most literature uses, but if time allowed better precision could have been obtained running more averages. The heuristic values p (the evaporation rate) and $\tau 0$ (pheromone initial values) for the ant algorithms, and the heuristic values for the genetic algorithm have been taken from literature and not experimented within this dissertation.

In Addition AntSolver was also compared to ACOTSP to verify the validity of the presented framework.

11.5 Future Work & Research

There are many area of this dissertation that could lead to further research.

Firstly self-adaptive parameters setting would be a good experiment that is ant algorithms could use another ant system embedded within itself to self-adapt to parameters such as evaporation, alpha, beta and all other heuristics. The embedded ant system could work on improving these while the main algorithm solves the problem at hand.

It may be possible to improve the memory management of Ant algorithms to enable them to find the shortest route of the Mona Lisa file and try to beat the current records. In order to achieve this, some advanced data structures would need to be implemented as defined by Applegate et al. (2009). The author is of the opinion that the ant systems could find the shortest route quickly in a few months, especially if combined with advanced LK heuristics (the exact algorithms are using 1000+ CPU years). A related area which could be the subject of further research is parallelisation - due to their nature, the ant system could be greatly improved by running each ant in its own CPU. This could be very straight forward, and would not be a considerable challenge as only the pheromone matrix needs to be locked. Some improvements from literature been discussed in Chapter 5. One additional suggestion is to create an array of locks the same size as the pheromone matrix and simply lock the edges but not the complete matrix. Resultant ant update waiting times would be reduced by approx. (n-1). In addition, as the matrix is symmetrical it could be made bigger and by keeping the symmetrical nature, each ant could access different parts and only lock when the ant is updating, and - if the other symmetrical node is free - copy the value across. This could also be expanded upon to enable distributed ant colonies on separate computers (Ku-Mahamud and Nasir, 2010) such that each ant would be on a different distributed node, and the matrix on its own node. Like the Ant colony clustering method, this method could be designed to ensure Mona Lisa victory.

The adaptation of the framework to solve other NP-hard problems Subject to further research (like those discussed in Chapters 2 and 5) could also be subject to further research. Further, the addition of extra algorithms (such as the particle swarm

optimiser, simulated annealing, intelligent water drops, artificial bee colony, bat algorithm, gravitational search algorithm and even crowd simulation) and then the comparison of these algorithms with each other side by side could benefit from additional experimentation and research.

Various other experiments on the travelling salesperson problem could also be evaluated, as only 13 have been experimented on within this dissertation, as some algorithms work better with different problems. Furthermore adapting the framework to run on dynamic problems and asymmetrical TSP files is subject to further research (Cai, 2008b).

Finally, further investigation into and improvements of mutated Ant and its mathematical convergence proof could be a worthwhile project. It could be possible to select the mutation vertices via a nearest neighbour list instead of all vertices in the complete solution. That way the algorithm will not swap vertices that have a long distance between them. In addition another improvement that could be implemented by adding an extra ant to follow some of the best so far path and some of the best path in the irritations similar to crossover in the genetic algorithm this could greatly improve the algorithm.

11.6 Conclusion

A brute force algorithm was used as an argument to prove that linear methods crumble with the Travelling Salesperson Problems. Even with the most optimal code they are still *O((n-1)!/2)* and are shown to take a considerable time (4,340 seconds) to solve a simple 15 nodes problem. Adding 1 extra node will increase this to approx. 25 hours, 17 nodes will increase to 75 hours and so on. In contrast, all ant algorithms can converge to the optimal solution of the same 16-city problem in a few seconds and can solve optimally a 1,000 node problem in a few minutes using the same machine. In Addition as shown experimentally Mutated Ant performed the best of all ant algorithms and it solved the problem usa13509 that is 13509 cities to within 3% of the optimal solution in 2400 seconds half the time of the brute force algorithm to solve 15 cities. To achieve this sort of a solution is truly an impressive display of the supremacy of ants.

Proving that ants *do* find the shortest path.

BIBLIOGRAPHY

Analyzing Application Performance by Using Profiling Tools [WWW Document], n.d. URL http://msdn.microsoft.com/en-us/library/z9z62c29.aspx (accessed 2.15.14).

