DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2019

Genetic Algorithms: Comparing Evolution With and Without a Simulated Annealing-inspired Selection

MATS ANDERSSON

DAVID MELLIN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Genetic Algorithms: Comparing Evolution With and Without a Simulated Annealing-inspired Selection

MATS ANDERSSON

DAVID MELLIN

Bachelors in Computer Science
Date: June 6, 2019
Supervisor: Jörg Conradt
Examiner: Örjan Ekeberg
School of Electrical Engineering and Computer Science
Swedish title: Genetiska algoritmer: En jämförelse av evolution med och utan simulerad härdning i urvalet

Abstract

The Genetic Algorithm (GA) is an interesting problem solving algorithm which takes inspiration from evolution in order to self-improve and reach good solutions to problems by reproduction and mutation. This thesis compares a GA with and without a Simulated Annealing (SA) inspired selection when it comes to solving three different instances of the Traveling Salesman Problem (TSP). SA was found to be able to help the GA reach better solutions, but the results also depended on other parameters within the GA itself.

Sammanfattning

Den genetiska algoritmen (GA) är en intressant problemlösningsalgoritm som hämtat inspiration från evolutionen och förbättrar sig själv genom fortplantning och mutering. Denna uppsats jämför två implementationer av en GA. Den ena är en ren GA och den andra är samma GA med tillagd urvalsfunktionalitet inspirerad från simulerad härdning (SA). De båda algoritmerna jämfördes med varandra på att lösa tre olika instanser av handelsresandeproblemet (TSP). SA visade sig kunna hjälpa en GA att nå bättre resultat, men resultatet berodde även på andra parametrar inom GA:n.

Contents

Chapter 1

Introduction

Striving for efficiency is a part of many areas of human interest. Completing tasks faster means that more tasks can be completed in the same amount of time which in turn frees up time for other activities, such as performing even more tasks or even doing something else with the saved time. Efficiency leads to all sorts of things, such as products being produced faster, ideas getting advanced more rapidly and humanity evolving more swiftly.

In computer science, efficiency is mainly about getting machines to compute algorithms and perform tasks faster. If a computer task is performed efficiently, the computer generally has to use less power to compute the task. Efficient programs help save energy and makes more time available for other computational purposes. Sometimes the desired computation would even be impossible without efficient algorithms [1].

Within computer science, the task of improving programs and making them more efficient is called optimization. Optimization problems consist of finding the best solution from all possible solutions existing. In order to find the best solution, one can try different algorithms and different approaches at solving a certain problem. Since the word computer contains the word compute, or calculate, it is no surprise that faster calculations and performances in a computer is sought after within computer science. As Donald Knuth, "the father of the analysis of algorithms" puts it: *"In established engineering disciplines a 12 % improvement, easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering"* [2]

One of the most famous optimization problems is the Traveling Salesperson Problem (TSP). It is formulated as "given n cities and the distances between them, find a shortest route that visits each city exactly once". This problem has yet to be generally solved by any known algorithm. Instead, approxi-

mation techniques are used to find reasonably good solutions to the problem. The TSP is well suited as a benchmark of efficiency for a lot of optimization algorithms.

Since the TSP can be represented as a graph consisting of edges and vertices, a lot of problems can be represented as TSP problems. Some examples are "which is the best path for a robot to traverse a warehouse?" and "in which order should we visit these places to travel as little as possible in our vacation?".

A lot of optimization techniques have been tried when it comes to solving the TSP. Two examples are genetic algorithms and simulated annealing, both of which use probabilistic techniques.

1.1 Genetic Algorithms

Genetic Algorithms (GA) were invented in the 1960s by John Holland [3]. They stem from the Darwinian idea of “survival of the fittest”. John Hollands’ idea was that algorithms could take inspiration from the way species survive. GA:s utilize a population of individuals that perform a task and then get a score depending on how well the perform compared to either the goal or/and the rest of the individuals in the population. The most successful individuals get to “reproduce” more than the others and the idea is that the genes of the successful individuals will be inherited to the next generation every iteration. And so it continues, creating better individuals each generation and eventually reaching an optimal solution.

1.2 Simulated Annealing

Simulated Annealing (SA) was first presented in 1983 [4]. SA takes inspiration from the metallurgic process of annealing, which is when a heated metal is cooled slowly in order to reduce the defects in the solidified metal and gain a better end result. SA does the same thing, by iteratively searching for an optimal solution by doing more random changes in the beginning (when the temperature is high), and slowly decreasing the randomness as time goes on (as the “metal” cools itself). When SA was introduced in 1983 it was used on the TSP and was shown to be useful on problems of multivariate and combinatorial nature [4], including the TSP itself.

1.3 Purpose

The goal of finding efficient techniques is a substantial part of computer science. Previous research has shown that both GA and SA are valid implementations when searching through a lot of possible solutions, such as when solving the TSP. The purpose of this thesis is to investigate whether a GA can be improved with the help of SA when it comes to solving the TSP.

1.4 Research Question

This study aims to investigate how a standard GA performs compared to the same GA with an SA-inspired parent selection when it comes to solving the TSP on three different TSP problem instances.

Chapter 2

Background

In this chapter, the basic terminology and the core concepts used in this essay are presented. First the problem is introduced more thoroughly, which is then followed by the relevant algorithms developed and used.

2.1 Traveling Salesman Problem

The traveling salesman problem (TSP) is one of the oldest problems in combinatorics and comes from as early as 1832 [5]. The problem is defined as "given n cities and the distances between them, find a shortest route that visits each city exactly once".

The TSP is an NP-complete problem, which means that it can not be solved efficiently by any known algorithm [6]. The reason for the TSP being a hard problem is reasonably intuitive just with a quick example: Imagine if we have 3 cities. The possible orders in which we visit these cities is that we start with any of the three, then select one of the two remaining ones and lastly visit the only one left. In mathematical terms that would be 3x2x1 = 3! = 6 possible solutions to search through to find the best one.

If the amount of cities increase, the amount of possible routes increases rapidly. Just by doubling the amount of cities we get 6! = 720 different routes. Just 10 cities yield over 3.5 million different routes. Therefore, searching through all the possible solutions to find the best one quickly becomes unfeasible as the number of cities increase. Instead, we have to resort to optimization algorithms that approximate the best value and limit what part of the solution space to search through by utilizing heuristics and other "smart" problem solving methods.

The TSP is one of the most studied optimization problems in the world,

partly perhaps due to its catchy name, but primarily due to the fact that it is a simple sounding problem but has yet to get a general solution. The TSP can also be represented by a graph with vertices and edges, which means that it is applicable in a lot of areas such as genetics, neuroscience, telecommunication and computer science, thus furthering its appeal [7].

2.2 Genetic Algorithms

The Genetic Algorithm (GA) is part of an area called evolutionary computation. Evolutionary computation takes inspiration from biological evolution in order to compute answer and solutions to problems [3]. One of the main appeals of evolution when it comes to solving computational problems is the way it can be used as a method of "designing innovative solutions to complex problems" [3].

The GA was presented by Holland in 1975 [8][3] and in particular takes inspiration from the way genes are passed on by individuals in one generation to their offspring in the next. The idea is that if the best performing members of a generation get to reproduce more into the next than worse performers get to, we iteratively get better and better solutions each generation. This should then yield iteratively better performing members for each new generation until an optimal or "good enough" solution has been found.

2.2.1 GA Terminology

GA:s use a few important concepts. Those are described briefly here.

Model

An **individual**, also known as a **chromosome**, is a single instance of a solution to a problem. In the TSP, an individual can be represented as a single route within the solution space. As an example, this 8-city route is one individual:

(7, 2, 4, 5, 6, 3, 1, 8)

A **gene** is a part of an individual. The gene usually represents a trait within the individual, and is commonly represented with single bits. However, they can be represented in other ways. An example is as letters in a protein structure. In terms of the TSP, a gene can represent a certain city visited at a certain position in the order of cities. As an example, the bolded city 4, in the third position, is a single gene in this individual (as is every other city):

(7, 2, **4**, 5, 6, 3, 1, 8)

A **population** is a collection of individuals in a single generation. As an example, here is a population of size 3:

(7, 2, 4, 5, 6, 3, 1, 8)
(1, 2, 5, 8, 6, 4, 3, 7)
(5, 3, 8, 7, 2, 1, 4, 6)

A **generation** is a single instance of a population within a single iteration of searching the solution space. The first generation is completely random, while the rest of the generations are heavily generated from the previous one. As an example, the example above not only is an example of a population, but also shows how it can look in a single generation.

