Visualization and comparison of randomized search heuristics on random traveling salesman problems

Thorkil Burup

s122506

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

The traveling salesman problem is in a class of problems, that are assumed to not be computable efficiently. Using randomized search heuristics is a way of handling problems such as the traveling salesman problem. In this report, *randomized local search*, *simulated annealing*, *(1+1) evolutionary algorithm* and $\mathcal{MAX}$–$\mathcal{MIN}$ *ant system* are observed to determine their strengths and weaknesses in relation to each other, and to provide a benchmark on different random problem instances. Experiments conducted in this study show that randomized local search and (1+1) evolutionary algorithm are the most adaptable and generally applicable, but simulated annealing can perform exceptionally well if given optimal conditions. $\mathcal{MAX}$–$\mathcal{MIN}$ ant system find reasonable solutions fast, but cannot improve on them at a high rate compared to the other three. An experiment using random problem instances with vertices placed in clusters indicated that the graph structure does not affect the performance of the algorithms.

Based on data provided by the experiments, recommendations on the algorithms can be made: Randomized local search and (1+1) evolutionary algorithm are based on continuously improving their tours. If given a good initial tour (e.g. using an approximation algorithm), they are not only able to reach good solutions fast, but more surprisingly (especially for randomized local search), they are able to consistently reach better solutions. Simulated annealing works mediocre, unless it can be provided with very problem-specific parameters. In this case, it excels and is shown to consistently improve its tour cost for a problem. $\mathcal{MAX}$–$\mathcal{MIN}$ ant system is shown to rely heavily on its distance heuristic in the tested environments, making its pheromone heuristic insignificant.

The randomized search heuristics are compared to Concorde on three different problem instances. This comparison shows that the randomized search heuristics are capable of calculating solution with costs approximately 115% of optimal solutions, but can do so in half the time the state-of-the-art solver Concorde uses. The data presented in this study show that randomized search heuristic are convenient when an exact solution is not needed and when exact solvers are not practical; for instance when time or computation resources are limited.

Preface

The project was done as a Master's thesis in Computer Science at the Department of Applied Mathematics and Computer Science of the Technical University of Denmark. The project ran from September 1st 2014 to February 1st 2015, and was supervised by Associate Professor Carsten Witt.

The study is intended for computer scientists. The report consists of a literature review and an experimental part. The literature review covers the traveling salesman problem, different types of problem instances (including random instances) and four different randomized search heuristic algorithms. The experimental part compares the four algorithms on a variety of scenarios, using a program that was developed for this purpose. The study provides guidelines and results for four different randomized search heuristics.

I wish to thank Carsten Witt for his supervision and guidance during the project period, and for piquing interest in this area of computer science through his course *Computationally Hard Problems* taught at the Technical University of Denmark.

Contents

CHAPTER 1

Introduction

The essence of the traveling salesman problem is to find the shortest tour which visits a given set of cities, and returns to the starting point. The problem is very well-studied, in part because it is an $\mathcal{NP}$-hard problem and in part because its solutions can be applied to a wide range of problems related to logistics, but also other areas, such as genome sequencing, can be expressed as traveling salesman problems.

The traveling salesman problem cannot be efficiently solved to optimality, thus measures in which we find suboptimal, but still reasonable solutions in less time are practical. Randomized search heuristics such as evolutionary algorithms, simulated annealing and ant colony optimization are frequently used to provide such solutions to difficult combinatorial optimization problems, including the traveling salesman problem. The purpose of this project is to implement, visualize and experimentally compare simple randomized search heuristics on different random problem instances, both in relation to each other but also with an advanced exact solver.

In addition to this introduction, the report contains into six chapters. Chapter 2 defines the traveling salesman problem formally, and discuss variations thereof, including problem instances. The chapter also includes a section describing the problem instances used in this study, and how they are created. Chapter 3 describes a selection of randomized search heuristics. The different algorithms are accompanied by pseudocode that sketches their implementations. In chapter 4, a program developed in the project to implement the algorithms and visualize their solutions is described. Several interesting implementation details are presented in this chapter, as well as the limitations and ideas for future features. The experiments conducted in this study are presented in chapter 5. Each type of experiment is explained separately, and results of the experiments are explained in this context. In chapter 6, a discussion of the used methods and a recommendations for future work in the area are found. Chapter 7 contains the conclusions of the experiments.

CHAPTER 2

Traveling salesman problem

The *traveling salesman problem* (TSP) is one of the best-known combinatorial problems in computer science. For a salesman who has to visit certain locations during his day to sell his goods, and return home at the end of the day, the problem is to find the optimal route to do so. The following formal definitions are used for the problem:

DEFINITION 2.1 (HAMILTONIAN CYCLE) Given a graph $G = (V, E)$, a Hamiltonian cycle is a closed loop through edges in E visiting every vertex in V exactly once.

DEFINITION 2.2 (TRAVELING SALESMAN PROBLEM) Given an complete, undirected graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}$, compute the Hamiltonian cycle of minimal cost according to c.

The problem is interesting for several reasons. First, the problem has a wide range of practical applications; spanning from logistics to genome sequencing [2]. Second, the problem belongs to the set of $\mathcal{NP}$-hard problems, meaning that, unless $\mathcal{P} = \mathcal{NP}$, it cannot be solved efficiently (i.e. in polynomial time). Many have tried (and failed) to come up with efficient algorithms for TSP in order to solve the $\mathcal{P}$ versus $\mathcal{NP}$ problem which is one of the seven Millenium Prize Problems, whose solutions award a million dollars each [5]. Developing an algorithm with polynomial time-complexity for TSP would prove that $\mathcal{P} = \mathcal{NP}$. The most common opinion amongst experts is that $\mathcal{P} \neq \mathcal{NP}$. This popular claim is reflected in *The Second P=?NP Poll* by Gasarch [9], which presents expert opinions from 2002 and 2012 on the matter.

2.1 Variations of TSP

The problem described in definition 2.2 is a very common version of TSP. In this report the term TSP will use exactly that definition. Other variations of problem do exist though, and they can be formed by changing different aspects of the problem.

In definition 2.2, G is assumed to be complete, meaning that each vertex is incident on all other vertices. This assumption is not strictly necessary. As long as just one single Hamiltonian cycle exists in G, a solution meeting the criteras of TSP as defined can be found. However, determining if such a cycle exists is itself an $\mathcal{NP}$-complete problem called the *Hamiltonian cycle problem*.

The restriction that G is undirected can also be relaxed. If some edges are only traversable in one direction (e.g. one-way streets), the graph is directed. Furthermore, the problem known as *asymmetric TSP* (ATSP) allows the cost function to assign different distances between two vertices depending on the direction you travel; $c(i, j) \neq c(j, i)$ for some pair of vertices (i, j). ATSP can be transformed into TSP in order use techniques that require the graph to be symmetric. Such a transformation is proposed by Jonker and Volgenant [15], in which dummy vertices and infinite-weight edges are added to create the undirected graph that simulates the symmetry of the original graph. The resulting undirected graph contains twice as many vertices as the original asymmetric instance[1], which is a very considerable overhead for an $\mathcal{NP}$-hard problem. The transformation is mentioned because many advanced TSP solvers cannot solve ATSP [12].

Finally, the cost function c can follow different rules. Perhaps the most obvious one is *metric TSP* (MTSP), in which the edge weights map to distances and the cost function follow the triangle inequality theorem. The theorem states that it will always be at least as costly to go from a to b and then to c, as it is just going straight from a to c. More formally:

$$c(a, b) + c(b, c) \geq c(a, c) \tag{2.1}$$

[1]This number can be reduced if connections between vertices have the same weight in either direction, as we do not need to insert dummy vertices in these cases.

If you consider distances between points, the theorem makes sense, but if you weigh your graph by other means, it may not apply. If the costs associated with the edges of the graph denote travel time for instance, it may be more optimal to take a detour (in terms of distance) in order to reach your destination faster. The problem in which there are no restrictions on the cost function is called *general TSP* (GTSP).

Further reading on the topic of TSP variations can be found in *The Traveling Salesman Problem: A Computational Study* by Applegate *et al.* [2].

2.2 Solving TSP

The naive way to solve TSP is to compute every single tour and determining the best one. There are $\frac{(n-1)!}{2}$ such tours where $n \geq 3$ is the number of vertices. Unless n is very small, the amount of tours becomes such a large number that we cannot compute them all in reasonable time. The Held-Karp algorithm is an improved algorithm to find the exact solution to TSP using a dynamic programming approach. It has a time-complexity of $O(n^2 2^n)$ which is better than factorial time, but still very much unfeasible for large problems [13].

Instead of trying to solve TSP, one can instead try to *approximate* a solution or use *heuristics*, addressed in the following subsections. First, however, it should be noted that the terms *TSP tour* and *solution* are used to describe a Hamiltonian cycle, which may be misleading as a Hamiltonian cycle is not necessarily

even though the terms are not technically describing solutions to TSP (i.e. not necessarily minimized in cost). The terms are used merely for simplification, convenience and the lack of better terms.

2.2.1 Approximation

To approximate a solution to TSP is to find a Hamiltonian cycle, whose cost is not necessarily optimal. By removing the requirement optimality, it is very simple to come up with algorithms that run in polynomial time, but the problem then also becomes irrelevant – any tour through all the vertices will suffice. The essential idea behind approximation algorithms is to be able to prove just *how* much worse the solutions produced by the algorithms can be, compared to the optimal solution. The currently best known approximation algorithm for metric TSP is *Christofides' algorithm*, which was published in 1976. It is a 3/2-approximation[2] with a reasonable time-complexity of $O(n^3)$. The algorithm works by generating a minimum spanning tree over the graph, and then transforming it into a TSP tour in several steps. The transformation is performed in a way to retain a guarantee of the approximation solution size relative to the optimal solution size [4].

Since the best known approximation algorithm for MTSP can result in solutions up to 50% worse than the optimal, they might not suffice for some applications. Furthermore, GTSP cannot be approximated within polynomial time [25]. Research has shown that MTSP cannot be approximated to a ratio lower than 123/122 in polynomial time, under the assumption that $\mathcal{P} \neq \mathcal{NP}$[16]. Being the strictest lower bound found yet, this result means that optimistic computer scientists can still hope to see an improvement over Christofides' algorithm in the future.

2.2.2 Heuristics

Heuristics are alternative ways of approaching a problem in order to calculate close-to-optimal solutions in short time. A good example of a heuristic for TSP is the *nearest neighbor* heuristic; a vertex is picked as the starting point and then the tour is constructed by moving to the nearest yet-unvisited vertex. The hope is that the approach will produce a reasonable tour in the end. The heuristic most certainly is able to produce good solutions, but it may in fact produce the worst possible solution if the graph is constructed in a certain way [10]. As such, heuristics sound to be similar to approximation algorithms, without the bonus of guaranteeing an approximation factor, but heuristics should be perceived more as the idea or guidance behind such an algorithm.

To get beyond approximation algorithms, heuristics can be combined with randomization. The idea is to let a random element decide on moves in the search space, guided by the used heuristic. This is called *randomized search heuristics*. For instance, we might apply randomness to the nearest neighbor heuristic and instead of always selecting the nearest vertex, when constructing the tour, we select the next vertex at random but giving the nearest vertices a higher probability of selection than those far away. This specific idea is employed by the $\mathcal{MAX}$–$\mathcal{MIN}$ ant system, which will be described in chapter 3 along with randomized local search, simulated annealing and (1+1) evolutionary algorithm.

[2]For an α-approximation, α denotes the approximation factor, which is the worst-case upper or lower bound for the approximation relative to the optimal solution. For TSP specifically, the factor tells us how much greater the cost of the TSP tour will be in the worst case compared to the exact solution.

Randomized search heuristics are the focus of this study, and the topic will be presented in detail in chapter 3. However, approximation algorithms remain relevant for two reasons; (1) they can be used as initial guidance for the search heuristics and (2) they can serve as a benchmark standard for the search heuristics. Experiments using Christofides' algorithm are found in section 5.4.2 on page 50.

2.3 Graphs

In this report we will focus on undirected, complete graphs. The cost function will follow a Euclidean metric, referred to as *Euclidean TSP*, which means that the cost of the edge between any two vertices i and j is calculated as the distance between the two points (x, y) on the plane:

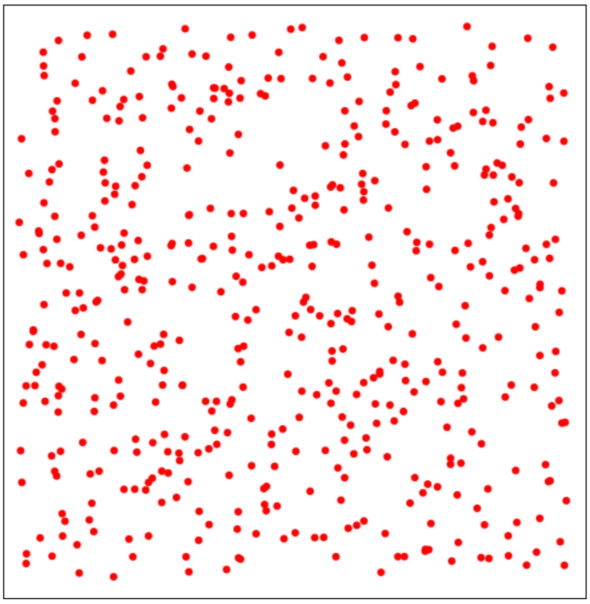
$$c(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{2.2}$$

This is a special case of MTSP, as Euclidean geometry inherently abides by equation (2.1) on page 4. The graph will be given by a set of vertices, each represented by an x and y coordinate, with the edge-costs calculated according to equation (2.2).

The graphs used will have more or less structure to them. It is interesting to look at different types of graphs in order to examine the performance of different algorithms under different circumstances. Some algorithms might perform better, when a certain structure exists on the graph, whereas others might excel when there is no particular structure. In the following subsections we look at three different ways of generating graphs randomly: Distributing vertices uniformly on a graph, placing vertices in clusters and adding randomness to existing graphs. Furthermore, a library of existing TSP instances is presented.

2.3.1 Uniform distribution of vertices

Creating a graph, where the coordinates of each vertex are selected at random, causes the vertices to be somewhat equally distributed in the given area. A consequence is that the optimal tour will be comprised of edges which are all similar in length, a consequence that is emphasized as the problem size increases. When creating a TSP tour on such a graph, no part of the graph is expected to be more important to optimize than another part. A graph with random uniformly distributed vertices can be seen in figure 2.1a.


(a) Uniform distribution of vertices

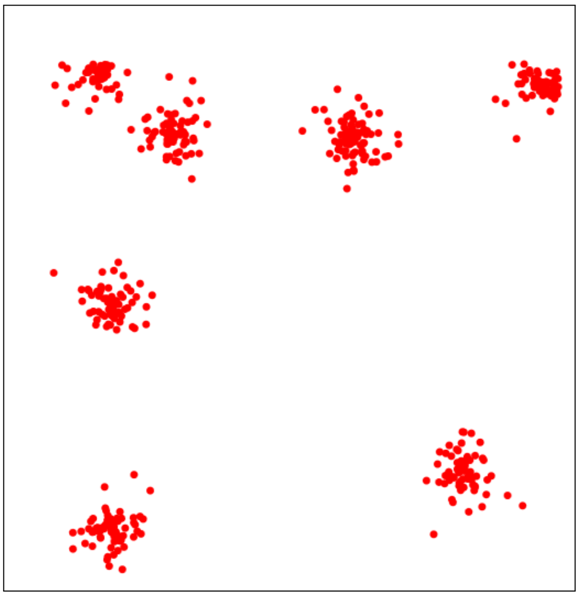

(b) Vertices grouped in clusters

Figure 2.1: Sample graphs of different types ($n = 500$).