Andy, n.d. Yet Another HeartBleed. Pentura Labss Blog.

Applegate, D., Bixby, R., Cook, W., Chvátal, V., 1998. On the solution of traveling salesman problems. Rheinische Friedrich-Wilhelms-Universität Bonn.

Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W., Espinoza, D.G., Goycoolea, M., Helsgaun, K., 2009. Certification of an optimal TSP tour through 85,900 cities. Oper. Res. Lett. 37, 11–15. doi:10.1016/j.orl.2008.09.006

Balaprakash, P., Birattari, M., Stützle, T., Dorigo, M., 2006. Incremental local search in ant colony optimization: Why it fails for the quadratic assignment problem, in: Ant Colony Optimization and Swarm Intelligence. Springer, pp. 156–166.

Banerjee, S., El-Bendary, N., ella Hassanien, A., Kim, T., 2011. A modified pheromone dominant ant colony algorithm for computer virus detection, in: Multitopic Conference (INMIC), 2011 IEEE 14th International. Presented at the Multitopic Conference (INMIC), 2011 IEEE 14th International, pp. 35–40. doi:10.1109/INMIC.2011.6151503

Beck, K., 2002. Test Driven Development: By Example, 1 edition. ed. Addison-Wesley Professional, Boston.

Beni, G., Wang, J., 1993. Swarm Intelligence in Cellular Robotic Systems, in: Dario, P., Sandini, G., Aebischer, P. (Eds.), Robots and Biological Systems: Towards a New Bionics?, NATO ASI Series. Springer Berlin Heidelberg, pp. 703–712.

Beyer, K., Goldstein, J., Ramakrishnan, R., Shaft, U., 1999. When is "nearest neighbor" meaningful?, in: Database Theory—ICDT'99. Springer, pp. 217–235.

Birattari, M., Pellegrini, P., Dorigo, M., 2007. On the Invariance of Ant Colony Optimization. IEEE Trans. Evol. Comput. 11, 732–742. doi:10.1109/TEVC.2007.892762

Bonabeau, E., 1999. Editor's Introduction: Stigmergy. Artif. Life 5, 95–96. doi:10.1162/106454699568692

Browne, G., n.d. Sydney eScholarship Repository [WWW Document]. URL http://ses.library.usyd.edu.au/handle/2123/11091 (accessed 7.22.14).

Bullnheimer, B., Hartl, R.F., Strauss, C., 1997a. A new rank based version of the Ant System. A computational study.

Bullnheimer, B., Hartl, R.F., Strauss, C., 1997b. A new rank based version of the Ant System. A computational study.

Cai, Z., 2008a. Multi-Direction Searching Ant Colony Optimization for Traveling Salesman Problems, in: International Conference on Computational Intelligence and Security, 2008. CIS '08. Presented at the International Conference on Computational Intelligence and Security, 2008. CIS '08, pp. 220–223. doi:10.1109/CIS.2008.151

Cai, Z., 2008b. Multi-Direction Searching Ant Colony Optimization for Traveling Salesman Problems, in: International Conference on Computational Intelligence and Security, 2008. CIS '08. Presented at the International Conference on Computational Intelligence and Security, 2008. CIS '08, pp. 220–223. doi:10.1109/CIS.2008.151

Chaiyaratana, N., Zalzala, A.M.S., 1997. Recent developments in evolutionary and genetic algorithms: theory and applications, in: Genetic Algorithms in Engineering Systems: Innovations and Applications, 1997. GALESIA 97. Second International Conference On (Conf. Publ. No. 446). Presented at the Genetic Algorithms in Engineering Systems: Innovations and Applications, 1997. GALESIA 97. Second International Conference On (Conf. Publ. No. 446), pp. 270–277. doi:10.1049/cp:19971192

Chen, L., Sun, H.-Y., Wang, S., 2008. Parallel implementation of ant colony optimization on MPP, in: 2008 International Conference on Machine Learning and Cybernetics. Presented at the 2008 International Conference on Machine Learning and Cybernetics, pp. 981–986. doi:10.1109/ICMLC.2008.4620547

Clark, A., 1997. Being There: Putting Brain, Body, and World Together Again. MIT Press.

