Eötvös Loránd University
Department of Mathematics

Bence Keresztury

Genetic algorithms and the Traveling Salesman Problem

BSc thesis

Supervisor : Tamás Király

ELTE, Department of Operations Research

2017., Budapest

Abstract

My thesis is about the nature of evolutionary algorithms and how they can be applied to a well-known optimization problem. A brief introduction about the elements of evolutionary algorithm is given. I further investigated the Traveling Salesman Problem (TSP), a popular problem in combinatorial optimization. Some of the widely used methods for solving TSP including tour construction and tour improvement heuristics are presented. I also discuss one possible generalization of the original problem (GTSP) and the solutions proposed. In the second part of the thesis I present how the TSP and GTSP can be solved using genetic algorithms. In the final part, genetic algorithmic solution is compared to a traditional tour construction heuristic on several instances. Limitations and possible future improvements are discussed.

Contents

Abstract ii

Acknowledgements vi

1 Evolutionary algorithms 1

1.1 Representation of individuals 2
1.2 Evaluation function 2
1.3 Population 3
1.4 Parent selection 3
1.5 Recombination 5
1.6 Mutation 5
1.7 Survivor selection 6
1.8 Generating the EA cycle 6
1.9 The Genetic Algorithm 7

2 Traveling salesman problem 9

2.1 Applications of the symmetric TSP 10
2.2 Methods to solve the traveling salesman problem 11
2.3 The generalized TSP 18
2.4 Applications of the symmetric GTSP 18
2.5 Methods to solve the GTSP 19

3 Genetic algorithm for the traveling salesman problem 21

3.1 Representation of tours 21
3.2 Crossover 24
3.3 Mutation 31
3.4 Setting parameters 33
3.5 GA for the GTSP 33

4 Experimental results 35

4.1 Performance on traditional TSPs 36

4.2 Performance on 50-25 gTSPs . . . 36
4.3 Performance on 100-20 gTSPs . . . 36
4.4 Limitations and future improvement . . . 37

Biblography **39**

Attachment **42**

List of Figures

1.1 Demonstration of Roulette Selection . . . 4
1.2 General scheme of the EA cycle. Source: [11] . . . 6

2.1 Pseudocode for Double Minimum Spanning Tree heuristic . . . 13
2.2 The graph consisting of edges of T and M. Source: [21] . . . 14
2.3 Pseudocode for Christofides heuristic . . . 14
2.4 Illustration of Node Insertion. Source: [21] . . . 15
2.5 Node Insertion algorithm . . . 16
2.6 Illustration of Edge Insertion algorithm. Source: [21] . . . 16
2.7 Edge Insertion algorithm . . . 17
2.8 Illustration of 2-opt exchange algorithm. Source: [21] . . . 17
2.9 2-opt algorithm . . . 18
2.10 The generalized 2-opt exchange. Source: [12] . . . 20

3.1 Binary representation of cities in the TSP . . . 22
3.2 PMX step 1. . . . 25
3.3 PMX step 4. . . . 25
3.4 PMX step 6. . . . 26
3.5 Edge table . . . 30

4.1 An example run . . . 35
4.2 35 city TSP . . . 36
4.3 50 city, 25 states gTSP . . . 37
4.4 100 city, 20 states gTSP . . . 37

Acknowledgements

I want to thank my thesis supervisor Tamás Király who helped me a lot with his feedbacks both on math and English grammar. I also thank Dorka Keresztury for the long philosophical talks on the topic of artificial intelligence and many more. The inspiration she gave me helped me tremendously to develop interest in related fields. Huge appreciation for Fanni Dandé and Zsombor Sárosdi who fought on my side from exam to exam through these three years. Without Fanni's lecture notes and Zsombi's insights I would have never made it to the final exam. Last but not least, I want to thank Renáta Fáy who challenged me to finish my thesis in a very short time. The vision of winning the challenge helped me to stay focused and motivated through the whole process.

Chapter 1

Evolutionary algorithms

Sources: [5], [11]

Evolutionary computation (EC) is a subfield of artificial intelligence. It serves as an umbrella term for different iterative methods which are based on adopting Darwinian principles. EC can be classified as a trial and error (generate and test) problem solver, a generic global optimization method with a metaheuristic or stochastic nature. These methods store a multi-set of possible solutions (called: a population) instead of just one element in the search space. Typically, EC methods are applied when the size of the search space is too big for using exact algorithms. EC can be divided into three different categories: methods inspired by biological evolution (evolutionary algorithms), methods inspired by some biological behavior and methods using mathematical models.

Evolutionary algorithm (EA) is one of the EC methods. EA uses mechanisms that are prevalent in biological evolution, such as selection, reproduction, recombination and mutation. It is a very diverse field consisting of many processes originated at different times and places: genetic algorithms, evolutionary strategies, evolutionary programming and genetic programming. All these techniques are based on the same underlying principle. We take a population of randomly generated elements of the search space (individuals), then environmental pressure through natural selection is introduced (survival of the fittest). We can apply the given objective function as an abstract fitness measure to be maximized. Candidates with higher fitness function have better chance to become parents and seed the next generation. These good individuals pass their genes to the next generation, thus the average fitness in the population will rise.

To solve a problem with EA we need to address three different issues:

1. how to represent individuals
2. how to create and apply the variation operations (recombination and mutation)
3. what kind selection and replacement should be used

1.1 Representation of individuals

An individual of an Evolutionary Algorithm is an element of the search space. The search space consists of all solutions of the given problem. Representation of an individual should bridge the gap between the original problem context and the problem-solving space of the EA, thus linking the two worlds together. In other words, it means we must convert possible solutions to a form that can be manipulated by a computer. In the EA literature solutions in the original problem context are often called phenotypes while their encoding in the EA context are referred to as genotypes. The whole evolutionary search takes place in the genotype space, which can be very different from phenotype space (space of all possible candidate solutions). The solution to the original problem is obtained after decoding the best genotype. Therefore, all the feasible solutions are expected to be represented in the genotype space. Potential representations are integer or real numbers/vectors, permutations, function parameters or data structures (e.g. tree). The form of representation usually depends on the problem: the less complex the search space in which the EA operates, the better the representation. The three most commonly used form of representation are real (or integer) vector, binary vector and permutation.

Real or integer numbers can be assigned to each attribute and represent the individuals as vectors: I=(x_1,x_2,...,x_n), where x_i is the variable for the i. attribute. Typically, numbers from only a given interval can be assigned to the variables, thus the EA has to search in this bounded and closed n-dimensional space.

Binary vector representation of an individual is similar to the integer representation. It is often used in genetic algorithms as a genotype. In this case, the search takes places in an n-dimensional hypercube (where n is the length of the string that represents the individual). We search for the vertex with the highest fitness value.

Solutions of combinatorial optimization problems like the traveling salesman problem, timing problems or the quadratic assignment problem can be represented as permutations. These problems are about ordering a given number of objects in a way to find the best order for a given objective function. Individuals can be seen as I=(π_1,π_2,...,π_n), which denotes the permutation of the first n elements where the i-th position is occupied by the element π_i.

1.2 Evaluation function

The evaluation function is prevalently called fitness function in EA terminology. It maps from the search space to the real numbers and assigns the 'goodness' as a solution to each individual, in other words it measures the quality of the genotypes. It imposes requirements to which the population need to adapt and defines what improvement means. It forms the basis for selection and therefore facilitates improvement. Ultimately, the goal of every EA is to maximize the fitness function, i.e. find the global maximum of the search space. There are many problems where the objective function is given. Some of them can easily be converted into fitness function, however, more complex problems sometimes require the creation of a fitness function which is made of different fitness functions. These should facilitate

the attainment of partial goals.

1.3 Population

The population is a multiset of possible solutions, i.e. a set of genotypes in which even multiple copies of an element are allowed. During the evolutionary process the individuals are never changing or adapting, while the population does, thus the population forms the unit of evolution. Given a specific representation, defining a population is about setting a size of it, which remains constant in most of the EA applications. However, in some sophisticated EA algorithms an additional spacial structure with a distance measure or neighborhood relation has to be defined. Selection operators (parent and survivor selection) get a given population as argument, and return with a multiset of individuals. An important characteristic of a given population is the diversity measure which indicates the variability of solutions. Too little diversity can cause the process to get stuck at a local optimum, while too much diversity can lead to a non, or slowly converging algorithm. There exists no single diversity measure, typically the number of different fitness values, phenotypes or genotypes can be referred to as the diversity of the population, nevertheless more sophisticated indicators, such as entropy, are also widely used.

1.4 Parent selection

Similarly to biological evolution, parent selection or mating selection in an EA should be responsible for the improvement of the population average. This quality improvement is typically measured by the fitness function. The basis of the selection is the observation that the fitness value of the parents correlates with their children's fitness. Therefore, this type of selection allows better individuals to be the parents of the next generation. This is usually a stochastic process, where good solutions have a greater chance of seeding the following generations, thus passing on their genes and characteristics. Nevertheless, low quality solutions also get a small chance to become parents in order to sustain the desirable diversity of the population. I introduce some of the most commonly used parent selection operators: the Roulette Selection, the Stochastic Universal Sampling, the Tournament Selection and the Truncation Selection.

1.4.1 Roulette Selection

Roulette selection is one of the first and best-known selection operators. It is a fitness proportional selection, which means that individuals' chance of becoming parents is proportional to the ratio of their fitnesses. It is a replacement sampling process, i.e. an individual can be selected multiple times. Let I_i be the i-th individual, and f is a nonnegative function that indicates the fitness of the individuals. The chance of the individual I_i to be selected as a parent is $\mathrm{p}(I_i) = \frac{f(I_i)}{\sum_{j=1}^{n} f(I_j)}$. The process of this selection can be illustrated with the help of a roulette wheel. Lets map every individual onto a roulette, where each individual I_i is represented by a space that proportionally corresponds to its fitness. A fix pointer

is located at the outside of the wheel. We spin the roulette wheel and we select the individual to be a parent on which the pointer points after the wheel stopped spinning (see Figure 1.1). When the wheel

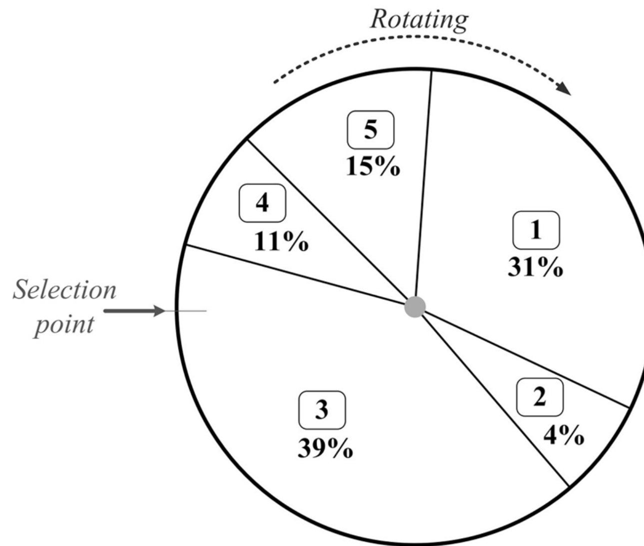


Figure 1.1: Demonstration of Roulette Selection

is repeatedly spun |P| times (where |P| is the number of individuals in the population), the expected number of copies of I_i will be |P|·p(I_i). The disadvantage of this selection is that if some of the fitness functions are nearly equal, the best solution does barely have more chance than the others, although this slight differences in fitness can be very important in finding the global optimum. To eliminate this problem, scaling and fitness-transforming methods were suggested, but other forms of selection are most commonly used.

1.4.2 Stochastic Universal Sampling

This type of selection is also a fitness proportional sampling method. However, the Stochastic Universal Sampling (SUS) tries to minimize the number of duplicates among the parents. Unlike roulette selection, SUS assigns an arc length to every individual proportional to their expected number of copies:

$$\mathrm{E}(I_i) = |\mathrm{P}| \cdot \mathrm{p}(I_i).$$

|P| pointers are uniformly distributed along the arc. We select |P| parents simultaneously by randomly rotating the roulette and choosing the individuals whose circle segments are in front of the fixed pointers.

1.4.3 Tournament Selection

Contrary to fitness proportional selection, Tournament Selection only uses the order of fitness values to select the parents of the next generation. This helps to maintain variance in the population by limiting the number of copies among the parents. This method selects parents in two steps. Firstly,

Tournament Selection chooses *tour* number of individuals with equal probability, where *tour* is a parameter. Secondly, out of these chosen individuals, the one with the highest rank will be selected to be a parent. This process is repeated |P| times.