This type of graph is generated by first selecting upper and lower bounds on x and y coordinates, defining the dimensions of the graph. Then, each vertex is placed on the canvas by selecting the x and y coordinates randomly according to a uniform distribution between the selected bounds. This is probably the most intuitive way of generating a random graph.

Problems with uniformly distributed vertices are interesting because they are surprisingly similar. Even if two random problem instances look somewhat different, they probably have a tour cost close to each other. This phenomenon is explained by the Beardwood-Halton-Hammersley theorem [3], which states that as the number of vertices n goes towards infinity, the tour cost tends towards a specific value dependent on n and a constant β. For the simplest case, where the x and y coordinates of the vertices are uniformly distributed in the interval $[0;1]$, the theorem states that with probability 1;

$$\lim_{n\to\infty} \frac{L}{\sqrt{n}} \to \beta \tag{2.3}$$

where L is the optimal tour length. If we know the constant β, we can estimate tour costs of random instances – more precisely as the problem increases in difficulty. An exact value for β has not been found, but emperical experiments suggest that the value is approximately 0.714 [2].

2.3.2 Vertices in clusters

Clustered graphs are interesting to investigate because the placement of vertices mimic how cities are naturally developed; large density of cities at the center locality and increasing space between them as we move further away. An example of such a problem is shown in figure 2.1b.

In this type of problem, navigation inside a single cluster is pretty similar to instances with uniformly distributed vertices, though the vertices are more dense in the center of the cluster. If clusters contain only few vertices each and the clusters are far from each other, the connections *between* clusters will be responsible for the majority of the cost of a TSP tour. Choosing suboptimal ways of connecting the clusters have greater impact on the overall solution.

The clustered graph is created by selecting an amount of cluster centers, which are then distributed randomly on the graph (i.e. exactly like the distribution of vertices described in section 2.3.1). Then, each vertex is assigned to a cluster at random. To decide where to place a vertex, a direction and a distance are chosen at random, determining how far and in which direction the vertex is placed from the center of its cluster. The random distances from the centers are selected from a Gaussian distribution with mean value $\mu = 0$, and variance σ being user-specified. That is, a placement of vertices close to cluster-centers are more probable than far from them. To avoid negative distances, the absolute value of the number chosen from the Gaussian distribution was chosen to be used. Allowing negative values would not affect the resulting graphs much; it would just mean that half the time, vertices were placed in the opposite direction of the chosen direction. Disallowing negative distances was purely a design decision

2.3.3 Introducing randomness in existing graphs

To make new problem instances, we can take an established TSP instance and move its vertices around randomly. This idea is related to *smoothed analysis* [27], which is a way of analysing performance by adding small amounts of noise to the worst-case scenario for the algorithm.

Obviously, the amount of noise added decides how much the graph will be altered. For instance, given a problem instance where the x and y coordinates are in the range 0 to 1000, moving every vertex 1 unit in a random direction will not change the appearance of the graph by much. On the other hand, if the vertices can instead be moved up to 200 units away, then the modified graph will most likely become completely unrecognizable. If the process is performed on a graph with some form of structure, e.g. a clustered graph, the structure is degraded in an increasing degree when the move distance is increased.

To apply randomness to an existing graph, a maximum shift distance, d_m, is selected. All vertices are then moved away from its original position in a random direction. The randomly selected distance is uniformly selected from the interval $[0; d_m[$.

2.3.4 TSPLIB

Instead of creating new graphs, we can reuse already existing instances. *TSPLIB*, a library of TSP instances, was introduced by Gerhard Reinelt in 1991 as a way of providing the scientific community with a base of TSP instances – as well as their solutions once acquired [22]. All symmetric TSP instances found in the library at the time of writing have been solved to optimality [21], which makes the library an excellent basis for benchmarking algorithms.

The TSPLIB specification [23] defines the Euclidean distance function as:

$$c(i,j) = \left\lfloor \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right\rceil \tag{2.4}$$

where $\lfloor\ \rceil$ denotes the *rounding to nearest integer* on a real number. Note that the rounding causes this alternative distance function to *not* comply with the triangle inequality.

Even though TSPLIB uses this distance function, the regular Euclidean distance in equation (2.2) on page 7 will be assumed in this study, unless explicitly stated otherwise.

2.4 Representing a TSP tour

There are various ways of representing a TSP tour. The *adjacency representation*, *matrix representation*, *ordinal representation* and *path representation* presented in [26] are all useful for different possibilities in crossover-operations in evolutionary optimization. In this study the latter representation was used to represent a TSP tour. The tour is represented simply by the sequence of vertices in the order they appear on the tour, beginning at an arbitrary vertex. The sequence $[v_1, v_2, \ldots, v_n]$ represents the tour $v_1 \to v_2, \to \cdots \to v_n \to v_1$. Note that the vertices at the first and last position of the sequence are connected on the tour. Of course, for the path to qualify as a TSP tour, every vertex in the graph must appear exactly once. Also note that there are $2n$ ways of encoding the same tour of length n; the sequence can be shifted and reversed without affecting the tour. This is true because the graphs are undirected and complete. Reversing path sequences for tours on directed or incomplete graphs may result in an invalid tour or a tour with a different cost. An example of the encoding is shown in figure 2.2.

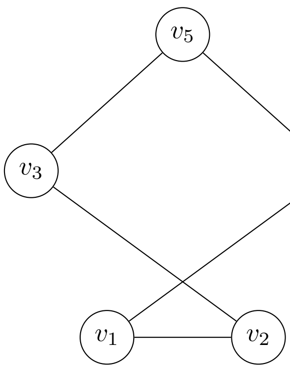


Figure 2.2: The TSP tour represented by the sequence $[v_4, v_5, v_3, v_2, v_1]$.

Chapter 3

Randomized search heuristics

Randomized search heuristics use some method of calculating solutions, and then refine them through many iterations. Essentially, they aim to produce new solution candidates using knowledge from previous iterations.

Due to the iterative nature of the randomized search heuristic algorithms, they can be terminated after any given number of iterations and yield the best found solution. This allows us to specify a timespan for the algorithm to execute in, which can be valuable in some scenarios, especially considering setups with limited resources. On the other hand, search heuristics do not give any guarantees on the solutions they produce.

The search heuristics for TSP detailed in this chapter are: *randomized local search*, *simulated annealing*, *(1+1) evolutionary algorithm* and $\mathcal{MAX}$–$\mathcal{MIN}$ *ant system*. The random search heuristics for TSP proceed by always having a candidate solution and then try to improve on that solution. Of the four algorithms, three need a starting point, the so-called initial guess. This can be either (1) a random guess, or (2) a solution known to be good – possibly calculated by an approximation algorithm.

The random guess can be formed by selecting a random permutation of the vertices. The cost of such a tour is expected to be bad, because the expected edge cost from a vertex to a randomly selected vertex is much higher than the cost of the edge connecting the vertex in the optimal TSP tour. However, such a guess is trivial to compute.

With the second approach we can for instance use Christofides' approximation algorithm (discussed in section 2.2.1). The algorithm is a 3/2-approximation, so the initial tour will be at most 3/2 times bigger than the optimal tour. The cost of this initial tour will, with a very high probability, be much lower than that of the random permutation. With a time-complexity of $O(n^3)$, the algorithm takes more time than selecting a random guess.

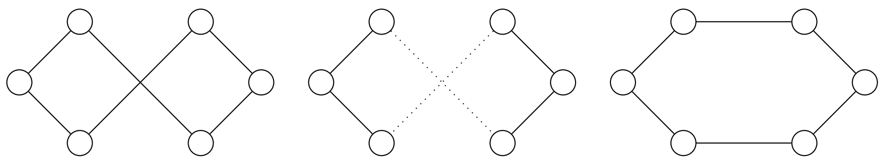
Then, how do we determine a starting point for the algorithms? The hypothesis is that a good initial guess will let the algorithms form a good solution faster, because it has already been helped along the way. If the guess is very bad, the algorithms need to alter the solution tour through many iterations to reach the same fitness of a good guess. In this report, we will mainly focus on the random guess. However, an experiment conducted using a Christofides' approximation is presented in section section 5.4.2 on page 50.

3.1 2-opt

Before presenting the algorithms, we will introduce the *2-opt swap*. Most of the algorithms presented in the following section are merely general approaches to iteratively improve solutions for a problem by introducing som random modification. Being general algorithms, they do not specify exactly what this modification is or how it is applied. However, the modification should be able to produce any solution from any given starting point in a finite number of steps, otherwise the search space becomes disjoint.

For TSP, we obviously need a modification that can change the tour. Since we use the path representation (described in section 2.4), we need a way to rearrange the path to create new permutations. The 2-opt modification allows us to do exactly that. The 2-opt swap is a technique first proposed by Croes in [6]. The idea behind the operation is to resolve edge intersections. The swap is executed by selecting two edges in the tour. These two edges are deleted, which will turn the TSP tour into two seperate paths. By reconnecting the two paths the only other possible way, the swap is completed. This entire process is depicted in figure 3.1. The 2-opt swap procedure can be implemented to run in linear time to the size of the graph (see section 4.2.2 for details).

The number of 2-opt swaps required to transform a tour into any other tour is



(a) The tour before 2-opt **(b)** Deleting the edges **(c)** Reconnecting paths

Figure 3.1: A complete 2-opt swap on a sample TSP tour.

finite. Let us denote the starting tour as the *original* and the goal as the *target*. An approach to transforming the original tour into the target tour could be to look at an arbitrary edge in the target tour. If that edge is not present in the original, a single 2-opt swap can introduce it at the cost of some other edge by making sure to select edges for the swap such that the two vertices we wish to connect are in seperate paths and are both at either end of their respective path when the selected edges are deleted (refer to figure 3.1b). After the swap we then move on to the next edge (in either direction) on the target tour and repeat the process, making sure to never select edges for the 2-opt swap that have previously been introduced. As a result, we can transform a solution into any other solution in $O(n)$ swaps, n being the number of vertices in the graph. It may well be that there are even faster ways of performing this transformation, but this paragraph exists merely to point out that it can be done in a finite number of steps. This fact is important because it allows us to escape any local optimum given enough consecutive 2-opt swaps.

In this report, the term *random 2-opt swap* will be mentioned occasionally. What is meant is simply to pick two edges from the tour at random and perform a 2-opt swap on those edges.

3.2 Neighborhoods and local optima

In this study, a neighbor to a TSP tour is a solution that can be reached with a single 2-opt swap. A local optimum is a solution that has no neighbors with a lower cost, and hence cannot be improved by performing only a single 2-opt swap.

```
input: initial guess
output: TSP tour
begin
  best ← initial guess
  while stop−condition not met
    candidate ← 2−opt(best)

    if cost(candidate) ≤ cost(best)
      best ← candidate
  end
  return best
end
```

Figure 3.2: Pseudocode for randomized local search algorithm.

3.3 Algorithms

In the following subsections, the randomized search heuristic algorithms used in this study will be described. The descriptions will be accompanied by pseudocode, and in this code you will find that the algorithms loop until a "stop-condition" is met. A stop-condition could constitute for instance reaching a certain number of iterations, reaching a certain tour cost (possibly obtained by using the Beardwood-Halton-Hammersley theorem described in section 2.3.1 on page 7) or exceeding a time limit. The latter is used in this study. When using the term *iteration* for the algorithms, this outer loop is the subject. For this chapter, the cost function presented in the pseudocode is assumed to have a time-complexity of $O(n)$ (this bound is explained with the implementation details on 2-opt in section 4.2.2).

3.3.1 Randomized local search

Randomized local search (RLS) is the simplest of the considered algorithms. It is an iterative approach, which takes the previous best found solution, performs a random 2-opt operation on it, and consider the outcome. If the cost of the resulting solution is better than or equal to that of the previous best, the new solution is accepted. Otherwise it is rejected. The pseudocode for RLS is shown in figure 3.2.

The algorithm needs an initial solution guess to get started. This can be a random guess or an approximation solution from Christofides' algorithm.

RLS is a local search, which means that it cannot reach every possible solution from its current solution, because it never accept a solution that is worse than the current best. Because of this, RLS can enter local optima from which it cannot escape. A situation can easily occur where the algorithm finds a good solution, but a better one could be found if multiple 2-opt operations could be performed.

Every iteration of RLS has a time-complexity of $O(n)$ because of the 2-opt modification and because of calculating the tour cost.

3.3.2 Simulated annealing

Annealing is the process of heating up metal and then slowly cooling it down to let it steadily find a new microstucture with better properties. When the metal is hot it will have a more fluent structure, and during the cooling it will slowly form its preferred structure. *Simulated annealing* (SA) is an optimization algorithm for TSP that employs the ideas behind annealing [17]. In the beginning of the execution of the algorithm, when temperatures are high, restructuring of the solution will be accepted with high probability, even if the TSP cost is increasing. As the temperature decreases, so does the probability of accepting bad state changes. The intention is that a rough structure is found while the temperature is high, and as the temperature decreases, the algorithm enters a refinement phase in which it finds the finer details of the solution.

Pseudocode for the algorithm can be found in figure 3.3. The algorithm maintains a temperature T, the evolving solution and the yet best found solution. In each iteration, first a random modification will occur, then the modification may or may not be accepted, and finally the temperature is decreased according to a cooling scheme. When a modification is made, it is always accepted if it improves the solution. If the modification worsened to solution, its acceptance will follow the *Boltzmann selection*, meaning that the modification will still be accepted with probability

$$p_B(\Delta C, T) = e^{-\Delta C/(k_B T)} \tag{3.1}$$

where ΔC is the difference in cost between the new and old solution, k_B is a constant and T is the current temperature. If a change is rejected, the previous solution is simply used as basis for the next iteration. The Boltzmann selection will have a higher probability of accepting modification with only a small difference in cost and especially when the temperature is high.

In addition to be initialized with a starting guess, SA is also provided with a cooling scheme. The cooling scheme defines what temperature T the procedure starts with, and how it is cooled down in each iteration. In this study, a cooling

```
input: initial guess, cooling scheme
output: TSP tour
begin
  best ← initial guess
  current ← best
  T ← initial temperature according to cooling scheme

  while stop−condition not met
    candidate ← 2−opt(current)
    $\Delta C$ ← cost(candidate) − cost(current)

    if $\Delta C \leq 0$
      current ← candidate
      if cost(candidate) ≤ cost(best)
        best ← candidate

    else
      r ← random number in interval [0;1[
      if r < $p_B(\Delta C, T)$
        current ← candidate

    T ← next temperature according to cooling scheme
  end
  return best
end
```

Figure 3.3: Pseudocode for simulated annealing algorithm.

scheme due to Klaus Meer [19] will be used. It has the recursive definition:

$$\begin{aligned} T_0 &= m^3 \\ T_i &= T_{i-1} \cdot \left(1 - \frac{1}{cm^2}\right) \end{aligned} \tag{3.2}$$

where $c > 0$, $m > 0$ are parameters and i is the iteration count. This cooling scheme assumes $k_B = 1$, so that value will be used in this project as well. A suggested value for m is $20n$ where n is the number of vertices in the graph. The parameter c should be a small constant [19]. The temperature T_i is given as a result of slicing a small percentile from the previous temperature T_{i-1}. The absolute decrease in temperature will therefore slow down as i increases, and it will never reach zero. In theory, always be able to escape local optima because acceptance of cost-increasing moves always have a non-zero probability, but practically the probability of doing so will become very low with a low temperature. In other words, the algorithm with be more likely to accept bad moves in the beginning of the execution, and will then slowly, as the temperature decreases, tend towards RLS.