Coleman, C.M., Rothwell, E.J., Ross, J.E., 2004. Investigation of Simulated annealing, ant-colony optimization, and genetic algorithms for self-structuring antennas. IEEE Trans. Antennas Propag. 52, 1007–1014. doi:10.1109/TAP.2004.825658

Cook, W., n.d. Concorde Home [WWW Document]. URL http://www.math.uwaterloo.ca/tsp/concorde/index.html (accessed 5.3.14).

Cordón, O., de Viana, I.F., Herrera, F., 2002. Analysis of the best-worst Ant System and its variants on the TSP. Mathw. Soft Comput. 9, 177–192.

Cordón, O., de Viana, I.F., Herrera, F., Moreno, L., 2000. A new ACO model integrating evolutionary computation concepts: The best-worst Ant System.

Cordon, O., Herrera, F., Stützle, T., 2002. A Review on the Ant Colony Optimization Metaheuristic: Basis, Models and New Trends. Mathw. Soft Comput. 9, 141–175.

Donald, D., 2011. Traveling Salesman Problem, Theory and Applications.

Dorigo, M., Birattari, M., Stutzle, T., 2006a. Ant colony optimization. IEEE Comput. Intell. Mag. 1, 28–39. doi:10.1109/MCI.2006.329691

Dorigo, M., Birattari, M., Stutzle, T., 2006b. Ant colony optimization. IEEE Comput. Intell. Mag. 1, 28–39. doi:10.1109/MCI.2006.329691

Dorigo, M., Birattari, M., Stutzle, T., 2006c. Ant colony optimization. IEEE Comput. Intell. Mag. 1, 28–39. doi:10.1109/MCI.2006.329691

Dorigo, M., Di Caro, G., 1999. Ant colony optimization: a new meta-heuristic, in: Proceedings of the 1999 Congress on Evolutionary Computation, 1999. CEC 99. Presented at the Proceedings of the 1999 Congress on Evolutionary Computation, 1999. CEC 99, p. -1477 Vol. 2. doi:10.1109/CEC.1999.782657

Dorigo, M., Gambardella, L.M., 1997a. Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Trans. Evol. Comput. 1, 53–66. doi:10.1109/4235.585892

Dorigo, M., Gambardella, L.M., 1997b. Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Trans. Evol. Comput. 1, 53–66. doi:10.1109/4235.585892

Dorigo, M., Maniezzo, V., Colorni, A., 1996. Ant system: optimization by a colony of cooperating agents. IEEE Trans. Syst. Man Cybern. Part B Cybern. 26, 29–41. doi:10.1109/3477.484436

Dorigo, M., Socha, K., 2006a. An introduction to ant colony optimization.

Dorigo, M., Socha, K., 2006b. An introduction to ant colony optimization.

Dorigo, M., Stützle, T., 2004a. Ant Colony Optimization. MIT Press.

Dorigo, M., Stützle, T., 2004b. Ant Colony Optimization. MIT Press.

Dr. Dobb's | Good stuff for serious developers: Programming Tools, Code, C++, Java, HTML5, Cloud, Mobile, Testing [WWW Document], n.d. Dr Dobbs. URL http://www.drdobbs.com/parallel/ant-colony-algorithms/parallel/sourcecode/ant-colony-algorithms/30100184 (accessed 4.11.14).

Fejzagic, E., Oputic, A., 2013. Performance comparison of sequential and parallel execution of the Ant Colony Optimization algorithm for solving the traveling salesman problem, in: 2013 36th International Convention on Information Communication Technology Electronics Microelectronics (MIPRO). Presented at the 2013 36th International Convention on Information Communication Technology Electronics Microelectronics (MIPRO), pp. 1301–1305.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., more, & 3, 1999. Refactoring: Improving the Design of Existing Code, 1 edition. ed. Addison-Wesley Professional, Reading, MA.

Gamma, E., 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading, Mass.

Geetha, R.R., Bouvanasilan, N., Seenuvasan, V., 2009. A perspective view on Travelling Salesman Problem using genetic algorithm, in: World Congress on Nature Biologically Inspired Computing, 2009. NaBIC 2009. Presented at the World Congress on Nature Biologically Inspired Computing, 2009. NaBIC 2009, pp. 356–361. doi:10.1109/NABIC.2009.5393321

Gerhard Reinelt, R., n.d. TSPLIB.

