This document can be stored and made available to the public on the open internet pages of Aalto University. All other rights are reserved.

Jussi Hakosalo

Genetic algorithms in symmetric Travelling Salesman Problem

School of Science

Bachelor's thesis
Espoo 31.8.2019

Thesis supervisor:

Prof. Harri Ehtamo

Thesis advisor:

M.Sc. (Tech.) Juho Andelmin

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
BACHELOR'S THESIS

Author: Jussi Hakosalo

Title: Genetic algorithms in symmetric Travelling Salesman Problem

Date: 31.8.2019 Language: English Number of pages: 6+31

Degree programme: Mathematics and Systems analysis

Supervisor: Prof. Harri Ehtamo

Advisor: M.Sc. (Tech.) Juho Andelmin

The purpose of this study is to shed light on the combinatorial optimization problem called symmetric travelling salesman problem (STSP). STSP has been a prominent topic on optimization research for decades with several applications in, for example, logistics, biocomputing and microelectronics manufacturing.

The STSP is easy to understand, but due to its NP-hard nature it is extremely difficult to solve by brute force search. Lately, new metaheuristic algorithms based on genetic inheritance, known as genetic algorithms, have proven to be efficient on tackling this age-old problem.

In this study, I will give a brief overview on the problem, the genetic algorithmic approach, and discuss different stopping criteria and selection methods. Finally, I will compare results of several runs with varying combinations of mutation operators, crossover operations and population sizes on a set of benchmark instances.

Keywords: TSP, Combinatorics, Genetic Algorithms

Preface

My sincerest thanks to Juho Andelmin and Prof. Harri Ehtamo for excellent guidance. In addition, Guild of Physics Board of 2018 has made this whole ordeal much more bearable. Perttu Yli-Opas and Juuso Kylmäoja for their continuous support with my software developer aspirations.

Otaniemi, 31.08.2019

Jussi J. Hakosalo

Contents

Abstract ii

Preface iii

Contents iv

Symbols and abbreviations vi

1 Introduction 1

2 Background 3
- 2.1 Konigsberg bridges . . . 3
- 2.2 Vandermonde and Kirkman . . . 4
- 2.3 Hamilton and The Icosian Game . . . 4
- 2.4 Flood and Whitney . . . 5
- 2.5 Dantzig, Fulkerson, Johnson . . . 5
- 2.6 Branch & Bound . . . 5
- 2.7 Dynamic Programming . . . 6
- 2.8 Lin-Kernighan Heuristic . . . 6
- 2.9 Genetic and Evolutionary Algorithms in TSP . . . 7

3 Materials and methods 8
- 3.1 The genetic algorithm . . . 8
- 3.2 Experiment set-up . . . 9
  - 3.2.1 General notes of the setup . . . 9
  - 3.2.2 The program and functionality . . . 10
  - 3.2.3 Used hardware and software . . . 10
- 3.3 Used data . . . 10
- 3.4 Variation in experiment . . . 11
  - 3.4.1 Crossover operators . . . 12
  - 3.4.2 Mutation operators . . . 14
  - 3.4.3 Population size . . . 17
  - 3.4.4 Stopping criteria . . . 17
  - 3.4.5 Selection methods . . . 17

4 Results 19

5 Conclusions and summary 22
- 5.1 General conclusions . . . 22
- 5.2 Possible improvements to the current algorithm . . . 23
- 5.3 Possible improvements to the program . . . 24
- 5.4 Next steps . . . 24

Appendices

A Best found solutions **28**

B Simulation data **30**

C Using the simulation software **31**

C.1 Getting Started 31

C.2 Prerequisites 31

C.3 Using the software 31

Symbols and abbreviations

Operators

$\sum_i$	Sum over index i
$\mathcal{O}(n!)$	Big-Oh-notation for algorithmic complexity. The notation characterizes how the running time or memory usage of a particular algorithm behaves when the input size grows. $\mathcal{O}(n!)$ describes factorial complexity, $\mathcal{O}(n^c)$ polynomial complexity etc.

Abbreviations

TSP	Travelling salesman problem
STSP	Symmetric Travelling salesman problem
SFML	A C++ library used in media and graphics programming (Simple and Fast Multimedia Library)
GA	Genetic algorithm
GP	Genetic programming
TSPLIB	A collection of data related to symmetric TSP, hosted by University of Heidelberg

1 Introduction

Travelling salesman problem (TSP) has been a prominent and well-known mathematical problem for nearly two centuries. The problem of finding the shortest route that visits a given number of locations has practically always existed, but the first mathematical formation for the problem was given in the 19th century by the Irish mathematician W.R.Hamilton [4]. The first mathematical approach for solving the problem was given in 1930 by Merrill Flood, and the name Travelling salesman problem was coined soon after by Princeton University's Hassler Whitney [2].

The TSP asks the following question: "Given a list of cities and distances between all pairs of cities, what is the shortest possible route that visits each city once and returns to the origin city?". Although cities serve as a good example for the problem, it can be generalized to any n-dimensional Euclidean space where the "cities" are depicted by points in $\mathbb{R}^n$. This study will focus only on the common symmetric variants of TSP (STSP, symmetric travelling salesman problem), where the cost of travelling between two vertices is invariant of the direction of travelling. Hence the cost of travelling along an edge is the euclidean distance between the vertices of that edge.

Let $G = (N, E)$ be an undirected symmetric graph with $N = \{1, \ldots, n\}$ nodes (cities) and $E = \{e = (i, j) : i, j \in N\}$ edges between the nodes where each edge $e \in E$ has a cost c_e. Let $\delta(i) = \{(u, v) \in E : u = i, \text{ or } v = i\}$ be the set of edges that include node i for all $i \in N$, and let $E(S) = \{(i, j) \in E : i \in S \text{ and } j \in S\}$ be the set of edges for any node subset $S \subseteq N$. Let $x_e \in \{0, 1\}$ for all edges $e = (i, j) \in E$ where $x_e = 1$ if edge $e \in E$ is part of the TSP tour, and $x_e = 0$ otherwise. The symmetric TSP can be modeled as follows:

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & \sum_{e \in E} c_e x_e && && (1)\\
\text{subject to} \quad & \sum_{e \in \delta(i)} x_e = 2, && \forall i \in N && (2)\\
& \sum_{e \in E(S)} x_e \leq |S| - 1, && \forall S \subseteq N \setminus \{1\} : 2 \leq |S| \leq n - 1 && (3)\\
& x_e \in \{0, 1\}, && \forall e \in E && (4)
\end{aligned}$$

The objective (1) minimizes the TSP tour cost. The constraints (2) ensure that each city is adjacent to exactly two edges. The constraints (3), typically called *subtour elimination constraints*, guarantee that the solution is a complete tour, and the constraints (4) enforce integrality on the decision variables x_e for all $e \in E$.

STSP is one of the most common and intensively studied problems in the field of combinatorial optimization. It has applications in various fields, such as logistics, DNA sequencing and astronomy. Thus, solving the STSP has practical relevance but due to it being an NP-hard problem [10] makes it difficult to find the optimal solution efficiently. In worst-case scenario the basic brute-force algorithm has a computational complexity of $\mathcal{O}(n!)$, since it must iterate through every single possible permutation of the given points.

As a prominent problem in its field, there have been countless studies regarding different approaches to tackling the STSP. One of these approaches is a mixed metaheuristic called a genetic algorithm. The genetic algorithm is modelled after the genetics occurring in living organisms. This can be divided into five parts:

1. Generate a base population. Usually this process is randomized, and the members of said population only need to be eligible for the problem.

2. Selection: Choose several members from a base population to be the new parent solutions, using a ranking system of some sort, based on the fitness of the individuals. Fitness is simply a metric how well a solution performs in the context of the problem. In STSP, fitness is the length of the route. Keep track of the best solutions obtained so far.

3. Crossover: Combine these new parent solutions using different crossover operators to obtain new child solutions that have inherited traits from their parents.

4. Mutation: Mutate some combinations of the parent solutions to have enough variation in the new population, using again various mutation operators

5. Repeat steps 2-4 until a satisfactory solution has been found or some other stopping criterion is satisfied.

The purpose of this study is to examine the genetic algorithms used in solving STSP. We aim to study different combinations of mutation and crossover operations, and also different sizes of starting and evolving populations. We will also study different types of selection as well. The aim is to find a good combination of parameters and operators to solve a set of STSP instances of the problem as efficiently as possible.

The rest of this thesis is structured as follows. In section 2, we provide historical background for both GA and (S)TSP. Section 3 reviews the materials and methods used in this study. Results of the study are presented in section 4, and section 5 concludes and summarises the work.