Like RLS, each iteration of SA requires $O(n)$ time to complete due to the 2-opt swap and calculation of tour-cost.

3.3.3 $(1+1)$ evolutionary algorithm

Evolutionary algorithm (EA) is, as the name suggests, based on evolution. The algorithm lets a population (of solutions) mutate to produce a new generation. Its general form, $(\mu + \lambda)$ EA, has a population of μ *individuals* (solutions), and to produce the next generation, a set of λ mutated individuals is generated. The best μ individuals from the parent set and the offspring set are selected to be the new population. EA is in the family of *genetic algorithms*, which usually deals in the mutation of bit strings because they correspond well to genes (see the *Handbook of Natural Computing* by Rozenberg *et al.* [24] for further reading on genetic algorithms, evolutionary algorithms and other nature-inspirired approaches).

(1+1) EA is the simplest form of the $(\mu + \lambda)$ EA, with only a single solution in the population and a single offspring solution. Only if the offspring solution is better than the parent solution will it serve as a replacement. So far, the algorithm resembles RLS, but in evolutionary algorithms any solution should be able to mutate into any other solution. When dealing with bit strings, a mutation could involve iterating the bits and flip them with probability p_m. The probability p_m should be $1/s$ where s is the length of the string, meaning that an average of 1 bit is flipped in each iteration [8]. This mutation can turn any solution into any other solution with a non-zero probability, allowing the heuristic to break out of local optima. Unlike bit strings, we cannot just "flip" the edges in a TSP tour, so the procedure cannot be directly translated. Instead, as suggested Sutton & Neumann [29], we can in each iteration do k number of 2-opt operations, where k follows a Poisson distribution.

A Poisson distribution has a parameter[3] $\lambda > 0$ which defines the mean value and variance of the distribution. The distribution supports numbers $k \in \{0, 1, 2, ...\}$. If we choose $\lambda = 1$ for instance, the mean value of the distribution is 1 which in turn means that we will experience an immense number of zeros every time a large number (e.g. 100) is picked. When modifying a TSP solution, it does not make sense to do zero 2-opt operations – we will end up with the same solution. To overcome this small inconvenience, Sutton & Neumann decided on using $k + 1$ modifications, where k follows a Poisson distribution [29]. This effectively turns every zero from the distribution into a one, every one into a two and so forth. The disadvantage of doing this is, that the distribution is now centered around $\lambda + 1$ instead of λ as it affects every number picked from the distribution. Another approach is to simply replace any encountered zero with a one. In this case, the distribution is only changed in regards to the occurrences of zeros and ones, not the numbers above that point. Both of these procedures to pick the number of 2-opt operations per iteration will be experimented with.

[3]Not to be confused with the number of offspring in $(\mu + \lambda)$ EA.

```
input: initial guess, $\lambda$
output: TSP tour
begin
  best ← initial guess

  while stop-condition not met
    k ← pick from Poisson distribution with parameter $\lambda$
    candidate ← current

    i ← 0
    while i < k
      candidate ← 2-opt(candidate)
      i ← i + 1
    end

    if cost(candidate) ≤ cost(best)
      best ← candidate
  end
  return best
end
```

Figure 3.4: Pseudocode for (1+1) evolutionary algorithm.

The former approach will be called *(1+1) EA (k+1)* and the latter *(1+1) EA (substitution)*.

The algorithm for (1+1) EA can be found in figure 3.4. It is initialized with two parameters. Like RLS and SA, it needs an initial solution, and the additional parameter λ is used for the Poisson distribution. The algorithm has a time-complexity of $O(kn)$ for each iteration, because it has to perform k 2-opt swaps, each requiring $O(n)$ time. However, because the mean value of the Poisson distribution, which k is selected from, is constant, the algorithm has $O(n)$ amortized time-complexity per iteration.

3.3.4 $\mathcal{MAX}$–$\mathcal{MIN}$ ant system

When real-world ants move towards their food source, and are presented with an obstacle, they have a choice to make: go left or go right. The ants may pick either of these options at random, thus it is expected that half the ants will go left and the other half right. Whenever the ants move, they leave behind a trail of pheromone, which is a substance the ants are attracted to. When given a choice between a route with a low amount of pheromone and one with a high amount, they are more inclined to take the route with the high amount. The ants will move back and forth past the obstacle faster on the shortest route around the obstacle, resulting in the trail of pheromone accumulating faster on that side. Because the pheromone trail is stronger on the short route, more ants will choose that route, thus further adding to the pheromone trail. This positive feedback will eventually result in all the ants going the shorter route around the obstacle. Ant colony optimization algorithms try to apply the behavior of real-world ants to solve combinatorial problems.

The *ant colony system* was developed to find good TSP tours by letting artificial ants construct tours on the graph [7]. To this end, the graph is extended with a representation of the pheromone levels τ_{ij} for the edge between every pair of vertices i and j, with the additional constraint that $\tau_{ij} = \tau_{ji}$ for symmetric TSP. The algorithm works by sending out artifical ants to traverse the graph. This is done by placing m ants on different start vertices, and then one by one letting them construct a tour. The ant constructs the tour by employing the so-called *state transition rule* to select the next vertex in the tour. The state transition rule gives the probability of the ant moving from its current vertex i to another vertex j by

$$p_{ij} = \begin{cases} \dfrac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{u \in J} \tau_{iu}^{\alpha} \cdot \eta_{iu}^{\beta}} & \text{if } j \in J \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

with $\eta_{ij} = 1/c(i,j)$ being the heuristic function, J being the set of vertices not yet visited by the ant and the exponents α and β being parameters to the algorithm stating the relative importance of pheromone and distance, respectively. The state transition rule favors selection of edges with low costs and high pheromone levels.

When all the ants have constructed their TSP tours, the algorithm emulates the accumulation of pheromone by updating the τ values across the graph according to the *global update rule*[4]

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij_{old}} + \rho \cdot \Delta\tau_{ij} \tag{3.4}$$

where $0 \leq \rho \leq 1$ is the evaporation factor of the pheromone and

$$\Delta\tau_{ij_k} = \begin{cases} 1 & \text{if edge } (i,j) \in B \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

with B being either the best-so-far (also called the global-best, globally-best or best-yet) or the iteration-best tour.

It is clear to see that edges that are part of B will collect more pheromone whereas those not in B will slowly have their pheromone decay. The global update rule causes the positive feedback effect known from the real ants.

[4]Several different global update rules exist. The chosen rule defined by equations (3.4) and (3.5) is a simplification of that found in [28]. A comprehensive collection and discussion of global update rules can be found in [11].

```
input: $\alpha$, $\beta$, $\rho$, $m$, $\tau_{max}$, $\tau_{min}$, update type
output: TSP tour
begin
  best ← random tour
  iterationbest ← best

  while stop-condition not met
    foreach $m$ ants
      v ← random vertex
      J ← empty set
      tour ← [v]
      add v to J

      while tour is incomplete
        v ← select vertex according to state transition rule (3.3)
        add v to J
      end

      if cost(tour) $\leq$ cost(iterationbest)
        iterationbest ← tour
      end
    end

    if cost(iterationbest) $\leq$ cost(best)
      best ← iterationbest
    end

    foreach edge in graph
      update $\tau_{edge}$ according to global update rule (3.4) using update type
    end
  end

  return best
end
```

Figure 3.5: Pseudocode for $\mathcal{MAX}$–$\mathcal{MIN}$ ant system algorithm.

There is a risk that the ant colony system will stagnate into a small set of solutions, because the pheromone trail diminishes on the edges that are not used, and keeps increasing on those who are. To solve this problem, Stützle & Hoos [28] developed the *$\mathcal{MAX}$–$\mathcal{MIN}$ ant system* (MMAS), which is based on the ant colony system.

MMAS introduces a new feature to overcome the stagnation issue. Two new parameters, τ_{max} and τ_{min}, set an upper respectively lower bound on the amount of pheromone that can be stored on each edge at any given time. If the pheromone quantity calculated on an edge in equation (3.4) is lower than τ_{min} or higher than τ_{max}, the pheromone will instead be set to τ_{min} or τ_{max}, respectively. This mechanism helps preventing the positive feedback effect causing stagnation, because the probability of picking an edge can only diminish to a certain point. In [28], local search and a so-called *proportional update rule* are also utilized after each iteration, but these features are omitted in this study to keep the algorithm simple and comparable to the other presented heuristics.

Pseudocode for MMAS can be found in figure 3.5. The time-complexity for a single iteration is $O(mn^2)$: Each of the m ants need to construct a tour. The tour requires n vertices to be added, and each time we add a vertex, the state transition rule is used. This rule requires us to calculate a sum based on $O(n)$ vertices. It is therefore quite apparent that a single iteration of MMAS has much more computational work than RLS, SA and (1+1) EA, especially for high values of m. It is expected that MMAS is unable to undergo as many iteration as the other heuristics. Of course, the strong guidance that MMAS has when constructing tours, in the form of the state transition rule, is a compensation for this fact.

3.4 Concorde

Concorde is a software suite developed by David Applegate, Robert E. Bixby, Vašek Chvátal and William J. Cook in the 1990s to find optimal solutions for TSP [1]. Concorde is considered to be a state-of-the-art TSP solver [12]. The program has solved every instance present in TSPLIB. The largest of these problems contains 85 900 vertices, and it is the largest instance yet to be solved [1].

With such a great program for obtaining exact solutions, what do we need search heuristics for? There are problems that even Concorde cannot handle, amongst others the 1 904 711-city world problem. On this problem, however, a heuristic algorithm due to [14] based on the Lin-Kernighan heuristic [18] has found a tour with a cost at most 0.0474% higher than the optimal solution (lower bound determined by Concorde), which goes to show that heuristics do indeed have their merits.

Experimenting with the algorithms used by the Concorde software is beyond the scope of this study. Still, Concorde will be used to serve as a baseline for the randomized search heuristics to compete with.

Chapter 4

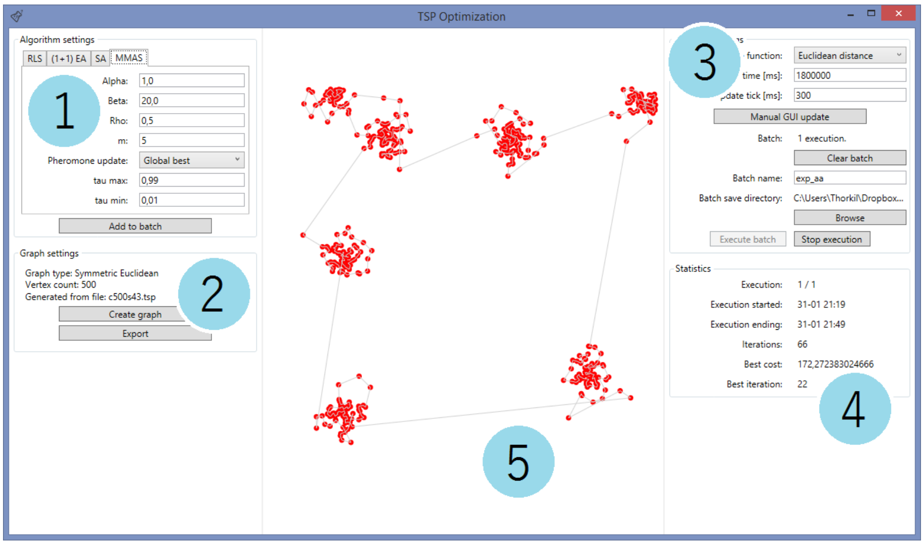
Program implementation

In order to experiment with the search heuristics discussed in chapter 3, a program was created. It was determined following features needed to be supported:

- Creating, loading and modifying graphs and saving them to the hard drive in TSPLIB format.
- Setting up algorithms and running them in batches.
- Storing results and statistics on algorithm execution on the hard drive.
- Showing solutions as they evolve during execution.

The user interface of the resulting program is shown in figure 4.1. The program was written in C# 5.0, which ships with the .NET 4.5 framework.

This chapter will describe the program and discuss some implementation details as well as limitiations of the program and ideas for future features.



Algorithm settings

Alpha: 1,0
Beta: 20,0
Rho: 0,5
m: 5
Pheromone update: Global best
tau max: 0,99
tau min: 0,01

Graph settings

Graph type: Symmetric Euclidean
Vertex count: 500
Generated from file: c500s43.tsp

function: Euclidean distance
time [ms]: 1800000
date tick [ms]: 300
Batch: 1 execution.
Batch name: exp_aa
Batch save directory: C:\Users\Thorkil\Dropbox...

Statistics

Execution: 1 / 1
Execution started: 31-01 21:19
Execution ending: 31-01 21:49
Iterations: 66
Best cost: 172,272383024666
Best iteration: 22

Figure 4.1: The graphical user interface of the program.

4.1 Program description

The graphical user interface has five sections. The first is the *Algorithm settings* area, which allows you to set up algorithms with specific parameters and to add them to the execution batch. The batch is merely a collection of algorithms to be run consecutively, such that the user can start a set of algorithms to run over night, for instance.

The second area is the *Graph settings* in which a few statistics on the currently shown graph can be seen. There are two buttons here as well; the *Create graph* button and the *Export* button. Both of these buttons open dialogs to create graphs or export the current graph, respectively. Creation of graphs can be done as described in sections 2.3.1, 2.3.2 and 2.3.3 on pages 7—10. Graphs can also be loaded from a TSPLIB file without adding randomness.

The third area handles *Runtime settings*. This includes setting the execution time for each of the algorithms in the batch (i.e. 1 800 000 ms for half an hour), how often the solution shown in the graph area is updated (a zero here means the solution is only updated by a click on the *Manual GUI update* button) and settings on where to store execution statistics for the algorithms. Each completed execution creates an individual statistic file. This way, an execution can be aborted without losing the experiments that have been successfully conducted.

Finally, buttons to start and end executions are found in the bottom of the area.

The fourth area, the *Statistics area*, shows some statistics on the current executing batch. It shows which algorithm is currently running and how well it is faring. It also presents information for the entire batch; when it was started, when it will end and how many algorithms are left to execute.

The fifth and final area is the middle section showing the graph and the current evolving solution. The vertices of the graph are presented as red dots, and the current solution is drawn by grey lines.

4.2 Implementation details

In this section, specific interesting implementation details behind the program are presented. Specifically, the implementation of tour cost calculations, 2-opt swaps and MMAS tour construction will be discussed, and the structure of the program will be described.

