WWW.JAMK.FI

Traveling salesman problem

Jakub Štencek

Bachelor’s Thesis
May 2013

Degree Programme in Software engineering

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Author(s)
Štencek, Jakub

Type of publication
Bachelor´s Thesis

Date
8. 5. 2013

Pages
53

Language
English

Permission for web publication
(X)

Title
Traveling salesman problem

Degree Programme
Software Engineering

Tutor(s)
Salmikangas, Esa

Assigned by
JAMK University of Applied sciences

Abstract

The main target of this bachelor's thesis was to analyze and solve this classic optimization task. The traveling salesman problem is a very demanding optimization problem for which there is no universal algorithm to solve this problem optimally.

The purpose or this thesis was not to find an optimization algorithm but to find and to try out the known and usually used algorithms and compare their solution with a solution of the author's own algorithm. The first part focuses on the origin and partly expert description of this problem, the actual state of possibilities of solutions and supposed development in “not so far” future. The second part concentrates on the software which was made for solving the traveling salesman problem. This software was made in Java programming language and it is attached to this thesis. The third part is focused on the analysis and implementation some of current possible solutions. Four known algorithms are implemented and the fifth is an exact algorithm. This fourth part is focused on developing, testing the author's own algorithm and its evaluation. This algorithm is based on a simple genetic algorithm but radically adjusted. The final chapter is focused on evaluation of algorithms and comparison of their results.

This thesis showed that even a simpler algorithm can achieved quite good value of the solution. Probably the best implemented solution was Christofides algorithm which can be recommended as primary algorithm to solve the traveling salesman problem or to start with the heuristic solution.

Keywords
Traveling salesman, problem, algorithm, optimization, the shortest distance, network

Miscellaneous

Contents

1 Introduction.....5

2 Theory.....7

- 2.1 Theory of the traveling salesman problem.....7
- 2.2 Heuristic algorithms.....9
- 2.3 Formulation of problem as integer programming problem.....11
- 2.4 History.....11
- 2.5 Current situation.....13
- 2.6 Known solutions.....13
- 2.7 Future.....16

3 Software for solving traveling salesman problem.....17

- 3.1 Developed software.....17
- 3.2 Input data.....18
- 3.3 Structure of data in application and drawing of vertexes and solution.....19
- 3.4 Matrix of destinations.....20
- 3.5 Choosing algorithm.....22

4 Implementation of some known solutions.....24

- 4.1 Chosen sources.....24
- 4.2 Nearest neighbor.....25
- 4.3 2-approximation algorithm.....27
- 4.4 Christofides algorithm.....31
- 4.5 Genetic algorithm.....37
- 4.6 Exact algorithm.....41

5 Stencek algorithm.....43

6 Conclusion.....50

References.....53

FIGURES

FIGURE 1: Matrix of destinations..21
FIGURE 2: Choosing algorithm...22
FIGURE 3: Results..23
FIGURE 4: Graphical output..23
FIGURE 5: Seven bridges of Königsberg..35
FIGURE 6: Generation of the random way without sub-cycles..........................46
FIGURE 7: Mutation...47
FIGURE 8: Crossover...48
FIGURE 9: Graphical output of Stencek's algorithm. 49

TABLES

TABLE 1: Count of permutations for solve TSP with exact algorithm..................................6
TABLE 2: The results of Nearest neighbor algorithm...27
TABLE 3: The results of 2-approximation algorithm...31
TABLE 4: The results of Christofides algorithm..37
TABLE 5: The results of Genetic algorithm...40
TABLE 6: The results of Exact and other implemented algorithms.....................42
TABLE 7: The results of Stencek's algorithm...49
TABLE 8: The summary of solutions 52

1 Introduction

It does not have to be so obvious, but the **traveling salesman problem** is a really difficult optimization problem. This problem is mathematical and it generalizes the problem of finding the shortest way around all points in a set of vertexes.

This problem can be formulated as:

- There are ***n*** points and ways between all of them with known lengths (which means it is simple to find the shortest way between any two points). The target is to find the shortest way, which passes all points right at once, start and end in the same point. It means the target is to find the shortest possible round trip.

The real problem is not to find the simplest way to looking for the shortest round trip – one that exists and to find the optimal round trip. There is, however, a problem, a major one. In this case it is necessary to try out all possible ways between all points, which is exactly the problem. Even a "layman" has to see this solution has to be abnormally time-consuming. In a case of a few points (e.g. four or five) the problem is not so complicated. Even though everybody can solve this with a pen and piece of paper, some proficient arithmetician can solve this problem even without any tools and help. In case of more point (e.g. eight or nine) it is possible to use some simple algorithm even this one with try out all of possible solutions. But what if there is a case where there is the need to solve traveling salesman problem with thousands or tens thousands of points? Is it possible with this quantum of computation to solve (it means in real time, when computer will be computing for several years it is useless) with current computers?