2 Background

2.1 Konigsberg bridges

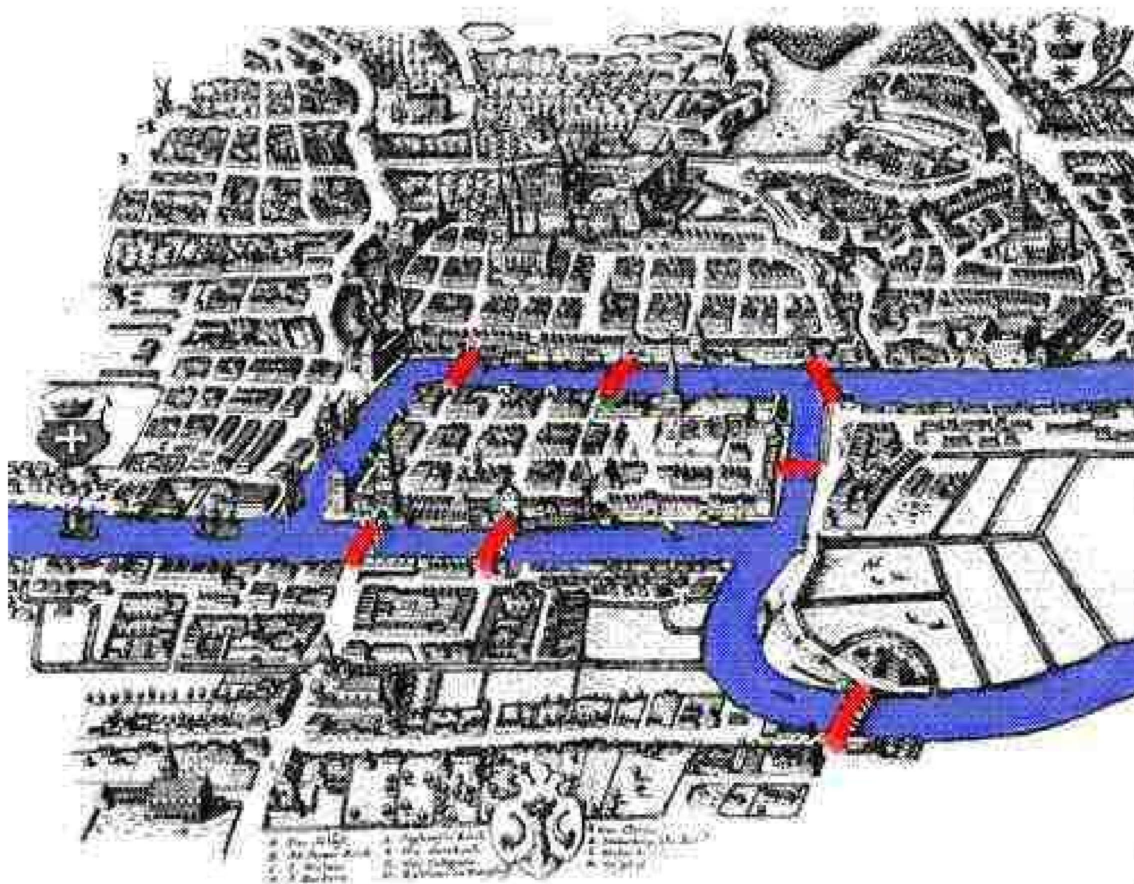

Figure 1: The bridges of Koningsberg. [15]

The then-Prussian city of Koningsberg, nowadays known as Kaliningrad, was the origin of an interesting problem. The city was shaped such that seven different bridges would cross the water in certain places. The shape of the city is shown in Figure 1. The local public wondered whether or not one could find a route where every bridge would be crossed just once, yet ending up in the same location. Seeing as the number of possible routes is quite large (7!=5040), an intuitive explanation did not emerge. This problem was not treated mathematically until 1735 by Euler.

The Koningsberg problem intrigued Euler, who later published a solution to the problem in his 1735 paper '*Solutio problematis ad geometriam situs pertinentis*', the solution of a problem relating to the geometry of position. In his paper, Euler brought the problem from a map to a simple figure with lines and vertices. This is possibly the first-ever occurrence of a graph in strictly mathematical context.

2.2 Vandermonde and Kirkman

The French polymath A.T. Vandermonde studied extensively the so-called *Knight's tour problem*, where a chess knight has to visit every single square of a chess board and return to the starting square. The knight can only move two steps to a direction parallel with one of the edges of the board, followed by one perpendicular step. Vandermonde published a paper in 1777, '*Remarques sur les problèmes de situation*, Remarks on problems of position, where he discusses the Knight's tour problem in a way faintly resembling modern graph theory. The problem itself can be generalised to following: is it possible, for a given graph, to find a circuit where each and every vertex is visited just once?

In the mid-1900th century an Englishman by the name of Thomas Kirkman studied polyhedra, that is, solids bounded by plane faces, such as pyramids or cubes. In his 1855 paper '*On the representation of polyhedra*' Kirkman asks if it is always possible, for a given graph of a polyhedron, to find a circuit visiting every vertex once. Although his findings were later found to be incorrect, he managed to find common features for graphs for which it is impossible to obtain such a circuit. He also provided terminology that would later be used in graph theory.

2.3 Hamilton and The Icosian Game

While Kirkman was working on his polyhedra, another famous mathematician was studying a similar problem. William Hamilton, who is quite renowned for his groundbreaking work on algebra and classical mechanics. Hamilton discovered a system of non-commutative algebra (algebra in which multiplication does not necessarily satisfy the equation $ab = ba$) and named it *The Icosian Calculus* and published it in 1856. He based a certain puzzle named *The Icosian Game* as the graphical interpretation of the aforementioned algebra. The layout of the Icosian Game can be seen in Figure 2 .

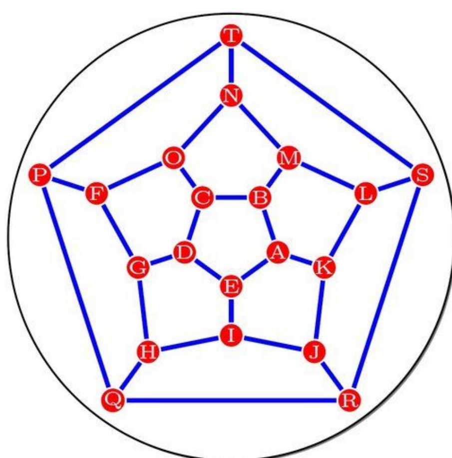

Figure 2: Layout of the Icosian Game.

The game involved several different objectives to find paths satisfying certain conditions. The first problem was indeed that of finding a path that passes once and only once through each vertex on the board. In the wake of Hamilton's work, a circuit passing through each vertex is known as a Hamiltonian circuit, and if it is possible to find such a circuit, the graph is said to be Hamiltonian.

2.4 Flood and Whitney

In the beginning of the 20th century, TSP resurfaced again and was widely studied at the renowned Princeton University. According to Merrill Flood [2], Hassler Whitney brought up the problem in its modern form at a seminar talk in 1934. Flood took the problem from Whitney, started working on it and publicized it in the mathematical community.

Flood moved to industry in 1940s, namely, to RAND Corporation, where many of the 40s discoveries regarding optimization were brought to light. In 1954, Flood worked at RAND with Ray Fulkerson, Selmer Johnson and George Dantzig, who would later become very well known for their work with TSPs.

2.5 Dantzig, Fulkerson, Johnson

At RAND, the TSP work revolved primarily on solving a particular example of the TSP in the United States [2]. Robinson (1949) describes this problem as follows:

"*One formulation is to find the shortest route for a salesman starting from Washington, visiting all the state capitals and then returning to Washington.*" [19]

With his colleagues, Dantzig, the original author of the popular Simplex LP algorithm, approached the problem by reformulating it to a form that they were able to solve. Their approach was to create an LP-relaxation that would give them a lower bound for the original problem. In the 1940s-1950s, LP-problems were a main focus for many studies and research teams.

The most important discovery of Dantzig et al. was that solving a certain LP-relaxation of the problem does not only give a lower bound; they would also use specific subtour elimination constraints which guide the relaxation iteratively closer to the optimal solution.

2.6 Branch & Bound

In the end of 1950s, a new method of solving TSPs emerged. Even though many different studies employing the same search methodology emerged during this period, the first ones to call the algorithm '*Branch and Bound (B & B)*' were Little et al. in 1963 [13]. While some features of B & B algorithms are extensions of the

plane cutting method were studied in the work of Dantzig et al., the method itself did not attract that much attention until the 1960s.

The core principle of B & B-method is to divide the problem into smaller subproblems that may be split into even smaller subproblems, and so on. A branch-and-bound- tree is formed, where the original problem is in the root of the tree and each leaf of the tree is a branched subproblem. Each leaf will then be solved and either discarded or branched further depending on the solution of the subproblem in the leaf node. The optimal solution is found when no nodes exist that can lead to a better solution.

2.7 Dynamic Programming

In 1959, Richard Bellman of the RAND Corporation published his book of dynamic programming. This method, nowadays used for many operations research and computer science problems, quickly found its applications in solving the TSP. The main idea is to assume that if a tour is optimal, any tailing path of the tour from node 1 to n, $n < N$, must be optimal as well. Bellman et al. used this methodology to break the TSP-solving records of the 1960s.

The colleagues of Bellman, Held and Karp, show in their work that the dynamic programming-based approach is able to solve all instances of TSP to optimality in time of $O(n^2 2^n)$ which is significantly better than the upper bound of the brute force search, namely $O(n!)$. Woeginger states that even in the present day, the HK dynamic programming algorithm is the most efficient algorithm for solving any TSP instance to optimality [23] based on its theoretical complexity.