Execution

The **fitness function** is the measurement of how well an individual performs at the objective. What type of value is desired depends on the problem. In the TSP, the total distance traveled is usually used as the measurement of performance. A lower distance traveled equals to better performance since the TSP is a minimization problem.

The **population size** is the amount of individuals in a generation. A bigger population size makes the program run slower, but gives a better chance for a single generation to find better solutions. The trade-off in increasing the population size usually means getting better results but having a slower program which converges to its optimal solution slower with respect to time.

Selection is the process of choosing which individuals in a generation to crossover into new individuals in the next generation. Selection can be implemented in simple ways, such as just selecting a percentage of the best performers within a generation. Selection can also be done in more advanced ways, such as giving each member a portion of a roulette wheel where every members portion size depends on its fitness level, and then spinning the wheel several times in order to select parents.

Retain rate is the portion of individuals selected to be parents for the next generation. If all individuals of a generation gets to reproduce equally, the next generation is basically random.

Crossover is the operation of structurally generating a new individual from parents in a previous generation. There are a substantial number of ways in which one can combine the genes of the parents into a new individual.

Mutation is the operation of randomly changing the genes somehow when generating a new individual or passing an individual from one generation to the next. A mutation within a TSP representation could be swapping the order in which one visits two cities. There are several types of possible mutations. It could be as simple as simply swapping two cities or more complicated, like selecting a small subset of the solution and reversing it.

Mutation rate is the probability of an individual having its genes mutated.

The details on how this essay implemented the GA is described in the method chapter, chapter 3.

2.2.2 Algorithm Description

A short description of a GA can be defined as:

1. Initialize random population from solution space.
2. Calculate the fitness of each individual in the population.
3. Reproduce the generation, using a combination of these methods, making sure the next generation has the same population size:
   - Select a portion of the most fit individuals and simply pass them onto the next generation.
   - Generate offspring by taking fit individuals and combining them using crossover.
   - Generate offspring by taking any individual and mutating it.
   - Generate new individuals randomly.
4. If a good enough solution is found, or if another termination criteria such as time limit or maximum amount of generations is reached, return the result. Otherwise, go to step 2.

2.3 Simulated Annealing

Simulated Annealing (SA) was presented as a concept in 1983 [4]. The idea is taken from metallurgy, where annealing is the process of heating and cooling a material in order to alter the properties of the material thanks to changes in its structure. Slower cooling processes yield materials with less defects, and thus better results.

In computer science, SA is the process of simulating this temperature usage by having a variable that "cools" down as the iterations of the program go on. When the temperature is high, the program will be more likely to accept and keep working with worse solutions than when the temperature has lowered. That results in the algorithm being able to jump out of local maxima/minima, thus increasing the chance of reaching a solution at or close to a global optimum.

2.3.1 Acceptance Function

The **acceptance function** is the mechanism behind deciding whether to keep working with the new solution or not. If the new solution is better, it is often accepted instantly. If the new solution is worse, the algorithm decides whether it is accepted or not. The acceptance function presented by Kirkpatrick et al. when they introduced SA was

$$P(e, e', T) = \begin{cases} 1 & \text{if } e' < e \\ exp(-\frac{e'-e}{T}) & \text{if } e' \geq e \end{cases} \tag{2.1}$$

which means that the probability of accepting a new solution e' is 100% if it is better and depends on randomness if it is worse. How likely the randomness is to occur then depends on the temperature T. The function can basically be defined in whatever way one wants the acceptance probability to change differently depending on what temperature the system currently has.

2.3.2 Cooling Function

The **cooling function**, or **cooling schedule** decides how the system cools down, and thus how it decreases the amount of "randomness" withing its calculations. When Kirkpatrick et al. presented their original SA-implementation they used the following cooling schedule, where one just lowers the temperature by multiplying with a constant α, choosing the constant depending on how fast one wants to cool the system:

$$T_{k+1} = \alpha T_k \tag{2.2}$$

Since the cooling function is just a mathematical function, there exists a lot of different variations on it. All depending on in what manner the system cooling is desired.

2.3.3 Algorithm Description

A run-through of SA can be described like this:

1. Set the initial temperature and generate a random solution.
2. Generate a new solution by making a small change in the current one.
3. If the new solution is better, accept it. If the new solutions is worse, use the acceptance function to decide whether to accept it.
4. If the new solution is good enough or if the temperature is low enough, return the result. Otherwise, decrease the temperature and go to step 2.

2.4 Exploration and Exploitation

Both GA and SA try to combine the ideas of exploration and exploitation. **Exploration** means traversing the search space in a broad way, trying to find good starting points to exploit. **Exploitation** is the way one makes small tweaks to an already good solution in order to find the best possible.

In GA, the crossover can be seen as the exploration part, since the crossover function generates new members in a broad way. Mutation on the other hand tries to exploit the good members into becoming even better ones by making small changes to them.

In SA, one goes from heavy exploration and minor exploitation to the inverse as the temperature decreases. In the beginning the algorithm is almost random, and as the system cools towards the freezing point one aims at reaching the best local optimum, which hopefully is the global one.

One of the hardest challenges in both GA and SA is balancing the algorithm well enough to utilize both exploration and exploitation. There is no definitive way to achieve this balance, it varies from problem to problem and situation to situation.

2.5 Previous Research

2.5.1 GA and SA in TSP

Using Genetic Algorithms and Simulated Annealing to solve the TSP are both established to be a valid concept. When the concept of SA was first presented

in 1983 by Kirkpatrick, Gelatt and Vecchi, it was presented with examples on how it could be used to solve the TSP [4].

In 1985, Greffenstette, Gopal, Rosmaita and van Gucht showed that the GA could solve the TSP. They also mentioned that careful comparisons between SA and GA could be interesting due to their GA implementation being competitive to SA implementations at the time [9].

Greffenstette et al. also mentioned the idea of combining the GA with other heuristics as an interesting one [9].

2.5.2 GA Crossover and Mutation Functions for TSP

In 1999, Larrañaga, Kuijpers, Murga, Inza and Dizdarevic performed a comparative study on 8 different crossover functions and 6 different mutation functions that had been presented as usable when performing a GA for the TSP at the time [10].

They concluded that among their tested functions, four crossover functions were superior, one of which is called OX1. Two of the best performing mutation functions were DM and ISM [10].

As a reminder, crossover occurs when an offspring is generated from a parent and mutation occurs randomly in a single individual right as it is passed into a new generation.

Order Crossover (OX1)

The order crossover operator was first presented by Davis in 1985 [11]. It generates offspring by choosing a part/sub-tour of the first parent and then placing the rest of the cities in the order they appear in the second parent. Here is an example taken from Larrañaga et al. [10]: Consider two parents:

(1 2 3 4 5 6 7 8)
(2 4 6 8 7 5 3 1)

First, decide where to cut out the sub-tours. We choose to cut between city 2 and 3, and between city 5 and 6:

(1 2|3 4 5|6 7 8)
(2 4|6 8 7|5 3 1)

Then, we place the sub-string into the offspring at the same position.

(* *|3 4 5|* * *)
(* *|6 8 7|* * *)

And finally, we put the remaining cities that are needed in offspring 1 by starting after the second cutoff point, and placing the cities in the order they are placed in parent 2, and vice versa. Then we get the following children:

(7 8|3 4 5|1 2 6)
(3 1|6 8 7|2 4 5)

Displacement Mutation (DM)

Displacement mutation is credited to a report by Michalewicz from 1992 [10]. DM selects a random sub-string from the individual and moves it into a random place within the individual. As an example [10]: Say our individual is

(1 2 3 4 5 6 7 8),

then we randomly take out (3 4 5) as an example. Then we get

(1 2 6 7 8).

Then we randomly place (3 4 5) back somewhere. If we place it after city 7 we get

(1 2 6 7 3 4 5 8).