4.2.1 Calculating costs of TSP tours

In order to avoid computational overhead, when calculating the cost of TSP tours, it was decided to precompute the edge costs (following equation (2.2) in section 2.3 on page 7) and to use an adjacency matrix representation to store them in. The adjacency matrix uses n^2 cells to store data for a graph with n vertices. Since all the graphs used in this project are undirected, the matrices will be symmetric across the diagonal, allowing us to settle with storing and precomputing only half as much. The cost of a TSP tour of length n can be computed as

$$\sum_{i=1}^{n-1} \big(c(s_i, s_{i+1})\big) + c(s_n, s_1) \tag{4.1}$$

where s_i is the ith element in the path sequence and c is the cost function. Because $c(i, j)$ can be looked up in the adjacency matrix in constant time, the cost of the entire tour takes $O(n)$ time to calculate with $O(n^2)$ space usage.

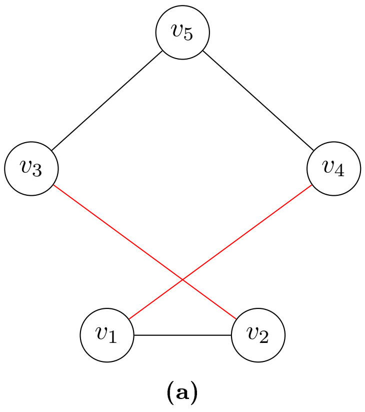


Figure 4.2: The TSP tour before (a) and after (b) a 2-opt swap.

4.2.2 Implementing the 2-opt swap

As described in chapter 3, RLS, SA and $(1+1)$ EA all the 2-opt swap algorithm to introduce random alterations to their TSP tours. The effect of a 2-opt swap is described in section 3.1 on 14, but how exactly does one introduce such a swap?

In figure 4.2 a 2-opt swap is shown. The edges colored red are involved in the swap. Let us assume that the sequence representing the tour in figure 4.2a is $[v_4, v_5, v_3, v_2, v_1]$. To apply the 2-opt swap, two steps are performed; (1) delete the selected edges and (2) reconnect the resulting paths the other possible way. How this is actually done is to take the first and the last vertex between the chosen edges. These two vertices mark the start and the end of one of the two paths appearing if deleting the selected edges. To connect the two paths, this selected path is simply reversed. In the example, we could select v_2 as the first vertex and v_1 as the last *or* v_4 as the first and v_3 as the last. Following the procedure, it would result in the sequence $[v_4, v_5, v_3, v_1, v_2]$ or $[v_3, v_5, v_4, v_2, v_1]$, respectively. Both of these sequences are representations of the tour shown in figure 4.2b.

Note that the selection of vertices must be able to wrap around the end of the sequence. For instance, in the tour $[v_4, v_5, v_3, v_2, v_1]$ if we select v_2 as the first vertex of the subsequence and v_4 as the last, the resulting tour should be $[v_2, v_5, v_3, v_4, v_1]$ (i.e. reversing the order of v_2, v_1 and v_4) and *not* $[v_2, v_3, v_5, v_4, v_1]$ (i.e. reversing the order of v_4, v_5, v_3 and v_2) which encodes a completely different tour. In other words; just selecting two vertices is not enough – it matters which of the vertices is selected as the start and which is selected as the end of the subsequence.

The time-complexity for a 2-opt swap is linear to the number of vertices. In the worst case, the implementation of the 2-opt swap requires us to reverse the entire sequence.

4.2.3 Algorithm architecture

A diagram of the algorithm classes is found in figure 4.3.

The algorithms derive from the `Algorithm` class, which contains basic information and methods needed; recording improvements for statistics, comparing a new-found solution with the best-yet solution and a method for running the algorithm for a given timespan. The pseudocode for the algorithms in chapter 3 all have their code inside a while-loop that runs until a stop condition is met. This while-loop is implemented in the `Algorithm` class, and in each iteration it invokes the `Iterate()` method implemented in each of its subclasses; RLS, (1+1) EA, SA and MMAS. This construction was chosen in order to let a common base class implement the logic that is universally needed by the algorithms.

The algorithms are implemented in a fashion to very easily extend them or change their behavior. Elements of the algorithms that are considered to be subject to change are modeled as objects. An example of this is the modification operation: The 2-opt operation is a class in itself, and when constructing the algorithm objects, such an operation is supplied in the form of an object implementing the `IModificationOperation` interface. In this study, 2-opt is used exclusively, but if one would want to use for instance 3-opt (deleting three edges and reconnect) instead, it is simple to implement it as a class and supply that object when constructing the algorithms instead. Similarly, if we wish to implement a new cooling scheme for SA which provides a constant (instead of relative) cooldown each iteration and stops cooling at $T = 0$, we can introduce a new class `ConstantCooling` which derives from the abstract `CoolingScheme` class, and implement the desired behavior in its `NextTemperature()` method.

The observant reader notices that `RandomizedLocalSearch` derives from `OnePlusOneEvolutionaryAlgorithm`; the reason being that the only difference between (1+1) EA and RLS is the amount of modifications in each iteration – a variable amount for (1+1) EA and always 1 for RLS. Implementation-wise, RLS is a special case of (1+1) EA where the `IModificationCount`'s `NumberOfModifications()` method always returns 1. This functionality is implemented in the `FixedCount` implementation, which `RandomizedLocalSearch` is forced to use.

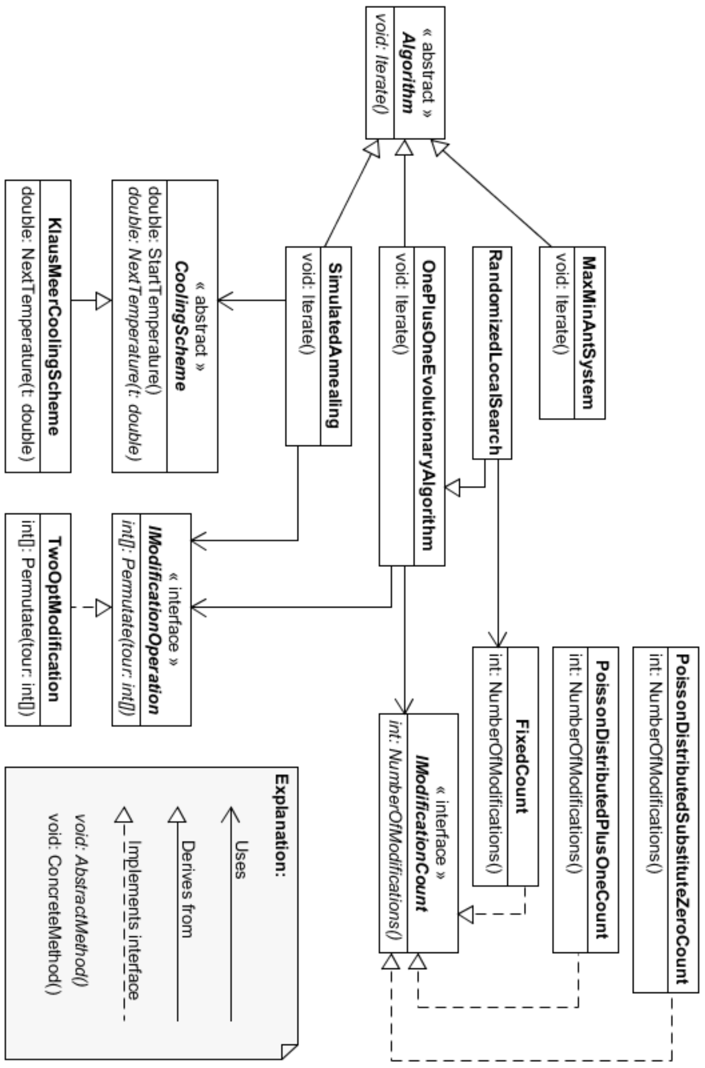


Figure 4.3: UML object diagram over classes related to the algorithms.

In addition to being easily extendible, a benefit of implementing functionality through interfaces is that logic is separated from the algorithms, allowing us to apply the same code in different algorithm (if applicable), thus avoiding repeating ourselves.

4.2.4 Executing algorithms asynchronously

In order to interact with the program while the algorithms execute, the algorithms need to execute asynchronously. To this end, a class called `AlgorithmRunner` was implemented. Its job is to run a batch of algorithms on the problem, provide information on its progress and save the algorithm statistics to files.

The `AlgorithmRunner` and the code that invokes it (i.e. the code handling the start button click event) use the *Task parallel library* of .NET, in which the `async` and `await` keywords are used to respectively declare and invoke asynchronous methods. When a method awaits a call to an asynchronous method, it returns control to its caller. The process is depicted in the form of a sequence diagram shown in figure 4.4.

When clicking the start button, an event-handling method is invoked, which initializes the `AlgorithmRunner`. When the method is ready, it makes a call to the runner to execute the algorithms. It does so by using the `await` keyword, meaning that it will stop blocking the main thread. The algorithms execute in a separate thread, which means that they can occupy their own CPU core in multi-core environments.

When the GUI needs to update the solution, i.e. when the user clicks the update button or the update time has elapsed, it probes the `AlgorithmRunner` for the current best solution. Additional thought went into this process, as it is always dangerous to share data between threads. Fortunately, in this program, data is only read from the other thread – not modified, so we just need to ensure that the data is in a valid form. If the `AlgorithmRunner` updated the best solution by modifying the existing solution and changing the positions of vertices therein, then, at the probe time, the solution might be incorrectly represented. Instead, every time the `AlgorithmRunner` updates the best solution, it changes the reference of the solution to a new array containing the new solution. When the GUI probes for the best solution, the runner either has one reference or another, but in either case the reference points to a valid solution.

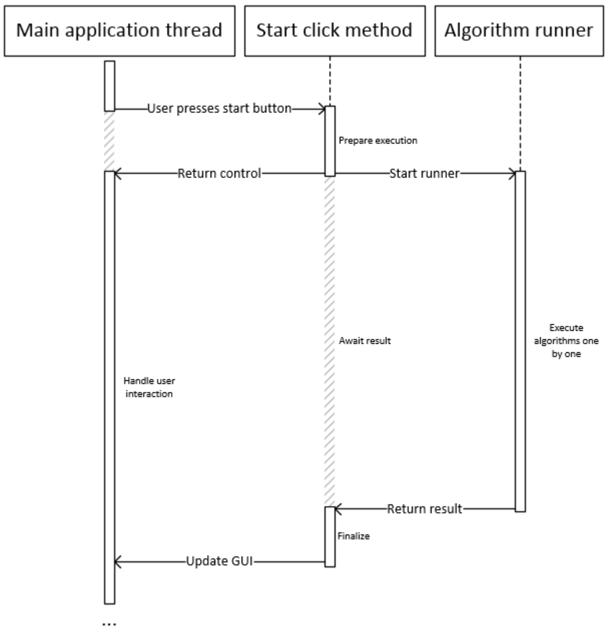


Figure 4.4: Sequence diagram showing the algorithms running in parallel with the main application thread.

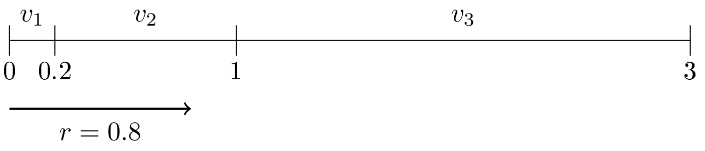


Figure 4.5: Selection of next vertex in MMAS tour construction.

4.2.5 Tour construction in $\mathcal{MAX}$–$\mathcal{MIN}$ ant system

In RLS, SA and (1+1) EA, the main random element was the random 2-opt modification, which is trivial to determine because it just requires us to pick two random discrete values (i.e. indices in the solution array). In SA, a cost-increasting modification could be applied with some probability, but this is also easy to determine by simply picking a random value between zero and one, and comparing it with the probability of acceptance.

In MMAS, however, a more difficult situation arises. When MMAS constructs the tours, it needs to pick the next vertex among the set J of valid vertices. This is done according to the probabilities given by the state transition rule shown in equation (3.3) on 21. It is not trivial to select a vertex as before, by simply choosing a random variable. Instead, we need to determine which vertex the randomly selected value maps to. The relevant part of the state transition rule from a vertex i to a vertex $j \in J$ is:

$$p = \frac{\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}}{\sum_{u \in J} \tau_{iu}^{\alpha} \cdot \eta_{iu}^{\beta}} \tag{4.2}$$

The denominator in equation (4.2) is the probability weight, denoted w, of each vertex in relation to each other. Consider the scenario depicted in figure 4.5, where $J = \{v_1, v_2, v_3\}$ and $w(v_1) = 0.2$, $w(v_2) = 0.8$ and $w(v_3) = 2$. Here, v_3 is ten times more likely to be picked by than v_1. In order to select a vertex based on these weights, we select a uniformly distributed number in the interval from zero to the sum of weights, i.e. the numerator from equation (4.2), which is 3 in the example. We then choose vertices in any order and start subtracting their respective weights from r. The vertex whose weight force r below 0 is selected as the next step on the tour. In figure 4.5, $r = 0.8$ is selected. We subtract $w(v_1)$ from r, resulting in $r = 0.6$. Then, $w(v_2)$ is subtracted from r, resulting in $r = -0.2$. Thus, in this case, v_2 is selected.

Essentially, the random number r indicates where on the spectrum we stop, and the vertex occupying that interval in the spectrum is selected. The order in which the vertices appear is of no concern. The random number will not be biased toward any point in the spectrum, and the increase in probability of selecting a vertex with a high weight stems from the fact that said vertex will occupy a larger interval on the spectrum (e.g. v_3 occupying 2/3 of the interval in figure 4.5).

Unfortunately, this method has the disadvantage that it can it take $O(|J|)$ operations to finish: The method may need to subtract the weight of every single vertex before concluding its choice. Precomputing the state transition values is not feasible, seeing that the values change every iteration and the state transition rule relies on which vertices are contained in J at the given time. A practical optimization could be to sort the vertices in the spectrum by their probabilities such that the vertex with the highest probability of being selected is first. This would result in the expected number subtraction made to r being reduced. However, because a sorting step is required, the approach will probably not have any positive impact – let alone a large one, so this attempt at optimization was not tried in this study.

4.3 Limitation and future features

The program was created with the mindset that algorithms with different parameters should be created and run in batches. The program was not build around letting different graphs or execution times be variables in the batch, so when a batch is run, the graph and execution time for each algorithm are fixed. If one wish to test a specific algorithm on a series of different graphs, one would have to manually switch graphs after each execution. A feature allowing the graphs and execution times to be a variable part of the batch would enable us to do this without difficulty. Furthermore, the program in its current state only allows adding algorithms one by one and deleting *all* algorithms in the batch. It would be beneficial to get more control over the batch, allowing us to delete individual algorithms or rearrange the ordering they are executed in.

The statistic files generated by executing algorithms in the program all contain the best found tour in the path representation. The idea was to allow the program to load statistic files and recreate the solution presented in them, but due to time limitations, this feature was never implemented.

Finally, the program is only able to read Euclidean graphs in a given TSPLIB file format. The program can only load symmetric TSP instances defined with a `NODE_COORD_SECTION` and a correct setup according to the descriptions in section 2.3 on page 7. If the program is to be used for different graph types, these should be implemented and the program should support all different graph types supported by the TSPLIB format.

CHAPTER 5

Experiments and results

In this chapter, results from various different experiments are presented. The goal is to provide basis for a comparison of the randomized search heuristics presented in chapter 3. Experiments were planned for four problem settings.