2.8 Lin-Kernighan Heuristic

In the aftermath of Held and Karp's findings, TSP research began to focus in finding tours that were not necessarily optimal, but still of low cost. Different methods for this, called heuristics, were developed in the late 1960s' and early 70s'. Even today, the algorithm that is widely regarded as the best for finding best tours for symmetric TSPs is the Lin-Kernighan- algorithm that was presented in their paper in 1973 [12].

The LK-heuristic algorithm works by trying to modify a given tour in order to produce a new tour with a lower cost. The method, inspired by Flood, removes k edges from the tour and attempts to reconnect the resulting disconnected paths in different orders, dubbed as a *k-opt move* [2]. During the following decades, LK-heuristic was pushed to larger and more complex TSP problems with great success, leading up to Keld Helsgaun solving the enormous TSP World Problem with the algorithm.

2.9 Genetic and Evolutionary Algorithms in TSP

The research on genetic and evolutionary algorithms has been quite separate from that of TSP's. Reeves claims that the term *Genetic Algorithm* first surfaced in 1975 when John Holland used the term to describe a new breed of searching algorithms. Since then, evolutionary and genetic algorithms have been applied to various problems, for instance, in mathematical optimization and product design. Due to the structure of the TSP, genetic algorithm's search methodology fits well into it and can give some excellent results.

In their article *The traveling salesman problem: A case study*, Johnson and Geoch formulate a basic schema for a genetic optimization algorithm that has been used since Brady in 1985 [10] (the algorithm used in this study also resembles their schema). Their schema is presented below:

1. Generate a *population* of k starting solutions $\mathbf{S} = \{S_1, ..., S_k\}$.
2. Apply a given local search algorithm A to each solution S in **S**, letting the resulting locally optimal solution replace S in **S**.
3. While not yet *converged*, that is the results are still improving, do the following:
   (a) Select k' distinct subsets of **S** of size 1 or 2 as *parents*.
   (b) For each 1-element subset, perform a randomized *mutation* operation to obtain a new solution.
   (c) For each 2-element subset, perform a (possibly randomized) *crossover* operation to obtain a new solution that reflects aspects of both parents
   (d) Apply local search algorithm **A** to each of the k' solutions produced in step c, and let **S**' be the set of resulting solutions.
   (e) Using a *selection strategy*, choose k *survivors* from $\mathbf{S} \cup \mathbf{S}'$, and replace the contents of **S** by these survivors.
4. Return the best solution in **S**.

Since Brady's research, various other studies have been conducted to improve the use of GAs in solving the STSP. Suh and Van Gucht improved Brady's solution method by using a heuristic crossover and an improved 2-opt-local search heuristic in 1987. Muhlenbein, Gorges-Schleuter, and Krämer in 1988 were the first ones to construct the whole algorithm in a parallelized way, thus improving the performance of such methods significantly. They also improved the previous crossovers and selection strategies.

In 1991, Johnson adapted the Lin-Kernighan algorithm as a part of the genetic algorithm, aptly naming it *Iterated Lin-Kernighan*. This iterated algorithm was able to outperform a parallel non-genetic LK-algorithm and other then state-of-art genetic algorithms. Nowadays, as far as we know, the modified Lin-Kernighan approach presented by Keld Helsgaun in 2000 is currently the top notch algorithm for solving the TSP [8].

3 Materials and methods

3.1 The genetic algorithm

Genetic algorithms are a class of search heuristics that aim to imitate the principles of evolution and genetic inheritance in nature. To facilitate the discussion in further chapters, the following defines a core terminology used in conjunction with genetic algorithms.

Sastry et al. (2005) describe genetic algorithms as methods that typically encode the problem variables as sets of strings. The strings that function as the candidate solutions to the problem are referred to as *chromosomes*, alphabets are referred to as *genes* and gene values *alleles*. In the case of the STSP, a chromosome is one of the candidate routes, while genes are the different cities and alleles basically the numerical ordering of the cities in a route. Another important definition is the concept of fitness: in nature, fitness is usually the lifespan from the beginning of certain species. In GAs the fitness must be measured in a way fitting to the problem at hand, so that the algorithm guides the evolution towards better solutions. In the STSP, we only care about the lengths of certain routes. Our fitness function therefore has to rank the routes based on their total length and nothing else. Indeed we also have to require that the route is legitimate: it has to visit each and every vertex (city) once and only once. In addition, the starting point for the route must also be its terminating vertex [20].

Population refers to all the candidate solutions currently under evaluation. Many search methods start the search from scratch, but a GA has to be provided a starting population in order to evolve and find better solutions. Starting population size plays a significant role in the performance of the algorithm. Too big a starting population might be too heavy and waste computational resources, whereas a starting population too small typically leads to fast convergence on local minima, thus hindering the performance of the algorithm significantly.

The steps of a GA go as follows (Sastry et al., 2005) [20] :

1. Start and initialization: Initial population is generated, usually randomly.

2. Evaluation: The candidate solutions i.e., chromosomes, are evaluated and ranked based on the specified fitness function. The choice of the fitness function allows us to easily adapt a GA to cater to our specific needs. For instance, we might want to find a solution where the average vertex-to-vertex-distance is maximized.

3. Selection: As discussed before, in this phase the chromosomes for our new generation are selected. Obviously we tend to favour those that rank higher on the fitness function and disregard the worse-performing ones. Selection can be based on purely the fitness of the chromosome, or it can be anything else. Different selection methods are discussed in more detail in the next section.

4. Recombination/Crossover: As in nature, the selected chromosomes are used as parents to create a new population via recombination. The general idea is to use the best *qualities* of the parent solutions, while still varying the solutions enough so that the algorithm does not instantly converge on a weak local solution.

5. Mutation: The new population resulting from the recombination undergo a mutation phase, like in nature, to increase the variation in the new chromosomes. Mutation usually only focuses on a certain part of the solution, changing just parts of its traits.

6. Replacement: The original population is replaced by the new recombined and mutated population of solutions aka. *new generation*.

7. Iteration: We iterate steps 2-6 until the algorithm stops. Stopping criteria can be, for instance, a computation limit time or no improvement in the best solution over a predefined number of successive generations.

3.2 Experiment set-up

3.2.1 General notes of the setup

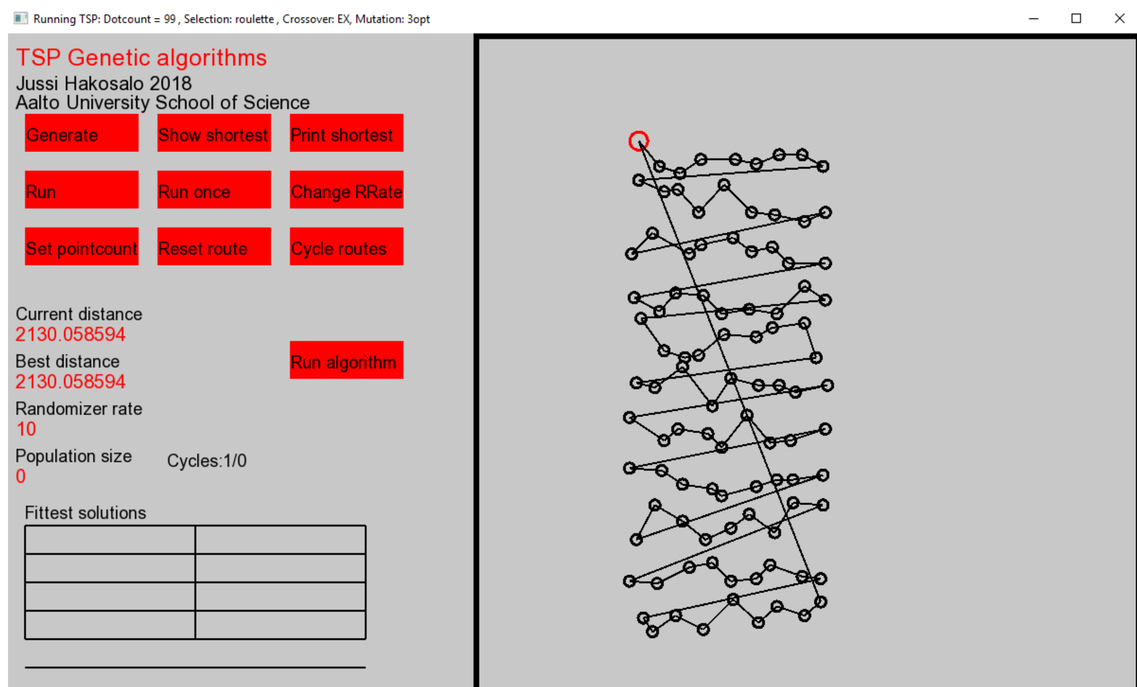

Figure 3: General view of the simulation.

The environment used in this study is a simple C++ - based simulation program that visualizes the TSP solution in a simple manner as seen in 3. The main idea is to make the algorithm as clear as possible. The simulation is capable of running the

algorithm with varying parameters easily. We can also easily generate new graphs and run the algorithm on them. Step-by-step-approach is also possible, should the user want it.

3.2.2 The program and functionality

The program is based on a number of different collections that keep track of current route populations. One of them presents the current work-in-progress route collection while others are used as auxiliary memory to facilitate the computation of new generations.