Insertion Mutation (ISM)

Insertion Mutation, by Fogel in 1988 and Michalewicz in 1992 [10] is similar to DM, but only moves a single city. It randomly selects a city in the individual, and inserts it randomly somewhere else. As an example:

(1 2 3 **4** 5 6 7 8)
(1 2 3 5 6 7 **4** 8)

2.5.3 GA Selection Methods for TSP

Razali and Geraghty compared three different selection methods for the TSP. The first function was proportional roulette wheel selection, which is when each individual gets a portion of a roulette wheel where the portion is entirely proportional to its fitness level. The second was tournament selection, where one selects only a few members at random and then picks the best among these few. The third was rank-based roulette wheel selection, which is similar to the proportional roulette wheel, but the portion each individual gets is based entirely on the rank number, instead of the fitness level value. They concluded

that rank-based roulette wheel selection performed the best, with tournament selection in second place and proportional roulette wheel selection in last place [12].

Tournament Selection

Tournament selection is a selection method for selecting parents to use in crossover. It is performed by selecting a handful of individuals, comparing the fitness levels of these individuals and then selecting the best one to use as a parent for the next generation. This process is then repeated until the desired amount of parents have been acquired. As an example, take a population consisting of 10 individuals. The individuals are named 1 to 10 where 1 is the individual with the best fitness level and 10 is the individual with worst fitness. We want to pick out 3 parents to generate the next generation from. We start a tournament and select three individuals at random:

(3, 6, 7)

Since 3 is the individual with the best fitness level, it is selected as a parent. We start another tournament:

(2, 3, 10)

2 has the best fitness level, and is selected as parent. We perform the last tournament:

(7, 9, 10)

7 is the member with the best fitness level, and thus our parents to cross over are:

(2, 3, 7)

Tournament selection can be performed with and without repeated picks. By using tournament selection, diversity is maintained reasonably well, since one tournament might only have bad participants, there is a possibility that the selected parents might be poor performers, which means that even bad individuals within a generation has the chance to reproduce.

Goldberg and Deb compared both binary tournament selection (tournament size = 2) and bigger tournament selections to several other selection methods. In terms of time complexity, tournament selection performs reasonably well at $O(n)$ [13].

Goldberg and Deb also compared different sized tournament selections to each other and concluded that the more participants in a tournament selection, the faster the convergence of the algorithm [13].

The notion that selecting more members per tournament yields faster convergence is reasonably intuitive, since more participants in a tournament give better parents on average since the likeliness of selecting at least one good tournament participant increases. Thus, better offspring are generated on average, leading to a faster movement towards an optimum. Therefore, within the tournament selection, one can choose different number of tournament participants depending on how fast one wants the GA to converge. The number of participants in a tournament selection is usually in the 2-4 range, derived from what the writers of this thesis have read.

2.5.4 SA Cooling Functions for TSP

Cohn and Fielding compared 5 different cooling schedules over 100 runs on solving different sizes of TSP:s in order to see which cooling schedule performed best. They found that the cooling schedules that forced the temperature to cool faster performed better the larger the problem was. When the problems were small, they performed similarly. However, they also noted that performance seemed to also have to do with how the TSP problem was shaped [14].

Martín and Sierra compared 10 cooling schedules, and concluded that when comparing the performance of the algorithm in relation to the cooling schedule, the implementations that had a moderate slope in the initial and central parts of the temperature decrease curve together with a soft slope towards the end performed the best. Whether they had convex or sigmoid shape did not matter [15]. An example of an algorithm with convex shape is the original cooling schedule that Kirkpatrick et al. presented [4], which can be seen in section 2.3.2.

2.5.5 GA vs SA in TSP

Some studies have been made comparing the performance of GA implementations and SA implementations when it comes to solving the TSP. Adewole, Otubamowo and Egunjobi compared two implementations of GA and SA and concluded that their SA generally had a faster run time, while the GA found better solutions [16].

On the contrast, Mukhairez and Maghari tested GA and SA together with

Ant Colony Optimization (ACO), which is another optimization technique. Their GA implementation was worse at both finding the optimal solution and execution time when solving the TSP, compared to both SA and ACO [17].

2.5.6 SA with GA in TSP

When it comes to solving the TSP, SA and GA have also been combined before. Lin, Kao and Hsu used SA with a GA-inspired generation iteration and compared their result with classic SA [18] and performed tests on three different NP-hard problems, one of which was the TSP.

They called their implementation the Annealing-Genetic (AG) algorithm. Their AG implementation outperformed the SA implementation at both finding the best solution and in having a lower run-time [18].

Chapter 3

Methods

In this chapter, the methods chosen to compare the different implementations are described. We also describe how we chose to implement the two different algorithms.

3.1 Genetic Algorithm Implementation

In order to explore the research question, a framework for the GA was built. The program language used was Java, mainly due to the preference of the writers. In this section, we will describe how the GA was implemented and how we represented the TSP within it.

3.1.1 Setup

The entire project was built from the ground up in Java, and programmed from the ground up by the authors. A GitHub repository with the code is available in appendix A.

The TSP data was set up as a number of places in a two-dimensional space, taken from different websites but all as part of a well known TSP dataset named TSPLIB. The exact TSP instances used are described in section 3.4.

A single individual within a generation was represented as a path through these cities, mainly since path representation probably is the most intuitive representation method for a path in the TSP, and also the one preferred by the authors. As an example, a numberical path through a TSP instance of 8 cities would be (1, 2, 3, 4, 5, 6, 7, 8), which means that each city is visited in order, and finally one would return to the first city.

The GA was initialized by randomizing the entire population. Then the fitness of each individual was determined by calculating the total distance through the TSP for each individual, where lower was better. We computed the best individual, worst individual and mean values for the generation, saved the result and then reproduced the next generation.

The first generation was completely randomly generated. All of the subsequent generations were built up in two parts, by using the following three steps:

1. First we simply passed on the very best individual of the generation, without mutation or crossover.

2. Then we selected a number of parents from the previous generation, using tournament selection.

3. Then we generated the individuals for the next generation using crossover on parents.

After creation, an individual had the chance to randomly mutate. The implementation of mutation is described in section 3.1.4.

Pseudocodes for the algorithm and the different functions are described in section 3.3.

The reason for passing on the best individual from a generation to the next is that one is guaranteed to never forget the best result one has found so far. Since our population size was large enough, it seemed reasonable to expect this single member to not cause premature convergence.

The rest of the new generation was generated by crossing over individuals from the previous generation, where the better performers were more likely to reproduce. This is the core concept of GA:s, and the part that keeps it improving iteratively. The method of crossover is described in section 3.1.3.

3.1.2 Selection

We chose to use tournament selection, described in 2.5.3. The reason for selecting tournament selection as our method of choosing parents was that it is easily implemented, commonly used and well performing. We used 4 different sizes of tournaments, from 2 to 5, in order to investigate whether there was a difference if the algorithm used more tournament participants or not. The thought was that since the convergence was faster with more tournament participants, the SA might be more helpful in implementations with big tournament sizes compared to small ones.

3.1.3 Crossover

When generating offspring from previous generations, we used the crossover function named OX1, described in 2.5.2. The reason behind choosing OX1 was that it is one of the classic ones that at the same time seems to perform well at solving the TSP, according to the results of Larrañaga et al. [10].

3.1.4 Mutation

When generating new individuals, they were given a chance to randomly mutate. This helps add diversity to the generation in order to better search the solution space. We used two different mutation functions, again selecting from the best performing ones according to Larrañaga et al. [10].

The selected mutation functions we used were DM and ISM, described in section 2.5.2. They were given an equal chance to occur. As an example if the mutation rate was set to 0.1, the total chance of a mutation occurring was 10 percent. Since the two mutation functions had an equal chance of occurring, the total chance for each of them to occur would be 5 percent. They could not occur at the same time, however.

3.1.5 Parameters

The GA parameters that could be tweaked were population size, number of generations, retain rate, number of parents and mutation rate. The concepts were explained in more detail in 2.2.1.

Since there is a lot of combinations of tweaks one can do within a GA, we had to run a lot of tests to find reasonable parameter settings that were good enough to run the experiment with. Note that we do not claim to have experimented enough to find the best values possible, since we were alleviated from having a "best possible" program due to both GA and GA+SA implementations being built the same, apart from the single parameter varied by the SA implementation. The same build meant that the results of the two implementations should be comparable as long as the parameters selected were the same.

Parameter	**Value**
Population Size	Number of Cities * 4
Number of Parents	Population Size * 0.5
Mutation Rate	0.2
Tournament Size	2 - 5
Max. Number of Generations	150000

The population size selected was scaled to the problem size. The trade-off when selecting population size was that a smaller problem size decreased the execution time but if it was selected too low it led to the algorithm not converging to a sufficient solution.