1.4.4 Truncation Selection

The idea behind Truncation Selection is to eliminate the worst individuals from the gene pool. This method first orders individuals based on their fitness value, then selects the best T of them, where T is a parameter. From these selected individuals everyone has an equal possibility to become a parent. Unlike Tournament Selection which mimics biological selection, this is an artificial selection method.

1.5 Recombination

Crossover or recombination is the process where new solutions (offspring) are created from two (or sometimes more) parent solutions. It is a variation operator that enables the EA to search in the search space traced out by the parents, therefore the heritage of their attributes is warrantied. The main idea behind recombination is that by crossing two solutions with different but desirable features, an offspring can be created which combines advantages from both parents. Similarly to mutation, recombination typically contains stochastic elements, either the choice of what parts of the parents are crossed, or the method these parts are combined are random. There are many forms of recombination dependent on the representation of the individuals and the specific problem. Recombination is the main search process in most of the EAs.

1.6 Mutation

Variation operators (mutation and crossover) create new individuals from old ones, thus helping to sustain the diversity of the population. Mutation is a variation operators which gets an individual as input and creates a new one (offspring/child) from it. The new individual will only be slightly changed. Drastic changes would erase the good properties of the selected solution. It is worth noting that mutation is convenient for the fine-tuning of a search. while recombination searches in a bigger space (in the hypercube generated by their parents) and it is useful for finding local and global optima, it is not suitable for precise approximations. Mutation however, is competent for searching the neighboring environment of the offspring, thus finding the best possible solution. The efficacy of the search is dependent on more variables. The neighborhood of an individual consists of all the solutions reachable within a single move (crossover or mutation). In easier problems, changing a couple of values at random could be sufficient to sustain the desired variability, however in more complex problems with multiple local maxima, changes in the definition of neighborhood could be necessary. That means, at the beginning of a search a broader neighborhood should be defined, which helps to explore the search space quickly.

However, at the end of the search a tighter, smaller neighborhood is more useful in finding a more precise answer, while also facilitating convergence.

1.7 Survivor selection

After obtaining new individuals from the parents, the formation of a new generation is needed. In order to refresh the old population, some individuals of the old generation are replaced with offspring. This procedure is called survivor selection, replacement or reinsertion. This process is characterized by the ratio of new offspring in comparison to the population size (generation gap), and the reinsertion rate, which indicates the quotient of old individuals replaced by new ones. In case both rates are equal to 1, then the entire new generation is formed from offspring. That means all solutions live only one generation and there is no chance of survival. If the generation gap $\leq$ reinsertion rate, than all the offspring are reinserted into the population, although some selection mechanism is required to determine which individuals are to be replaced from the previous generation. Contrary to the parent selection it is often a deterministic process where the solutions with the lowest fitness rates are replaced while the other survive. This selection process is called elitism, because it enables the best solutions to survive more generations. Nevertheless, if generation gap $>$ reinsertion rate, than only a part of the newborn solution can be reinserted into the new population. Therefore, some "spartan" selection mechanism is required to distinguish between viable and nonviable offspring by eliminating the latter ones.

1.8 Generating the EA cycle

An EA solves a specific problem with the help of numerous populations generated after each other. The general scheme of an EA can be seen in the figure 1.2. A population at a given time is called a

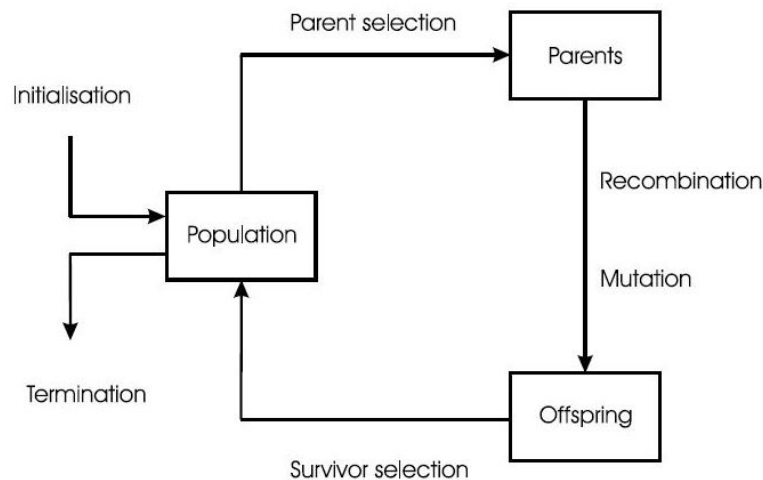


Figure 1.2: General scheme of the EA cycle. Source: [11]

generation. In order to get the EA running, the development of an initial population is needed. Usually a set of random solutions is generated, although more sophisticated methods with some heuristic to the specific problem can also be applied. In order to refine solutions, individuals as a result from a former EA, or based on experts opinion can also form the initial population. After the initialization, the strategic parameters must be specified. Strategic parameters are the specific values which define the structure of the algorithm, function of the above mentioned processes (selection, recombination, mutation, etc.) and the rate of the change in the population. These parameters are usually:

- size of the population
- p_r: the probability of recombination
- p_m: the probability of mutation
- generation gap
- reinsertion rate

These strategic parameters are very problem-specific. The success or failure of the EA method often depends on the fine-tuning of parameters. Due to the stochastic nature of the EA, it is hard to tell the moment when the EA reaches the desirable results. For this reason many termination conditions have been formulated. In case the exact optimum is known, the EA should be stopped if this (or a solution with a given $\varepsilon > 0$ precision) is reached. If such an optimal fitness value is unknown, a maximal number of generations or maximal running time can be defined as stopping condition. Another widely used condition for termination is when the best solutions are the same over a given number of generations. That means the search operators cannot create better solutions. That could indicate that the global optimum is reached, but likewise it could mean to be stuck at a local extremum. In real life applications there are many problems with a given criterion for goodness. In this case we can stop the search if the criterion is met. The last generally used termination condition is when the population becomes too homogenous, i.e. the individuals are very similar to each other. The problem with this is that the recombination operator becomes ineffective and cannot sustain the diversity of the population, therefore the search can be stopped.

1.9 The Genetic Algorithm

Genetic algorithms are one of the best-known and most widely used evolutionary algorithms. It is commonly used to generate high-quality solutions to optimization and search problems. This technique was introduced by Holland in 1975 [18]. Since then, many variations of the original algorithm were suggested. In my thesis, I will refer to Holland's original algorithm as the canonical genetic algorithm. In the canonical GA, individuals are represented with binary strings of fixed length. Fitness is defined by $f_i/\bar{f}$, where f_i is the fitness associated with string i and $\bar{f}$ is the average fitness of the population. Holland used roulette selection to choose the parents of the next generation. In this algorithm, crossover

is the main search operator, while mutation is used to map the nearby environment. Typically the chance of recombination is very high ($p_r \approx 0.9$) and the rate of mutation is low ($p_m \approx 0.01$).The canonical GA uses 1-pont crossover, that means Offspring 1 gets the first part of his string from Parent 1, and the second part from Parent 2. (Offspring 2 is generated in an analogous way, with the parents' role reversed). The cut-point between the two parts is chosen randomly. Holland used simple bit mutation. That means for each bit, there is a low probability of changing its value. Usually |P| number of parents are selected, which generate |P| offspring. These children solutions immediately replace the previous generation entirely, which means that each solution can only live for one generation. Over the years, empirical results have shown many flaws in the canonical GA. For a faster convergence, elitism was introduced. Also other selection mechanisms were suggested (see Section 1.4) which provided better results. Experience showed that many problems can be better solved using non-binary representation (for example see Chapter 2). Finally the problem of how to choose a fixed mutation and recombination rate was largely solved with adaptive GA-s, where the parameters are encoded as extra genes and allowed to evolve themselves (see Section 3.4). However, the canonical genetic algorithm is still widely used in some straightforward problems in which binary representation is suitable. Moreover, it was a unified base for theoretical results for decades, thus provided much insight into the behavior of evolutionary processes in combinatorial search spaces.

Chapter 2

Traveling salesman problem

The traveling salesman problem (TSP) is one of the best-known problems in combinatorial optimization. The problem was defined in the 1800s by W. R. Hamilton and T. Kirkman. The problem can be illustrated with a salesman who wants to travel to every city in the country to sell his product. He has a map, where all the distances between the cities are listed. What is the shortest possible route where he visits every city exactly once and ends up in his hometown? The constraint that the salesperson has to start from his hometown can be ignored, since the cheapest tour will be identical, no matter where he started the journey. Mathematically, this problem can easily be converted to a graph-theoretical problem.

1 Definition. The (symmetric) traveling salesman problem: Given a $G = (V, E)$ complete undirected graph with n nodes and the distance (/cost) matrix between each of them, what is the shortest (/cheapest) Hamiltonian cycle in G?

A Hamiltonian cycle is a closed walk through the graph that visits each node exactly once. In other words the problem can be stated as following: we are given the set of n cities $c_1, c_2, ..., c_n$ and for each par c_i, c_j, $i \neq j$ a distance $\mathrm{d}(c_i, c_j)$. Our goal is to find an ordering π of the cities that minimizes the quantity

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}).$$

This quantity is referred to as tour length. In my thesis, I concentrated on the metric TSP, in which the d distances satisfy the following three conditions:

1. $\mathrm{d}(c_i, c_j) \geq 0$ for $1 \leq i, j \leq \mathrm{n}$ and and $\mathrm{d}(c_i, c_j) = 0$ if and only if $i = j$.

2. $\mathrm{d}(c_i, c_j) = \mathrm{d}(c_j, c_i)$ for $1 \leq i, j \leq \mathrm{n}$.

3. $\mathrm{d}(c_i, c_j) \leq \mathrm{d}(c_i, c_k) + \mathrm{d}(c_k, c_j)$ for $1 \leq i, j, k \leq \mathrm{n}$.

The TSP minimization problem can be stated as a decision problem, that asks the following yes or no question: Given a number t, a set of cities, and distance between all city pairs, is there a tour visiting each city exactly once of length less than t?

2 Definition. TSPDECISION: Consider the set $\{(G, d, t) : G = (V, E)$ a complete graph, d is a function d: $V \times V \to \mathbb{Z}$, t $\in \mathbb{Z}\}$. Is there any Hamiltonian cycle in G with cost that does not exceed t?

3 Theorem. *The decision form of the traveling salesman problem is NP-complete.*

Proof. First, we prove that the TSPDECISION $\in NP$. If we want to check the tour four credibility, we have check that the tour contains each vertex exactly once and that the total sum of the tour does not exceed t. Both can easily be done in polynomial time.
Secondly we prove that TSPDECISION is NP-hard. We can prove this by showing that the Hamiltonian cycle problem can be reduced to the TSPDECISION problem in polynomial time. Since we know that the Hamiltonian cycle problem is NP-complete, the TSPDECISION must be NP-complete, too. (Otherwise we could find a Hamiltonian path in polynomial time with the help of the TSPDECISION). Assume $G = (V, E)$ is an instance of Hamiltonian cycle, where $|V|{=}n$. We construct an instance of TSPDECISION. We define a complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V and i \neq j\}$. Thus, the cost function is defined as:

$$d(i, j) = \begin{cases} 1, & \text{if (i,j)} \in \text{E.} \\ 2, & \text{if (i,j)} \notin \text{E.} \end{cases} \tag{2.1}$$

We prove that G has a Hamiltonian cycle if and only if G' has a tour of cost at most n. Suppose that a Hamiltonian cycle H exists in G. The cost of each edge in H is 1 in G', since each edge belongs to E. Therefore, H has a total cost of n in G'. Thus, if a graph G has a Hamiltonian cycle then graph G' has a tour of n cost.
Conversely if G' has a tour H' with a total cost of n, then each edge in the tour must have the cost of 1, because the cost of edges in E are 1 or 2 by definition and a tour contains n edges. Since all edges have 1 cost in H', therefore it contains only edges in E. Thus, the traveling salesman tour in G' is a Hamiltonian cycle in G as well, therefore TSPDECISION is NP-Complete.

□

2.1 Applications of the symmetric TSP

The popularity of the traveling salesman problem is partly due to its wide variety of applications. Apart from the obvious applications in optimizing traffic and transport routes, the symmetric TSP has been applied to many different areas like VLSI chip fabrication [22], X-ray crystallography [4], overhauling gas turbine engines [32], clustering of data arrays [26], chronological seriation in archeology [21] and vehicle routing [26].