The user is able to use either a custom number of randomized vertices, or a predefined vertex set that can be imported in the program. The application offers a simple framework where customizing the stopping criterion, population sizes, and different crossover-, mutation-, and selection algorithms are readily available. Visualizing the results is not done separately; instead, MS Excel may be used for this functionality.

3.2.3 Used hardware and software

All the simulations have been ran on the following setup:

- OS: Microsoft Windows 10 Education, version 10.0.17134 Build 17134
- Environment/IDE: Visual Studio Community 2017, version 15.7.1. SFML version 2.5.1.
- Compilation (built in VS): g++ with c++17 standard. Compiler flags: -fopenmp -Wall -O3 -march=native
- CPU: Intel Core i5-6600K CPU @3.5GHz, overclocked to 3.8GHz
- RAM: 16GB of DDR4

3.3 Used data

The data used in this thesis has been obtained from TSPLIB database maintained by University of Heidelberg [18]. The data repository hosts hundreds of different TSP instances and their confirmed optimal solutions. The problem instances used in the computational tests are listed in table 1.

Table 1: Instances used in the experiment and their respective node (city) counts.

Instance name	Node count
berlin52	52
bays29	29
rat99	99
kroA100	100
lin105	105
pr76	76
pr124	124
att48	48
st70	70

3.4 Variation in experiment

The genetic algorithms are tested on different problem graphs with varying parameters. The modified parameters are as follows:

1. Crossover operators: the operators that govern how the fittest members that are chosen will be combined with each other.

2. Mutation operators: these operators govern how the resulting routes from the crossover are mutated before they are added to the new generation, to ensure that the algorithm does not get stuck on local minima.

3. Population size: the size of the population from which fittest parents are chosen. An initial population too big will lead to slower execution, whereas a population too small typically converges fast but tends to get stuck on a local minima.

4. Stopping criteria: the rules by which the algorithm ceases its execution. For instance, if the cost (length) of the best route does not improve after a specified number of generations, the algorithm may terminate. Another example could be a time-based stopping criterion.

5. Selection methods: there are different ways to select the best candidate solutions for the GA. We can simply sort the solutions in ascending order based on their fitness, or randomly choose the new population and weigh the random selection with the fitness of each solution?

The main focus of this study is on the first two aspects, namely the crossover and mutation operators. However, the other mentioned variations will be experimented with as well.

3.4.1 Crossover operators

Let us give a brief overview on the operators used in this study. The following crossover operators will be used:

1. Order crossover (OX) (Olivier et al.1989, Goldberg 1987)
2. Edge recombination crossover (ER) (Whitley et al. 1987)

The OX is based on maintaining the implicit order of the parent tours. After the two parent chromosomes are selected, the first parent is cut from two points (these points can be randomly selected or pre-determined). The substring between the two cut points is copied on the offspring directly. Then we will complete the offspring by adding the following cities in the same order they appear in the second parent. J.Y. Potvin explains the process clearly as follows.

Let us study a simple TSP with 8 cities, identified by numbers 1-8. Via random selection and ranking, we have selected two parent solutions. We shall take an arbitrary substring from the first parent, substitute it directly to the second parent, and then add all other cities after the cut-and-pasted substring in the same order as they appear in parent 2. If they are already in the added cut, the next city in order is chosen. In the following example, the basic principle of the OX is demonstrated.

Starting with parents 1 and 2 as given in (5), we randomly choose an arbitrary subset of subsequent cities from the first parent. In this example, we choose the subset [2,7,3,6]. Then we add all the remaining cities outside the subset in the order the cities first appear in parent 2.

Parent 1: | 1 | 5 | 2 | 7 | 3 | 6 | 4 | 8 |
Parent 2: | 8 | 6 | 7 | 5 | 4 | 2 | 3 | 1 |

Offspring
Step 1: | – | – | 2 | 7 | 3 | 6 | – | – |
Step 2: | 5 | 4 | 2 | 7 | 3 | 6 | 1 | 8 |

(5)

The ER uses a specialized 'edge map', a simple map data structure that "maintains the list of edges that are incident to each city in the parent tours and that lead to cities not yet included in the offspring" (Potvin 1996). We add to the offspring a city that has the least number of edges to those cities that are not yet included in the offspring. Should two cities share the same number of such edges, one of them

will be chosen randomly.

According to Potvin, ER reduces the chances of our algorithm getting trapped in a local optimum. Let us use the following parent tours:

$$\begin{aligned} &\text{Parent 1: } | 1 | 4 | 2 | 6 | 3 | 5 | \\ &\text{Parent 2: } | 4 | 5 | 3 | 2 | 6 | 1 | \end{aligned} \tag{6}$$

Let us construct the edge map and keep track of the cities and their neighbors.

$$\begin{aligned} &\text{city 1 has edges to: } 4\ 5\ 6 \\ &\text{city 2 has edges to: } 3\ 4\ 6 \\ &\text{city 3 has edges to: } 2\ 5\ 6 \\ &\text{city 4 has edges to: } 1\ 2\ 5 \\ &\text{city 5 has edges to: } 1\ 3\ 4 \\ &\text{city 6 has edges to: } 1\ 2\ 3 \end{aligned} \tag{7}$$

All of these cities have the same number of distinct neighbors. Next, we will select one of the cities randomly; in this case the city 3. We remove the corresponding entry from the edge map and all references to it from the respective edge collections.

$$\begin{aligned} &\text{city 1 has edges to: } 4\ 5\ 6 \\ &\text{city 2 has edges to: } 4\ 6 \\ &\text{city 4 has edges to: } 1\ 2\ 5 \\ &\text{city 5 has edges to: } 1\ 4 \\ &\text{city 6 has edges to: } 1\ 2 \end{aligned} \tag{8}$$

Select city 5:

$$\begin{aligned} &\text{city 1 has edges to: } 4\ 6 \\ &\text{city 2 has edges to: } 4\ 6 \\ &\text{city 4 has edges to: } 1\ 2 \\ &\text{city 6 has edges to: } 1\ 2 \end{aligned} \tag{9}$$

Select city 2:

$$\begin{aligned} &\text{city 1 has edges to: } 4\ 6 \\ &\text{city 4 has edges to: } 1 \\ &\text{city 6 has edges to: } 1 \end{aligned} \tag{10}$$

Select city 4:

$$\begin{aligned} &\text{city 1 has edges to: 6} \\ &\text{city 6 has edges to: 1} \end{aligned} \tag{11}$$

The last two cities will be, again, chosen randomly. By collecting all the cities in order, our crossover result is the following:

$$\text{Offspring: } |\ 3\ |\ 5\ |\ 2\ |\ 4\ |\ 6\ |\ 1\ | \tag{12}$$

3.4.2 Mutation operators

The following mutation operators are to be used:

1. Swap mutation (SM)
2. Scramble mutation (CM)
3. 2-Opt heuristic (2O)
4. 3-Opt heuristic (3O)
5. Lin-Kernighan heuristic (LKH)

Even though the 2O, 3O and LKH could be treated as crossover operators, I will regard them as mutation operators for the sake of simplicity after Potvin [16].

The SM is a simple operator that swaps two randomly selected cities in a tour with each other. The change made is quite small, akin to the original concept of mutation. However, as the swaps are completely randomized, our algorithm may not lead to better solutions frequently.

The CM chooses a random substring from the offspring and randomly permutates this substring. If the chosen substring is too small, this mutation only affects locally and might not prevent us from getting stuck in local minima. On the other hand, choosing a substring too big will completely scramble a significant portion of the route, thus leading to excessive randomization.

The following three: 2O, 3O and LKH, are known as *k-opt* heuristics, operating around the principle of local hill climbing. The heuristic operates by removing k edges from the graph, creating 2 or more distinct paths depending on the k value. Thereafter the algorithm tries to form a shorter route by recombining these paths into a new route using every possible edge configuration and choosing the shortest one. The same process is iterated over every possible set of the vertices until all

possible combinations have been iterated trough and no further improvement can be achieved. A single k-opt swap will allow $k^2 - 1$ ways of recombining the resulting paths, thus resulting in a $O(n^k)$ computational complexity per single execution.