When selecting the number of parents the risk with choosing a too small number is that the next generation might not being diverse enough due to being based on the same parents. The aim when selecting this parameter was to have enough parents but not having such a high amount that it would lead to a high probability of not using all of them.

The mutation rate was decided upon by testing and a value between 0.1 and 0.3 seemed appropriate. A higher value affected the convergence in a negative way but a too low value made the population not be diverse enough.

A tournament size of 2 is the smallest possible value since a tournament size of 1 simply means picking a parent directly from the previous generation. The ceiling of size 5 was selected due to having at least four different sizes to compare, but could in hindsight have been set even higher.

The maximum number of generations the algorithm was allowed to run was selected as a number large enough for it to converge on all of the problem instances. To speed up the tests we considered that the algorithm had converged if it had not found a better solution in 20.000 generations.

3.2 Simulated Annealing-inspired Implementation

SA is, as described earlier, the idea of making an algorithm behave more randomly in the beginning of the run-time and converging it to be more focused on optimization as time goes on. In order to design our SA implementation properly, we thought a lot about how to properly achieve this behaviour in the GA, which has the same "balance" throughout the entire run-time.

One of the major parts of the implementation was to decide what parameter or function to vary during run-time. The standard way of SA is as described to generate a new solution, examine whether it is better than the current one and maybe keep working with it anyway if it is worse. However, we chose a slightly different approach.

Since a well structured GA usually generates a reasonably diverse population, we decided to not use the standard SA method of looking at newly generated solutions and accepting or disregarding them. Instead, we let the SA function control the quality of the parents. Selecting worse parents in the beginning of the runtime would in theory generate worse individuals in the

beginning, slow down convergence and improve the final result since it would be better at avoiding local optimums.

By worsening the selected parents our goal was to see whether a slower convergence helped the GA in achieving a better final result.

In order to accomplish this, we added an acceptance function to the tournament selection, described in section 2.5.3. This functionality depended on the SA temperature, and gave worse performers the possibility of winning the tournament and thus becoming parents for the next generation.

In theory, that would slow down the convergence in comparison to the pure GA, where the better members always won their tournaments, due to the algorithm not being as elitist. And thus, it would perhaps improve performance.

3.2.1 Cooling Schedule

When deciding what cooling schedule to use, we looked at the previous research indicating that a convex shape would be preferable, and selected a function that had a convex shape with a decent slope in the beginning, but at the same time had a lot of time closing in on zero, since the algorithm needed to get time to converge to the best optimum. We chose

$$T(x) = (x - 1)^8 \tag{3.1}$$

where x was the portion of the generations completed so far in the runtime. The graph of how the system cools down can be seen in figure 3.1. Note that since it is a GA, the system never really freezes. Instead, when the temperature is zero, the SA-implementation acts as the pure GA.

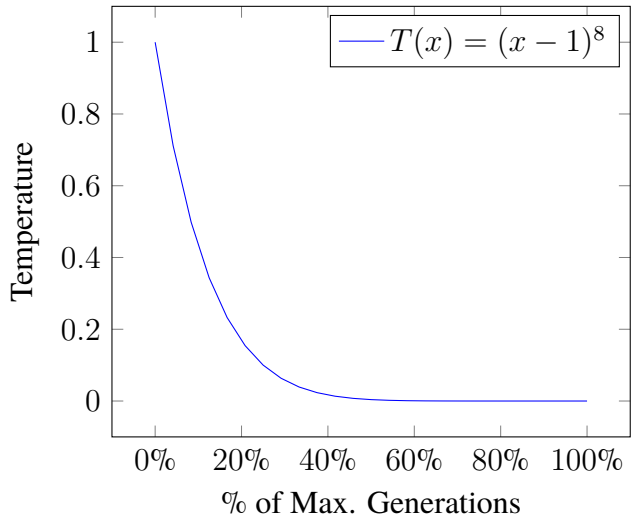

Figure 3.1: The Cooling Schedule chosen for this project.

3.2.2 Acceptance Function

The acceptance function was implemented in the tournament selection in order to allow for worse parent selection. In normal tournament selection, one chooses a number of candidates and selects the best one. In our SA-inspired implementation, we chose a number of candidates, picked one of them at random and if it was not the best candidate we could still accept it as a parent. How likely it was depended on the temperature.

The exact method is that we first picked out a number of candidate parents depending on how big of a tournament size we had. A tournament size of 3 gave 3 candidate parents and so on. Then, we picked whichever of the candidates was the best one, p, by looking at their fitness value. Finally, we selected one of the candidates, p' at random, and if it was the best one or had the same fitness as the best one, it was instantly accepted as a new parent. If it was not the best one, we used the function depending on the temperature T to either accept or reject it.

$$P(p' = parent|p, p', T) = \begin{cases} 1 & \text{if } p' \leq p \\ T & \text{if } p' > p \end{cases} \tag{3.2}$$

3.3 Algorithm Pseudocodes

Algorithm 1 shows how the general GA works. Algorithms 2 and 3 show how the differing parent selection in the SA-inspired implementation works and Algorithm 4 shows how the new population was generated after parent selection.

Algorithm 1: Genetic Algorithm

```
Output: Shortest TSP path found
population = random population;
generation = 1;
while generation < maxGenerations do
    parents = parents from Algorithm 2 (GA) or Algorithm 3 (SA);
    population = reproduce parents with crossover and mutation;
    generation += 1;
end
return best individual;
```

Algorithm 2: GA Tournament Selection

```
Input: Tournament Size size, Last Population, Number of Parents
Output: Parents for the Next Generation
parents = [];
while needing more parents do
    potentialParents = randomly select size number of candidates from
     previous population;
    bestParent = select the parent with the lowest fitness value from
     potentialParents parents.add(bestParent);
end
return parents;
```

Algorithm 3: SA Tournament Selection

```
Input: Tournament Size size, Last Population, Number of Parents
Output: Parents for the Next Generation
parents = [];
while needing more parents do
    potentialParents = randomly select size number of candidates from
     previous population;
    bestParent = select the parent with the lowest fitness value from
     potentialParents;
    randomParent = randomly select a potential parent from
     potentialParens;
    if randomParent > bestParent and random(0, 1) < temperature
     then
        parents.add(randomParent);
    else
        parents.add(bestParent);
    end
end
return parents;
```

Algorithm 4: Reproduction Algorithm

```
Input: Selected Parents
Output: A New Generation of Individuals
population = [];
while needing more individuals in population do
    parent1, parent2 = randomly select two parents;
    child = OX1 crossover between parent1 and parent2;
    if random(0, 1) < mutationRate then
        child.randomMutation();
    end
    population.add(child)
end
return parents;
```

3.4 Data Sets

In order to test our GA and GA with SA implementations we chose three existing instances of TSP problems from TSPLIB.

The chosen TSP instances were:

Country/City	TSP Problem Name	Cities	Optimal Length
Western Sahara	wi29	29	27603
Djibouti	dj38	38	6656
Berlin	berlin52	52	7542

The main reason for choosing to use existing TSP problems was that they have already been completely solved, which means we had an optimal solution to compare our results with. As an example, the instance with cities from Djibouti has 38 cities, with an optimal length of 6656.

Using these instances also aided with easier implementation, since we did not have to generate our own optimal TSP graphs and calculate the best distances on our own. Instead, we could as mentioned compare with already solved problems.

The exact data sets are added within appendix B and are also available online by searching for the TSP problem name.

3.5 Evaluation

In order to evaluate our results, we ran the GA implementation with the described parameters and the SA-inspired implementation with the same GA parameters and the cooling schedules described in section 3.2.1.

We ran the two algorithms 1000 times on wi29 and dj38, and 100 times on berlin52. We compared the best found values for the GA and GA+SA implementations and investigated how many of them reached within 1% of the optimal solution. Then we also compared the best distance found on average. Since the GA settings were the same, the results showed whether the SA implementation added into one of the GA implementations helped it get better results or not.

We also ran reliability tests on the wi29 instance to ensure that our results seemed reliable and were not just coincidental due to the randomness within the algorithms.

The results are presented in the next chapter.

Chapter 4

Results

In this chapter we present the results of the tests executed within this project.