The first – and perhaps the most obvious – setting to test, is the *size* of the graph. This setting is a great way of measuring the search heuristics because big problem instances limit the use of exact algorithms. Using randomization in big instances may result in too many subpar modifications, leading to the solution evolving at too slow a pace.

The second setting to look at is the allowed *execution time*. Allowing more or less execution time may affect the algorithms differently. Restricting time is also directly related to the graph size, in the sense that a large problem instance will take longer to compute than a small one. Obviously, restricting the time enough will cause the algorithms to produce worse solutions than otherwise – but how much worse?

The third setting is *graph structure*. To experiment with this, comparisons are made between graphs where vertices are uniformly distributed and graphs with vertices placed in clusters.

It is also interesting to see what happens if we slowly dissolve the structure of a clustered graph through many steps, to finally mimic the random graph. To do this, randomness is introduced into the structured graphs until they appear as random as the graph with uniformly distributed vertices.

The fourth and last setting, which will be experimented with, is to see how the algorithms respond to being given a good *initial guess*. Imagine that you already know a pretty good solution to the problem – why not let the search heuristics take advantage of that? Changing the starting point of the search from a random initial guess to a more reasonable solution is achieved by using an approximation algorithm (see section 2.2.1) prior to executing the search algorithm.

Furthermore, the randomized search heuristics will be compared to the Concorde TSP solver on TSP instances from TSPLIB, to determine how good the solutions provided by the algorithms are relative to the optimal solutions.

In every experiment, the results are presented as averages over several executions. When dealing with randomized algorithms, fluctuation in results are expected. By performing several repetitions and comparing averages, we try to balance out this occuring variance.

5.1 Test environment

All benchmarks presented in this chapter were performed on a Windows 8.1 Pro 64-bit computer with an Intel® Core$^{\mathrm{TM}}$ i5-3210M CPU running with clock-speeds of 2.5 GHz. The program was compiled in release mode with full code-optimization and run as a standalone executable.

5.2 Graphs used for experiments

For the experiments, graphs newly created by the techniques discussed in sections 2.3.1 and 2.3.2 on pages 7 — 9 are used. Table 5.1 gives an overview of created graphs used in the experiments. The *names* column specifies the name of each graph, which will be used in this chapter to identify the graphs. *Creation method* says how the graph was created in the program, including relevant parameters (e.g. cluster sizes for a clustered graph) such that the graphs can be reproduced in the program. Here, *Uniform* denotes the uniform distribution of vertices while *Clustered* denotes a clustered graph.

Table 5.1: Overview of created graphs.

Name	Size	Creation method
`r100`	100	Uniform, $x, y \in [0; 20[$, random seed 42
`r250`	250	Uniform, $x, y \in [0; 20[$, random seed 1
`r500`	500	Uniform, $x, y \in [0; 20[$, random seed 43
`r750`	750	Uniform, $x, y \in [0; 20[$, random seed 35
`r1000`	1 000	Uniform, $x, y \in [0; 20[$, random seed 44
`c100`	100	Clustered, $x, y \in [0; 20[$, random seed 42, 4 clusters, $\sigma = 0.8$
`c500`	500	Clustered, $x, y \in [0; 20[$, random seed 43, 7 clusters, $\sigma = 0.8$
`c1000`	1 000	Clustered, $x, y \in [0; 20[$, random seed 45, 15 clusters, $\sigma = 0.5$

5.3 Experiments with algorithm parameters

In order to find overall good parameters for the separate search heuristics, experiments were run on a wide range of parameter combinations on the graphs `r500` and `r1000`, both with an execution time of 5 minutes and 1 minute.

The *initial guess* parameter for the relevant algorithms will be a random permutation, except of course in the experiment using approximation initialization (see section 5.4.2).

The tested values and the results of these initial tests are found in the following subsections, each algorithm separately. Since RLS does not have any relevant parameters, it will not have such a subsection. Its behavior is very identical to that of (1+1) EA, except that it is unable to escape local optima.

5.3.1 Simulated annealing

In SA, the two parameters are c and m, both of which relates to Meer's cooling scheme described by equation (3.2) on page 18. In, [19] it is suggested to use a low value for c and to use $m = 20n$ where n is the number of vertices in the graph. It was decided to test the values $c \in \{0.01, 0.1, 0.5, 1, 2\}$, $m \in \{n, 10n, 20n, 10, 100, 1\,000\}$. The results of these tests can be found in table 5.2.

Looking at the tests we immediately notice that in some cases, we get results that exceed the norm. For instance, the executions on `r500` given 30 minutes to execute usually yield solution with costs in the range 360 — 390, but for

Table 5.2: Benchmarks of parameters c and m for simulated annealing on two different graphs (r500 and r1000 of size $n = 500$ and $n = 1000$, respectively) using two different time limits. Results are average costs over 5 executions. The average column presents the total combined average of all executions. The best average for each problem is marked with bold font.

c	m	r500 5 min	r500 1 min	r1000 5 min	r1000 1 min	Average
0.01	n	382.15	379.88	526.44	527.48	453.99
0.01	$10n$	368.20	510.12	9 684.91	9 767.63	5 082.71
0.01	$20n$	357.41	4 667.33	9 704.34	9 732.46	6 115.39
0.01	10	381.35	381.27	526.18	526.07	453.72
0.01	100	379.42	380.17	527.58	532.11	454.82
0.01	500	383.16	386.29	520.96	530.03	455.11
0.01	1 000	380.89	385.39	523.94	528.37	454.65
0.10	n	378.17	388.65	523.09	682.36	493.07
0.10	$10n$	4 622.74	4 649.65	9 711.99	9 723.68	7 177.02
0.10	$20n$	4 644.98	4 671.04	9 732.21	9 771.14	7 204.84
0.10	10	382.56	383.74	525.19	**524.93**	454.11
0.10	100	380.96	382.63	528.22	529.48	455.33
0.10	500	384.48	386.47	523.69	535.17	457.45
0.10	1 000	375.41	377.50	525.04	687.28	491.31
0.50	n	371.89	372.62	524.71	9 782.51	2 762.93
0.50	$10n$	4 651.61	4 658.00	9 717.98	9 706.56	7 183.54
0.50	$20n$	4 648.03	4 668.92	9 670.59	9 712.53	7 175.01
0.50	10	379.41	382.50	526.35	530.69	454.74
0.50	100	384.41	387.04	524.98	537.52	458.49
0.50	500	375.43	374.73	525.87	896.35	543.10
0.50	1 000	363.26	4 700.99	520.32	9 717.06	3 825.41
1.00	n	369.30	**369.48**	9 646.32	9 765.93	5 037.76
1.00	$10n$	4 659.24	4 695.40	9 703.14	9 743.95	7 200.43
1.00	$20n$	4 638.49	4 673.08	9 710.13	9 734.09	7 188.95
1.00	10	381.90	384.60	523.74	527.91	454.54
1.00	100	381.26	378.17	522.35	532.99	**453.69**
1.00	500	370.08	373.04	522.94	9 683.60	2 737.41
1.00	1 000	**356.16**	4 679.98	9 672.22	9 763.84	6 118.05
2.00	n	363.38	4 670.45	9 705.63	9 746.16	6 121.40
2.00	$10n$	4 626.51	4 663.33	9 704.70	9 757.64	7 188.05
2.00	$20n$	4 644.14	4 697.67	9 747.13	9 768.57	7 214.38
2.00	10	379.96	377.87	529.83	530.42	454.52
2.00	100	381.93	383.72	524.06	540.22	457.48
2.00	500	362.14	4 662.44	**517.45**	9 778.83	3 830.21
2.00	1 000	4 116.40	4 695.51	9 694.72	9 723.99	7 057.65

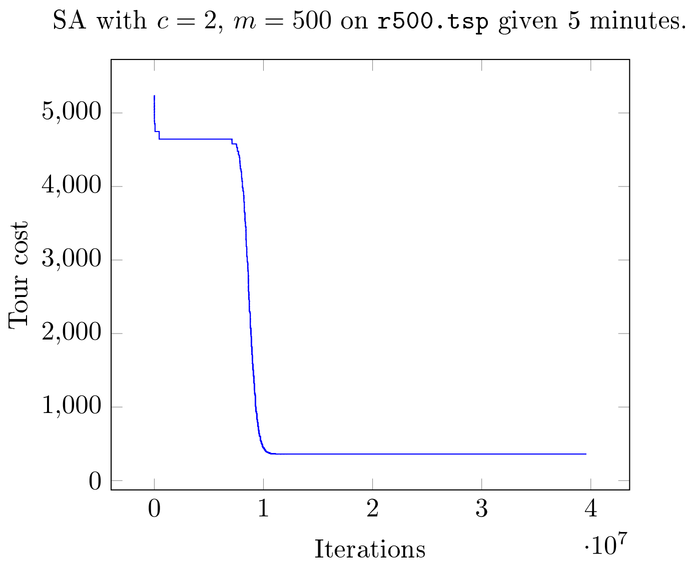


Figure 5.1: The typical development of a solution by simulated annealing.

some of the settings, the cost reaches more than 4 000. The reason for these bad performances is, the temperature never decreases enough for the the algorithms to reach the refinement phase. It is stuck in its initial phase, allowing too many bad decisions to actually get anywhere. If m is too big the starting temperature is too high. If c is too big, the temperature decreases too slowly. What values are classified as "too high" will of course depend on the execution time, because more time means more iterations, which in turn means more occasions for temperature decrease.

In figure 5.1, a plot of how SA typically develops its solution is shown. In the beginning, where bad decisions are prone to be made, SA improves the solution for a little while by chance. After approximately 500 000 iterations it completely stops evolving because it now chooses more bad decisions than good ones. After 7 million iterations, the temperature has decreases enough that the solutions develops rather quickly. Over the next 3 000 000 iterations, the solution improves from having a cost of 4 500 to a cost of about 400. After that, the process is slowed down as the number of good 2-opt swaps diminishes. After 12 million iterations, no more improvements are found.

The execution depicted by figure 5.1 has a lot of wasted time. The first 7 million iterations do much; in these iterations the algorithm is just decreasing its temperature in the hopes of eventually getting somewhere. The last 3/4 of the

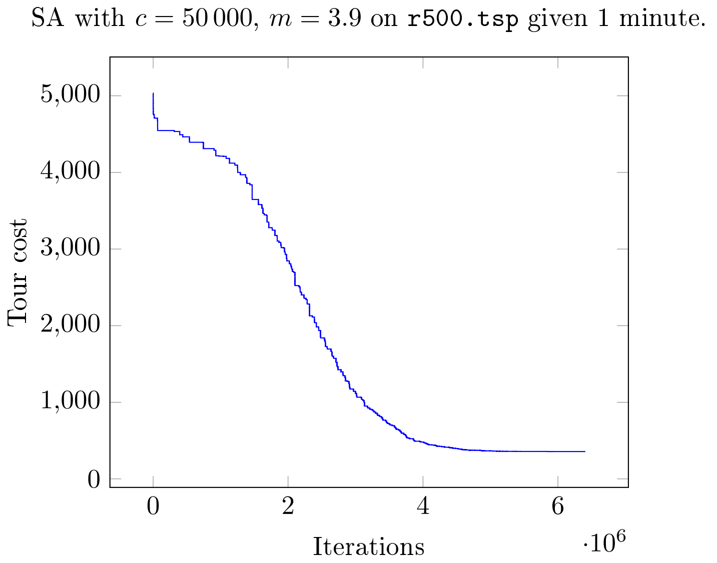


Figure 5.2: A more time-efficient use of simulated annealing.

entire execution are completely wasted as the algorithm yields no improvements there. In this phase, the temperature has decreased to such an degree that escaping local optima is improbable.

To fully take advantage of SA, c and m should be given values such that there is no wasted time on either side of "the big drop". In the example shown in the figure, a decrease in starting temperature (i.e. lowering m) would move the beneficial part of the plot to the beginning of the execution, and a slower cooling (i.e. higher values of c and/or m) would stretch the beneficial part over the entire execution, hopefully avoiding entering a local optimum too soon. The requirement of doing this is of course to have a fixed execution time. One can imagine a company doing planning with simulated annealing, giving each execution exactly 1 minutes to compute. In this case, very specific parameters can be found to ensure the 1 minutes are utilized completely by the algorithm. In figure 5.2 the above idea has been used to find some more ideal parameters[5] ($c = 50\,000$ and $m = 3.9$) for the same graph as in figure 5.1 but given only 1 minute. The solution produced is in this case better than the one shown in figure 5.1; it has less time to execute but uses that time much more efficiently. In average over 5 executions, for `r500` given 1 minute, the solution produced with this set of parameters has a cost of 358.97, beating all the other tested setups of

[5]The parameters were found empirically, as is proposed in [17]. The parameters are far from the suggested values in [19].

SA by at least 10.5. The drawback is that it cannot handle less execution time or a significantly larger graph.

The claim that stretching the cooling process over the entire duration is beneficial is also reflected by table 5.2, where we can see that high values of m (i.e. higher starting temperatures and slower cooling) generally yields better results. The exception is when the algorithm never reaches the refinement phase, in which case it yields horrible results with no noteworthy progress from the initial guess.

A systematic approach to determining good parameters for SA is presented in [20], but this approach is not further investigated. In this study, we choose to select a setup to do a wide variety of experiments on, thus very specific settings which only works well in certain scenarios are not considered. The tests mentioned earlier revealed that many combinations of values worked out almost equally well. It was decided to use $c = 1$ and $m = 100$ because it was ever so slightly better than the competition.

5.3.2 (1+1) evolutionary algorithm

The only concern with (1+1) EA is how many 2-opt swaps are applied within each iteration. As mentioned in section 3.3.3 on page 19, two approaches for finding the number of modifications in an iteration are used: Poisson distribution + 1 and Poisson distribution with 0 substituted by 1. For both of these methods, the parameters $\lambda \in \{0.01, 0.1, 0.5, 1, 2\}$ were tested for performance. The value 0.01 represents a "value close to zero", which will promote more ones with either type distribution. The larger values for λ will provide more iterations with multiple 2-opt swaps. The benchmarks are presented in table 5.3.

In figure 5.3, solution cost as typically developed by (1+1) EA is shown. In the beginning, many modifications lead to improvements causing the solution to rapidly improve. As the solution improves, it becomes increasingly difficult to find the 2-opt swaps that improve the overall solution. When the algorithm reaches a local optimum, it can occasionally break free by performing the right combination of multiple 2-opt swaps.

In the first phase, reaching a local optimum, low values of λ work best because there is a higher chance of selecting few rather than many 2-opt modifications that combined improves the tour. In the second phase, when a local optimum is reached and the algorithm needs to perform multiple 2-opt swaps to break out, a higher λ-value is beneficial, because it results in more iterations with a chance to escape the local optimum. Since the last phase is reached only when

Table 5.3: Benchmarks of parameter λ for (1+1) evolutionary algorithm on two different graphs (`r500` and `r1000` of size $n = 500$ and $n = 1000$, respectively) using two different time limits. Results are average costs over 5 executions. The average column presents the total combined average of all executions.