The 2-opt heuristic is the simplest k-opt heuristic. This algorithm removes two edges from the graph, thus creating 2 node-disjoint paths. It then tries to combine these paths with edges that make the tour complete while improving it as much as possible. As there are only two edge combinations for reconnecting any two paths into a tour, the 2-opt inherently removes all edge crossings. An example of 2-opt swap is presented in Figure 4. We shall present an interesting result regarding the 2-opt swap:

Theorem 3.1. *If any Hamiltonian path for a graph $G = (N, E)$ is a boundary to a convex set, the path in question is uniquely the shortest Hamiltonian path for the graph G. Additionally, this path will always be the result of a single iteration of the 2-opt hillclimb heuristic for any Hamiltonian path in G.*

Proof. Let $G = (N, E)$ be an undirected symmetric graph with $N = \{1, \ldots, n\}$ nodes (cities) and $E = \{e = (i, j) : i, j \in N\}$ edges between the nodes where each edge $e \in E$ has a cost c_e. A simple iteration of 2-opt heuristic iterates through every single possible 2-partition of the tour and tries to find the optimum by reversing the partitioned paths. Therefore 2-opt effectively removes all edges that cross each other. Since we are operating in $\mathbb{R}^2$ and the graph is symmetric, triangle inequality must hold. Therefore if a tour P resulting from 2-opt forms a boundary to a convex set, it is hence the shortest Hamiltonian path (tour), since any other tour would not satisfy the triangle inequality and would have crossing edges, which the 2-opt procedure removes, and thus can not be any shorter than P. □

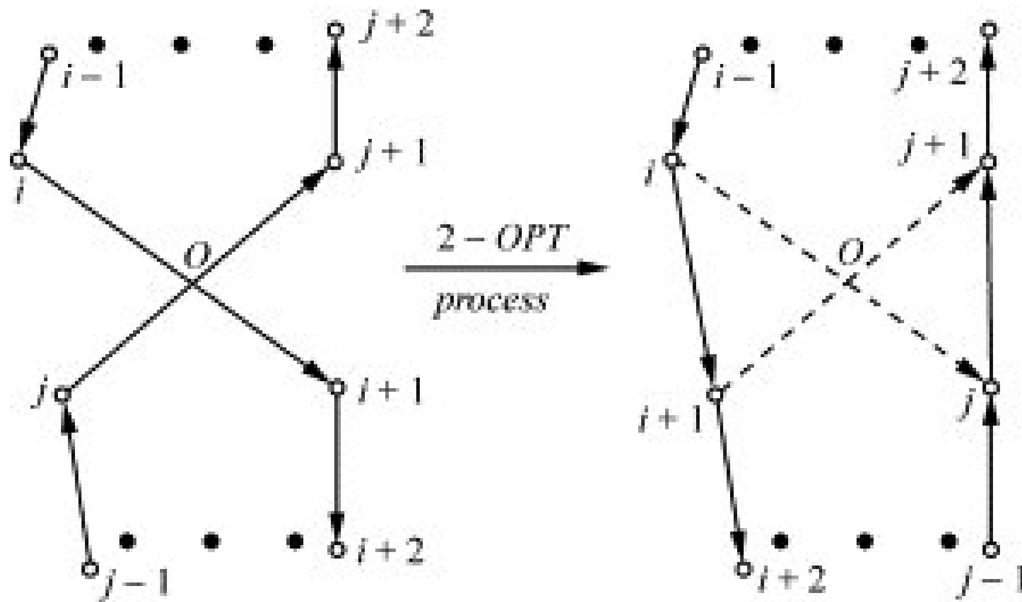

Figure 4: Example of a single 2-opt swap [24].

The 3-opt heuristic removes three edges from the graph, creating 1-3 node-disjoint paths that can be reconnected in 7 different ways to create new routes. Otherwise 3-opt works similarly to 2opt. After removing the three edges, it goes through every possible recombination of the three resulting node-disjoint paths, finding the best way of combining the paths. An example of 3-opt swap is presented in Figure 5

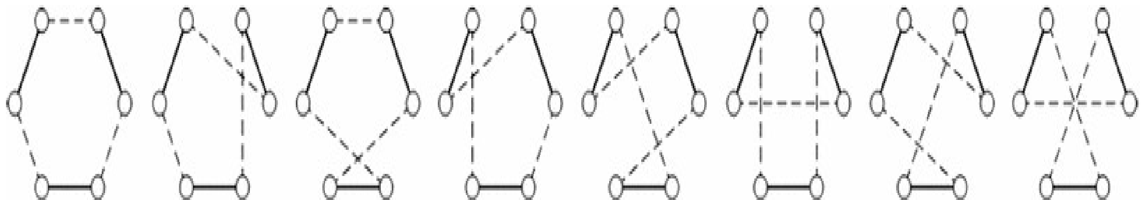

Figure 5: Example of possible 3-opt swaps [7].

The Lin-Kernighan algorithm published in 1973 [12] is hailed as the most successful heuristic for solving the STSP [8]. LK-heuristic is a generalization of the previously mentioned 2-opt and 3-opt heuristics. The LK heuristic works in the same fashion as the previous swap heuristics, with the exception that the swap size k is decided adaptively between each step. The algorithm deduces how many edges need to be removed in each stage in order to find a better route.

Most successful implementations of the genetic algorithm in TSP tend to revolve around LKH due to its relative simplicity and tremendous computational prowess. Keld Helsgaun of Roskilde University namely has developed an LKH-based algo-

rithm that holds the record for shortest route of the 1,904,711-city World TSP.

Due to lack of time and resources, the LKH implementation will be completed after the publication of this thesis.

3.4.3 Population size

The population size of the TSP genetic algorithm plays a tremendous role in its performance. In 1986 Grefenstette claimed that a population size of 60-110 is optimal for these purposes [6]. However, M.O. Odetayo presented studies that do not support this claim [14]. Using Grefenstette, Reeves and Odetayo as a basis, a population size varying between 40-100 will be used in this thesis.

3.4.4 Stopping criteria

Different stopping criteria can be devised in order to stop the execution of the algorithm if a desired outcome is achieved. For testing sets in which an optimal solution is known, the stopping criterion can be quite simply the relative difference of the costs of the route found and the known optimal route. In bigger instances or problems without a known optimal, one valid stopping criteria is the difference in the route length between subsequent generations. If the best route does not improve after a number of generations, satisfactory convergence has been reached and the algorithm will likely not find any better solutions. In the business world, a computational time limit might be used in problems where time or computational resources are more important than fine-tuning the obtained solutions.

In this work, combinations of various stopping criteria are used. The two main ones are:

1. A constraint requiring an improvement over a number of successive generations.
2. A clause that terminates the algorithm if
   (a) More than 4 generations have passed and
   (b) Two successive generations do not improve the length of the best route enough.

3.4.5 Selection methods

Selection should be related to the fitness in a way that better fit solutions are more likely chosen as the parents of the new generation. Choosing an efficient selection method is of utmost importance, as it ensures that the algorithm is progressing towards a better solution constantly while still maintaining a heterogeneous parent population that is unlikely to get stuck on local minima. We use two different selection methods: roulette wheel selection following Baker's *stochastic universal selection* [3], and a simple elitism selection. A survival rate of 50% is used in the

code, meaning that 50% of the candidate solutions are selected as a basis for the next steps of the algorithm.

The simplest form of selection, fitness selection (i.e., elitism selection), ranks the routes in fitness order, resulting in a simple and efficient selection. In short, we select the n best parents from a generation and use their combinations in the successive steps. Albeit this kind of a selection is very easy to implement, it converges easily and contributes to the homogeneousness of successive populations.

The fitness proportionate selection, i.e., *Roulette wheel selection*, operates similarly to a weighted roulette wheel. The selection probability for each route is proportional to its fitness so that routes with highest fitness scores are more likely to be selected as parent chromosomes for the next generation. We shall use a *stochastic universal selection* method proposed by Baker [3] to obtain a systematic and nonbiased selection.

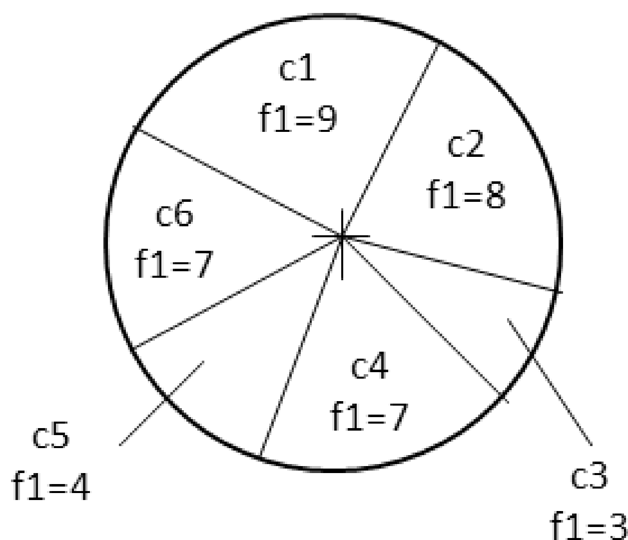

Figure 6: Figure of a roulette wheel selection [22]

The basic execution of a roulette wheel selection goes as follows:

1. A fitness value is assigned to every route in the generation.

2. Based on their respective fitness scores, the chromosomes will be assigned decimal values so that the sum over all of the chromosomes in the generation is one, e.g., a probability distribution, as shown in figure 6.

3. We select random values from a uniform distribution over $[0, 1]$ and select chromosomes based on the corresponding random values.

This way the algorithm favours those candidates that have higher fitness values and are thus more likely to be selected. However, the probability-based selection means that even chromosomes with low fitness values may occasionally be selected, thus contributing to the heterogeneity of the population.

4 Results

This section studies the genetic algorithm and the different algorithmic components and parameters introduced in Section 3 by solving a set of STSP benchmark instances. Due to the enormous number of possible parameter and algorithm settings, most of the parameter values and settings have been found by trial and error. The best found solutions and algorithm variations will be studied here in further detail. However, the reader is invited to get the software for themselves and play around with the experimental setup in order to confirm the findings presented in this study.

The used instances and their respective optimal solutions have been obtained from Universität Heidelberg's TSPLib, courtesy of Prof. Dr. Gerhard Reinelt.