4.1 Quality of the Best Found Solutions

Figures 4.1, 4.2 and 4.3 show the portion of the test runs that reached a solution within 1% of the optimal solution for the three differently sized problems respectively. The number on the x-axis represents the different tournament size in the parent selection step of the genetic algorithm.

All three figures show an advantage for pure GA implementation for tournament size 2. For tournament size 3 it depends on the problem, and at tournament sizes 4 and 5 the SA inspired implementation more commonly reaches a solution close to optimal on all three problem instances. The larger the problem size, the lower rate of runs reached a solution close to optimal in general.

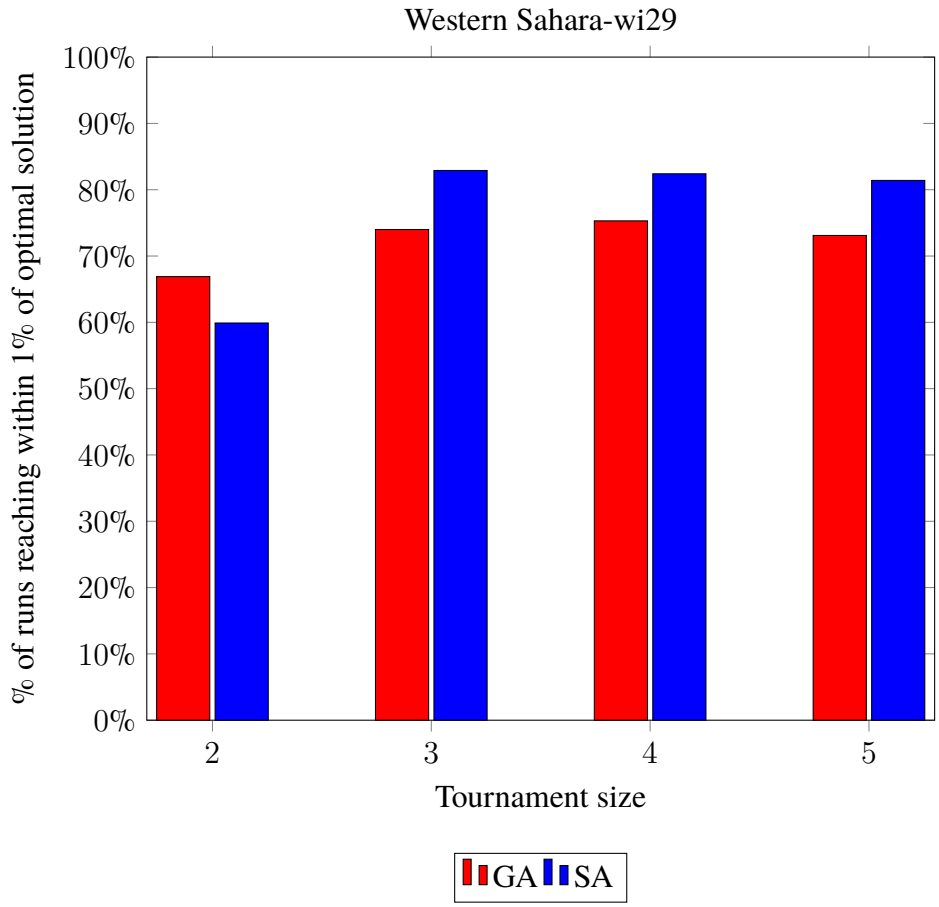

Figure 4.1: Share of test runs reaching a solution within 1% of the optimal solution on the wi29 problem.

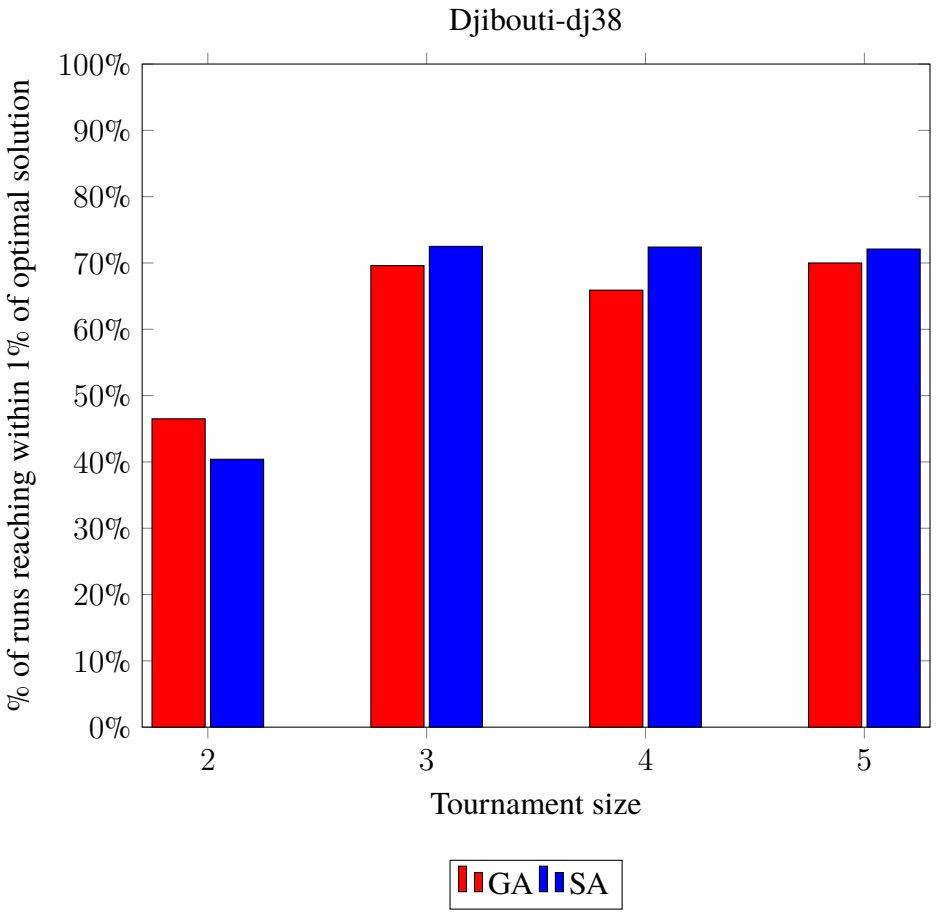

Figure 4.2: Share of test runs reaching a solution within 1% of the optimal solution on the dj38 problem.

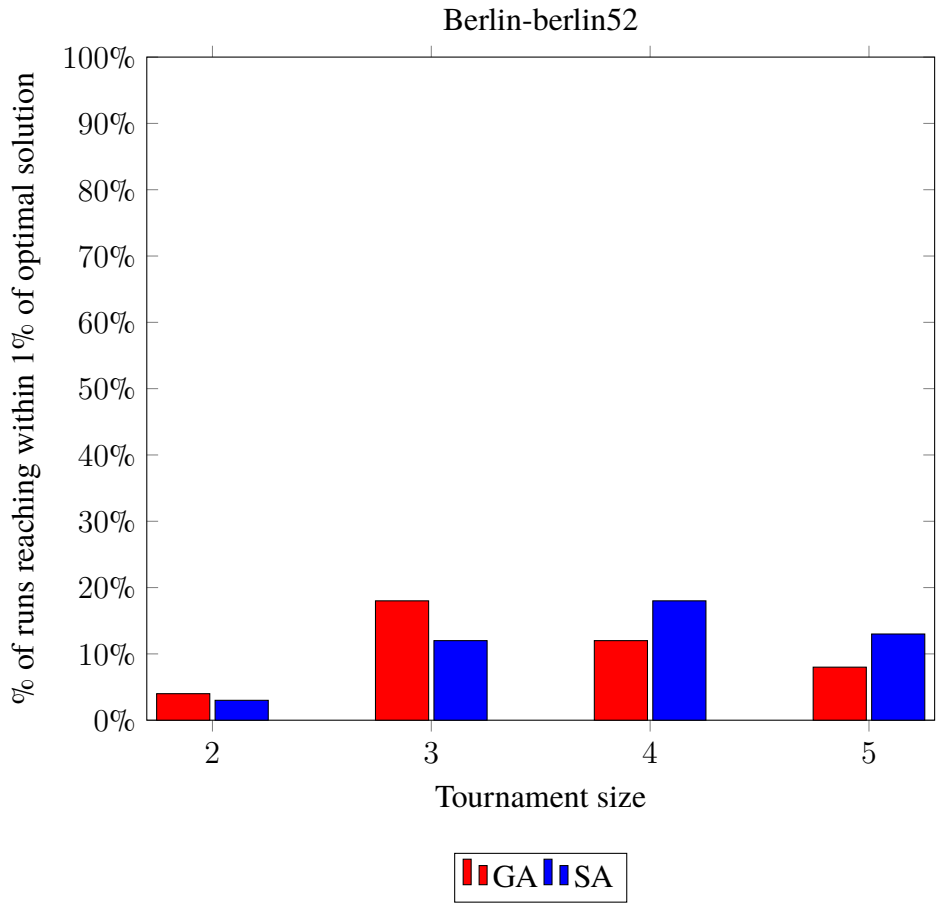

Figure 4.3: Share of test runs reaching a solution within 1% of the optimal solution on the berlin52 problem.