Glover, F., 1989. Tabu Search—Part I. ORSA J. Comput. 1, 190–206. doi:10.1287/ijoc.1.3.190

Goss, S., Aron, S., Deneubourg, J.L., Pasteels, J.M., 1989. Self-organized shortcuts in the Argentine ant. Naturwissenschaften 76, 579–581. doi:10.1007/BF00462870

Hardygóra, M., Paszkowska, G., Sikora, M., 2004. Mine Planning and Equipment Selection 2004: Proceedings of the Thirteenth International Symposium on Mine Planning and Equipment Selection, Wroclaw, Poland, 1-3 September 2004. CRC Press.

Hölldobler, B., 1990. The Ants. Harvard University Press.

Huang, H., Wu, C.-G., Hao, Z.-F., 2009. A Pheromone-Rate-Based Analysis on the Convergence Time of ACO Algorithm. IEEE Trans. Syst. Man Cybern. Part B Cybern. 39, 910–923. doi:10.1109/TSMCB.2009.2012867

Huaxin, M., Shuai, J., 2011. Design patterns in software development, in: 2011 IEEE 2nd International Conference on Software Engineering and Service Science (ICSESS). Presented at the 2011 IEEE 2nd International Conference on Software Engineering and Service Science (ICSESS), pp. 322–325. doi:10.1109/ICSESS.2011.5982228

Johnson, D.S., 1990. Local optimization and the Traveling Salesman Problem, in: Paterson, M.S. (Ed.), Automata, Languages and Programming, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 446–461.

Johnson, D.S., McGeoch, L.A., 1997. The traveling salesman problem: A case study in local optimization. Local Search Comb. Optim. 1, 215–310.

Johnson, D.S., Papadimitriou, C.H., Yannakakis, M., 1988. How easy is local search? J. Comput. Syst. Sci. 37, 79–100. doi:10.1016/0022-0000(88)90046-3

Karp, R.M., 1977. Probabilistic Analysis of Partitioning Algorithms for the Traveling-Salesman Problem in the Plane. Math. Oper. Res. 2, 209–224. doi:10.1287/moor.2.3.209

Khaer, M.A., Hashem, M.M.A., Masud, M.R., 2007. An empirical analysis of software systems for measurement of design quality level based on design patterns, in: 10th International Conference on Computer and Information Technology, 2007. Iccit 2007. Presented at the 10th international conference on Computer and information technology, 2007. iccit 2007, pp. 1–6. doi:10.1109/ICCITECHN.2007.4579432

Khan, S., Bilal, M., Sharif, M., Sajid, M., Baig, R., 2009. Solution of n-Queen problem using ACO, in: Multitopic Conference, 2009. INMIC 2009. IEEE 13th International. Presented at the Multitopic Conference, 2009. INMIC 2009. IEEE 13th International, pp. 1–5. doi:10.1109/INMIC.2009.5383157

Ku-Mahamud, K.R., Nasir, H.J.., 2010. Ant Colony Algorithm for Job Scheduling in Grid Computing, in: 2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation (AMS). Presented at the 2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation (AMS), pp. 40–45. doi:10.1109/AMS.2010.21

Li, C., Yang, S., Pelta, D.A., 2011. Benchmark generator for the IEEE WCCI-2012 competition on evolutionary computation for dynamic optimization problems. China Univ. Geosci. Brunel Univ. Univ. Granada Tech Rep.

Lin, S., Kernighan, B.W., 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. Oper. Res. 21, 498–516. doi:10.1287/opre.21.2.498

Lissovoi, A., Witt, C., 2013. Runtime Analysis of Ant Colony Optimization on Dynamic Shortest Path Problems, in: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, GECCO '13. ACM, New York, NY, USA, pp. 1605–1612. doi:10.1145/2463372.2463567

Lissovoi, A., Witt, C., n.d. Runtime analysis of ant colony optimization on dynamic shortest path problems. Theor. Comput. Sci. doi:10.1016/j.tcs.2014.06.035

Louden, K.C., Lambert, K.A., 2011. Programming Languages: Principles and Practices: Principles and Practices. Cengage Learning.