(1+1) evolutionary algorithm (k+1)

λ	`r500` 5 min	`r500` 1 min	`r1000` 5 min	`r1000` 1 min	Average
0.01	383.77	385.19	529.13	**531.42**	**457.38**
0.50	382.34	387.15	**525.97**	560.89	464.09
1.00	**381.42**	**378.73**	527.36	614.76	475.57
1.50	382.87	390.65	535.91	730.05	509.87
2.00	382.11	385.28	558.31	889.06	553.69

(1+1) evolutionary algorithm (substitution)

λ	`r500` 5 min	`r500` 1 min	`r1000` 5 min	`r1000` 1 min	Average
0.01	384.93	**377.39**	527.96	**533.68**	**455.99**
0.50	382.83	386.65	**519.69**	538.65	456.95
1.00	380.92	382.72	521.59	542.12	456.84
1.50	378.13	383.07	527.05	582.23	467.62
2.00	**376.68**	380.12	523.31	620.13	475.06

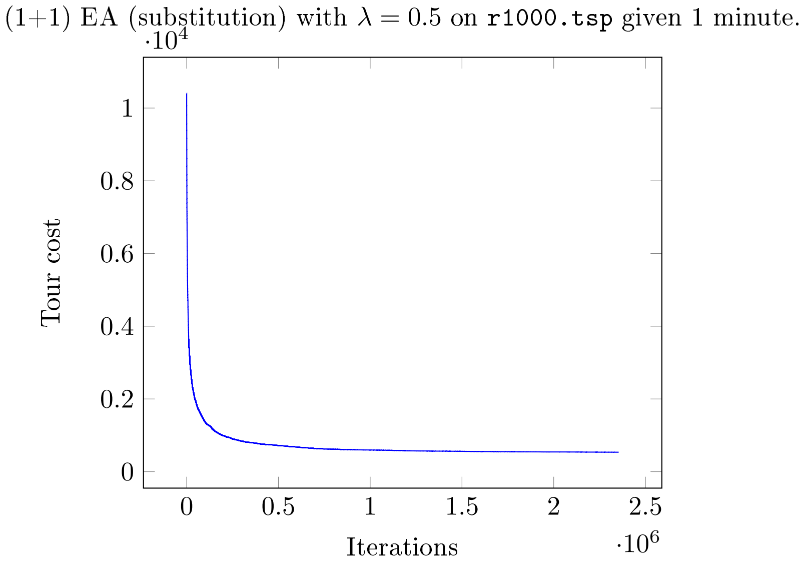


Figure 5.3: Typical solution development for (1+1) EA.

given enough time, executions that are given a long time to run favor a larger λ.

During the execution, a low value for λ lets the algorithm reach low-cost solutions faster than in case of high λ-values. However, once reached, a local optimum is easier to break free from with a higher λ. Interestingly, the algorithm seems able to reach local optima pretty fast. Given enough time, executions using a larger λ-value eventually reach the same solutions, but are then more likely to further improve on them due to their increased number of iterations with multiple 2-opt swaps.

The benchmarks in table 5.3 show that (1+1) EA (substitution) is superior to (1+1) EA (k+1). It also shows that $\lambda = 0.01$ performed best overall, but it has extensive variance in its performance, always being a top contender when given 1 minute, while being worst candidate when given 5. Instead, the choice landed on $\lambda = 1$ because it was nearly as good as 0.01, but it presented a decent performance in all the tested situations.

5.3.3 $\mathcal{MAX}$–$\mathcal{MIN}$ ant system

MMAS initializes with a lot of parameters. Recalling from section 3.3.4 on page 20, we have: α, β, ρ, m, τ_{min}, τ_{max} and the *update-type* (i.e. whether the update is based on the global-best or iteration-best solution). The parameters τ_{min} and τ_{max} are suggested by [11] to be set to $1/n$ and $1 - \tau_{min}$, respectively. Wanting a constant value for both parameters, $\tau_{min} = 0.01$ and $\tau_{max} = 0.99$ were selected. The other parameters will be benchmarked with $\alpha = 1$, $\beta \in \{1, 3, 7, 10, 13, 17, 20, 23\}$, $\rho \in \{0.01, 0.1, 0.25, 0.5\}$ and $m \in \{1, 5\}$. Because so many parameters were involved, the benchmark was only run on `r500` given 5 minutes execution time.

An excerpt of the benchmark is shown in table 5.4. The entire benchmark can be seen in appendix A.The results show that a high value for β is imperative for the success of the algorithm. With a high β, the other parameters seem to have little to no effect on the developed tours, which makes sense as the pheromone heuristic is downgraded, and the other parameters all relate to pheromone buildup on the graph. Picking a low β has a negative impact on the algorithm; it simply becomes too “random” when constructing its tours. In the end, the values $\alpha = 1$, $\beta = 20$, $\rho = 0.5$, $m = 5$ were chosen, and the global update rule uses the best-so-far tour to modify pheromone levels.

With a high β, MMAS starts mimicing the nearest neighbor heuristic. Looking at the solution created by MMAS, it seems that the distance heuristic forces the

Table 5.4: Excerpt of the benchmarks of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system's parameters. Here, $\alpha = 1$, $\tau_{min} = 0.01$, $\tau_{max} = 0,99$. The other parameters are presented in the table. Execution was performed on `r500` given 5 minutes execution time. The displayed costs are averages over 5 executions.

β	ρ	m	*update-type*	`r500` 5min
1	0.5	5	global-best	3 094.53
3	0.5	5	global-best	719.25
7	0.5	5	global-best	417.42
10	0.5	5	global-best	402.50
13	0.5	5	global-best	389.11
17	0.5	5	global-best	388.39
20	0.5	5	global-best	384.26
23	0.5	5	global-best	386.50

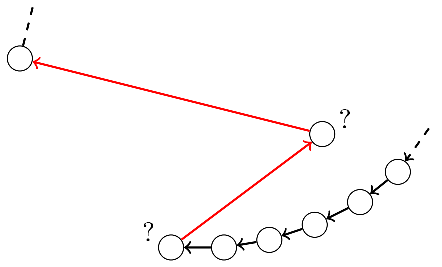


Figure 5.4: An example of bad decisions caused by the distance heuristic of MMAS.

algorithm into taking suboptimal choices during tour construction. The sitation is explained in figure 5.4 where the nearest neighbor is chosen at every turn (as is most probable); once the tour reaches a corner or a sitation where all nearby vertices have already been visited, it has to take a bad move in order to progress.

It is speculated that the pheromone heuristic has more success in non-metric problem instances. Also, in the original ant system by Dorigo *et al.* [7], all ants were used to update the pheromone trails, which may have given the pheromone a higher impact on the entire algorithm. In [28], where MMAS was introduced, local search was performed, which would be another way to ensure that the edges getting a benefit form the global update rule were in fact part of a good tour. However, using Euclidean TSP, MMAS as described in this study is very dependent on the distance heuristic, but to such a degree that it renders the pheromone heuristic practically useless. The problem is, that for the pheromone heuristic to be beneficial, it has to be provided with reasonable tours to per-

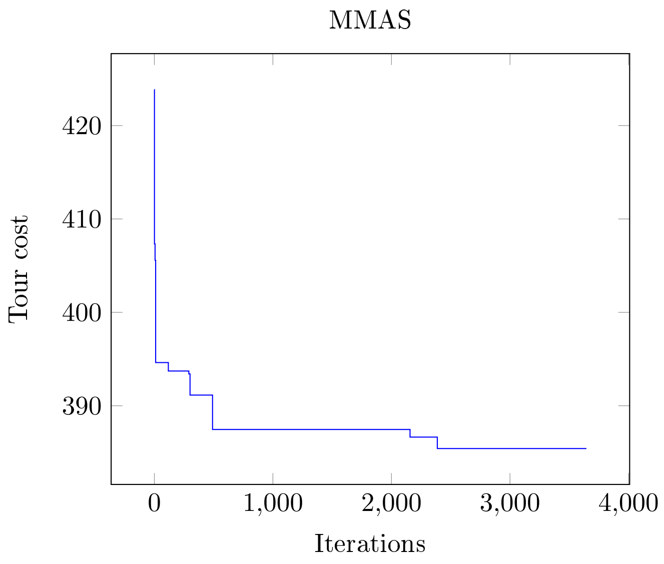


Figure 5.5: Typical solution development for MMAS.

form sensible pheromone updates; if β is too low, the pheromone levels become random gibberish, but if β is too high, the pheromone levels will never truly matter because the pheromone heuristic is overruled by the distance heuristic.

The usual solution development as seen in figure 5.5 looks like that of (1+1) EA, except that MMAS starts much lower and has fewer steps with actual improvement. The reason it starts out much better than the other algorithms is of course its distance heuristic used for constructing the tours. In general, MMAS only manage a small fraction of iterations compared to the other algorithms. When an improvement is found by MMAS, it is usually a complete restructuring of the solution, as every iteration builds its own solutions, which is in direct constrast to the other algorithms, where the tours are created by introducing a small change in a previous solution. As with (1+1) EA, the progress by MMAS slows down dramatically as the solution improves.

A point should be made that in comparison with the other presented algorithms, MMAS is rather complex. As such, the algorithm is rather rigid and firmly cemented in its approach: It cannot be easily modified to behave differently. For instance, RLS, SA and (1+1) EA can be modified to use a different modification instead of 2-opt, but such modifications seem to be hard to come by for MMAS.

5.4 Experiments with problem setups

In this section, we will conduct experiments on graph size, execution time, improvement of the initial guess and graph structure, as well as an experiement which compares the search heuristics to Concorde. In the following subsections, when referring to an algorithm, it is assumed to have the parameters chosen in section 5.3: SA has $c = 1$ and $m = 100$. (1+1) EA uses $\lambda = 1$ and substitutes zeros from the Poisson distribution with ones. MMAS has $\alpha = 1$, $\beta = 20$, $\rho = 0.5$, $m = 5$, $\tau_{min} = 0.01$, $\tau_{max} = 0.99$ and uses the global-best solution for pheromone updates.

5.4.1 Execution time and graph size

A very relevant question regarding the randomized search heuristics is how well they perform under time constraints and how increasing the graph size affects their results. These two aspects are contained in the same test as they are connected to each other: Increasing the graph size will of course increase the difficulty of the problem, making the search heuristics require more time to produce good results.

For the experiments we will use five graphs with increasing difficulties: `r100`, `r250`, `r500`, `r750` and `r1000`. We will operate with five different execution times; 5 seconds, 30 seconds, 1 minute, 5 minutes and 30 minutes.

The results of the experiments are shown in table 5.5. In the table we see that the overall trend is that the algorithms provide the best solutions when given the most time. Time and problem size are also undoubtedly connected; the bigger the problem, the more time is needed to provide a good answer.

In general, RLS and SA seem to settle in a local optima, and stop improving once they are reached. If we look at the rows for RLS and SA on `r100` and `r250`, the results are actually similar to each other, regardless of the execution time. This is because the algorithms generate close-to-optimal solutions within the first 5 seconds, and are unable to (noticeable) improve on them – hinting that they reach local optima within that time. On larger graphs, this behavior appears only later because the local optima are harder to reach. Take for instance RLS on `r1000`; here we see improvements for the first 5 minutes instead of the first 5 seconds. SA was described as being able to escape local optima, but in practice it does not work out because the temperature reaches too low values too fast. The algorithm avoids these local optima by selecting a good overall structure *during* cooling as discussed in section 3.3.2 on page

Table 5.5: Randomized local search (RLS), simulated annealing (SA), (1+1) evolutionary algorithm (EA) and $\mathcal{MAX}$–$\mathcal{MIN}$ ant system (MMAS) executed on five different graphs; `r100`, `r250`, `r500`, `r750` and `r1000`, with five different execution times; 5 seconds, 30 seconds, 1 minute, 5 minutes and 30 minutes. The displayed costs are averages over five attempts.

`r100`

Algorithm	30 min	5 min	1 min	30 sec	5 sec
RLS	167.48	167.56	165.72	164.16	165.81
SA	162.32	161.68	163.39	164.62	159.23
(1+1) EA	152.87	152.44	158.15	162.59	161.88
MMAS	152.03	153.52	153.60	155.19	157.94

`r250`

Algorithm	30 min	5 min	1 min	30 sec	5 sec
RLS	272.34	274.07	273.97	272.99	271.95
SA	272.00	271.82	276.81	272.34	270.55
(1+1) EA	258.67	270.36	274.21	270.83	277.48
MMAS	272.10	274.27	277.20	276.94	282.20

`r500`

Algorithm	30 min	5 min	1 min	30 sec	5 sec
RLS	376.84	381.47	379.13	383.81	392.54
SA	385.51	381.26	383.00	384.09	399.70
(1+1) EA	381.61	381.92	381.62	382.70	406.12
MMAS	384.39	385.21	387.90	394.09	395.11

`r750`

Algorithm	30 min	5 min	1 min	30 sec	5 sec
RLS	455.52	456.03	452.32	459.33	603.65
SA	451.41	450.76	450.75	449.91	728.67
(1+1) EA	451.56	452.81	452.16	458.72	692.71
MMAS	467.68	468.93	473.18	475.89	477.84

`r1000`

Algorithm	30 min	5 min	1 min	30 sec	5 sec
RLS	524.83	524.50	532.18	566.84	990.03
SA	530.28	522.35	535.19	577.72	1 987.45
(1+1) EA	525.51	523.59	544.58	609.03	1 167.46
MMAS	546.15	547.42	551.12	554.12	563.51

17; a restructure *after* the temperature has been significantly decreased is very improbable. Because RLS and SA get stuck in local optima, the experiments in which they do can seem random (e.g. on `r100` where SA is best when given only 5 seconds). This is because the performance is largely dependent on *which* local optimum the algorithm hit; whether it is close or far from the optimal solution. The results indicate that such behavior is more significant on small problem instances.

(1+1) EA usually starts slower than RLS, but eventually catches up with it. This is a clear indicator that (1+1) EA is absolutely able to escape local optima where RLS is not. The behavior also suggests that (1+1) EA does not benefit from multiple 2-opt swaps at the same time in the beginning of the execution, but rather benefit from them in the later stages where they are needed to escape local optima. This behavior is especially apparent in the experiments on `r1000`, where (1+1) EA has not caught up with RLS and SA even after 1 minute of execution, and it is expected that the trend will only be further highlighted as problem sizes increase.

MMAS is very quick to generate good tours compared to the other algorithms, especially on large problems. This is an obvious benefit from its distance heuristic. MMAS does improve its solution over time, but is usually outpaced by (1+1) EA. It also seems that the larger the graph becomes, the harder it becomes for MMAS to keep in front: On `r100` it stays ahead the entire time, only getting caught by (1+1) EA after 30 minutes, but on `r1000` it is behind all other algorithms after 1 minute, and only proceeds to fall further behind as more time is spent.

Overall, for the parameters chosen for the different algorithms, allowing 30 minutes of execution time seems to be too much, as the algorithms are mostly settled on good solutions after only 5 minutes.

5.4.2 Improving the initial guess

RLS, SA and (1+1) EA all use an initial guess as a starting point for their searches. If the initial guess is a good guess instead of a random guess with a high expected cost, the algorithms might be able to achieve better results, especially when their execution time is limited. In this experiment we see how the three algorithms perform when Christofides' 3/2-approximation algorithm is used to provide the initial guess. MMAS has not been included in this experiment because it does not use an initial guess. One could base the initial distribution of pheromone on the result from the approximation algorithm and see how it affects MMAS, but that experiment was chosen not to be conducted.