4.2 Average Best Distance

Figure 4.4, 4.5 and 4.6 show the average distance found of the two versions of implementation with different tournament sizes in the parent selection step of the algorithms. The x-axis represents the different tournament sizes. The vertical line across the charts indicates the optimal solution for the TSP-problem instance. The charts also shows the standard deviation.

For a tournament size of two the traditional GA approach generally finds better solutions. Similar to the previous section the implementation that found a better solution for tournament size three depends on the problem instance. For a tournament size of four and five the SA approach performs slightly better than the traditional GA approach. For all of the problem sizes the standard

deviation for tournament size 2 is larger while a tournament size of 3 to 5 is quite equal.

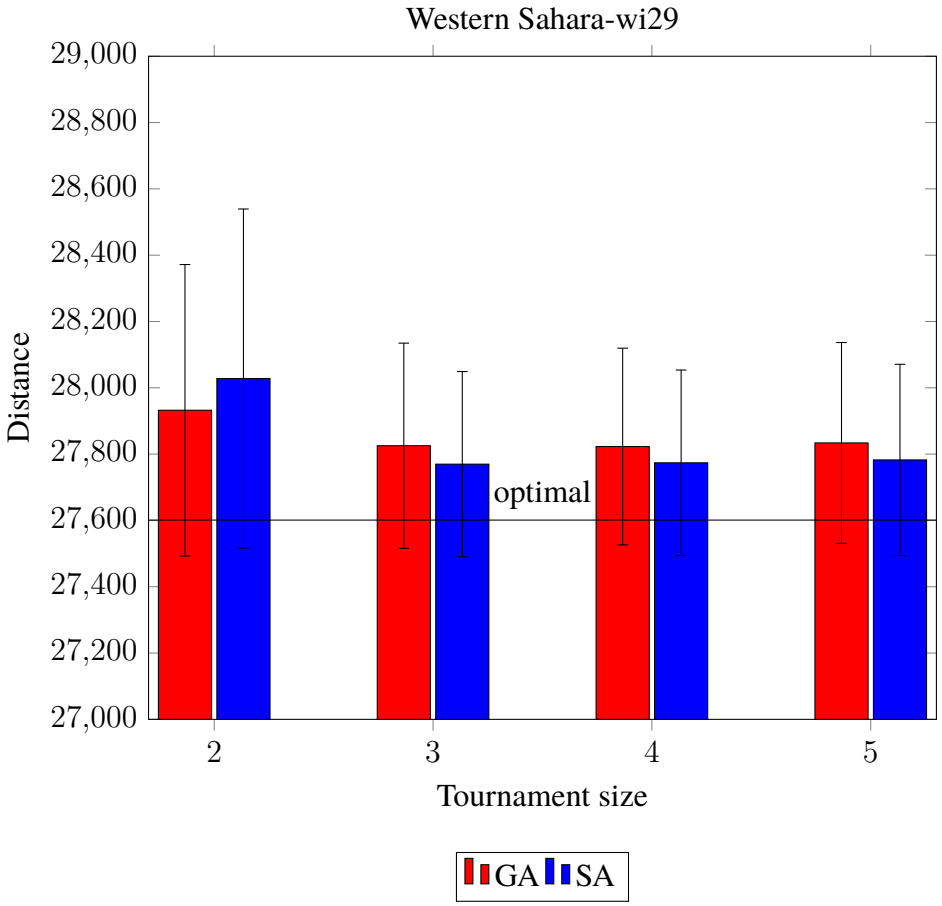

Figure 4.4: Distance of the selected route after convergence on the wi29 problem.

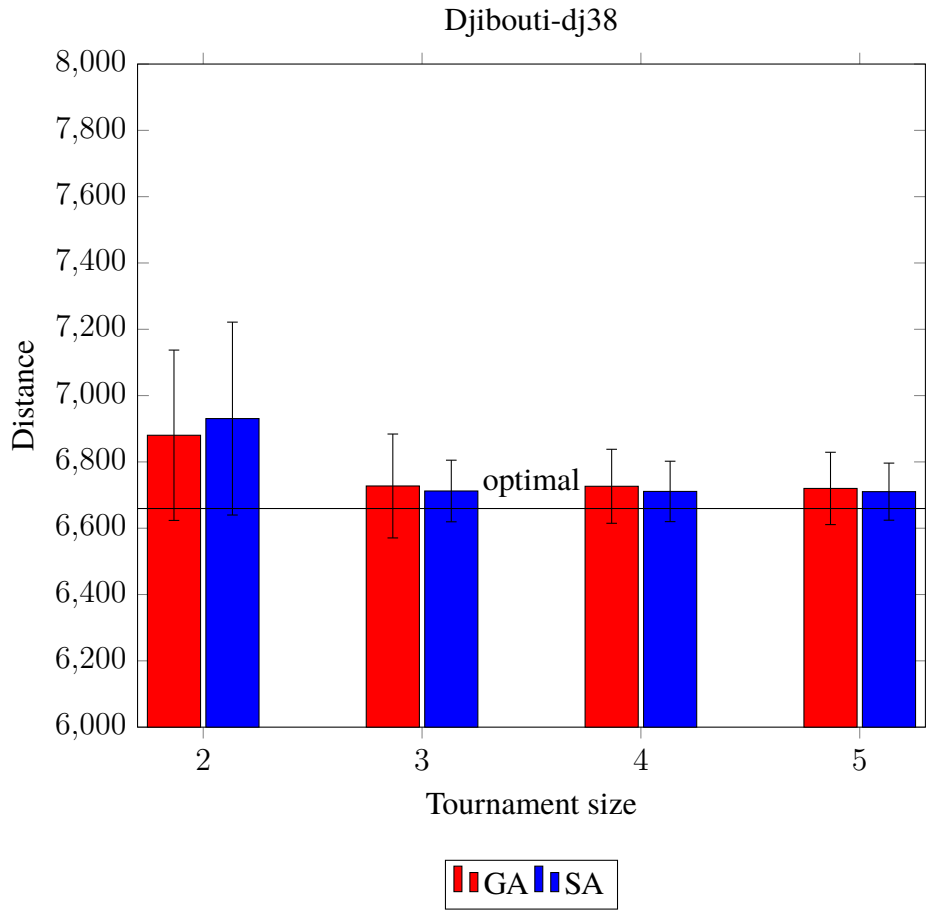

Figure 4.5: Distance of the selected route after convergence on the dj38 problem.