2.2 Methods to solve the traveling salesman problem

Source: [20], [21]

The main difficulty of the traveling salesman problem is the immense number of possible solutions. In a map with n cities, there are $\frac{(n-1)!}{2}$ different possible routes. Since the TSP minimization problem is NP-hard too, (assuming the widely-believed conjecture $P \neq NP$) any algorithm finding optimal tours must have a worst-case running time that grows faster than any polynomial. Because of the above-mentioned problems, instead of looking for an exact solution in polynomial time, researchers concentrated on two different alternatives. One approach was the attempt to develop optimization algorithm that work well on special real world instances, but ignore the worst-case scenarios. By contrast, the other approach was to look for heuristics that merely find near-optimal solutions, but do so quickly. In my thesis, I will concentrate on the latter ones.

2.2.1 Tour construction heuristics

Nearest Neighbor

Perhaps the most natural heuristic for the TSP is the well-known Nearest Neighbor algorithm. The salesman starts off his journey from a random city and always travels to the nearest, yet unvisited city. Mathematically, we construct an ordering $c_{\pi(1)}$, ..., $c_{\pi(n)}$, so that $c_{\pi(1)}$ is chosen arbitrarily, and $c_{\pi(i+1)}$ is chosen to be the city c_j that minimizes the $\{$d$(c_{\pi(i)}$,$c_j)$: π(k) $\neq j$, 1$\leq k \leq i\}$ distance. The salesman traverses the cities in the constructed order, finally returning from the last city ($c_{\pi(n)}$) to where he started ($c_{\pi(1)}$). The running time of the Nearest Neighbor algorithm is $\Theta(n^2)$, since we have to look at all the vertices and compute its distance from every other vertex, to choose the minimum. So, it is a relatively fast algorithm, however it is clear that it will not provide an optimal solution in most cases. Finding the nearest neighbors may "leave out" relatively close vertices, which than must be visited at the end. This can lead to long distances between remaining cities by the end of the tour. All we can guarantee is that in metric instances the ratio between the distance found by the Nearest Neighbor heuristics, denoted by NN(I) and the optimal tour OPT(I) is $\frac{NN(I)}{OPT(I)} \leq \frac{1}{2}$ ($\lceil log_2(n) \rceil$+1). [20]

Greedy algorithm

This tour construction heuristic is similar to the Nearest Neighbor algorithm in that it also builds up the Hamiltonian cycle one edge at a time (note that a Hamiltonian cycle is a graph containing all the vertices with a degree of two). However, this algorithm starts by sorting the edges by length, and always adding the shortest remaining available edge to the tour. An edge is available if it is not yet in the tour and if adding it would not create a 3-degree vertex or a cycle with edges less than n. This heuristics can be implemented to run in $\Theta(n^2 log(n))$ time, therefore it is somewhat slower than the nearest neighbor algorithm, however, its worst-case tour quality is somewhat better. [20]

Nearest Insertion

This algorithm belongs to the family of insertion heuristics. These methods build a tour by starting with a trivial tour of one city (c_i) and then selecting cities in the tour one-by-one based on some criteria until every city is visited. Nearest Insertion heuristic selects the node from the not yet visited cities which is the nearest to any city in the tour. That means a city c_j will be inserted if it minimizes the distance d(c_i,c_j) where city c_i is in the partial tour and c_j not yet in the tour. Finally, c_j will be visited either before or after c_i dependent on what is cheaper. This algorithm also runs in $\mathcal{O}(n^2)$ time, and guarantees the tour constructed not be longer than two times the optimal tour [34].

Cheapest Insertion

This algorithm is also of greedy nature. It proceeds by searching the node among all nodes not yet inserted which can be inserted with the lowest increase in the length of the tour. It starts with finding the city c_j which minimizes the distance d(c_i,c_j), $i \neq j$ and building the partial tour (c_i,c_j). A general step is to find the cities c_i, c_j and c_k, where c_i and c_j being at the two ends of an edge belonging to the partial tour and c_k not yet belonging to the partial for which d(c_i,c_k)+d(c_k,c_j)-d(c_i,c_j) is minimal. Insert this city c_k between c_i and c_j in the partial tour. We repeat this process until every node is visited. This algorithm runs in $\mathcal{O}(n^3)$ time (however, with careful programming it can be improved to $\mathcal{O}(n^2 log(n))$). This heuristic also guarantees the tour constructed not be longer than two times the optimal tour [34].

Random Insertion

Random Insertion proceeds by selecting a random, not yet visited city c_k in the map and insert it into the tour in the cheapest possible way (d(c_i,c_k)+d(c_k,c_j)-d(c_i,c_j) is minimal, where c_i and c_j are already in the partial tour). Although random insertion can only guarantee the found route to be shorter than log_2(n)+1 times the optimal solution, in practice, however, random insertion performs better than the previous two insertion heuristics [34].

Farthest Insertion

Farthest insertion is a very useful tour construction heuristic with good qualities. It runs in $\mathcal{O}(n^2)$ time and provides good results in practice, too. However, its worst case tour length is unknown [34]. It proceeds by inserting the city c_k in the tour if it is the farthest node, from the partial tour. That means c_j is inserted when its distance to any tour node is maximal. The idea behind this strategy is to fix the overall layout of the tour early in the insertion process.

Christofides algorithm

We can as the question whether it is possible to provide a polynomial algorithm that has a worst-case ratio better than two. In 1976, Christophides constructed an algorithm [6], that guarantees the ratio

to be 1,5. This is the best algorithm as far as performance guarantee is concerned. I firstly present a simple approximation algorithm and prove that it provides a tour with length at most two times the optimal solution, then show how Christofides improved this algorithm to provide an 1,5 approximation to the TSP.

The Double Minimum Spanning Tree heuristic starts by building a minimal-cost spanning tree T. Note that the cost of T is smaller than the optimal tour, because if any edge e from optimal tour is deleted we get a spanning tree and because of T's minimality and the nonnegativity of distances we get: d(OPT) $\geq$ d(OPT - e) $\geq$ d(T). Then we duplicate all the edges in T, thus every vertex will have an even degree. So, we can now easily construct an Euler cycle E. The Euler cycle will use every edge in the duplicated graph, therefore d(E) = 2*d(T) $\leq$ 2*d(OPT). Finally, we traverse the cycle but short-circuit past vertices that have been seen before. Starting at a random node, follow the Euler cycle, but mark vertices as already visited when we visit them. Later, when encountering a vertex already visited, we simply skip to the next unvisited node in the Euler cycle. If we keep going until we encounter the starting node again, we get a Hamiltonian tour H. Because of the triangle inequality the resulting length is no greater than the length of the Euler cycle, therefore: d(H) $\leq$ d(E) $\leq$ 2*d(OPT). In conclusion, we proved the following theorem:

4 Theorem. *The Double Minimum Spanning Tree heuristic is a two-approximation to the TSP.*

```
procedure doubletree
1. Build a minimum spanning tree from the set of all cities.
2. Duplicate all edges.
3. Construct an Euler cycle.
4.Traverse the cycle, but do not visit any node more than once, taking shortcuts when a node
  has been visited.
end doubletree
```

Figure 2.1: Pseudocode for Double Minimum Spanning Tree heuristic

The Christofides algorithm is similar to the Double Minimum Spanning Tree. Likewise, it starts with the construction of the minimum spanning tree T, but it constructs the Euler cycle in a cleverer way. For the minimum spanning tree T holds that d(T) $\leq$ d(OPT) (see above). Now, we add a minimum weight perfect matching M on the odd-degree vertices. Since the number of odd-degree vertices are even in every graph, this can be done. We claim that the total cost of the matching will be no greater than $\frac{1}{2}$d(OPT). Let $N*$ be an optimal TSP tour on just the odd-degree vertices. Let N_1 and N_2 be the two perfect matchings on the odd-degree nodes obtained by taking the edges of $N*$ alternatively. Then d(M) $\leq$ min {d(N_1), d(N_2)} $\leq$ d(N*)/2 $\leq$ d(OPT)/2. The first inequality is from the fact that N_1 and N_2 are perfect matchings on the odd-degree nodes and M is the minimum weight matching on

the same nodes. The second inequality is holds because the minimum of two numbers is at most the average. And finally, the last inequality is because if we get a TSP tour on the odd-degree vertices from OPT by skipping the even-degree vertices we get a cycle that costs at most d(OPT) due to the triangle inequality. And the resulting tour on the odd-degree vertices is not better than d($N*$), because $N*$ is optimal. We proved that d(M) *leq* $\frac{1}{2}$d(OPT). Finally, we take the graph with edges consisting of the edges T and M. This graph may contain double edges, since T and M may intersect (see Figure 2.2).

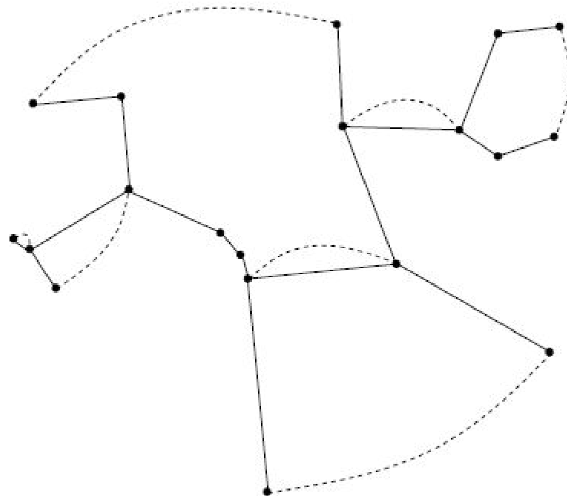


Figure 2.2: The graph consisting of edges of T and M. Source: [21]

Notice that all vertices have even degrees, because exactly one edge have been added to the odd-degree nodes of T. So we can construct an Eulerian cycle E that has the cost of d(T)+d(M). The last step is, as seen above, to produce a Hamiltonian cycle H by simply skipping the already visited edges and obtaining a cycle that is due to the triangular inequality no worse than the Eulerian cycle E. Therefore d(H) $\leq$ d(T)+d(M) $\leq$ d(OPT)+$\frac{1}{2}$d(OPT) = $\frac{3}{2}$d(OPT). Thus:

5 Theorem. *The Christofides algorithm is a 1,5-approximation to the TSP.*

---

procedure christofides

1. Build a minimal spanning tree from all the set of all cities.

2. Compute a minimum weight perfect matching on the odd-degree vertices of the tree and add it to the tree.

3. Construct an Euler cycle.

4.Traverse the cycle, but do not visit any node more than once, taking shortcuts when a node has been visited.

end christofides

Figure 2.3: Pseudocode for Christofides heuristic

The computation of the Christofides algorithm takes considerably more time than the previously mentioned construction heuristics. A minimum weight perfect matching can be computed in $\mathcal{O}(n^3)$ time [10]. Since a spanning tree may have $\mathcal{O}(n)$ odd-degree vertices, the Christofides heuristic runs in $\mathcal{O}(n^3)$ time. However, this algorithm is rarely implemented in practice, because insertion algorithms often perform better in real-world scenarios and are easier to implement.

2.2.2 Improving solutions

We saw that the quality obtained by tour construction can be moderate. Although these heuristics can be useful for some applications, they are not satisfactory in general. However, traveling salesman tours generated by the tour construction heuristics can be improved. In this section, we will have a look on local improvement heuristics for the TSP. They are all based on different but simple exchange operators (or moves) that converts one feasible TSP tour into another. It is a form of neighborhood search process where each tour has an associated neighborhood of adjacent tours, i.e. those who can be reached within a single move. The algorithm continually changes the current tour to an adjacent one that is better, until no such tour in the neighborhood exists (hill climbing algorithms). That means we reached a locally optimal solution. However, the running time of these algorithms cannot be polynomially bounded in the worst case since it is dependent on the size of reductions achieved by the tour modifications. Assuming integer input, the worst case scenario is that every improving move reduces the tour length by one unit and hence the running time depends on the initial and final tour length. In spite of these, these algorithms are often implemented because they usually provide very good results on "real-word" problems. The following heuristics differ merely in terms of the definition of adjacent tours.