Table 5.6: Randomized local search (RLS), simulated annealing (SA) and (1+1) evolutionary algorithm (EA) executed on three different graphs; `r100`, `r500` and `r1000`, with three different execution times; 5 seconds, 30 seconds and 1 minute, using an approximation by Christofides' algorithm as initial guess. The displayed costs are averages over five attempts. The cost of the approximation by Christofides' algorithm is also presented in the table.

`r100`

Algorithm	5 sec	30 sec	1 min
Christofides'		174.60	
RLS	157.78	156.6	157.15
SA	162.03	160.61	162.46
(1+1) EA	156.83	156.7	154.19

`r500`

Algorithm	5 sec	30 sec	1 min
Christofides'		398.21	
RLS	358.59	358.44	357.2
SA	396.63	382.47	382.09
(1+1) EA	360.92	358.56	356.79

`r1000`

Algorithm	5 sec	30 sec	1 min
Christofides'		550.16	
RLS	526.31	500.04	498.23
SA	550.16	550.16	532.53
(1+1) EA	531.26	504.39	497.8

The experiments are conducted on the graphs: `r100`, `r500` and `r1000`. The time restrictions is relatively low at 5 seconds, 30 seconds and 1 minute. These times constraints include only the search heuristics; the time it takes to calculate the initial guess by Christofides' algorithm is not counted towards this limit. As mentioned earler, Christofides' has a time-complexity of $O(n^3)$, which makes it require more clock cycles than the random guess: On `r1000`, it took approximately 1 second to compute the initial tour, which is not completely insignificant. For `r100`, however, the computation time was too low to measure.

The results of the tests are shown in table 5.6. The tours generated by Christofides' approximation algorithm have costs of 174.60, 398.21 and 550.16 for `r100`, `r500` and `r1000` respectively. The numbers tells us that for `r100` and `r500`, none of the algorithms get a serious benefit by executing for 1 minute compared to 5 seconds. For `r1000`, however, it is a different matter. Here, RLS and (1+1)

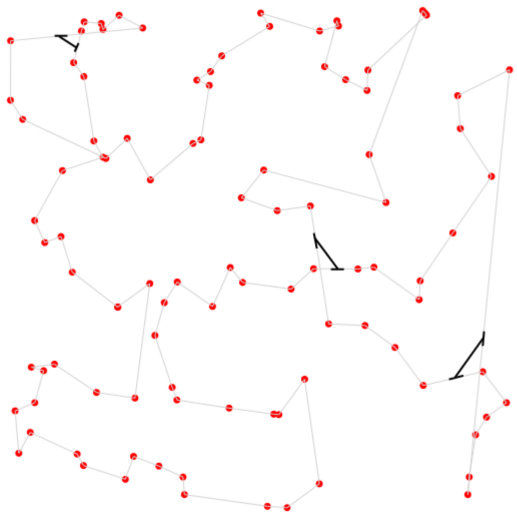


Figure 5.6: Tour calculated by Christofides' algorithm on `r100`. The black lines indicate suggested 2-opt swaps to improve the tour.

EA are able to improve their solutions quite a bit, because the development of solutions are slower when graphs are larger.

It is very interesting to compare the results in table 5.6 with those found in table 5.5. SA does not receive a significant improvement by using Christofides' algorithm for the initial guess, but RLS and (1+1) EA do to an almost extreme degree. In this experiment, both algorithm reach tour costs below 500 after one minute on `r1000`, whereas they end up at approximately 525 on the same graph after 30 minutes when using a random initial guess. To understand the reason why, we look at a tour given by Christofides' algorithm in figure 5.6. The algorithm makes use of a lot of sensible edges, but occasionally has to settle with some very bad edges. The inclusion of these tend to introduce intersections between edges, and these situations are easily resolved by 2-opt swaps. By performing just the few marked 2-opt swaps in the figure, we have suddenly produced a very reasonable result.

Taking a look at SA in table 5.6, we see that it is not able to take advantage of the good initial guess to the same degree as RLS and (1+1) EA. In fact, it produces results very similar to those in table 5.5. This is because SA is designed to break down the structure of the solution before slowly patching it together again, so in the beginning of the execution, when the temperature is low, it accepts a lot of bad swaps and moves away from an otherwise good starting point.

Table 5.7: Randomized local search (RLS), simulated annealing (SA), (1+1) evolutionary algorithm (EA) and $\mathcal{MAX}$–$\mathcal{MIN}$ ant system (MMAS) executed on three different graphs; `c100`, `c500` and `c1000`, with vertices moved up to d_m away in a random direction from their original position. The displayed costs are averages over five attempts, each execution given 5 minutes.

`c100`

Algorithm	$d_m = 0$	$d_m = 2$	$d_m = 4$	$d_m = 6$	$d_m = 8$	$d_m = 10$
RLS	64.54	80.38	90.63	103.62	102.66	107.67
SA	62.81	77.46	87.07	99.51	99.85	103.79
(1+1) EA	61.62	76.47	86.06	96.69	96.30	101.15
MMAS	64.35	78.03	90.75	101.45	98.42	101.72

`c500`

Algorithm	$d_m = 0$	$d_m = 2$	$d_m = 4$	$d_m = 6$	$d_m = 8$	$d_m = 10$
RLS	162.63	218.44	278.77	314.97	325.84	330.48
SA	159.37	214.55	283.17	310.21	322.96	329.58
(1+1) EA	158.15	215.05	281.28	310.00	326.30	334.35
MMAS	168.34	227.30	285.76	321.21	332.25	342.75

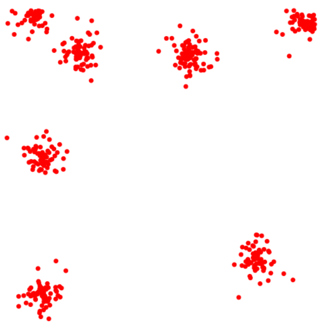
`c1000`

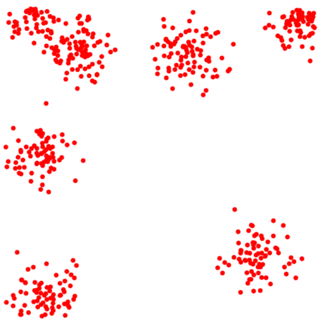
Algorithm	$d_m = 0$	$d_m = 2$	$d_m = 4$	$d_m = 6$	$d_m = 8$	$d_m = 10$
RLS	191.97	327.31	420.52	451.95	471.42	483.43
SA	194.01	323.31	423.28	457.39	477.29	483.37
(1+1) EA	193.50	324.75	423.61	450.91	468.26	479.46
MMAS	210.79	341.80	442.14	475.86	495.97	502.03

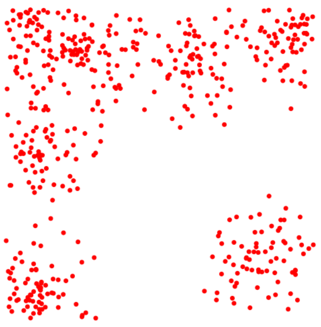
5.4.3 Gradual modification of graph structure

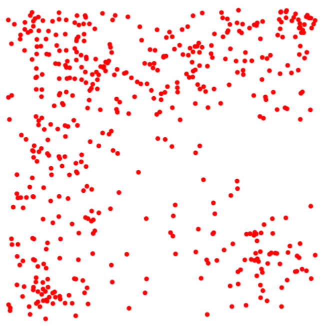
In this experiment we wish to test how a strong structure in the graph affects the different algorithms, and whether they behave differently than on an unstructured graph. To do this, we start out with graphs where the vertices are placed in clusters, and then introduce more and more randomness to the graph. The graphs `c100`, `c500` and `c1000` will be used, all of which have their vertices in a 20×20 area. Six different maximum coordinate shifts (as described in section 2.3.3 on page 10) will be examined for each graph: $d_m \in \{0, 2, 4, 6, 8, 10\}$, and every execution will be given 5 minutes to execute. Figure 5.7 shows how this introduced randomness affects a clustered graph.

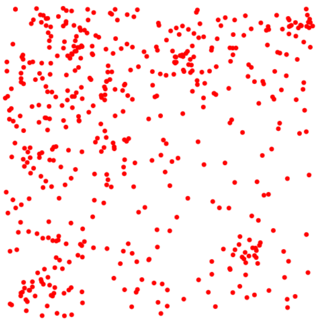
The results, presented in table 5.7, shows that the tour lengths increase in cost as the clusters are dissolved (i.e. higher shift). This is not surprising, as the vertices


(a) $d_m = 0$


(b) $d_m = 2$


(c) $d_m = 4$


(d) $d_m = 6$


(e) $d_m = 8$

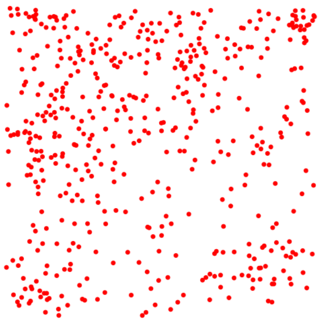

(f) $d_m = 10$

Figure 5.7: The graph `c500` with every vertex moved up to d_m from their original position in a random direction.

become spread on a larger area that the tours will have to cover. The results also indicate that there are no significant correlation between how clustered a graph is and how well the search heuristics perform. RLS, SA and (1+1) EA perform at a comparable level in regards to each other, as was the case in the experiments in section 5.4.1 (table 5.5), and MMAS is slightly behind. This is the picture, regardless of the problem instance and regardless of the selected maximum shift.

5.4.4 Comparisons with Concorde on TSPLIB instances

Hitherto, the experiments have focused on comparing the search heuristics with each other, but have not provided data on how well their solutions compare to optimal solutions. To make this comparison, the Concorde software is used.

To be able to compare the randomized search heuristics with Concorde, we have to employ the same cost function. Therefore, in this experiment, the cost function in equation (2.4) on page 11 is used by the search heuristics. The new cost function introduces a problem: Because of the relatively low dimensions on the created graphs (shown in table 5.1 on page 39), many edges have equal costs due to the rounding. This causes Concorde to perform exceptionally well, giving an incorrect impression of its power. To give a more fair comparison in this experiment, it was decided to use the TSPLIB instances: `pr439`, `u724` and `vm1084`.

The experiments were performed by first allowing Concorde to solve the problem instances, measuring the time t it took to do so. Then, the search heuristics were executed on the problems, first given t time, then $1/2t$ and finally only $1/10t$. The idea is to see how close the search heuristics get to Concorde in similar time, and how well they perform if their executions are limited to less than what Concorde uses.

The results of the experiments can be seen in table 5.8. The randomized search heuristic obviously cannot compete with Concorde when using as much time as it needs to find the optimal solution, but we do notice that the solutions calculated in time $1/2t$ lie at approximately 115% of the optimal solution, which is pretty acceptable. This time-saving is more important on bigger graphs, where $1/2t$ constitutes more time in absolute terms.

Again, search heuristics are useful in situations when an optimal solution is not strictly necessary, and when computation resources (in the form of power or time) are scarce. In the tested scenarios, half the computation time can be saved if you are willing to accept solutions that are 15% higher than the

Table 5.8: Randomized local search (RLS), simulated annealing (SA), (1+1) evolutionary algorithm (EA) and $\mathcal{MAX}$–$\mathcal{MIN}$ ant system (MMAS) executed on three different graphs; `pr439`, `u724` and `vm1084`. For each graph, C denotes the optimal solution and t the time Concorde used to find this solution. The algorithms were executed with different time limits relative to t. The displayed costs are relative to the optimal solution over five executions.

`pr439`

$t = 65$ seconds
$C = 107\,217$

Algorithm	$1/10t$	$1/2t$	t
RLS	114.19%	114.17%	114.17%
SA	113.24%	113.69%	112.22%
(1+1) EA	116.89%	112.28%	113.93%
MMAS	115.43%	115.16%	115.01%

`u724`

$t = 85$ seconds
$C = 41\,910$

Algorithm	$1/10t$	$1/2t$	t
RLS	132.82%	114.24%	113.40%
SA	146.36%	113.62%	113.60%
(1+1) EA	141.41%	114.11%	113.06%
MMAS	119.59%	118.88%	118.91%

`vm1084`

$t = 275$ seconds
$C = 239\,297$

Algorithm	$1/10t$	$1/2t$	t
RLS	140.64%	115.29%	114.35%
SA	147.28%	115.62%	113.70%
(1+1) EA	170.03%	116.11%	115.22%
MMAS	120.26%	119.26%	118.10%

optimal. Also, if a tour needs to be found within a given time frame, a TSP solver such as Concorde will not suffice, as it cannot guarantee finishing in time, and stopping it prematurely causes it to yield no solution.

CHAPTER 6

Discussion

When experimenting with randomized algorithms, two isolated results cannot be compared because the randomization can cause a fluctuation in behavior. Instead, multiple executions are performed, and the average results are compared. This was how the experiments were carried out in this study. The sampling size, i.e. the number of executions these averages were based on, was somewhat low. The averages were calculated based on five repetitions, but this number cannot be considered large enough if one wants to balance the random variation. Symptoms of this can be seen in several of result tables, for instance table 5.5 and page 49, where RLS seem to perform worse when given 30 or 5 minutes, compared to 60, 30 and 5 seconds.

The low number of repetitions is a consequence of time constraints. As a result of these uncertainties, small variations in the results were ignored. The analyses of the experiments only consider those results that are very clear. A consequence of this carefulness is, that it might have caused one to miss results that are valid, but too insignificant to be considered as such. Another consequence of the uncertainty is, that results that fit well with the author views are more likely to be taken up, and if results contradict those views they might be more likely to be rejected as “random variation”, e.g. when RLS performed worse given long time. Such tendencies should be avoided as it puts a bias on the analyses, which is yet another reason for increasing the number of repetitions.

In addition to increasing the number of repetitions, more fluent experiments could be favorable. For instance, when experimenting with problem sizes, graphs with 100, 250, 500, 750 and 1000 vertices were used. If the increments between these experiments were reduced, it would be easier to more precisely pinpoint where the differences in behavior between algorithms manifest themselves. Doubly so for execution times; if an experiment shows that (1+1) EA overtakes RLS between the 5 and 30 minutes marks, it would be nice to know more precisely when it happens.

The results from this study can be extended further. The focus of this study was narrowed down to a single problem type, the Euclidean TSP. It would be interesting to explore how the search heuristics perform in other problems, for instance those mentioned in section 2.1: Variations of TSP (on page 4. There was experimented, to a limited degree, with a slightly modified distance formula, but especially general TSP, where the cost function does not comply with the triangle inequality theorem and can even give negative values, and asymmetric TSP where costs differ depending on the direction, would be interesting problems to examine.

It was found that SA could greatly benefit from fine-tuning its parameters to a given problem setup. It would be interesting to examine if a reliable method to determine these parameters, such as presented in [20], can provide competitive parameters on a wide range of problems.

As the project evolved, it became clear that the different search heuristics have their strength in different aspects of the searches. For instance, RLS reaches good solutions faster than (1+1) EA, but (1+1) EA is able to improve its solution where RLS would reach a local optimum. MMAS beats RLS and (1+1) EA when the algorithms are given only very short time, or when they execute on relatively small graphs. SA performed poorly compared to RLS and (1+1) EA, but as mentioned it still has its merit because it can benefit greatly from problem-specific parameterization.