The following results are calculated with various parameter and operator combinations. The starting population size varies between 40 and 70 and the generation size is 70. The stopping criteria used consists of two distinct components:

1. If the average route length of a given generation is less than 2% shorter than the average route length of the preceding generation, the algorithm will terminate and returns the best solution found over all generations.

2. If the current generation is at least the nth one (where $n = 4$) and two successive generations contain identical routes, the algorithm will terminate, returning the best solution found over all generations.

All the simulation runs presented in this section were repeated 5-10 times, each run initiated randomly by selecting the initial population size to be between $40 - 70$. The smaller problems were repeated 10 times while the larger ones only 5 times due to the time constraints to finish the simulation tool to complete the thesis.

The purpose of the first criterion is simply to save time. Albeit better solutions may still be found, the required 2% improvement indicates that the algorithm has already converged significantly, thus rendering further calculations and generations ineffective relative to the additional computation time required. The second criterion ensures that successive generations remain heterogeneous.

Table 2 presents the shorthand notations for the different algorithmic components. The names consist of three parts: selection, crossover and mutation. For instance, *Ru-EX-2opt* means fitness proportionate (roulette) selection, edge recombination crossover and 2-opt hillclimb mutation.

Table 2: Acronyms and their meanings for different algorithmic components.

Legend	Meaning
R	Ranked/elitism selection
Ru	Roulette selection
OX	Order crossover
EX	Edge recombination crossover
2opt	2-opt local hillclimb mutation
3opt	3-opt local hillclimb mutation

Table 3 lists the instances used in the experiment, their number of cities, the optimal tour lengths for the instances, the best solution found with the used GA variant, the parameters of the used GA algorithm, the average time elapsed over 5-10 runs, and the relative difference (Δoptimal) between best found solution cost and the known optimal cost. The relative differences are computed as

$$\Delta\text{optimal} = (\text{Best}/\text{Optimal} - 1) \times 100\%$$

The complete results are presented in appendix B. The use of the mutation methods swap and scramble have been omitted from the results since it quickly became evident that, due to their high degree of randomness, they are not able to produce significant improvements even with longer computation times. However, they are still implemented in the software. Figure 7 shows the best solution obtained (which is also optimal) for the instance pr76 which has 76 cities.

Table 3: Results of the GA on the selected benchmark instances over 5-10 runs.

Instance	Algorithm	Time/iteration	Avg Best	Optimum	Best	Δoptimal
berlin52	Ru-EX-3opt	26.06s	7700	7542	7544	0.027%
att48	R-EX-2opt	44.34s	33719	33523	33523	0.000%
bays29	Ru-OX-2opt	1.32s	9116	9074	9078	0.044%
st70	Ru-OX-2opt	109.98s	678	675	677	0.296%
kroA100	Ru-EX-3opt	422.20s	21436	21282	21285	0.014%
lin105	Ru-OX-2opt	715.93s	14506	14375	14406	0.216%
pr124	Ru-OX-2opt	1531.04s	59527	59030	59030	0.000%
rat99	Ru-EX-3opt	495.62s	1240	1211	1223	0.997%
pr76	R-OX-2opt	123.64s	108825	108159	108159	0.000%
Average	-	385.57s	28527	28319	28325	0.177%

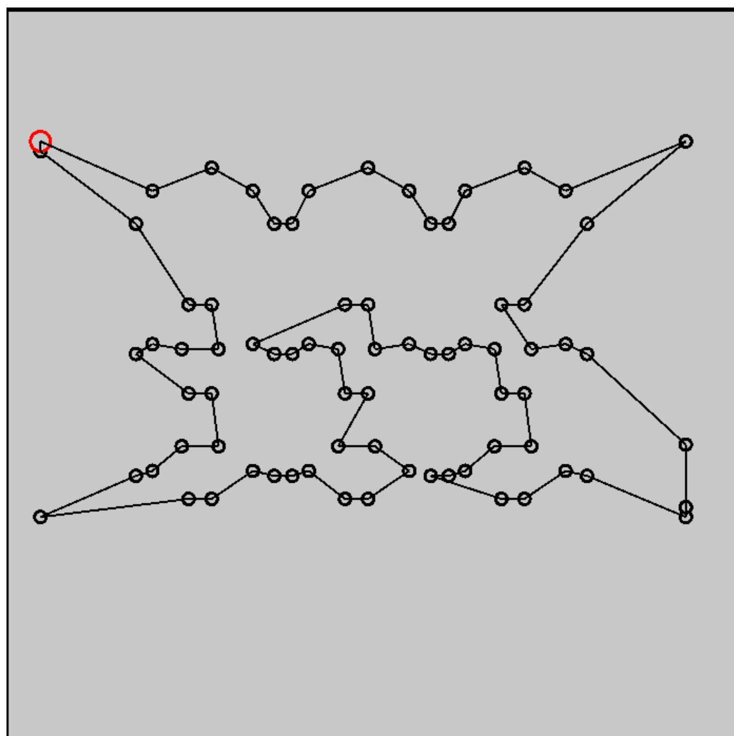

Figure 7: One of the problem instances, pr76, solved to optimality. This solution was obtained with the Ranked-OX-2opt variant. The optimal cost is 108159.

5 Conclusions and summary

5.1 General conclusions

Based on the results, the studied algorithm for the STSP is able to solve mid-sized problem instances very close to optimality in a relatively short time using a standard desktop computer. The custom built simulation software also provides meaningful visualizations for the instances.

Albeit the achieved success depends greatly on the chosen algorithm and how suitable it is for the problem instance at hand, several conclusions can be drawn from the data. It is also worth noting that the computed test instances are not too broad in variety or in size, and thus the conclusions are not as robust as they could be in a more extensive study.

First thing to note is that most of the problem instances were solved to less than 0.3% from optimality. Sensible approaches to achieve optimality could be, for example, using less strict stopping criteria, adding an extra layer of randomized mutation operators after the local hillclimb procedure, or simply re-weighting the roulette selection. Improvements to the current algorithm are discussed in more depth in Section 5.2.

Another noteworthy aspect is that the algorithm which employed 3-opt local hillclimb operated relatively faster in almost all of the used instances compared to 2-opt. 3-opt hillclimb has a theoretical time complexity of $\mathcal{O}(n^3)$, whereas 2-opt has that of $\mathcal{O}(n^2)$. This indicates possible problems in the code, because 2-opt algorithm should perform much faster than 3-opt.

Moreover, of all the 9 problem instances, algorithms employing the 2-opt mutation produced the best results for 6 of the 9 instances. Since 3-opt also includes 2-opt as a sub-procedure, one likely reason for this anomaly is that 3-opt causes the population to become too homogeneous much faster than the 2-opt, thus getting more easily stuck in local minima. A countermeasure for this would also be some forced method to mutate the generation after the 3-opt procedure, perhaps by the same methods as suggested in Section 3.

As Potvin found is his research [16], the EX and OX are the premier crossover operations for the GA TSP. His research showed that EX was able to find marginally more optimal solutions than OX, but the difference was not huge. In our data, 4 out of 9 of the best solutions were produced by an algorithm employing EX, whereas in 5/9 cases OX performed better. Thus both of these methods should be used on mid-sized instances to improve the chances of finding an optimal solution. From a time intensity point of view, EX was approximately 10-30 % slower than the OX on the same instances.

Regarding the selection operators, the results were as expected: the roulette wheel selection outperformed the simple elitism selection in 7 of the 9 test instances. This is to be expected, since even though simple elitism selection always ensures the fitness of the parent population to be maximized, the downside is that the not-so-fit candidates are never selected, even though some of their characteristics might be useful in escaping possible local minima. Using a simple ranked selection comes at a cost of risking the heterogeneousness of the population. However, as the results show, the differences between these two selection methods are not too significant. If the instances would have been bigger in scale, the difference between these two would perhaps have been clearer due to the aforementioned reasons.

5.2 Possible improvements to the current algorithm

As we briefly discussed in the previous section, most of the instances were solved to optimality or close to optimality. This demonstrates that the algorithm indeed converges quite well. The biggest issues of solving the instances relate to the problem of converging to local minima that is suboptimal. However, the results clearly show that the optimization method produces near-optimal solutions.

Thus, one of the main improvements would be to further increase the heterogeneousness of the populations in order to avoid local optima pitfalls. One possible method of achieving this is adding another step to the method: an extra mutation operation that is solely based on randomness and not local hillclimbing. This way we would be able to increase the variety of the populations. For instance, instead of using, e.g., Ru-OX-2opt, the diversity of the generations could be increased by adding a small randomized mutation (swap or scramble) in the end, resulting in Ru-Ox-2opt-swap. However the algorithm would then likely require more time to converge to a (probably improved) solution, so a suitable trade-off must be decided.

Another idea akin to the previous one would be to simply adjust the weights of the roulette wheel selection. In its classical form, the probability of choosing a specific individual for the next generation is directly proportional to its fitness value relative to the fitness values of other chromosomes. If we would rescale the probability weights, for instance, by taking some root of all the fitnesses and doing a similar roulette wheel selection on those, the likelihood of choosing suboptimal individuals would be increased. As with the previous improvement suggestion, this would likely increase the time it takes for the algorithm to converge.

Furthermore, the stopping criterion should not stop the algorithm if the best tour does not improve *enough* over a number of generations. Instead, this criterion should require that the the algorithm stops when the best found tour does not improve over a number of successive iterations.