Node Insertion

The Node Insertion removes one vertex from the existing tour and reinserting it at the best possible position (see Figure 2.4). To check whether an improvement by node insertion is possible takes $\mathcal{O}(n^2)$

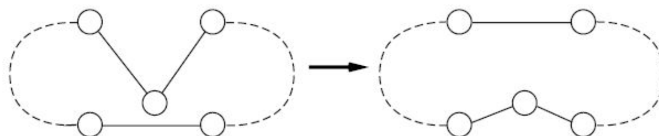


Figure 2.4: Illustration of Node Insertion. Source: [21]

time, since every possible insertion point for every possible vertex has to be examined. After finding an improving insertion move, the tour can be updated in $\mathcal{O}(1)$ time. The pseudocode for the heuristic based on Node Insertion can be implemented as seen in Figure 2.5:

nodeinsertion

Let T be the current tour.

while *not finished* **do**

For every node i=1,2..., n:

Examine all possibilities to insert i at a different position in the tour. If the tour length can be decreased, choose the best node insertion move and update T.

if *If no improving moves could be found* **then**

declare finished

end

end

end nodeinsertion

Figure 2.5: Node Insertion algorithm

Edge Insertion

A similar procedure can be defined, if we consider the edges instead of nodes. A single edge is removed from its place and it is reinserted at the best possible place (see Figure 2.6).

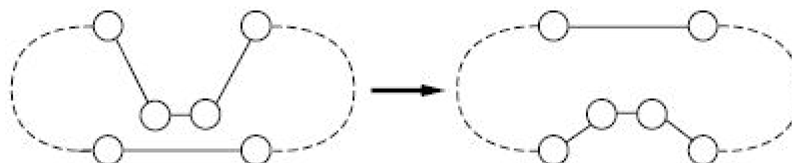


Figure 2.6: Illustration of Edge Insertion algorithm. Source: [21]

The corresponding algorithm can be be implemented in a similar way to Node Insertion (see Figure: 2.7)

Regarding the running time, the same remarks as for Node Insertion apply. That means, it also takes $\mathcal{O}(n^2)$ time to check whether an improving move exist in the current graph. Similarly, the worst case running time also depends on the measure of the improvements, therefore it depends on the initial and final tour length.

2-opt and 3-opt

2-opt is the most famous among the simple local search algorithms. It was first proposed in 1958 by Croes [7]. The motivation for this move is the following observation: considering the Euclidean case if a tour crosses itself, it can be improved by eliminating the crossing edges and reconnecting the paths in the way they do not cross (this can always be done). The new tour will be shorter than the old one because the sum of the two diameters of a convex quadrilateral is bigger than the sum of two opposite sides. A 2-opt exchange in general consists of deleting two random edges and reconnecting the paths

edgeinsertion

Let T be the current tour.

while *not finished* **do**

For every node i=1,2..., n:

Examine all possibilities to insert the edge between i and its successor at a different position in the tour. If it is possible to decrease the tour length this way choose the best such edge insertion move and update T

if *If no improving moves could be found* **then**

declare finished

end

end

end edgeinsertion

Figure 2.7: Edge Insertion algorithm

in a different way to obtain a new TSP tour. Note that there is only one other way of reconnecting paths to get a new tour. Even in the Euclidean case the eliminated edges do not have to necessarily cross in order obtain improvement (see Figure 2.8).

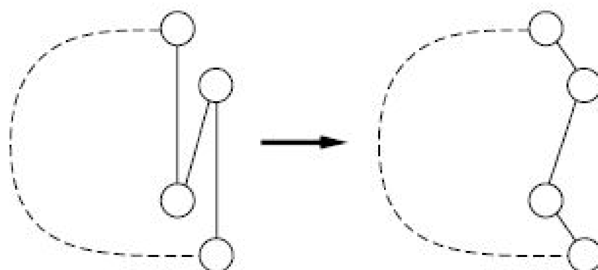


Figure 2.8: Illustration of 2-opt exchange algorithm. Source: [21]

The implementation of the 2-opt heuristics can be seen in Figure 2.9.

To check whether an improving move can be found in a given situation takes $\mathcal{O}(n^2)$ time, because we have to check all pairs of tour edges . The 3-opt algorithm is the natural extension of the 2-opt, where the exchange replaces up to three edges of the current tour. The number of combinations to eliminate the three edges are $\binom{n}{3}$ and eight ways are possible to reconnect the three paths to form a tour (assuming each of them contains at least one edge). Note that Node Insertion, Edge Insertion and 2-opt are special 3-opt operators. Node and Edge Insertion is obtained when one path in a 3-opt move consists of a single vertex or edge. A 2-opt exchange is a special 3-opt move, where one of the deleted edge is used for reconnecting two paths. Therefore, out of the eight 3-opt moves, there is only three real 3-opt exchanges that is not contained by Node/Edge Insertion or 2-opt. There are also k-opt methods for k>3 in which one move consists of replacing up to k edges but due to rapidly growing running time

2-opt

Let T be the current tour.

while *not finished* **do**

For every node i=1,2..., m:

Examine all 2-opt moves involving the edge between i and its successor in the tour. If it is possible to decrease the tour length this way, choose the best such 2-opt move and update T.

if *If no improving moves could be found* **then**

declare finished

end

end

end 2-opt

Figure 2.9: 2-opt algorithm

and only slight improvement in performance these heuristics are implemented more rarely.

2.3 The generalized TSP

Source: [36]

Since the 19th century, the traveling salesman problem has been generalized in multiple directions. One possible generalization of the original problem is the so called generalized Traveling Salesman Problem (GTSP, also known as set TSP, Multiple Choice TSP or Covering TSP). This problem describes the salesman who wants to travel to exactly one city in each state and finally returning to his starting position. Of course, he wants to minimize the total length of the trip. Similarly to the original problem, it also can be formulated as a graph-theoretical question.

6 Definition. We are given a complete undirected graph $G = (V, E)$, with n nodes and the distances (/cost) between each of them. Moreover, the node set c_1, c_2, ..., c_n is partitioned into m clusters C_1, C_2, ..., C_m (m≤n). What is the shortest (/cheapest) cycle that visits exactly one node from each cluster?

Note that in case $m = n$ we get back the original TSP. The decision form of the generalized traveling salesman problem, of which the TSPDECISION is a special case, is obviously also NP-complete.

2.4 Applications of the symmetric GTSP

Similarly to the original problem, its generalization proved to be useful in numerous real-world applications. For problems that are hierarchical by nature, the GTSP offers a more accurate model than the TSP. For example, if the traveling salesman wants to distribute his product between all his dealers in the country, he could meet all the local dealers in only one out of many possible cities in each state, so

he can minimize the costs of his trip. In "real-world", the GTSP was used in airplane routing [30], mail delivery [23], warehouse order picking [30], welfare agency routing [35], material flow system design [23], vehicle routing [23] and computer file sequencing [1].

2.5 Methods to solve the GTSP

In solving the set-TSP, the first approach is to solve the problem with adaptations of TSP algorithms. Simple tour construction heuristics are relatively easy to convert to the generalized version. Consider the Nearest Neighbor heuristic as an example. This algorithm can be modified to include the nearest city from a different, not yet visited cluster in every step. The process stops when all the produced tour covers all clusters. Similarly, insertion heuristics can also be adapted if we keep track of the already visited states. Conversely, improvement heuristics need more careful adaptation to the generalized problem. I will present an algorithm called RP1, that is based on 2-opt and 3-opt exchanges [12]. Let T be the current GTSP tour, visiting exactly one node from every cluster. Let $S \subseteq \{c_1, c_2, ..., c_n\}$ be the the set of visited cities. Although classical improvement algorithm can also provide an improved GTSP solution on the subgraph induced by S, it never changes the set of visited cities. In order to remove this restriction, Fischetti et al. proposed the following generalized 2-opt scheme: Let (..., C_α, C_β, ..., C_γ, C_δ,...) be the cluster sequence of the current tour T. In Figure 2.10 T is shown with continuous line. All the edges of T not incident with the nodes $C_\alpha \cup C_\beta \cup C_\gamma \cup C_\delta$ are fixed and drawn with heavy lines. In this example let us denote the distance between city c_i and city c_j with c_{ij}. We try to change the cluster sequence to (..., C_α, C_γ, ..., C_β, C_δ,...) in the most efficient way. To do that we determine two node pairs $(u*, w*)$ and $(v*, z*)$ such that
$c_{iu*} + c_{u*w*} + c_{w*h} = \min \{c_{ia} + c_{ab} + c_{bh} : a \in C_\alpha, b \in C_\gamma \}$ and
$c_{jv*} + c_{v*z*} + c_{z*k} = \min \{c_{ja} + c_{ab} + c_{bk} : a \in C_\beta, b \in C_\delta \}$, where nodes i, j, k (see Figure 2.10) are the nodes visited by T belonging to the clusters preceding C_α, following C_β, preceding C_γ and following C_δ, respectively. This process requires $|C_\alpha|\ |C_\gamma| + |C_\beta|\ |C_\delta|$ comparisons. On the whole, trying all possible pairs (C_α, C_γ) and (C_β,C_δ) runs in $\mathcal{O}(n^2)$ time, since every edge of G needs to be considered only twice.

Moreover, Fischetti's RP1 algorithm also considers the 3-opt exchanges, in which we try to modify the cluster sequence (..., C_α, C_β, C_γ, ..., C_δ, C_ϵ, ...) into (..., C_α, C_γ, ..., C_δ, C_β, C_ϵ, ...).

However, using modified TSP algorithms is not the only approach. Another possible solution is to transform the generalized problem into the original, non-generalized traveling salesman problem in an efficient way. One trivial way to transform the GTSP to the TSP is by decomposing one GTSP into multiple, smaller size TSP problems. Each of the TSP problems is defined by a distinct set of cities, with exactly one from every cluster. The different TSP instances correspond to the different possible choices from the cluster. A tour of any of these TSP instances is obviously a (generalized) tour of the original GTSP problem. The best of the TSP solutions will be the best generalized Tour of the original instance. Unfortunately, this decomposition is very inefficient, since the number of problems

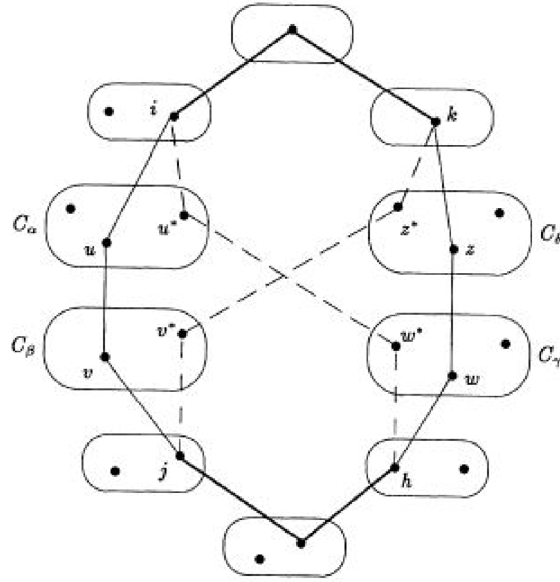


Figure 2.10: The generalized 2-opt exchange. Source: [12]

to be solved can grow rapidly. One possible solution is to transform the GTSP to exactly one TSP problem with larger size (for example see: [27]). Behzad and Modarres even achieved to transform the set-TSP into the standard TSP problem, where the number of nodes in the transformed TSP does not exceed the number of nodes in the original problem [3]. However, according to Noon and Bean [30], this approach may not outperform the GTSP specialized algorithms, but gives researchers means for verifying optimality on smaller problems. Such GTSP-specific solutions include dynamic programming [1], integer programming [24], Lagrangian relaxation [29], branch-and-cut algorithms [12] and genetic algorithms, which I will further investigate in my thesis.

Chapter 3

Genetic algorithm for the traveling salesman problem

Source: [25]

Genetic algorithms are often used to solve NP-hard problems. GAs often perform well approximating solutions to all types of problems because they ideally do not make any assumption about the underlying fitness landscape. The runtime of genetic algorithms depends on the representation of individuals and the implementation of GA operators.

3.1 Representation of tours

The first task to solve in every Genetic Algorithm is to represent individuals, i.e. possible solutions or tours in this case. A good representation supports GA operators to perform easily and fast on them. Many different representations have been proposed to the TSP using Genetic Algorithms. Among others, solutions were represented by binary strings, matrices, adjacency lists and paths.

3.1.1 Binary representation

Binary representation is perhaps the most commonly used representation in Genetic Algorithms. It is the most theoretically researched representation in the field of GAs [40]. In the traveling salesman problem, each city is encoded as a $\lceil log_2(n) \rceil$ long string, thus an individual is a string of length $n\lceil log_2(n) \rceil$. For example, in a TSP with 5 cities, the cities can be represented as a 3-bit string (see Figure 3.1).