In table 1 it is possible to see that when the traveling salesman problem has only ten points the count of possible ways is really a huge number. The reader is invited to try to imagine how long it would take to try out all possible ways in case of "just" 20 points. This is such a large quantum of computations that it is impossible to solve it in real time. There were a great number of scientists and other smart people in history who tried to find universal optimization algorithm for this problem, however, nobody was successful.

TABLE 1: Count of permutations for solve TSP with exact algorithm

Count of points	Count of possible ways
4	24
5	120
6	720
7	5 040
8	40 320
9	362 880
10	3 628 800
11	39 916 800
12	479 001 600
13	6 227 020 800
14	87 178 291 200
15	1 307 674 368 00
16	20 922 789 888 000
17	355 687 428 096 000
18	6 402 373 705 728 000
19	121 645 100 408 832 000
20	2 432 902 008 176 640 000
25	15 511 210 043 330 985 984 000 000

2 Theory

2.1 Theory of the traveling salesman problem

Traveling salesman has to visit all of his customers, come back home and take the shortest way. The way in which the visits to all points in the graph can be modeled is implemented with Hamilton's sequence. The sequence in a graph G is Hamilton's sequence in graph G which contains all the points of graph G. This is even the definition of Hamilton's way and Hamilton cycle, because both are special cases of Hamilton's sequence. Graph G is Hamilton's graph, if in graph G exists Hamilton's cycle. The target is in coherent edge rated graph to find the shortest Hamilton's cycle / the shortest closed Hamilton's sequence (Paluch, 2008).

The optimization version of traveling salesman problem belongs to NP-hard (NP means non-polynomial) problems. In general case it is not known how to find an optimal solution for every possible input in real time nor is it yet known. If a polynomial complex algorithm can exist.

Algorithm – is a mechanical procedure containing any simple element steps which for any input produces an output.
Algorithm can be assigned for:

- verbal description in general language
- as computer program in any programming language
- as hardware circuit etc.

Time of compute means that computers compute really fast (billions of operations in second) but of course not infinitely fast. To run any operation takes some non-null time. To compute for the same input can cost different time-consumption for different algorithms.

This could depend on:

- implementation of algorithm
- programming language
- used interpreter
- used hardware (processor, memory etc.).

Polynomial algorithms, class P and NP can be defined as follows: **I**n theory of complexity there is class P one of the basics classes. It contains all problems which are resolvable with deterministic Turing machine in polynomial time. P is obviously considered as class of problems which can be effectively resolvable but even in this class it is possible to find effectively non-resolvable problems (with complexity e.g. $N^{1000000}$ etc.). P contains many natural problems, looking for the smallest common divisor or finding the maximal paring in graph. In 2002 was proved that the problem decision of primarily belongs to class P. Superset P is NP. Class NP is a class of problems resolvable in polynomial time-consuming on non-deterministic Turing machine. The relationship between P and NP is not still resolved. But here is possibility that these two sets are equal. Even if the proof still does not exist, most of the word wide experts think that P is a proper subset NP.

In real world these kinds of problems are solved just approximately – with the use of heuristic algorithm (genetic algorithm, tabu search etc.). With these heuristic algorithms relatively good solutions can be achieved but the cost of this is to give up the entitlement for optimal solution. It should not be forgotten that, even a heuristic algorithm can find an optimal solution.

Traveling salesman problem can be split into two different kinds of problems – asymmetric and symmetric.
In the symmetric case of the traveling salesman problem is the distance between two points which are the same in both directions. The basic structure of graph is an undirected graph. In asymmetric case the distance between two points in both directions can be different or there can be just a way in one direction from/to the point. This structure is called oriented

graph. In the example for asymmetric traveling salesman problem it can be looking for the shortest tour in a city with one-way streets.

The most used case type of traveling salesman problem is so-called metric traveling salesman. In this case all distances are in graph fulfills triangular inequality. This type of traveling salesman problem coincides with the most of real problems. This attitude allows construction of approximate algorithms.

2.2 Heuristic algorithms

Heuristics is procedure to solve a problem which is not entirely accurate but is found in short (real) time. It is the procedure for searching exact enough solution which is not possible to prove. Heuristic algorithms are usually used when it is not possible to use any other and better algorithms (e.g. exact algorithm) which could give an optimal solution with general proof.

Heuristic algorithms were begun to be used with the advent of computers. They are still developed and used to solve even very complicated problems. It is useful to use them to solve e.g. functions with many parameters, complex processes and many extremes. They are used where it is not useful because of terms of time, which is the main disadvantage of exact algorithms. In this time there are usually methods to solve problems which combine heuristic and exact attitude.