Manjurul Islam, M.M., Waselul Hague Sadid, M., Mamun Ar Rashid, S.M., Kabir, M.J., 2006. An Implementation of ACO System for Solving NP-Complete Problem; TSP, in: International Conference on Electrical and Computer Engineering, 2006. ICECE '06. Presented at the International Conference on Electrical and Computer Engineering, 2006. ICECE '06, pp. 304–307. doi:10.1109/ICECE.2006.355632

Martello, S., Pisinger, D., Toth, P., 2000. New trends in exact algorithms for the 0–1 knapsack problem. Eur. J. Oper. Res. 123, 325–332. doi:10.1016/S0377-2217(99)00260-X

Meyers, S., 1995. More Effective C++: 35 New Ways to Improve Your Programs and Designs, 1st ed. Addison-Wesley Professional. Part of the Addison-Wesley Professional Computing Series series.

Micro Autonomous System Technologies (MAST) [WWW Document], n.d. URL http://www.mast-cta.org/ (accessed 5.6.14).

Milanovic, A., Duranovic, M., Nosovic, N., 2013a. Performance analysis of the traveling salesman problem optimization using ant colony algorithm and OpenMP, in: 2013 36th International Convention on Information Communication Technology Electronics Microelectronics (MIPRO). Presented at the 2013 36th International Convention on Information Communication Technology Electronics Microelectronics (MIPRO), pp. 1310–1313.

Milanovic, A., Duranovic, M., Nosovic, N., 2013b. Performance analysis of the traveling salesman problem optimization using ant colony algorithm and OpenMP, in: 2013 36th International Convention on Information Communication Technology Electronics Microelectronics (MIPRO). Presented at the 2013 36th International Convention on Information Communication Technology Electronics Microelectronics (MIPRO), pp. 1310–1313.

Mladenović, N., Hansen, P., 1997. Variable neighborhood search. Comput. Oper. Res. 24, 1097–1100.

Narasimha, K.S.V., Kivelevitch, E., Kumar, M., 2012. Ant Colony optimization technique to Solve the min-max multi depot vehicle routing problem, in: American Control Conference (ACC), 2012. Presented at the American Control Conference (ACC), 2012, pp. 3980–3985.

Ochoa, G., Harvey, I., Buxton, H., 2000. Optimal Mutation Rates and Selection Pressure in Genetic Algorithms., in: GECCO. Citeseer, pp. 315–322.

Parsons, S., 2005. Ant Colony Optimization by Marco Dorigo and Thomas Stützle, MIT Press, 305 pp., $40.00, ISBN 0-262-04219-3. Cambridge Univ Press.

Petzold, C., 1998. Programming Windows®, Fifth Edition, 5th edition. ed. Microsoft Press, Redmond, Wash.

Pour, H.M., Atlasbaf, Z., Mirzaee, A., Hakkak, M., 2008. A hybrid approach involving artificial neural network and ant colony optimization for direction of arrival estimation, in: Canadian Conference on Electrical and Computer Engineering, 2008. CCECE 2008. Presented at the Canadian Conference on Electrical and Computer Engineering, 2008. CCECE 2008, pp. 001059–001064. doi:10.1109/CCECE.2008.4564699

Prabhakar, B., Dektar, K.N., Gordon, D.M., 2012. The Regulation of Ant Colony Foraging Activity without Spatial Information. PLoS Comput Biol 8, e1002670. doi:10.1371/journal.pcbi.1002670

Prechelt, L., Unger, B., Tichy, W.F., Brossler, P., Votta, L.G., 2001. A controlled experiment in maintenance: comparing design patterns to simpler solutions. IEEE Trans. Softw. Eng. 27, 1134–1144. doi:10.1109/32.988711

Purohit, A., Sun, Z., Mokaya, F., Zhang, P., 2011. SensorFly: Controlled-mobile sensing platform for indoor emergency response applications, in: 2011 10th International Conference on Information Processing in Sensor Networks (IPSN). Presented at the 2011 10th International Conference on Information Processing in Sensor Networks (IPSN), pp. 223–234.