In this problem, the tour (1-2-3-4-5) (than back to 1) is represented as

$$(000001010011100).$$

It is important to notice that there are 3-bit long strings that do not represent any city: 101,110,111. This is a problem that has to be solved when considering crossover and mutation operator. For example, the original crossover operator proposed by Holland [18] (although, he proposed it for another

i	city i	i	city i
1	000	4	011
2	001	5	100
3	010		

Figure 3.1: Binary representation of cities in the TSP

problem) has to be supplemented by some kind of repair algorithm. Consider for example the following two tours: (1-2-3-4-5) and (1-5-2-4-3) represented as

(000001010011100) (000100001011010)

In the canonical GA algorithm (see 1.9) a random crossover point is selected. This breaks the strings into two different parts. Lets suppose for example that we randomly select the crossover point to be between the 9th and the 10th string:

(000001010 | 011100) (000100001 | 011010)

Recombinating the different paths we obtain:

(000001010011010) and (000100001011100),

which do not represent legal tours: (1-2-3-4-3) and (1-5-2-4-5). Some extra effort is necessary to correct offspring. The role of repair algorithms is to transform those individuals that do not belong to the search space into individuals of that search space. Similar problem arises with the canonical mutation operator proposed by Holland [18] (see: 1.9). The operator changes one or more bits with a probability equal to the mutation rate which is close to zero. For example, if the second or third bit changes in the representation of the fifth city (100), we get a representation which does not correspond to any city, thus correction is necessary. Due to these concerns, the binary representation in the TSP is rarely used in practice.

3.1.2 Matrix representation

At least two considerable attempts have been made to represent tours as binary matrices.

1. Fox and McMahon [13] suggested representing an individual as a matrix M=$\{m_{ij}\}$, $1 \leq i, j \leq n$, where m_{ij}=1 if, and only if, in the tour city i is visited before city j. For example the tour (2-4-1-3) is represented as following:

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Note that a valid TSP tour represented by a matrix has the following properties:

1. $\sum_{i=1}^{n}\sum_{i=1}^{n} m_{ij} = \frac{n(n-1)}{2} \qquad (i, j \in \{1, ..., n\})$

2. m_{ii}=0 $\qquad (i \in \{1, ..., n\})$

3. $m_{ij}=1 \wedge m_{jk}=1 \Rightarrow m_{ik}=1 \qquad (i, j, k \in \{1, ..., n\})$

In case there are less than $\frac{n(n-1)}{2}$ ones in the matrix, but 2. and 3. are satisfied, one can always add ones to the matrix in a way that it represents a legal tour. On that note, new crossover operators were developed (for more details see: [13]). However, Fox and McMahon did not define mutation operator.

2. Seniw represented tours with matrices in a different way [8]. He defined the matrix M=$\{m_{ij}\}$, $1 \leq i, j \leq n$, where m_{ij}=1 if, and only if, in the tour city j is visited immediately after city i. That means a TSP tour is represented by a matrix which has exactly one 1 in every row and every column. Conversely, matrices with exactly one 1 in each row and column does not necessarily represent a legal tour. For example

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

represents the tour (1-4-2-3), but

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

represents the set of subtours {(1-2),(3-4)}. Various crossover and mutation operators with repair algorithms have been defined [19], [8]. Although there are several good experimental results on moderate-size TSP instances using matrix representation, this representation is not commonly used either.

3.1.3 Adjacency representation

In this representation proposed by Grefenstette [15], a solution is represented by a list of n cities. The city j is listed in the i-th position if, and only if, the tour leads from city i to city j. Thus, the tour

(3-6-2-1-4-5)

is represented by the list

(132654).

Although every tour has a unique adjacency list representation, a list can represent an illegal tour. For example, the list

(231564)

represents two subtours: {(1-2-3), (4,5,6)}. It is easy to see that the classic crossover and mutation operator may lead to illegal tours, therefore other operators had to be defined. Grenfenstette defined various crossover operators (Alternating Edge Crossover, Subtour Chunks Crossover and Heuristic Crossover [15]). However, all these operators gave poor experimental results ([25]). This can be attributed to the facts that the above-mentioned operators either destroy good subpaths in parents or do not take into account any information available about edge length, thus this representation is not widely used either.

3.1.4 Path representation

Path representation is by far the most natural representation of a tour. Again, a TSP tour is represented by a list of n cities. The tour

(2-3-6-4-5-1)

is simply represented by

(236451),

that means if a city i is at the j-th position if the list, the city i is the j-th city to be visited. Advantages of the path representation include simplicity of fitness evaluation and usefulness of final representation. The fitness of a given individual (a TSP tour) is easy to evaluate, since it can be calculated by summing the costs of each pair of adjacent nodes. The final representation is useful, because it directly outputs the list of the cities in the order in which they have to be visited. Due to these good characteristics, path representation is the most widely used representation both in TSP and GTSP instances solved by Genetic Algorithms. In the following parts of my thesis, I will exclusively concentrate on this representation of individuals. Again, the classical crossover and mutation operators are not suitable for the path representation of the TSP, therefore other operators had to be defined.

3.2 Crossover

Source of Figures: [11]

Crossover (or recombination) is the process where parents give their genes and characteristics to their offspring, thus forming the basis of the next generation. The ideal recombination operator should recombine the critical information from parents' structure in a non-destructive, meaningful manner. Moreover, the other important role of this operator is to maintain population diversity and therefore avoiding premature convergence. Crossover operators can be roughly divided into three main categories: interval-preserving, position-based and edge-based crossover [39]. In an interval-preserving crossover one sub-path is copied from one parent to the offspring and then the other cities are added accordingly to their relative order in the second parents, so as to create an offspring that represents a legal tour.

A position-based crossover preserves the relative position of cities in parents. It attempts to create a child, where every position is occupied by the corresponding element from one of the parents. An edge-based crossover tries to preserve good edges and add new ones heuristically. Different methods use different preserving and adding algorithms. Without the claim of completeness, I present some of the most commonly used recombination operators.

3.2.1 Partially-Mapped Crossover (PMX)

The Partially-Mapped Crossover is an interval-preserving recombination operator created by Goldberg and Lingle [14]. It is one of the most widely used operator for path-type problems. Many slight variations appeared in the literature, here I use Whitley's definition [2], which works as follows:

1. Choose two crossover points at random, and copy the segment between them from the first parent ($P1$) into the first offspring (see: 3.2).

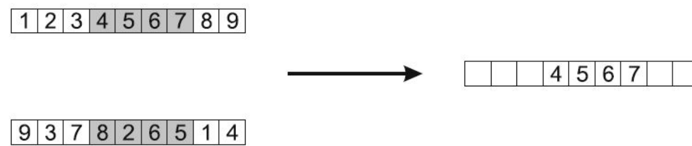


Figure 3.2: PMX step 1.

2. Starting off from the first crossover point look for elements in that segment of the second parent ($P2$) that have not been copied.

3. For each of these (say i), look in the offspring to see what element (say j) has been copied in its place from $P1$.

4. Place i into the position occupied by j in $P2$, since we know that we will not be putting j there (as we already have it in our string) (see: 3.3).

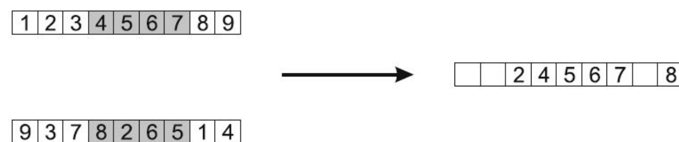


Figure 3.3: PMX step 4.

5. If the place occupied by j in $P2$ has already been filled in the offspring by an element k, put i in the position occupied by k in $P2$.

6. Having dealt with the elements from the crossover segment, the remaining position in this offspring can be filled from $P2$ (see: 3.4). The second child is created analogously with the parental roles reversed.

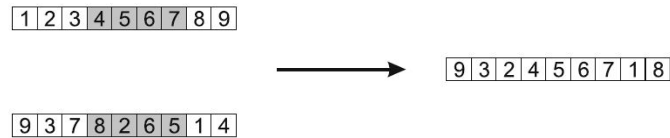


Figure 3.4: PMX step 6.

If we look at the offspring created in out example, we can see that six out of nine links, present in the offspring are present in at least one of the parents. A desirable property of any crossover operator would be to preserve edges present in both parents. However, the edge {7-8} is present in both parents, but not in the offspring.

3.2.2 Order Crossover (OX)

Order crossover is also one that can be categorized as an interval-preserving operator. As with PMX, OX also has more different variations in the literature. It was first proposed by Davis [9], who observed that the only important characteristic of a tour is the order of the cities, but not their absolute position. It creates an offspring by choosing a subtour of one parent and preserving the relative order of cities in the second one. Consider the following example, where two parent tours are:

$$(12345678) \text{ and } (24687531).$$

Lets suppose that the two randomly selected cut point are between the second and the third bit and between the fifth and the sixth bit. Hence, we get

$$(12|345|678) \text{ and } (24|687|531).$$

Now we create the first offspring in the following way:

1. The tour segment between the cut points are copied from $P1$ into the offspring. We obtain:

$$(**|345|***).$$

2. Starting from the second cut point of the first parent, the remaining cities are copied in the order in which they appear on the second parent, also starting from the second cut point. If a city is already presented in the offspring, we simply skip that city. When the end of a parent is reached, we continue from the first position, since the last and the first cities are connected in the tour. In our example this results in the offspring

$$(87|345|126).$$

The second offspring is created with the parental roles revered, i.e. copying the segment between the two cut points from $P2$ and filling in the remaining bits in the order of $P1$.

3.2.3 Maximal Preservative Crossover (MPX)

The Maximal Preservative Crossover [28] tries to prevent the destruction of good edges, but at the same time assures that there is enough information exchange between the parents. It works similar to the PMX, however there are some restrictions on the length of selected substring. Offspring 1 is created as following:

1. Randomly select a substring from $P1$, whose length is greater than or equal to 10 (except for very small instances), but smaller than or equal to the number of cities divided by 2.
2. All the elements of the chosen substring are removed from $P2$.
3. The chosen substring from $P1$ is copied to the first part of the offspring.
4. The end of the offspring is filled up with cities in the same order as they appear in $P2$.

A second offspring is created in an analogous manner with the parents' role reversed. For instance consider the following parent tours with substring $\{345\}$ chosen from the first parent:

$$(12|345|678) \text{ and } (24687531).$$

The MPX copies the substring $\{3,4,5\}$ to the offspring, then filling the rest with the elements of $P2$, without duplicating any numbers, thus we obtain:

$$(34526871).$$

The advantage of this crossover is that the number of destroyed edges have an upper limit. This MPX operator cannot destroy more edges than the length of the chosen substring. Usually at the beginning of the execution it reaches this maximum number of destroyed edges, but later on this number decreases, since solutions tend to have more common edges. Mühlbein et al. [28] decided to apply an additional mutation operator when less than 10% of the edges were destroyed.

3.2.4 Cycle Crossover (CX)

The cycle crossover suggested by Oliver et al. [31] is a position-based crossover, since it is concerned with preserving as much information as possible about the absolute position in which cities occur. This operator tried to create an offspring where every bit is occupied by a corresponding element from one of the parents. It is easier to understand CX through an example, thus consider the following parent tours again:

(12345678) and (24687531).

The first offspring can be created the following way:

1. We can choose the first element of the offspring to be equal to the first element in $P1$ or $P2$. Lets suppose we choose it to be equal to the first element of $P1$, therefore it the offspring will have 1 in the first position:

   (1*******)

2. The number 1 appears as the last element in $P2$, thus we know that the last element in the offspring must also be chosen from $P1$, otherwise the city 1 would be visited two times and it would not be a legal tour. Therefore, we get:

   (1******8).

3. Because the number 8 appears to be at the fourth bit in $P2$, the fourth bit in the offspring must also be occupied by the fourth element of $P1$. Analogously, the second element also has to be chosen from $P1$. Note that we cannot continue this sequence, since the city number 2 is in the first position in $P2$, which is already occupied, hence we got a so-called cycle which produces the following offspring so far:

   (12*4***8).

4. Now consider the third element in the offspring. Because we have chosen the first element to be from $P1$, we should select the third element to be from the second parent $P2$. To follow the previously introduced logic, the fifth, sixth and sevenths elements also has to be chosen from $P2$, thus we obtain the final offspring:

   (12647538).

Oliver et al. [31] concluded from theoretical and empirical results, that the CX operator performs better on the Traveling Salesman Problem than the previously discussed PMX.