5.3 Possible improvements to the program

Regarding the simulation program, a lot of improvements could be made. First off, as stated previously, our 2-opt routine takes more time than the 3-opt in practically all of the instances. That should not be the case due to the computational complexities of these being $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively. This could be resolved by using exactly the same style of programming in both of these. The current implementation of 2-opt is vastly different from that of the 3-opt, even though it performs very similar operations (remember that 3-opt includes 2-opt as subroutine). Therefore, by refactoring the 2-opt to be similar to the 3-opt routine would already speed up all of the 2-opt computations by a vast margin, thus leading to a more efficient execution of the program.

Possibly the biggest improvement that can be made to the program is moving the computation from CPU to GPU. Tsutsui (2013) has listed genetic algorithms as an *embarrassingly parallel* problem, meaning that the problem can be efficiently parallerized easily [21]. Modern day GPUs are very powerful when computing simple arithmetic operations, being able to do them in a massively parallerized fashion. As Brodtkorb et al. (2013) state, their 1000-city TSP instance takes 175 seconds on the CPU, compared to 2 seconds on GPU [5], a 80-fold speedup. Thus using GPU for the computation would allow faster simulation runs on bigger instances with similar hardware.

Furthermore, since the hillclimb heuristics take up about 99% of all the computation time, they are the only part of the program that has been parallerized in the code. Most of the instances used here are so small that parallerizing selection or crossover would not benefit us all too much, since spawning and killing the parallel threads also takes a bit of time. However, if this simulation would be used for bigger instances, parallerizing the other methods as well would benefit in the long run.

The whole program is coded in C++ using standard collection vectors as primary data structures. This leaves some room for improvement, as according to Daniel Lemire of University of Quebec[11], standard library vectors require specific operations to use the memory efficiently and thus require extra care when used in high performance code. Another software engineering approach is to use hashed sets where necessary to further improve the memory access pattern and therefore the performance of the entire simulation.

5.4 Next steps

Next steps to this research would be:

1. Evaluate more instances than we have here. Drawing dependable conclusions from a limited sample size makes the performance evaluation insufficient. Since that requires more time and computing resources, doing so was not possible in the scope of this study.

2. Solve larger STSP instances. The largest tackled instance in this study consisted of 124 cities, which is still very small in the world of combinatorial optimization. Optimizing the simulation software is the priority here starting with the improvements discussed in the previous chapter, such as GPU paralerizing. In order to scale up the simulation, both code optimization and more computing resources would be necessary.

3. To tackle the most demanding STSP instances, improving the algorithms used in this study would be to change the 2-opt and 3-opt to a Helsgaun modified Lin-Kernighan heuristic, the current state-of-the-art TSP algorithm.

References

[1] Adewole et al. A Genetic Algorithm for Solving Travelling Salesman Problem. International Journal of Advanced Computer Science and Applications. Vol. 2, No. 1, January 2011

[2] Applegate et al. The Traveling Salesman Problem: A Computational Study. Princeton University Press 2007.

[3] Baker, J. 1987. Reducing bias and inefficiency in the selection algorithm. Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application. Pp. 14-21.

[4] Biggs et al. 1976. Graph Theory 1736-1936. Oxford University Press.

[5] Brodtkorb, A.R. et al. 2013. GPU computing in discrete optimization. Part I: Introduction to the GPU. Euro Journal on Transportation and Logistics. 2(1-2), pp.129-157.

[6] Grefenstette, J.J. 1986. Optimization of Control Parameters for Genetic Algorithms. IEEE Transactions on Systems, Man and Cybernetics. SMC-16(1), pp. 122-128.

[7] Gülcü, Ş.A parallel cooperative hybrid method based on ant colony optimization and 3-Opt algorithm for solving traveling salesman problem et al. Soft Computing (2018) 22: 1669.

[8] Helsgaun, K. An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. European Journal of Operational Research 126(1) 2000, 106-130

[9] Helsgaun, K. (2009). General k-opt submoves for the Lin–Kernighan TSP heuristic. Mathematical Programming Computation. 1. 119-163. 10.1007/s12532-009-0004-6.

[10] Johnson, D.S., L.A, McGeoch. 1997. The traveling salesman problem: A case study. E. Aarts, J.K. Lenstra, eds. *Local Search in Combinatorial Optimization.* John Wiley & Sons, Chichester, UK. 215-310.

[11] Lemire, D. 2012. Do not waste time with STL vectors [online]. Daniel Lemire's blog (viewed 31/08/2019). Available at https://lemire.me/blog/2012/06/20/do-not-waste-time-with-stl-vectors/.

[12] Lin, S., B. W. Kernighan. 1973. An effective heuristic algorithm for the travelling-salesman problem. Operations Research 21(1), 498-516.

[13] Little, J. et al. 1963. An algorithm for the traveling salesman problem. *Operations Research* 11, pp. 972-989.

[14] Odetayo, M.O.. 1994. "Optimal population size for genetic algorithms: an investigation," IEE Colloquium on Genetic Algorithms for Control Systems Engineering, London, UK. pp. 2/1-2/4.

[15] Paloetti, T. 2011. Leonard Euler's Solution to the Konigsberg Bridge Problem. Available from https://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem

[16] Potvin, J. Genetic algorithms for the traveling salesman problem. Annals of Operations Research 63(1996) 339-370.

[17] Reeves, C.R. Handbook of Metaheuristics. International Series in Operations Research & Management Science 146. Springer Science+Business Media, LLC 2010.

[18] Reinelt,G. 2013. TSPLIB [online]. University of Heidelberg (visited 31/08/2019). Available at https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

[19] Robinson, J. 1949. On the Hamiltonian game (a traveling salesman problem). RAND Research Memorandum RM-303. RAND Corporation, Santa Monica, California, USA.

[20] Sastry K., Goldberg D., Kendall G. (2005) Genetic Algorithms. In: Burke E.K., Kendall G. (eds) Search Methodologies. Springer, Boston, MA.

[21] Tsutsui, S., Collet, P. (2013). Massively Parallel Evolutionary Computation on GPGPUs. Springer Science & Business Media. ISBN 978-3-642-37959-8.

[22] Vrajitoru,D. 2008. C463 / B551 Artificial Intelligence [online]. Indiana University of South Bend (visited on 31/08/2019). Available at https://www.cs.iusb.edu/ danav/teach/c463/6_local_search.html

[23] Woeginger, G.J. 2003. Exact algorithms for NP-hard problems: A survey. M.Junger, G. Reinelt, G. Rinadli, eds. *Combinatorial Optimization-Eureka, You Shrink!* Lecture motes in Computer Sciece **2570**, pp. 185-207. Springer, Heidelberg.

[24] Yang, J. & S., Xiaohu & Marchese, Maurizio & Liang, Y. (2008). Ant colony optimization method for generalized TSP problem. Progress in Natural Science - PROG NAT SCI. 18. 10.1016/j.pnsc.2008.03.028.

A Best found solutions

Presented are the best solutions for each problem instance. Best solution is that that has the lowest cost. If two costs are the same, the one with lowest runtime will be chosen