Because the algorithms have their individual strengths, it would be interesting to see if one could take advantage of these differences by combining the algorithms. For instance, it is speculated that it would be beneficial to start by using RLS and switch over to (1+1) EA when closing in on a local optimum. The same idea could be applied to parameters within the same algorithm. A low λ could be given to (1+1) EA in the beginning of an execution, and it could then slowly be increased as the solution develops. In case of MMAS we saw that the distance heuristic was probably too dominant. If the algorithm could start out with a large β, and slowly decrease it as the pheromone levels reach sensible levels on the edges, the pheromone heuristic might prove to be more useful. These ideas would all depend on some form of meta-knowledge to be able to decide when and/or how the parameters should change during execution.

Chapter 7

Conclusion

In this study, four randomized search heuristic were implemented; randomized local search (RLS), simulated annealing (SA), (1+1) evolutionary algorithm (EA) and $\mathcal{MAX}$–$\mathcal{MIN}$ ant system (MMAS). The search heuristics were compared by running experiments on random Euclidean TSP instances. The experiments covered the use different problem sizes and time limits, clustered graphs and uniform graphs as well as using an improved initial guess (not applicable to MMAS). In addition to this, they were compared to the state-of-the-art TSP solver called Concorde.

RLS found good solutions fast, compared to the other algorithms. However, when given too much time, an obvious weakness presented itself; it reaches local optima from which it is unable to progress. This downside is more severe on small graphs or when given a long time to execute; randomized local search simply cannot take advantage of the extra time because at some point, it simply has no moves it can possibly make.

SA was shown to work exceptionally well when it was possible to find and use parameters that match a given problem. If the size of the problem instance and the amount of execution time are known, parameters could be found for simulated annealing such that it performs exceptionally well. This is beneficial if you have a lot of similar problems that needs solving. However, such parameters are very fragile; a change in execution time, computation power and/or graph size will significantly decrease the efficiency of SA with such parameters. For this reason, some more generally viable parameters were used in the experiments, but these parameters caused the cooldown to happen to fast, and the simulated annealing ended up comparable to randomized local search in most cases.

(1+1) EA proved to be the uncontested best of the tested algorithms when given a long execution time. It proved to have a progression that was a little slower than randomized local search, but where randomized local search reached local optima, (1+1) evolutionary algorithm continued to improve, albeit at a slow rate. The (1+1) evolutionary algorithm has a parameter which allows one to balance the speed of initial progression versus the chance of breaking local optima down the road. (1+1) EA draws its number of modification for a given iteration from a Poisson distribution. In [29] it is suggested to use the number + 1 to avoid iterations with zero modifications. In this study, it was suggested to simply substitute zeros with ones, which proved to be an improvement in the tested cases.

The MMAS was shown to be very good on the short run because of its distance heuristic. On small problems it proved to be very competitive compared to the other search heuristics, but when reaching the large graphs with 750 or 1 000 vertices, it would fall behind. The pheromone heuristic turned out to be mostly insignificant. Changing the parameters related to this heuristic had no noticeable difference on the performance of the algorithm.

Experiments involving changing the time limit and graph size helped to emphasize the strengths and weaknesses of the algorithms. It clearly demonstrated how RLS was unable to take advantage of increased time – and how (1+1) evolutionary was. Unsurprisingly, a low amount of time combined with large graphs resulted in the algoritms not being able to produce good solutions. On a 1 000 vertex graph, the solutions provided after 5 seconds were approximately 2 — 4 times worse than those found after 30 seconds. The exception to this is MMAS, which was able to produce good solutions immediately because of its distance heuristic. For smaller graphs, however, solutions produced in 5 seconds are much more optimal. For a 100 vertex graph, the maximum difference between 5 seconds and 30 minutes executions were shown by (1+1) EA, with the 5 second execution giving a solution approximately 6% higher than what was produced after 30 minutes.

An experiment in which RLS, SA and (1+1) EA were given an initial guess in the form an approximation from Christofides' approximation algorithm, was also conducted. Compared to the random initial guess, it turned out to improve RLS and (1+1) EA to a surprisingly high degree. On the basis of the approximation, they could on a 1 000 vertex problem in 1 minute produce solutions that were approximately 5% below the solutions they could otherwise produce in 30 minutes. It turned out that Christofides' algorithm makes a great overall structure, but occasionally has to choose some bad moves. RLS and (1+1) EA using the 2-opt swap proved to be very potent at cleaning up these bad moves. SA did *not* have this trait; it simply accepts too many bad moves in the beginning of the execution, destroying the advantage the approximation supplied it with. Thus, the performance of simulated annealing in this experiment was not improved noticeably.

Experiments where a clustered graph was step-wise transformed to a more uniform graph were also performed. At each step of this transformation, the algorithms were executed to see if the structure of the graph had a significance for any of the algorithms. The restructuring of the graph showed no sign of affecting any of the algorithms. In relation to each other they performed similar to earlier experiments. This result indicated that there is no basis for a correlation between clustered graphs and the performance of the algorithms.

Finally, an experiment was conducted to compare the randomized search heuristics with the exact TSP solver Concorde, on graph with 439, 724 and 1084 vertices, respectively. The results from this experiment show that the randomized search heuristics were in their basic forms consistently able to produce solutions with costs of approximately 115% compared to the optimal solution in half time than Concorde needed. A point was made, that selection of search heuristic against an exact solver can depend on available computational resources and/or the need for establishing time limits. Concorde is clearly a powerful piece of software, but the randomized search heuristics still have their place.

Bibliography

[1] D. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. Concorde TSP solver. `http://www.math.uwaterloo.ca/tsp/concorde/index.html`. Accessed: 2015-01-24.

[2] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2006.

[3] J. Beardwood, J. H. Halton, and J. M. Hammersley. The shortest path through many points. *Mathematical Proceedings of the Cambridge Philosophical Society*, 55:299–327, 10 1959.

[4] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[5] S. Cook. The P versus NP problem. In *Clay Mathematical Institute; The Millennium Prize Problem*, 2000.

[6] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[7] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS-PART B*, 26(1):29–41, 1996.

[8] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1–2):51 – 81, 2002.

[9] W. I. Gasarch. Guest column: The second P=?NP poll. *SIGACT News*, 43(2):53–77, 2012.

[10] G. Gutin, A. Yeo, and A. Zverovich. Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the tsp. *Discrete Applied Mathematics*, 117(1-3):81–86, 2002.

[11] W. Gutjahr. Mathematical runtime analysis of ACO algorithms: survey on an emerging issue. *Swarm Intelligence*, 1(1):59–79, 2007.

[12] M. Hahsler and K. Hornik. Tsp—infrastructure for the traveling salesperson problem. *Journal of Statistical Software*, 23(2):1–21, 12 2007.

[13] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. In *Proceedings of the 1961 16th ACM National Meeting*, ACM '61, pages 71.201–71.204, New York, NY, USA, 1961. ACM.

[14] K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 2000.

[15] R. Jonker and T. Volgenant. Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, 2(4):161 – 163, 1983.

[16] M. Karpinski, M. Lampis, and R. Schmied. New inapproximability bounds for TSP. *CoRR*, abs/1303.6437, 2013.

[17] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[18] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the travelling-salesman problem. *Operations Research*, 21:498–516, 1973.

[19] K. Meer. Simulated annealing versus metropolis for a TSP instance. *Information Processing Letters*, 104(6):216 – 219, 2007.

[20] M.-W. Park and Y.-D. Kim. A systematic procedure for setting parameters in simulated annealing algorithms. *Computers and Operations Research*, 25(3):207 – 217, 1998.

[21] G. Reinelt. TSPLIB. `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`. Accessed: 2015-01-25.

[22] G. Reinelt. TSPLIB – a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, Nov. 1991.

[23] G. Reinelt. TSPLIB95, 1995.

[24] G. Rozenberg, T. Bäck, and J. N. Kok. *Handbook of Natural Computing*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[25] S. Sahni and T. Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, July 1976.

[26] D. Simon. *Evolutionary Optimization Algorithms*. Wiley, 1st edition, Apr. 2013.

[27] D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, May 2004.

[28] T. Stutzle and H. Hoos. MAX-MIN ant system and local search for the traveling salesman problem. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 309–314, Apr 1997.

[29] A. M. Sutton and F. Neumann. A parameterized runtime analysis of evolutionary algorithms for the euclidean traveling salesperson problem. *CoRR*, abs/1207.0578, 2012.

Appendix A

Parameters for $\mathcal{MAX}$–$\mathcal{MIN}$ ant system

Table A.1: Benchmarks of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system's parameters. Here, $\alpha = 1$, $\tau_{min} = 0.01$, $\tau_{max} = 0,99$. The other parameters are presented in the table. Execution was performed on `r500` given 5 minutes execution time. The displayed costs are averages over 5 executions.

β	ρ	m	*update-type*	`r500` 5min
1	0.01	1	iteration-best	3171.45
3	0.01	1	iteration-best	722.54
7	0.01	1	iteration-best	430.04
10	0.01	1	iteration-best	403.01
13	0.01	1	iteration-best	396.12
17	0.01	1	iteration-best	391.52
20	0.01	1	iteration-best	390.92
23	0.01	1	iteration-best	386.88
1	0.1	1	iteration-best	3089.49
3	0.1	1	iteration-best	739.76
7	0.1	1	iteration-best	426.41
10	0.1	1	iteration-best	402.84
13	0.1	1	iteration-best	394.18
17	0.1	1	iteration-best	392.64
20	0.1	1	iteration-best	390.04
23	0.1	1	iteration-best	385.17

Table A.1: Benchmarks of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system's parameters. Here, $\alpha = 1$, $\tau_{min} = 0.01$, $\tau_{max} = 0,99$. The other parameters are presented in the table. Execution was performed on `r500` given 5 minutes execution time. The displayed costs are averages over 5 executions.

β	ρ	m	*update-type*	`r500` 5min
1	0.25	1	iteration-best	3168.23
3	0.25	1	iteration-best	732.15
7	0.25	1	iteration-best	425.68
10	0.25	1	iteration-best	403.00
13	0.25	1	iteration-best	397.49
17	0.25	1	iteration-best	389.68
20	0.25	1	iteration-best	387.68
23	0.25	1	iteration-best	385.26
1	0.5	1	iteration-best	3095.42
3	0.5	1	iteration-best	732.84
7	0.5	1	iteration-best	427.54
10	0.5	1	iteration-best	398.43
13	0.5	1	iteration-best	393.10
17	0.5	1	iteration-best	390.34
20	0.5	1	iteration-best	391.98
23	0.5	1	iteration-best	388.13
1	0.01	1	global-best	3122.99
3	0.01	1	global-best	738.85
7	0.01	1	global-best	431.74
10	0.01	1	global-best	398.56
13	0.01	1	global-best	395.31
17	0.01	1	global-best	392.64
20	0.01	1	global-best	386.30
23	0.01	1	global-best	388.80
1	0.1	1	global-best	3120.22
3	0.1	1	global-best	728.16
7	0.1	1	global-best	427.28
10	0.1	1	global-best	391.00
13	0.1	1	global-best	396.61
17	0.1	1	global-best	390.84
20	0.1	1	global-best	390.36
23	0.1	1	global-best	387.90
1	0.25	1	global-best	3128.87
3	0.25	1	global-best	727.69
7	0.25	1	global-best	425.28
10	0.25	1	global-best	405.72
13	0.25	1	global-best	397.69
17	0.25	1	global-best	391.60
20	0.25	1	global-best	392.07
23	0.25	1	global-best	390.56
1	0.5	1	global-best	3177.14

Table A.1: Benchmarks of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system's parameters. Here, $\alpha = 1$, $\tau_{min} = 0.01$, $\tau_{max} = 0,99$. The other parameters are presented in the table. Execution was performed on `r500` given 5 minutes execution time. The displayed costs are averages over 5 executions.

β	ρ	m	*update-type*	`r500` 5min
3	0.5	1	global-best	733.59
7	0.5	1	global-best	427.69
10	0.5	1	global-best	402.01
13	0.5	1	global-best	395.76
17	0.5	1	global-best	389.61
20	0.5	1	global-best	385.82
23	0.5	1	global-best	389.26
1	0.01	5	iteration-best	3066.15
3	0.01	5	iteration-best	707.49
7	0.01	5	iteration-best	421.93
10	0.01	5	iteration-best	402.41
13	0.01	5	iteration-best	387.61
17	0.01	5	iteration-best	388.68
20	0.01	5	iteration-best	387.67
23	0.01	5	iteration-best	384.47
1	0.1	5	iteration-best	3121.18
3	0.1	5	iteration-best	703.11
7	0.1	5	iteration-best	416.94
10	0.1	5	iteration-best	394.37
13	0.1	5	iteration-best	389.13
17	0.1	5	iteration-best	389.00
20	0.1	5	iteration-best	389.54
23	0.1	5	iteration-best	387.29
1	0.25	5	iteration-best	3125.57
3	0.25	5	iteration-best	705.21
7	0.25	5	iteration-best	419.42
10	0.25	5	iteration-best	401.11
13	0.25	5	iteration-best	393.57
17	0.25	5	iteration-best	385.05
20	0.25	5	iteration-best	386.01
23	0.25	5	iteration-best	399.38
1	0.5	5	iteration-best	3100.83
3	0.5	5	iteration-best	715.57
7	0.5	5	iteration-best	423.65
10	0.5	5	iteration-best	401.62
13	0.5	5	iteration-best	391.73
17	0.5	5	iteration-best	386.24
20	0.5	5	iteration-best	385.42
23	0.5	5	iteration-best	385.39
1	0.01	5	global-best	3142.04
3	0.01	5	global-best	701.87

Table A.1: Benchmarks of $\mathcal{MAX}$–$\mathcal{MIN}$ ant system's parameters. Here, $\alpha = 1$, $\tau_{min} = 0.01$, $\tau_{max} = 0,99$. The other parameters are presented in the table. Execution was performed on `r500` given 5 minutes execution time. The displayed costs are averages over 5 executions.

β	ρ	m	*update-type*	`r500` 5min
7	0.01	5	global-best	423.82
10	0.01	5	global-best	400.86
13	0.01	5	global-best	394.49
17	0.01	5	global-best	388.54
20	0.01	5	global-best	388.41
23	0.01	5	global-best	385.23
1	0.1	5	global-best	3114.79
3	0.1	5	global-best	708.92
7	0.1	5	global-best	426.00
10	0.1	5	global-best	395.53
13	0.1	5	global-best	393.29
17	0.1	5	global-best	388.52
20	0.1	5	global-best	389.27
23	0.1	5	global-best	388.42
1	0.25	5	global-best	3107.21
3	0.25	5	global-best	710.00
7	0.25	5	global-best	417.45
10	0.25	5	global-best	399.98
13	0.25	5	global-best	389.03
17	0.25	5	global-best	390.28
20	0.25	5	global-best	388.15
23	0.25	5	global-best	387.39
1	0.5	5	global-best	3094.53
3	0.5	5	global-best	719.25
7	0.5	5	global-best	417.42
10	0.5	5	global-best	402.50
13	0.5	5	global-best	389.11
17	0.5	5	global-best	388.39
20	0.5	5	global-best	384.26
23	0.5	5	global-best	386.50