3.2.5 Position Based Crossover (PBX)

The Position Based Crossover suggested by Syswerda [38] starts by selecting a set of random positions and copies the selected elements from $P1$ into the first offspring. The remaining positions are filled in the order of unused cities in $P2$. Consider the following example with the random choice of the second, third and sixth position.

(1**23**45**6**78) and (24687531).

We proceed as following:

1. Copy the selected elements from $P1$ to the offspring:

$$(*23**6**)$$

2. Fill the empty slots with elements of $P2$ in the original order without using the cities already present in the offspring. In our example, we obtain:

$$(42387651)$$

3.2.6 Heuristic Crossover (HX)

The Heuristic Crossover [16] is an edge-based operator, since it tries to prevent good edges of the parents from being destroyed. The process of offspring creation can be described in 4 steps.

1. Select a random element to be the current city of the offspring.

2. Consider the four edges that are incident to the current city in the parental paths. Based on their relative costs, define a probability distribution over these edges. The probability associated with an edge that leads to a previously visited city should be defined as 0.

3. Select an edge on this distribution. In case none of the parental edges leads to an unvisited city a random edge is selected. Update the current city.

4. Repeat the steps 3 and 4 until a complete tour has been constructed.

If HX is used with uniform distribution, about 30% of the edges are passed to the offspring from each parent and 40% of the edges are randomly selected [25].

3.2.7 Edge Recombination Crossover (ERX)

The main idea of Edge Recombination operator is that as far as possible an offspring should only use edges that are present in at least one of its parents, thus passing maximum amount of information to the next generation. The breaking of edges is seen as an unwanted change. According to Grefenstette [16], edge-based operators often have the problems of leaving cities without a continuing edge in the parents. These cities become isolated and new edges (that have not been present in any of the parents) have to be included in the offspring tour. The ERX tries to cope with this problem by first choosing cities which have as few unused edges as possible. The only edge that is chosen without taking its characteristic into consideration is the returning edge from the last city to the starting point. Due to its philosophy, only a limited amount of new edges appear. The operator had many slightly different forms over the years. I describe the probably most commonly used version of Whitley [2], also known as edge-3 crossover, which ensures common edges to be preserved. The operator works as follows:

1. Construct the edge table. Cities which are connected to the current city in both parent solutions should be indicated with a '+' sign.

2. Pick an initial city at random and put it in the offspring and let that city be the current element.

3. Remove all the references to the current element from the edge table

4. If a current city has an entry in its edge table signed with '+', that city should be chosen as the next element. If there is no city with '+' sign, chose the city which itself has the shortest edge list. Ties should be split at random. In case of reaching an empty list, the other end of the offspring is examined for extension. If the list of the other end is empty as well a new city is chosen at random, hereby introducing a new edge. Finally go to step 2, until there are no more unvisited cities.

Lets consider the following example, where the parent solutions are

(123456789) and (937826514).

Firstly, we calculate the edge table of the two parents (see: 3.5):

Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Figure 3.5: Edge table

We start by randomly selecting a city, let us say city number 1 for example. Then we look the entries in its edge list and choose the city with the shortest edge list. Since cities 2,4 and 9 have 3 entries in their edge lists (number 1 is deleted from all the lists!) and city number 5 has only 2 entries, we choose city number 5 as the next city to be visited, thus obtaining the partial tour

(15).

In the next step, we see that the edge table of 5 contains an entry with '+' sign, therefore we chose city 6 as the new current city and get

(156).

After that, both 2 and 7 have a 2-entry list, therefore we chose randomly between them. Lets say we chose city 2. We select city 8 as the next city because it has the shortest list, then 7 because it is connected the 8 in both parent paths. By now we have the partial tour of

(156287).

Finally, we choose 3 because it is the only remaining item in 7's edge list, then we decide randomly between its entries (4 and 9). At the end, we visit the only unvisited city. Supposing we chose 4 randomly at the last step, we obtain the offspring

(156287349),

and it is composed entirely of edges from the two parents. Note that only one child per recombination is created by this operator. However, ERX can be modified to produce two children. Another important observation is that the edge recombination operator indicates, that the path representation alone might be too poor to represent important properties of a tour. For that reason, it was complemented by another structure, in this example an edge table. Whitley et al. [41] tested the ERX operator experimentally. They reported solutions with the ERX operator to be better than any previously found solutions.

3.3 Mutation

Unlike in the binary representation, it is no longer possible to mutate each gene (city) independently, because we would get offspring that do not represent a legal tour. Rather we consider moving alleles around the genome, i.e. we change the place and internal sequence of a subpath in the tour. That means the mutation rate parameter should be interpreted as the possibility of a solution undergoing a mutation, while the genes in the genome remain unchanged. I will present some of the most commonly used mutations of paths.

3.3.1 Swap Mutation

Swap mutation (also called exchange mutation or point mutation) selects two random cities in the tour and exchanges them. For example, we select randomly the second and the fifth place in the tour

(123456789),

the mutation will cause the following change:

1	2	3	4	5	6	7	8	9

→

1	5	3	4	2	6	7	8	9

3.3.2 Insertion Mutation

The insertion mutation randomly selects a city from the tour, removes it, then replaces at a randomly selected place. For instance, in our beloved example

(123456789)

if we first randomly select the fifth place with the value 5, remove it from the tour and then we replace it in a randomly selected position (third in our example).

3.3.3 Displacement Mutation

The displacement mutation is a natural extension of the insertion mutation, where we select a substring instead of a single gene, remove it from the tour and reinserting it in different position. For instance if we select the substring {2345} in our example and then replace it after the seventh position, we change

(1|2345|6789) to

(167234589).

3.3.4 Scramble Mutation

In scramble mutation, we chose a random subpath again and then scramble the order of the cities in this subpath. For example, if we select the substring {2345} in

(123456789)

once again, we could get a result like this:

It is important to mention that the scramble mutation was suggested in scheduling problems and not for the TSP. In problems where the adjacency is important it is rarely used, because it can destroy many good connections in a subpath.

3.3.5 Inversion Mutation

The inversion mutation has two different forms: the simple inversion mutation and the general inversion mutation. The simple inversion randomly selects a substring and reverses the order of this subpath. This operator effectively breaks the tour into three parts and preserves all the links inside these substrings. Note that only two links between parts are broken, thus the inversion mutation is the smallest change that can be made in adjacency-based problems. All other changes can be easily constructed as a sequence of inversions. Once again, if we consider the example

(123456789)

with the substring {2345} to be chosen, we obtain the following mutation:

The simple inversion mutation can be generalized in a way where the chosen substring is not only inverted, but removed from the tour and then it is reinserted in another position analogously to the displacement mutation.

3.4 Setting parameters

One of the most important part of any GA is the right setting of parameters. Ultimately the success or failure depends on the fine-tuning of these variables. Such parameters include the population size, the rate of recombination (p_r) and the mutation rate (p_m). The choice of population size is heavily dependent on the problem size. Too many individuals in a population can lead to very long computational time, while too little population size may not be enough to map the whole fitness landscape. Following Holland's canonical genetic algorithm, the mutation rate is usually low (between 0.05 and 0.005) and the crossover rate is high (0.5-1.0) [37]. In the last three decades, a new approach emerged in the GA literature. In an adaptive genetic algorithm, p_m and p_r are variables that change adaptively during the run of the algorithm. Crossover and mutation have a two-sided goal in any genetic algorithm. Firstly, they are responsible for exploring new regions in the solution space in search of the global optimum. And secondly, they should encourage the convergence to the global optimum, after locating the region. Increasing values of p_r and p_m promote exploration at the expense of exploitation, while low rates prohibit the exploration of new regions. Researchers try to achieve this trade-off by varying p_m and p_r adaptively in response of the solutions. The rates are increased, when the population tends to get stuck at local optimum and are decreased when the population is too scattered. The experimental results show that GAs with carefully designed adaptive p_m and p_r rates provide much better solutions than GAs with fixed rates [37].

3.5 GA for the GTSP

Due to previously mentioned considerations (see: 3.1), solutions of the generalized traveling salesman problem are also usually represented using permutations (path representation). However, using crossover operators created for the original TSP might transform individuals into new ones outside the solution space (offspring solutions may visit certain clusters more than once, while leaving out others). That means we must look for crossover techniques that create offspring representing legal GTSP tours.

3.5.1 Generalized crossover

One possible solution was proposed by Gutin and Karapetyan [17], where a random fragment $p1_a$, $p1_{a+1}$, ..., $p1_{a+l-1}$ is selected from the first parent solution and it is copied to the beginning of the child solution. Next, we create a sequence q' from the second parents, starting off by the $a + l$-th element. We remove vertices from q' which have already been visited by the offspring, thus we obtain a fragment q from the second parent with length $m - l$. Now q is affixed to the end of the offspring. The second child is created in an analogous manner with parents' role reversed.

3.5.2 Generalized mutation

Traditional mutation operators can be applied to the generalized TSP, however the set of visited nodes never changes during mutation. To eliminate this problem, more complex mutation operators were developed. One possible approach is to first determine C_1, C_2, ..., C_m the order in which of clusters are visited, then calculating the shortest tour visiting the node subsets in this fixed order. This can be done in polynomial time, with a dynamic programming algorithm developed by Renaud and Boctor [33]. Let d_{ij} denote the distance between city i and city j and L_{ij} the length of the shortest path from node i from C_1 to node j from C_k while passing by only one node in each of the clusters C_2, C_3, ..., C_{k-1}. Let L_i denote the length of the shortest m-edge tour starting from and coming back to the node i while visiting one node in every cluster C_2, C_3, ..., C_m. Then L, the length of the shortest tour visiting each cluster exactly once in the chosen order can be computed as follows:

$$L = \min_{i \in S_i} L_i,$$

$$\text{where } L_i = \min_{j \in S_m} [L_{ij} + d_{ji}]; \forall i \in S_1,$$

$$\text{where } L_{ij} = \min_{l \in S_{k-1}} [L_{il} + d_{lj}]; \forall i \in S_1, \forall j \in S_k, k > 2;$$

$$\text{and } L_{ij} = d_{ij} \forall i \in S_1, \forall j \in S_2.$$

.

The time complexity of this algorithm is $\mathcal{O}(n^3/m^3)$, although it can be reduced by choosing as S_1 the cluster containing the smallest number of nodes [33].

Chapter 4

Experimental results

I used MATLAB R2014a version to create a program on which I can test how the genetic algorithm performs on GTSP instances. The program randomly distributes cities in a two dimensional square grid and tries to solve the generated GTSP. The user can determine the area size, the number of clusters and nodes, the size of the population, the number of generations and the rate of mutation. Vertices are randomly assigned to the clusters while assuring that each cluster contains at least one node. The rate of crossover was constant 1 in my experiments, thus every parent solution undergone recombination. However, elitism was applied, thus the best of each generation was automatically added to the next one. Throughout the experiments, the area size was 10. I compared the genetic algorithm to the Nearest Neighbor algorithm (see Subsection 2.2.1). An animated user interface has been programmed to visualize the results and the improvement of the genetic algorithm. On the left subplot, the result of the Nearest Neighbor algorithm has been displayed. The subplot on the right displayed the best path in each generation of the GA. The colors of the different states have been randomly generated. An example plot can be seen in Figure 4.1.