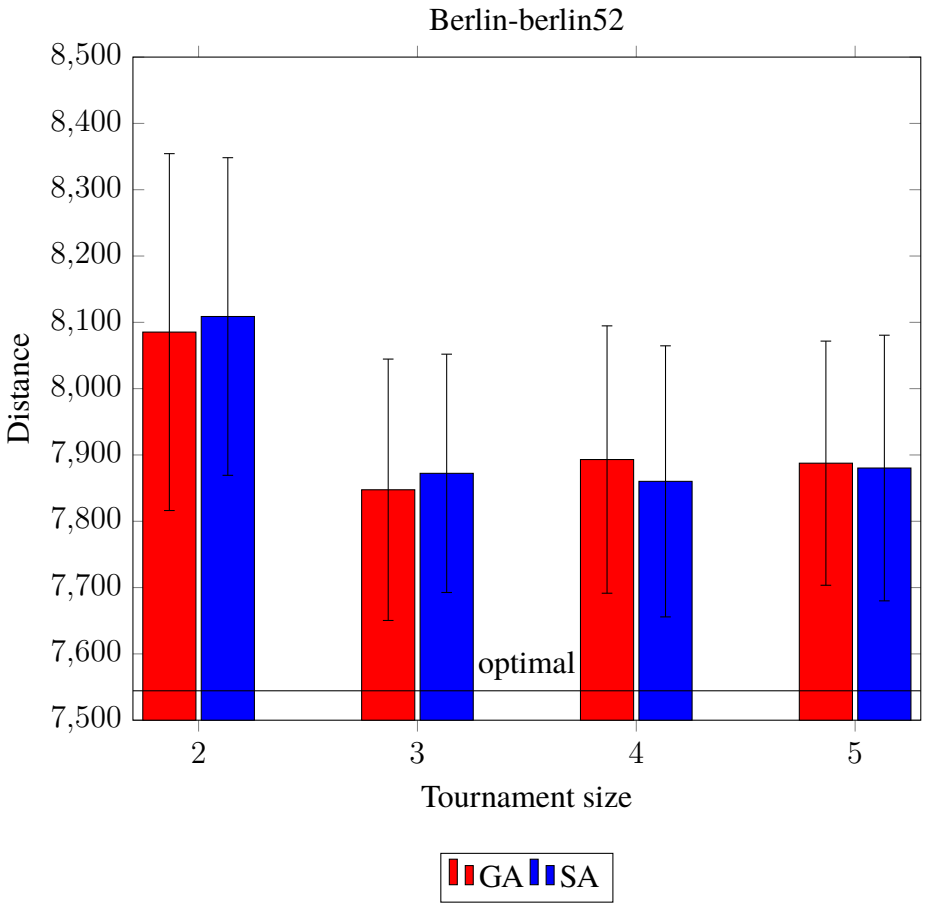

Figure 4.6: Distance of the selected route after convergence on the berlin52 problem.

4.3 Convergence

In figure 4.7 the difference in convergence rate between the different implementations is shown. The traditional GA implementation converges much faster than the SA one. For a higher tournament size the algorithms converges faster.

	2	**3**	**4**	**5**
wi29 GA	11960	1950	480	400
wi29 SA	25610	21310	15080	12420

Figure 4.7: Generations to convergence for the different implementations and tournament sizes on the wi29 problem instance

4.4 Reliability

In order to verify that our results were not simply due to the randomness within the program, we ran the tests on Western Sahara 10 times and compared the results. Figure 4.8 show that the reliability test yielded similar results every time.

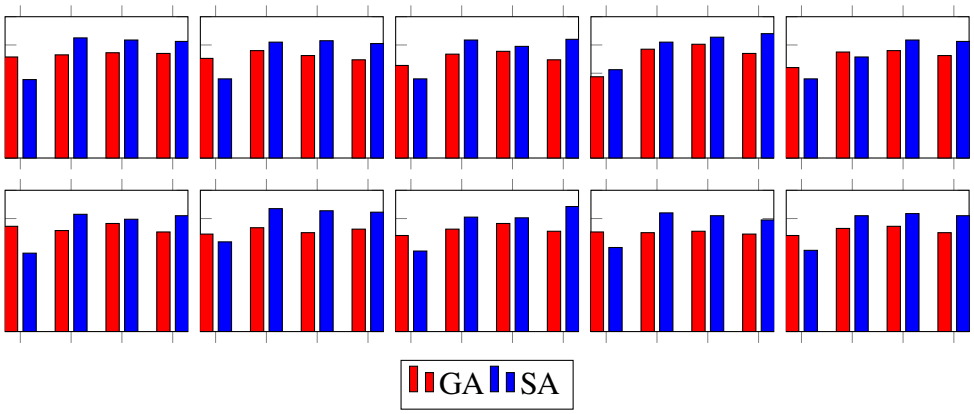

Figure 4.8: This figure show 10 reliability test runs on Western Sahara, with 2, 3, 4 and 5 in tournament size. The results show that the difference in performance was similar every run.

Chapter 5

Discussion

In this chapter we reflect on the findings in the previous chapter and on what other things we could have taken into account when designing the study.

5.1 Results

The results show that both implementations perform the worst with tournament size 2. The reason for this is probably that having parent selection with tournament size 2 makes the algorithm too random. Having only two candidates makes it a bit too easy for poor parents to reproduce, which in turn makes the final result worse.

The SA implementation performed worse than the pure GA with tournament size 2 on all three problems. The reason for this is likely due to the SA algorithm worsening an already somewhat poor parent selection, which amplified the general effect seen with tournament size 2.

The results also show that the SA-inspired algorithm performed better on the two smallest problem instances with tournament sizes 3-5 and on the biggest problem with tournament sizes 4 and 5. This indicates that slowing the convergence of the algorithm did help the algorithm in reaching a better final result on the higher tournament sizes, which had faster convergence rates. This seems reasonable since it aligns with the presumption that slowing the algorithm convergence rate down helps it find a better final value.

These results also indicate that in general, the bigger the tournament sizes the more the SA implementation helped by slowing the convergence down which aligns with the even more general idea of that counteracting premature convergence is helpful. It is noteworthy however, that there are a lot of other ways to slow down convergence in a GA, and that our method necessarily, and

perhaps likely, is not the best one.

A tournament size that have a higher probability of finding a solution within 1% of optimal also on average finds a shorter path. The standard deviation is similar on the graphs but are usually slightly smaller on the SA implementation. The standard deviation being smaller for the SA implementation removes possible doubt that it is due to outliers the SA implementation performs better than the GA implementation. This indicates that the two different measurements are linked together and the SA is slightly better.

Note that the deviation going below the optimum does not indicate that there are individual data points better than optimal.

It is possible that the difference between the algorithms would be even bigger by increasing the tournament sizes. However, the results indicate that there is an optimal balance between exploration and exploitation at tournament sizes 3-5 so while the difference between the algorithms might get bigger, the overall result might get worse.

The results are also interesting since they show the similar idea of that combining GA and SA leads to better performance than just the pure implementation of one of them. Lin et. al. got results showing that their SA+GA was better than their pure SA implementation [18]. Similarly, our results indicate an improvement of GA+SA compared to just GA.

5.2 Method

5.2.1 TSP Instance Sizes

Initially, we had ambitions of performing tests on TSP instances as big as with 194 cities. However, this was not feasible within the time frame of the project. We would however have preferred to have at least a fourth test bigger than berlin52 in order to see if the same trend could be seen in even bigger test sizes.

5.2.2 Parameters

There were a lot of parameters to set within our GA, such as tournament size, population size and number of generations. All of these could affect the results differently.

We do not claim to have used the best possible parameters in our implementation since finding exactly what the parameters would be was a too big of

a task for the scope of this thesis. Instead, we had to settle for "good enough" parameters by testing and using reasonable values.

One way to find out what the best parameters would be could have been to perform an exhaustive search over all of the possible combination of parameters that could have been used. That technique is also known as grid search or parameter sweeping.

Our project would definitely have benefited from such a verification of the parameters, but the time it would have taken made it too big for the scope of this project.

5.2.3 Tournament Sizes

It would be interesting to increase the tournament size even further, at least to find if there is a point where the large tournament size makes the algorithm perform much worse. That also could help observing difference in performance of the different implementations.

However, we suspect that the trend of GA+SA being better than the pure GA seen when increasing the tournament size would be similar and keep going when going up to tournament sizes such as 6, 8, 10 and so on.

5.2.4 Other SA Implementations

The SA method of worsening the parent selection early on in the run-time still seems reasonable in hindsight, even if there are other ideas on what to change that are as easily motivated as the parameter we went with. Some other ideas of a GA with SA implementation are:

- Traditional SA where one could generate an entirely new generation, looking at whether it is better than the current one and then selecting whether to work with it or not.
- Letting the SA function worsen the single individual generation, where one might mutate the individuals more early during the run-time and less later.

5.2.5 Final Notes

The general result that we got is also mainly about slowing the convergence, and convergence can be controlled in a lot of other ways that one does not need to have an SA implementation to do. Some examples are changing the mutation rate, the tournament size or simply using other parent selection, crossover

and mutation methods. Therefore, even though our results show that using an SA can work, it is not necessarily the method of choice for a programmer that just wants to make sure he has a decent GA.