The basic difference between heuristic and exact algorithm is the fact that heuristic algorithm does not give optimal solution (but it is not impossible to get an optimal solution) but gives an approximate solution which is close (more or less) to optimal. It is possible that there no exact algorithm to some problem. Exact algorithms are usually so demanding and cannot solve the problem in polynomial time. In the case of traveling salesman problem the exact algorithm means to try out all possible permutations which means to find an optimal solution, however, even a problem with a few points is really demanding on computing time.

Genetic algorithms

Genetic algorithm is a heuristic algorithm which simulates principles of evolution biology for finding solutions of complex problems which cannot be solved with any other exact algorithms. Genetic algorithms and any other algorithms belonging to so-called evolution algorithms use techniques as evolution principles known from biology – inheritance, mutation, natural selection, hybridization – for “selective breeding” of solution of a basic problem.

The principle of work of a genetic algorithm is a gradual creation of generations of solutions for different problems. It saves a so-called population in which each individual is one possible problem's solution. When the population develops the solutions become better and better. The solution is traditionally represented by binary numbers, string of zeros and ones but it is possible to use any other representation. Typically it is first a population from random individuals. In transition is a so-called “fitness function” is counted which expresses the quality of solution for a given individual. According to this quality individuals are scholastically selected, modified and new population is created. This procedure is iteratively recurrent what means the quality of solutions is better and better with any new population and iteration. The algorithm is stopped when enough quality of solution is achieved or after a predetermined time.

Tabu search

Tabu search is a meta-heuristic searching algorithm which can be used for solving combinatory problems (where it is necessary to achieve optimal arrangement and selection). The solution of this method greatly depends on the initial solution. Tabu search uses a local or surrounding search procedure, iteratively passes through possible solutions until criteria to end it is done. In many cases it happens that local procedures get stuck in local extreme. It is possible to prevent this with searching in areas which were skipped. The solution can be saved in any possible structure.

2.3 Formulation of problem as integer programming problem

Traveling salesman problem can be transformed on integer programming problem and solved with some solver (e.g. Express IVE etc.).

$$\min \sum_{i \neq j} c_{ij} x_{ij}$$
$$\sum_{i=0, i \neq j}^{n} x_{ij} = 1 \qquad j = 0, \dots, n$$
$$\sum_{j=0, j \neq i}^{n} x_{ij} = 1 \qquad i = 1, \dots, n$$
$$0 \leqslant x_{ij} \leqslant 1 \qquad \forall i, j$$
$$x_{ij} \in Z \qquad \forall i, j$$
$$u_i - u_j + n x_{ij} \leqslant n - 1 \qquad 1 \leqslant i \neq j \leqslant n$$
$$u_i \geqslant 0 \qquad u \in \mathbb{Z} \; \forall \; 1 \leqslant i \leqslant n$$

***Formula 1** – Formulation of traveling salesman problem as integer programming problem*

Constant c_{ij} means distance between two points *i* and *j*. Constant *n* means count of points. Binary variable x_{ij} indicates if it is the used way from point *i* to point *j* (1 – yes, 0 - no). *The s*econd and third condition secure that every point can be visited only once. The last condition prevents creation of cycles (Ping Ji & William Ho).

2.4 History

This sub-chapter is based on Laporte (2006). The roots of the traveling salesman problem are ambiguous. There is no doubt that people around the world needed to solve a problem with visiting more places and walk the shortest way. On science level this problem began to be solved sometime between 18th and 19th century. In 1835 was created instruction for traveling salesman with practical examples from Germany and Switzerland. This manual did not contain any mathematical or any other supporting documents to these examples.

The traveling salesman problem was firstly officially defined around 1800 by Irish mathematician W. R. Hamilton and British mathematician Thomas Kirkman. Hamilton developed some kind of game for solving traveling salesman problem - “Icosian game”, which is the so called Hamilton's game. The principle of this game is simple, the “player” has to find Hamilton's circle in the set of points. This problem got the final form around 1930 when professors began teaching about it at the University of Vienna and at Harvard University.

In the 1950s and 1960s this problem gained popularity in scientific circles not only in Europe but in the USA as well. Remarkable articles were published by George Dantzig, Delbert Ray Faulkerson and Selmer M. Johnson from RAND Corporation in Santa Monica. They transformed this problem onto a linear programming problem and developed method “cutting plane” for solving it. With this new method it was possible to solve the traveling salesman problem with 49 cities with an optimal solution which means this method found the shortest possible way around all points. For the next decades this problem was studied by many other scientists, mathematicians, computer scientists, physicists and many other scientists and enthusiasts.

Richard M. Karp proved that the problem of Hamilton's circle is NP-hard in 1972. This detection gave more apparent difficulty for finding optimal route.

In the 1970s and 1980s significant progress was registered when Grötschel, Padberg, Rinaldi and team resolved this problem with 2392 cities with “cutting planes” method and method “branch and bound”.