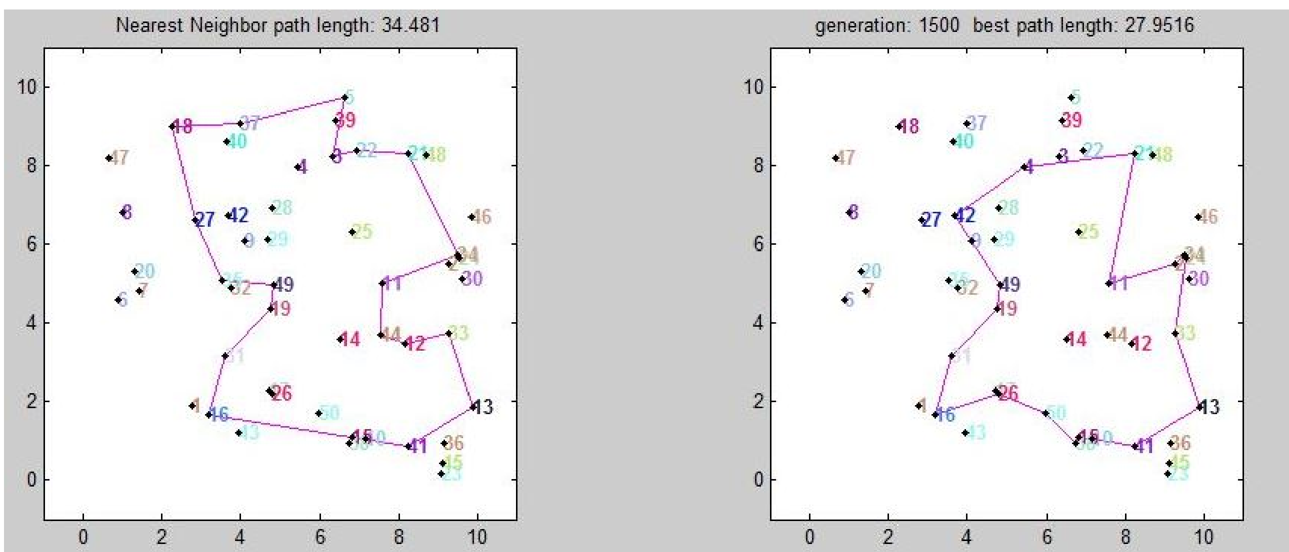
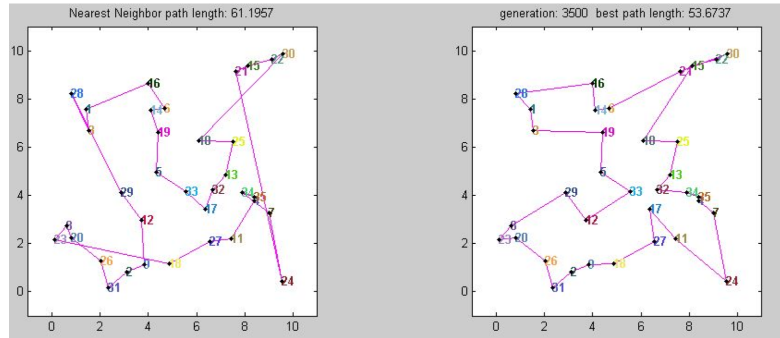


Figure 4.1: An example run

NN	GA	difference
63,3699	52,6253	10,7446
65,4729	63,3406	2,1323
67,3895	57,3576	10,0319
54,9306	51,0956	3,835
68,9894	52,1155	16,8739
49,8493	58,3281	-8,4788
64,8079	61,8189	2,989
58,5287	54,6172	3,9115
53,4497	54,4574	-1,0077
57,4724	55,683	1,7894

(a) Table of results



(b) Example run of a 35 city TSP instance

Figure 4.2: 35 city TSP

4.1 Performance on traditional TSPs

The algorithms have been run 10 times. Each time a new TSP instance with 35 cities was generated. The genetic algorithm was running for 3500 generations, the chance of mutation was 0.5%, and the size of population was 30. The algorithm took approximately 63 seconds to run. The length of paths and an example run on a 35 city TSP can be seen in Figure 4.2.

As we can see the GA outperforms the NN algorithm eight out of ten times. The length difference can be as big as 16.8 units. However, in two TSP instances the NN algorithm found shorter tour than the GA (highlighted red in the table).

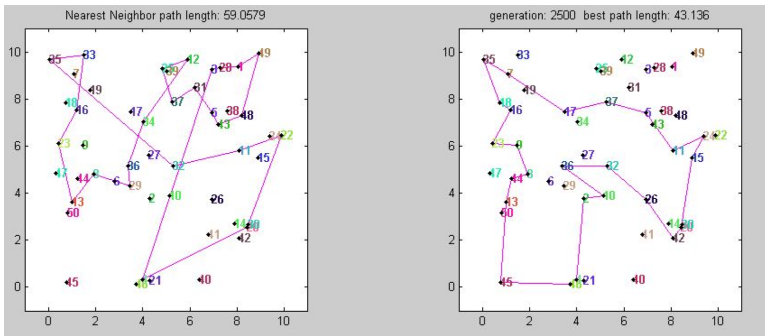
4.2 Performance on 50-25 gTSPs

The algorithms have been run 10 times. Each time a new TSP instance with 50 cities and 25 states was generated. The genetic algorithm was running for 2500 generations, the chance of mutation was 1%, and the size of population stayed 30. The algorithm took approximately 43 seconds to run. The length of paths and an example run on a gTSP instance with 50 cities and 25 states can be seen in Figure 4.3.

The genetic algorithm performed better on nine instances. We can conclude that the GA regularly outperforms the Nearest Neighbor algorithm in such instances.

4.3 Performance on 100-20 gTSPs

The shortcoming of the Nearest Neighbor algorithm is that at the end of its run it can include very long edges in the tour because some vertices are "left out". My conjecture is that NN performs better on instances where the state to city ratio is smaller, because it has more opportunity to choose from, thus the salesman is not required to travel such long distances towards the end of its journey. For that reason, I tested whether the genetic algorithm is able to outperform the Nearest Neighbor algorithm



(a) Example run on a 50-25 gTSP instance

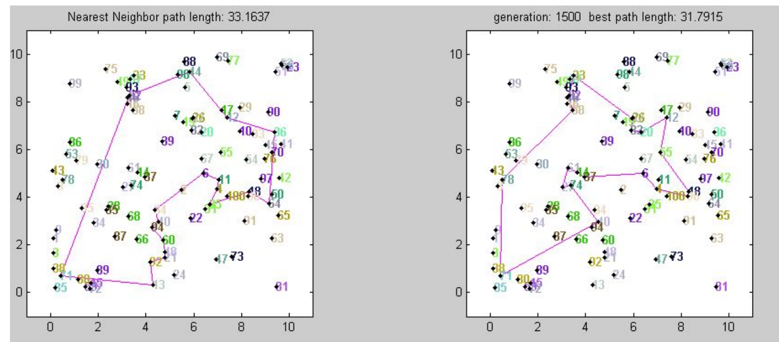
NN	GA	difference
59,0579	43,136	15,9219
57,859	42,1475	15,7115
44,2025	42,106	2,0965
39,0601	42,6546	-3,5945
41,9323	33,0406	8,8917
49,1435	43,6355	5,508
52,6823	39,5625	13,1198
44,2718	44,0494	0,2224
44,236	42,5833	1,6527
48,1444	42,8162	5,3282

(b) Table of results

Figure 4.3: 50 city, 25 states gTSP

NN	GA	difference
25,8212	28,1569	-2,3357
27,1298	27,0337	0,0961
26,2285	29,9323	-3,7038
28,485	26,5464	1,9386
29,8773	25,192	4,6853
26,2356	25,5667	0,6689
33,3183	31,6737	1,6446
29,4752	35,1421	-5,6669
34,8846	28,0007	6,8839
37,4165	29,0967	8,3198

(a) Table of results



(b) Example run on a 50-25 gTSP instance

Figure 4.4: 100 city, 20 states gTSP

on bigger instances with 100 cities and 20 states. In this instance the state to city ratio decreased from 1:2 to 1:5 compared to the previous experiment. Due to the more complex environment the population size has been significantly increased. In this experiment the size of population was 150. To maintain reasonable running time, the algorithm was running only 1500 generations. The algorithm took approximately 73 seconds to run. Mutation rate was 0.05%. The table of results and an example run can be seen in Figure 4.4.

As we can see, the GA does not dominate as clearly as in smaller instances. However, it still outperformed the NN algorithm seven out of ten times. We can conclude that GA is a useful algorithm that is able to provide better results than the Nearest Neighbor heuristic even on bigger gTSP instances with smaller state to city ratios.

4.4 Limitations and future improvement

The Nearest Neighbor algorithm could have been improved using tour improvement heuristics like insertion or generalized 2-opt exchange heuristics (see: 2.2.2). This way we would have created shorter tours, thus a tougher opponent to compete with. However, the genetic algorithm used in my experiments

can also be improved in many ways. One possible improvement is to write an adaptive genetic algorithm (3.4), thus continuously optimizing the mutation rate and crossover rate during the run. It would help mapping the environment and avoid getting stuck at local optima. Another possible improvement is to use the Shortest Tour algorithm 3.5.2 to find the shortest tour assuming the sequence of clusters is given. In that way, finding the global optimum is reduced to finding the right order of the clusters. Nevertheless, such an algorithm can lead to dramatically increased running times.

Bibliography

[1] H. AL. Record balancing problem-a dynamic programming solution of a generalized traveling salesman problem. *Revue Francaise D Informatique De Recherche Operationnelle*, 3(NB 2):43, 1969. 19, 20

[2] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Evolutionary computation 1: Basic algorithms and operators*, volume 1. CRC press, 2000. 25, 29

[3] A. Behzad and M. Modarres. A new efficient transformation of the generalized traveling salesman problem into traveling salesman problem. In *Proceedings of the 15th International Conference of Systems Engineering*, pages 6–8, 2002. 20

[4] R. G. Bland and D. F. Shallcross. Large travelling salesman problems arising from experiments in x-ray crystallography: a preliminary report on computation. *Operations Research Letters*, 8(3):125–128, 1989. 10

[5] I. Borgulya. *Optimalizálás evolúciós számításokkal.* Typotex, 2012. 1

[6] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document, 1976. 12

[7] G. A. Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958. 16

[8] S. D. A genetic algorithm for the traveling salesman problem. *Msc Thesis, University of North Carolina at Charlotte*, 1991. 23

[9] L. Davis. Applying adaptive algorithms to epistatic domains. In *IJCAI*, volume 85, pages 162–164, 1985. 26

[10] J. Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of Research of the National Bureau of Standards B*, 69(125-130):55–56, 1965. 15

[11] A. E. Eiben and J. E. Smith. *Introduction to evolutionary computing*, volume 53. Springer, 2003. v, 1, 6, 24

[12] M. Fischetti, J. J. Salazar González, and P. Toth. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research*, 45(3):378–394, 1997. v, 19, 20

[13] B. Fox and M. McMahon. Genetic operators for sequencing problems. *Foundations of genetic algorithms*, 1:284–009, 1990. 22, 23

[14] D. E. Goldberg, R. Lingle, et al. Alleles, loci, and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, volume 154, pages 154–159. Lawrence Erlbaum, Hillsdale, NJ, 1985. 25

[15] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–165, 1985. 23, 24

[16] J. J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. *Genetic algorithms and simulated annealing*, 4:42–60, 1987. 29

[17] G. Gutin and D. Karapetyan. A memetic algorithm for the generalized traveling salesman problem. *Natural Computing*, 9(1):47–60, 2010. 33

[18] J. H. Holland. Adaptation in natural and artificial systems. an introductory analysis with application to biology, control, and artificial intelligence. *Ann Arbor, MI: University of Michigan Press*, 1975. 7, 21, 22

[19] A. Homaifar, S. Guan, and G. E. Liepins. A new approach on the traveling salesman problem by genetic algorithms. In *Proceedings of the 5th International Conference on Genetic algorithms*, pages 460–466. Morgan Kaufmann Publishers Inc., 1993. 23

[20] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1:215–310, 1997. 11

[21] M. Jünger, G. Reinelt, and G. Rinaldi. The traveling salesman problem. *Handbooks in operations research and management science*, 7:225–330, 1995. v, 10, 11, 14, 15, 16, 17

[22] B. Korte. Applications of combinatorial optimization. In *talk at the 13th International Mathematical Programming Symposium, Tokyo*, 1988. 10

[23] G. Laporte, A. Asef-Vaziri, and C. Sriskandarajah. Some applications of the generalized travelling salesman problem. *Journal of the Operational Research Society*, 47(12):1461–1467, 1996. 19

[24] G. Laporte and Y. Nobert. Generalized travelling salesman problem through n sets of nodes: An integer programming approach. *INFOR: Information Systems and Operational Research*, 21(1):61–75, 1983. 20

[25] P. Larranaga, C. M. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999. 21, 24, 29

[26] J. K. Lenstra and A. R. Kan. Some simple applications of the travelling salesman problem. *Journal of the Operational Research Society*, 26(4):717–733, 1975. 10

[27] Y.-N. Lien, E. Ma, and B. W.-S. Wah. Transformation of the generalized traveling-salesman problem into the standard traveling-salesman problem. 20

[28] H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer. Evolution algorithms in combinatorial optimization. *Parallel Computing*, 7(1):65–85, 1988. 27

[29] C. E. Noon and J. C. Bean. A lagrangian based approach for the asymmetric generalized traveling salesman problem. *Operations Research*, 39(4):623–632, 1991. 20

[30] C. E. Noon and J. C. Bean. An efficient transformation of the generalized traveling salesman problem. *INFOR: Information Systems and Operational Research*, 31(1):39–44, 1993. 19, 20

[31] I. Oliver, D. Smith, and J. R. Holland. Study of permutation crossover operators on the traveling salesman problem. In *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*. Hillsdale, NJ: L. Erlhaum Associates, 1987., 1987. 27, 28