Finally, it is possible that the algorithms would have performed better if they were allowed to run for more generations, particularly in the berlin52 instance. Due to the time frame this option was discarded.

5.3 Further Research

Some ideas on future research could be investigating whether increasing the tournament sizes even further shows the same trend we found.

One could also test other cooling schedules and other SA implementation variations and investigate whether the results seemed similar to ours.

And of course, verifying that one is using the best possible parameters by using a grid search such as mentioned in section 5.2.2 and then seeing if the results are similar would also be interesting.

Chapter 6

Conclusions

This thesis has described the comparison of two different Genetic Algorithms, one "standard" and one with a Simulated Annealing-inspired cooling schedule. The cooling schedule in the SA inspired algorithm controlled the parent selection by making it worse in the beginning of the runtime and standard towards the end. The algorithms were tested on three different sizes of the Traveling Salesman Problem with four different sizes of tournament selection within the parent selection. All other parameters were the same in both algorithms.

The results show that with bigger tournament sizes the SA inspired implementation performed better, but that the difference between the algorithm were within each others standard deviation.

This project has shown that SA can be implemented in order to improve a GA, but that it also depends on other parameters within the GA, and is not necessarily the best method of choice.

Bibliography

[1] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[2] Donald E. Knuth. "Computer Programming As an Art". In: *Commun. ACM* 17.12 (Dec. 1974), pp. 667–673.

[3] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1996.

[4] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by simulated annealing". In: *SCIENCE* 220.4598 (1983), pp. 671–680.

[5] Alexander Schrijver. "On the History of Combinatorial Optimization (Till 1960)". In: *Discrete Optimization*. Ed. by G.L. Nemhauser K. Aardal and R. Weismantel. Vol. 12. Elsevier, 2005, pp. 1–68.

[6] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.

[7] David L. Applegate et al. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton, NJ, USA: Princeton University Press, 2007.

[8] John H. Holland. *Adaptation in Natural and Artificial Systems*. second edition, 1992. Ann Arbor, MI: University of Michigan Press, 1975.

[9] John J. Grefenstette et al. "Genetic Algorithms for the Traveling Salesman Problem". In: *Proceedings of the 1st International Conference on Genetic Algorithms*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1985, pp. 160–168.

[10] P. Larrañaga et al. "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators". In: *Artificial Intelligence Review* 13.2 (1999), pp. 129–170.

[11] Lawrence Davis. “Applying Adaptive Algorithms to Epistatic Domains”. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*. Los Angeles, California: Morgan Kaufmann Publishers Inc., 1985, pp. 162–164.

[12] Noraini Mohd Razali and John Geraghty. “Genetic Algorithm Performance with Different Selection Strategies in Solving TSP”. In: 2011.

[13] David E. Goldberg and Kalyanmoy Deb. “A Comparative Analysis of Selection Schemes Used in Genetic Algorithms”. In: ed. by Gregory J.E. Rawlins. Vol. 1. Foundations of Genetic Algorithms. Elsevier, 1991, pp. 69–93.

[14] H. Cohn and M. Fielding. “Simulated Annealing: Searching for an Optimal Temperature Schedule”. In: *SIAM Journal on Optimization* 9.3 (1999), pp. 779–802.

[15] José Fernando Díaz Martín and Jesús M. Riaño Sierra. “A Comparison of Cooling Schedules for Simulated Annealing”. In: Jan. 2008.

[16] Adewole A.p., Otubamowo K., and Egunjobi T.o. “Article: A Comparative Study of Simulated Annealing and Genetic Algorithm for Solving the Travelling Salesman Problem”. In: *International Journal of Applied Information Systems* 4.4 (2012). Published by Foundation of Computer Science, New York, USA, pp. 6–12.

[17] Hosam Mukhairez and Ashraf Maghari. “Performance Comparison of Simulated Annealing, GA and ACO Applied to TSP”. In: *International Journal of Intelligent Computing Research (IJICR)* Volume 6 (Dec. 2015), pp. 647–654.

[18] Cheng-Yan Kao Feng-Tse Lin and Ching-Chi Hsu. “Applying the genetic approach to simulated annealing in solving some NP-hard problems”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 23.6 (1993), pp. 1752–1767.

Appendix A

Source Code

https://github.com/Matski14/Bachelors_Thesis_Code

Appendix B

Data Sets

B.1 Western Sahara - wi29

City	X-coordinate	Y-coordinate	City	X-coordinate	Y-coordinate
1	20833.3333	17100.0000	16	26150.0000	10550.0000
2	20900.0000	17066.6667	17	26283.3333	12766.6667
3	21300.0000	13016.6667	18	26433.3333	13433.3333
4	21600.0000	14150.0000	19	26550.0000	13850.0000
5	21600.0000	14966.6667	20	26733.3333	11683.3333
6	21600.0000	16500.0000	21	27026.1111	13051.9444
7	22183.3333	13133.3333	22	27096.1111	13415.8333
8	22583.3333	14300.0000	23	27153.6111	13203.3333
9	22683.3333	12716.6667	24	27166.6667	9833.3333
10	23616.6667	15866.6667	25	27233.3333	10450.0000
11	23700.0000	15933.3333	26	27233.3333	11783.3333
12	23883.3333	14533.3333	27	27266.6667	10383.3333
13	24166.6667	13250.0000	28	27433.3333	12400.0000
14	25149.1667	12365.8333	29	27462.5000	12992.2222
15	26133.3333	14500.0000			

B.2 Djibouti - dj38

City	X-coordinate	Y-coordinate	City	X-coordinate	Y-coordinate
1	11003.611100	42102.500000	20	11690.555600	42686.666700
2	11108.611100	42373.888900	21	11715.833300	41836.111100
3	11133.333300	42885.833300	22	11751.111100	42814.444400
4	11155.833300	42712.500000	23	11770.277800	42651.944400
5	11183.333300	42933.333300	24	11785.277800	42884.444400
6	11297.500000	42853.333300	25	11822.777800	42673.611100
7	11310.277800	42929.444400	26	11846.944400	42660.555600
8	11416.666700	42983.333300	27	11963.055600	43290.555600
9	11423.888900	43000.277800	28	11973.055600	43026.111100
10	11438.333300	42057.222200	29	12058.333300	42195.555600
11	11461.111100	43252.777800	30	12149.444400	42477.500000
12	11485.555600	43187.222200	31	12286.944400	43355.555600
13	11503.055600	42855.277800	32	12300.000000	42433.333300
14	11511.388900	42106.388900	33	12355.833300	43156.388900
15	11522.222200	42841.944400	34	12363.333300	43189.166700
16	11569.444400	43136.666700	35	12372.777800	42711.388900
17	11583.333300	43150.000000	36	12386.666700	43334.722200
18	11595.000000	43148.055600	37	12421.666700	42895.555600
19	11600.000000	43150.000000	38	12645.000000	42973.333300

B.3 Berlin - berlin52

City	X-coordinate	Y-coordinate	City	X-coordinate	Y-coordinate
1	565.0	575.0	27	1320.0	315.0
2	25.0	185.0	28	1250.0	400.0
3	345.0	750.0	29	660.0	180.0
4	945.0	685.0	30	410.0	250.0
5	845.0	655.0	31	420.0	555.0
6	880.0	660.0	32	575.0	665.0
7	25.0	230.0	33	1150.0	1160.0
8	525.0	1000.0	34	700.0	580.0
9	580.0	1175.0	35	685.0	595.0
10	650.0	1130.0	36	685.0	610.0
11	1605.0	620.0	37	770.0	610.0
12	1220.0	580.0	38	795.0	645.0
13	1465.0	200.0	39	720.0	635.0
14	1530.0	5.0	40	760.0	650.0
15	845.0	680.0	41	475.0	960.0
16	725.0	370.0	42	95.0	260.0
17	145.0	665.0	43	875.0	920.0
18	415.0	635.0	44	700.0	500.0
19	510.0	875.0	45	555.0	815.0
20	560.0	365.0	46	830.0	485.0
21	300.0	465.0	47	1170.0	65.0
22	520.0	585.0	48	830.0	610.0
23	480.0	415.0	49	605.0	625.0
24	835.0	625.0	50	595.0	360.0
25	975.0	580.0	51	1340.0	725.0
26	1215.0	245.0	52	1740.0	245.0

TRITA-EECS-EX-2019:388