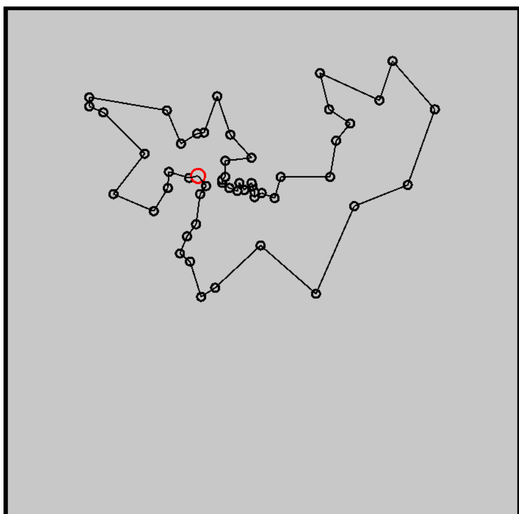

Figure 8: Berlin52. Cost:7544. Roulette-EX-3opt

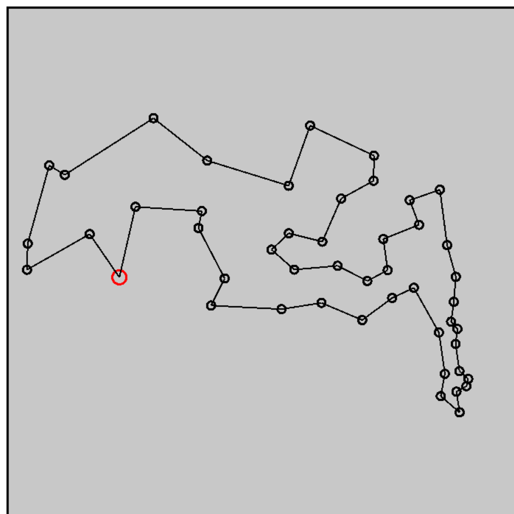

Figure 9: Att48. Cost:33523. Ranked-EX-2opt

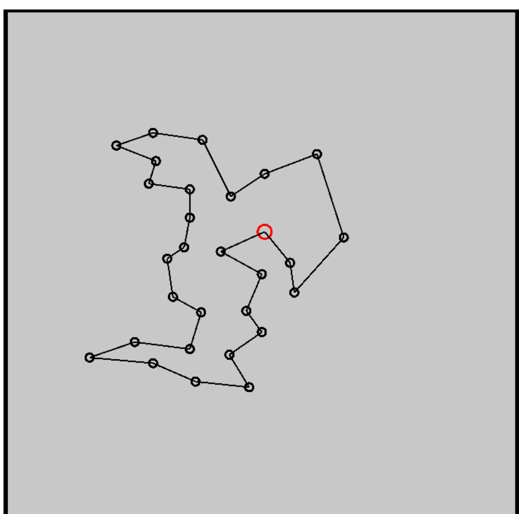

Figure 10: Bays29. Cost:9074. Roulette-OX-2opt

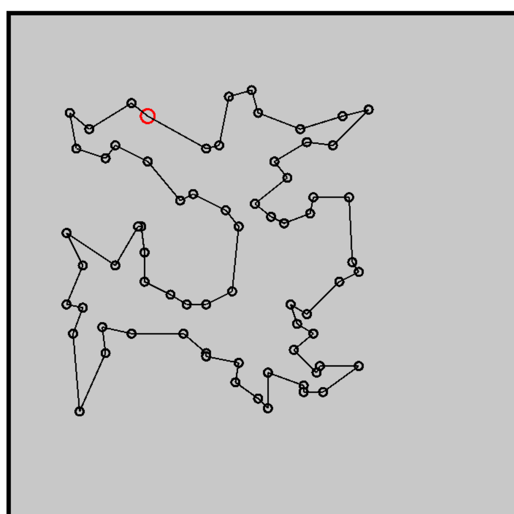

Figure 11: St70. Cost:677. Roulette-OX-2opt

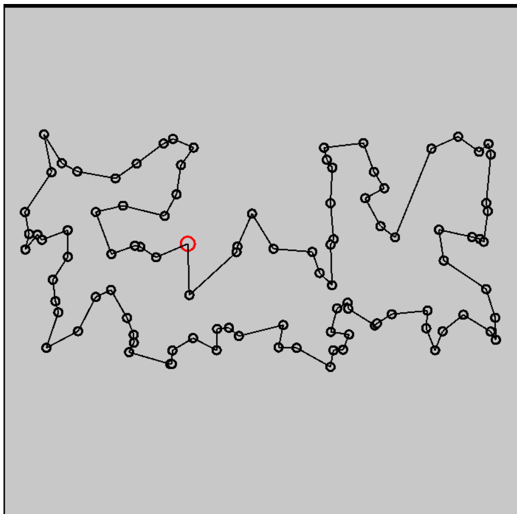

Figure 12: KroA100. Cost:21285. Roulette-EX-3opt

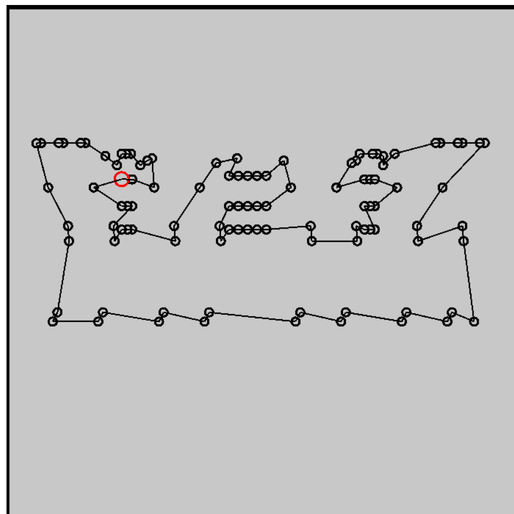

Figure 13: Lin105. Cost:14406. Roulette-OX-2opt

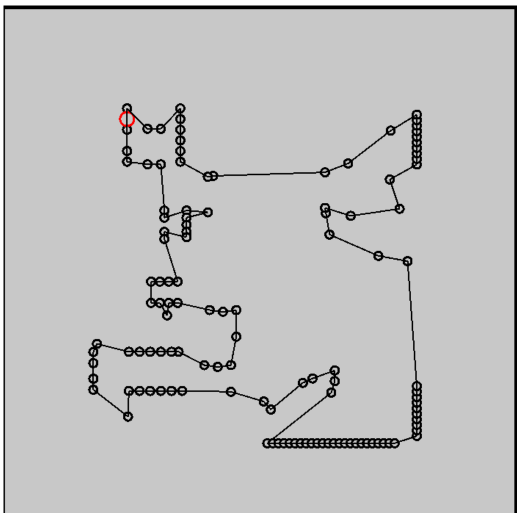

Figure 14: Pr124. Cost:59030. Roulette-OX-2opt

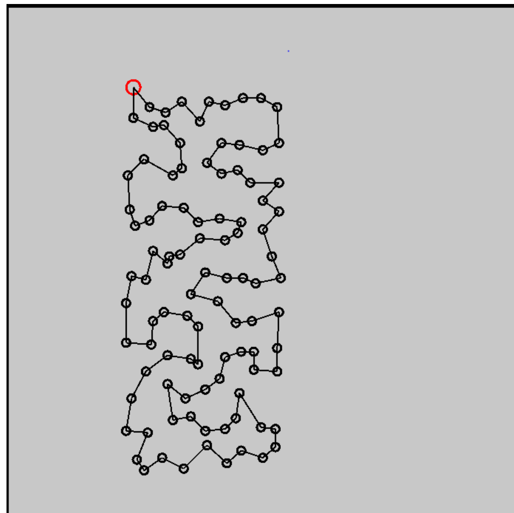

Figure 15: Rat99. Cost:1223. Roulette-EX-3opt

B Simulation data

Instance	Selection	CO	Mutate	Start	Gen	N	AVG best(N)	Runtime	Best	
berlin52	Roulette	OX	2-opt	70	60	10	7550	4.57	7544	
berlin52	Roulette	EX	3-opt	70	60	10	7700	4.34	7544	
berlin52	Ranked	OX	3-opt	70	60	10	7565	4.64	7544	
att48	Ranked	EX	2-opt	70	60	10	33719	7.39	33523	
att48	Ranked	OX	2-opt	70	60	10	33616	3.36	33555	
att48	Roulette	OX	3-opt	70	60	10	33651	2.8	33555	
att48	Roulette	EX	3-opt	70	60	10	33769	3.03	33600	
bays29	Ranked	EX	3-opt	70	40	10	9562	0.27	9219	
bays29	Ranked	OX	3-opt	70	40	10	9423	0.22	9151	
bays29	Roulette	EX	2-opt	70	40	10	9089	0.53	9074	
bays29	Roulette	OX	2-opt	70	40	10	9078	0.39	9074	
pr76	Ranked	OX	2-opt	70	70	8	108825	16.4	108159	
pr76	Ranked	EX	3-opt	70	70	8	113227	12.82	113086	
pr76	Roulette	OX	2-opt	70	70	8	108440	19.7	108159	
pr76	Roulette	EX	3-opt	70	70	8	108803	17.2	108347	
kroA100	Ranked	OX	3-opt	70	70	5	22391	24.23	21970	
kroA100	Ranked	OX	2-opt	70	70	5	21398	42.66	21285	
kroA100	Roulette	OX	2-opt	70	70	5	21494	49.59	21356	
kroA100	Roulette	EX	3-opt	70	70	5	21436	35.18	21285	
lin105	Ranked	OX	3-opt	70	70	5	14629	74.53	14518	
lin105	Ranked	EX	2-opt	70	70	5	14458	76.44	14420	
lin105	Roulette	OX	2-opt	70	70	5	14506	59.66	14406	
lin105	Roulette	EX	3-opt	70	70	5	14442	64.14	14428	
pr124	Ranked	EX	3-opt	70	70	5	61199	44.18	61992	
pr124	Ranked	EX	2-opt	70	70	5	59616	102.01	59519	
pr124	Roulette	OX	2-opt	70	70	5	59527	124.25	59030	
pr124	Roulette	OX	3-opt	70	70	5	59302	82.98	59074	
rat99	Ranked	EX	3-opt	70	70	6	1273	36.59	1259	
rat99	Ranked	OX	2-opt	70	70	6	1255	40.46	1234	
rat99	Roulette	OX	2-opt	70	70	6	1237	49.34	1234	
rat99	Roulette	EX	3-opt	70	70	6	1240	49.56	1223	
st70	Ranked	OX	3-opt	70	60	8	704	15.39	685	
st70	Ranked	EX	2-opt	70	60	8	690	31.26304	679	
st70	Roulette	OX	2-opt	70	60	8	678	14.66	677	
st70	Roulette	EX	3-opt	70	60	8	684	19.7072	682	

C Using the simulation software

C.1 Getting Started

Simply clone the repository at https://github.com/hakosaj/KandiTSP to a directory, compile and we are good to go!

C.2 Prerequisites

- SFML, at least version 2.5.1.

C.3 Using the software

Get the software from Github and simply make in the folder.

Then just run the file, the software tells you what to do! If using Unix systems, simply type 'make'.

In order to use further instances, those can be downloaded here for example: https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

Just save the instances in the Data-folder and voilá.

Make sure to format them to csv, same format as the other files!