Reinelt, G., 1991. TSPLIB—A Traveling Salesman Problem Library. ORSA J. Comput. 3, 376–384. doi:10.1287/ijoc.3.4.376

Reynolds, C.W., 1987. Flocks, Herds and Schools: A Distributed Behavioral Model, in: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '87. ACM, New York, NY, USA, pp. 25–34. doi:10.1145/37401.37406

Sedgewick, R., 1998. Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Third Edition, 3 edition. ed. Addison-Wesley Professional, Reading, Mass.

Sim, K.M., Sun, W.H., 2003a. Ant colony optimization for routing and load-balancing: survey and new directions. IEEE Trans. Syst. Man Cybern. Part Syst. Hum. 33, 560–572. doi:10.1109/TSMCA.2003.817391

Sim, K.M., Sun, W.H., 2003b. Ant colony optimization for routing and load-balancing: survey and new directions. IEEE Trans. Syst. Man Cybern. Part Syst. Hum. 33, 560–572. doi:10.1109/TSMCA.2003.817391

SimpleXlsxWriter [WWW Document], n.d. SourceForge. URL http://sourceforge.net/projects/simplexlsx/ (accessed 8.6.14).
Song, X., Li, B., Yang, H., 2006. Improved Ant Colony Algorithm and its Applications in TSP, in: Sixth International Conference on Intelligent Systems Design and Applications, 2006. ISDA '06. Presented at the Sixth International Conference on Intelligent Systems Design and Applications, 2006. ISDA '06, pp. 1145–1148. doi:10.1109/ISDA.2006.253773
Srinivas, M., Patnaik, L.M., 1994. Genetic algorithms: a survey. Computer 27, 17–26. doi:10.1109/2.294849
StreetChicago, I.I. of T.S.F., n.d. About Karl Menger [WWW Document]. URL http://science.iit.edu/applied-mathematics/about/about-karl-menger (accessed 5.3.14).
Stützle, T., Dorigo, M., 1999. ACO algorithms for the traveling salesman problem. Evol. Algorithms Eng. Comput. Sci. 163–183.
Stutzle, T., Hoos, H., 1997. MAX-MIN Ant System and local search for the traveling salesman problem, in: , IEEE International Conference on Evolutionary Computation, 1997. Presented at the , IEEE International Conference on Evolutionary Computation, 1997, pp. 309–314. doi:10.1109/ICEC.1997.592327
Stützle, T., Hoos, H.H., 1999. Analyzing the run-time behaviour of iterated local search for the TSP, in: III Metaheuristics International Conference. Citeseer.
Stützle, T., Hoos, H.H., 2000. MAX–MIN ant system. Future Gener. Comput. Syst. 16, 889–914.
The C++ Programming Language 4th Edition [Opsylum], n.d.
TSP Gallery [WWW Document], n.d. URL http://www.math.uwaterloo.ca/tsp/gallery/index.html (accessed 5.6.14).
Visual C [WWW Document], 2014. . Wikipedia Free Encycl. URL http://en.wikipedia.org/w/index.php?title=Visual_C%2B%2B&oldid=605323774 (accessed 5.3.14).
Whitley, D., Lunacek, M., Knight, J., 2004. Ruffled by ridges: How evolutionary algorithms can fail, in: Genetic and Evolutionary Computation–GECCO 2004. Springer, pp. 294–306.
Zhang, B., Gao, J., Gao, L., Sun, X., 2013. Improvements in the Ant Colony Optimization Algorithm for Endmember Extraction From Hyperspectral Images. IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. 6, 522–530. doi:10.1109/JSTARS.2012.2236821
Zhang, C., Budgen, D., 2012. What Do We Know about the Effectiveness of Software Design Patterns? IEEE Trans. Softw. Eng. 38, 1213–1231. doi:10.1109/TSE.2011.79
Zhang, Y., Wang, H., Zhang, Y., Chen, Y., 2011. BEST-WORST Ant System, in: 2011 3rd International Conference on Advanced Computer Control (ICACC). Presented at the 2011 3rd International Conference on Advanced Computer Control (ICACC), pp. 392–395. doi:10.1109/ICACC.2011.6016438
Zhang, Z., Zhang, N., Feng, Z., 2014. Multi-satellite control resource scheduling based on ant colony optimization. Expert Syst. Appl. 41, 2816–2823.

APPENDIX A

full class diagram