[32] R. D. Plante, T. J. Lowe, and R. Chandrasekaran. The product matrix traveling salesman problem: an application and solution heuristic. *Operations Research*, 35(5):772–783, 1987. 10

[33] J. Renaud and F. F. Boctor. An efficient composite heuristic for the symmetric generalized traveling salesman problem. *European Journal of Operational Research*, 108(3):571–584, 1998. 34

[34] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. An analysis of several heuristics for the traveling salesman problem. *SIAM journal on computing*, 6(3):563–581, 1977. 12

[35] J. P. Saksena. *Mathematical Model of Scheduling Clients Through Welfare Angencies: II.* Depts. of Electrical Engineering and Medicine, University of Southern California, 1967. 19

[36] J. Silberholz and B. Golden. The generalized traveling salesman problem: A new genetic algorithm approach. In *Extending the horizons: advances in computing, optimization, and decision technologies*, pages 165–181. Springer, 2007. 18

[37] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):656–667, 1994. 33

[38] G. Syswerda. Schedule optimization using genetic algorithms. *Handbook of genetic algorithms*, 1991. 28

[39] H.-K. Tsai, J.-M. Yang, Y.-F. Tsai, and C.-Y. Kao. Some issues of designing genetic algorithms for traveling salesman problems. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 8(10):689–697, 2004. 24

[40] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994. 21

[41] D. Whitley, T. Starkweather, and D. Shaner. *The traveling salesman and sequence scheduling: Quality solutions using genetic edge recombination.* Colorado State University, Department of Computer Science, 1991. 31

Attachment

TSP.m

```
% noc = number of cities
% as = area size
% ps = population size
% distmtx = distance matrix of the cities
% P = population (ps * noc)
% bp = best path
% fbp = fitness value of best path
% Gb = best path of a generation
% cities = randomly distributed cities
% cityloc = indicates to which cluster a point belongs
tic
noc=100; %number of cities
nos=20; %number of states
as=10; %area size
ps=150; %population size
nog=1500; %number of generations
Mr=0.005; %rate of mutation
if noc<nos
    error('There are more states than cities')
end
% randomly distribute cities
cities=as*rand(2,noc);
%Clustering cities into 'states'
[cityloc, statecolor]=clustering(nos,noc);
%calculating the distance matrix
distmtx=calculatedist(noc,cities);
%initializing population
P=initializepop(ps,noc,nos,cityloc);
```

```
29 %drawing the cities on the map
30 [hpb,rtitle]=drawfigure(as,noc,nos,cities,cityloc,statecolor);
31 %Nearest Neighbor heuristic
32 NNpath=NN(cities,cityloc,distmtx,noc,nos,as,hpb,rtitle);
33 for i=1:nog; %generation loop
34     %calculate probabilities function
35     [prob,Gb,bp,fbp,invfitness]=calcFitness(ps,distmtx,P);
36     % update best path on figure
37     updatebestpath(cities,Gb,hpb,rtitle,invfitness,bp,i)
38     %Roulette selection:
39     parentindices=roulette_selection(ps,prob);
40     %Crossover
41     P=crossparents(P,parentindices,cityloc);
42     %Mutation
43     P=mutation(P,Mr);
44     %Elitism
45     P(ps,:)=Gb;
46     %diversity measure
47     %div=diversity(P)
48 end
49 toc
50 pathlength(Gb,distmtx)
```

clustering.m

```
1 function [cityloc, statecolor]=clustering(nos,noc)
2 %This function randomly assigns vertices to clusters, while assuring that
3 %every cluster at least one vertex
4 cityloc=zeros(1,noc);
5 isLocated=zeros(1,noc);
6 for i=1:nos %assigning one random city to every state, so we know every
    state has at least one city
7     randomcity=ceil(noc*rand);
8     while(isLocated(randomcity)~=0)
9         randomcity=ceil(noc*rand);
10    end
11    isLocated(randomcity)=1;
12    cityloc(randomcity)=i;
```

```
13 end
14 for i=1:noc
15     if(isLocated(i)==0)
16         cityloc(i)=ceil(nos*rand);
17     end
18 end
19 stateSize=zeros(1,nos);
20 for i=1:nos
21     c=0;
22     for j=1:noc
23         if cityloc(j)==i
24             c=c+1;
25         end
26     end
27     stateSize(i)=c;
28 end
29 statecolor=zeros(1,3*nos);
30 for i=1:3*nos
31     statecolor(i)=rand;
32 end
```

calculatedist.m

```
1 function distmtx=calculatedist(noc,cities)
2 %this function calculates the distance matrix between all vertices
3 distmtx = zeros(noc,noc); %distance matrix
4 for i=1:noc-1
5     city1=cities(:,i);
6     for j=i+1:noc
7         city2=cities(:,j);
8         dcities=sqrt((city1-city2)'*(city1-city2));
9         distmtx(i,j)=dcities;
10        distmtx(j,i)=dcities;
11    end
12 end
```

initializepop.m

```
function P=initializepop(ps,noc,nos,cityloc)
%this function initializes the first population by generating random
%permutations of vertices
visitall=zeros(ps,noc); %random permutations of all cities, base for
    creating the population
for i=1:ps
    visitall(i,:)=randperm(noc);
end
P=zeros(ps,nos); %population
for i=1:ps
    c=0;
    isStateVisited=zeros(1,nos);
    for j=1:noc
        if isStateVisited(cityloc(visitall(i,j)))==0
            c=c+1;
            P(i,c)=visitall(i,j);
            isStateVisited(cityloc(visitall(i,j)))=1;
        end
    end
end
```

drawfigure.m

```
function [hpb,rtitle]=drawfigure(as,noc,nos,cities,cityloc, statecolor)
%this function draws the vertives on a two-dimensional as*as grid
figure('units','normalized','position',[0.1 0.2 0.8 0.5]);
hbp=zeros(1,2);
rtitle=zeros(1,2);
for j=1:2
    subplot(1,2,j);
    hpb(j)=plot(NaN,NaN,'m-');
    rtitle(j)=title(' ');
    hold on;
    for i=1:noc
        text(cities(1,i),cities(2,i),num2str(i),'FontWeight','bold','color
            ',[(statecolor(cityloc(i)*3-2)) (statecolor(cityloc(i)*3-1)) (
            statecolor(cityloc(i)*3))]);
    end
```

```
14      for i=1:nos
15       plot(cities(1,:),cities(2,:),'k.'); % plot cities
16      end
17      axis equal;
18      xlim([-0.1*as 1.1*as]);
19      ylim([-0.1*as 1.1*as]);
20  end
```

NN.m

```
1  function NNpath = NN(cities,cityloc,distmtx,noc,nos,as,hpb,rtitle)
2  %Nearest neighbor construction heuristics
3  %Starts with one node and always visits the nearest node from unvisited
4  %states
5  %Finally draws solution
6  diam=sqrt(2)*as;
7  distmtx2=distmtx+eye(noc)*diam;
8  start=ceil(rand*noc);
9  c=1; %counts how many states we have visited already
10 act=start;
11 NNpath=zeros(1,nos);
12 NNpath(c)=act;
13 while c~=nos
14 %we set city distances in the same state to diam, so it won't be the
       closest city
15     for i=1:noc
16         if cityloc(i)==cityloc(act) && i~=act
17                 distmtx2(i,:)=diam;
18                 distmtx2(:,i)=diam;
19         end
20     end
21     [minval, minplace]=min(distmtx2(act,:));
22     %do not visit visited city again
23     distmtx2(act,:)=diam;
24     distmtx2(:,act)=diam;
25     act=minplace;
26     c=c+1;
27     NNpath(1,c)=act;
```

```
28 end
29 NNlength=pathlength(NNpath,distmtx)
30 set(hpb(1),'Xdata',[cities(1,NNpath) cities(1,NNpath(1))],'YData',[cities
       (2,NNpath) cities(2,NNpath(1))]);
31 set(rtitle(1),'string',['Nearest Neighbor path length: ' num2str(NNlength)
       ]);
32 drawnow;
```

calcFitness.m

```
1 function [prob,Gb,bp,fbp,invfitness]=calcFitness(ps,distmtx,P)
2 %calculates fitness of individuals, marks the generation's best solution
3 invfitness=zeros(ps,1);
4 prob=zeros(ps,1);
5 for j=1:ps %population size loop
6      invfitness(j)=pathlength(P(j,:),distmtx);
7 end
8 fitness=1./invfitness; % we want to maximize fitness, inverse of path
       length
9 prob=fitness/sum(fitness);
10 [fbp,bp]=max(prob); %max is giving us two values: the maximum value and
       the number of position where the max element is
11 Gb=P(bp,:); %best path of the generation
```

updatebestpath.m

```
1 function update=updatebestpath(cities,Gb,hpb,rtitle,invfitness,bp,i)
2 %this function draws best path on the figure
3      set(hpb(2),'Xdata',[cities(1,Gb) cities(1,Gb(1))],'YData',[cities(2,
           Gb) cities(2,Gb(1))]);
4      set(rtitle(2),'string',['generation: ' num2str(i) '  best path
           length: ' num2str(invfitness(bp))]);
5      drawnow;
```

roulette selection.m

```
1 function parentindeices=roulette_selection(ps, prob)
2 %this function selects parents fitness proportionally
```

```
3 parentindeices=zeros(1,ps);
4 for i=1:ps
5       r=rand;
6       for j=1:ps
7            if sum(prob(1:j))>r
8                 parentindeices(i)=j;
9                 break
10           end
11      end
12 end
```

crossparents.m

```
1 function newgen=crossparents(P,parentindices,cityloc)
2 %this function creates a new generation by crossover
3 %Crossover rate = 1
4 parents=P(parentindices,:);
5 newgen=zeros(size(P));
6 [m,n]=size(P);
7 for i=1:floor(m/2)
8      P1=parents(2*i-1,:);
9      P2=parents(2*i,:);
10     cp1=ceil((n)*rand); %cutpoints are chosen at random from range [1..nos]
11     cp2=ceil((n)*rand);
12     if (cp2<cp1)
13          temp=cp2;
14          cp2=cp1;
15          cp1=temp;
16     end
17     l=cp2-cp1+1;
18     %Creating first offspring
19     newgen(2*i-1,1:l)=P1(cp1:cp2);
20     isVisited=zeros(1,n);
21     for j=1:l
22          isVisited(cityloc(P1(cp1+j-1)))=1;
23     end
24     %we create q by rotating P2 and then deleting cities in the already
          visited states
```

```
25      q=zeros(1,n);
26      for j=1:n
27          if cp2+j<=n
28              q(j)=P2(cp2+j);
29          else
30              q(j)=P2(cp2+j-n);
31          end
32      end
33      for j=n:-1:1
34          if isVisited(cityloc(q(j)))==1
35              q(j)=[];
36          end
37      end
38      newgen(2*i-1,l+1:n)=q;
39      %creating second offspring
40      newgen(2*i,1:l)=P2(cp1:cp2);
41      isVisited=zeros(1,n);
42      for j=1:l
43          isVisited(cityloc(P2(cp1+j-1)))=1;
44      end
45      %we create q by rotating P2 and then deleting cities in the already
            visited states
46      q=zeros(1,n);
47      for j=1:n
48          if cp2+j<=n
49              q(j)=P1(cp2+j);
50          else
51              q(j)=P1(cp2+j-n);
52          end
53      end
54      for j=n:-1:1
55          if isVisited(cityloc(q(j)))==1
56              q(j)=[];
57          end
58      end
59      newgen(2*i,l+1:n)=q;
60  end
```

mutation.m

```
function P=mutation(P,Mr)
%Displacement Mutation (based on Larrange 1999)
[m,n]=size(P);
for i=1:m
    if (rand<Mr)
        cut=[ceil(rand*n) ceil(rand*n)];
       if cut(2)<cut(1)
       temp=cut(2);
       cut(2)=cut(1);
       cut(1)=temp;
       end
       subtour=P(i,cut(1):cut(2));
       q=P(i,[1:cut(1)-1, cut(2)+1:n]);
       insertionp=ceil(rand*length(q));
       P(i,:)=[q(1:insertionp) subtour q(insertionp+1:end)];
    end
end
```

pathlength.m

```
function s = pathlength(path,distmtx)
%this function calculates the length of a given path
s=0; %path length summation
for k=1:length(path)-1
    s=s+distmtx(path(k),path(k+1));
end
s=s+distmtx(path(length(path)),path(1));
```