In 1990s Applegate, Bixby, Chvatal and Cook developed a program which used more methods to resolve this problem. Gerhard Reinelt published TSPLIB with examples of different difficulties for solving the traveling salesman problem in 1991. These examples are used by many researchers around the world and they share solutions of their own algorithms.

In 2005 Cook and team presented a solution for problem with 33 810 cities. It was the biggest (re)solved problem in this time. They guaranteed the solution which will not be worse than 1% compared with optimal tour for any other kind of this problem.

2.5 Current situation

Even in this time it is really interesting to see how many incredible experiments are done for solving the traveling salesman problem, and not only with the usage of computers. One of the most interesting could be to let traveling salesman problem to be resolved with a bee. Even the author of this thesis was really surprised when he read it for the first time. The basic principle of this experiment was to rearrange flowers and see how bees can adapt for this new situation. Every bee wanted to fly the shortest possible route. Of course it cannot be accept it as a stateful scientific experiment but even with this kind of bizarre experiment it is possible to get new ideas and inspiration for looking for new possible algorithms.

Optimal solution

As author of this thesis said before it is still unknown if some polynomial complex algorithm exists which can resolve any general traveling salesman problem with an optimal solution. Even the smartest people of our age still could not just confirm or disprove the existence of this algorithm possible. This problem has been solved for long time and even when good progress id made the point of perfection (optimal algorithm) is still so far.

2.6 Known solutions

The thesis deals with possible and known algorithms for solving traveling salesman problem but and even the author's own algorithm for solving is presented. This part is about a few known and used algorithms.

2-approximation algorithm

This algorithm solves a metric kind of traveling salesman problem in polynomial time. The main idea is to construct a minimal spanning tree. From the definition of the graph theory is known $cost(min\ spannig\ tree) \leqslant cost(optimum)$ because the minimal spanning tree contains *U-1* edges and circle contains *U* edges.

The second step applies a deep search of the first algorithm on this tree and saves all routes across all vertexes – some of the vertexes are handled twice.

The last step means going through all vertexes from step two and ignore all duplication. It makes a circle. Also, here is valid the triangular inequality, thus the cost of circle will be maximal two times higher than original tour.

Christofides algorithm

This algorithm can solve the traveling salesman problem with a solution which is maximal 1.5 multiple of the optimal solution. This less good solution is not "for free". The implementation is really demanding and it is proved on real data that this solution is not much better than 2-approximation algorithm.

The first step is to construct a minimal spanning tree of the graph and with deep search of the first algorithm from random vertex save the vertexes with odd count of edges and construct a complete graph from them.

The second step is to find minimal perfect matching and add these new edges to the minimal spanning tree. This graph is so-called Eulerian's graph. It means there exists a move which contains all of the edges right at once.

The last step is to make Eulerian's move.

Nearest neighbor

This algorithm is usually called Greedy's algorithm. It is probably the simplest algorithm but it means not so good quality of solutions. The principle of this algorithm is really simple. The user chooses the first vertex and algorithm will iteratively choose the nearest vertex until all vertexes are chosen.

The implementation is really simple and the quality of solution can depends on choosing the first vertex. In this time a lot of possible upgrades of this method are known. It is usually not used method in real situations.

Random improvement

This method can find an extremely short tour so close to an optimal tour on an example with maximum of 700 or 800 vertexes.

It can be said that these kind of algorithms are the best that there are in this time. In real life a problem with 100 000 vertexes can be solved.

The basic principle is to find a random tour, choose four close vertexes and change their order and made a new tour. This algorithm looks for a local minimum. This algorithm can be really useful for solving this problem with thousands of vertexes.

Ant colony

The pathfinder of artificial intelligence Marco Dorigo described a method to find “good enough” solutions in 1997. This method is the simulations of “ant colony”. This algorithm is inspired by the behavior of ants looking for the shortest tour between a source of food and their nest. The basic of this algorithm is the behavior of an individual which arises from following “steps” of other ants.

This algorithm sends many virtual ants so-called “agents” which explore possible tours. The selection depends on heuristic joining distance to the nest and the count of virtual steps left by other ants. Each agent evaluates the step before passing to another vertex. If any agent found a new shortest tour, this tour is marked as the actually shortest. The count of steps is inversely proportional to the length of the tour where: more steps means a shorter tour.

This method is very effective but so demanding for implementation and in case of a large count of vertexes could be demanding and time-consuming.

2.7 Future

Future development of this problem is at least really unclear. As it was said there is not yet any polynomial complex algorithm for solving this problem, however, it is not impossible to find someone. Every year many organizations around the world make contests and give grants to find the best possible solution. In this time the progress of development in information technology is so fast and with it goes hand in hand the progress of the development of software solutions, not excepting traveling salesman problem. It can be compared how powerful computers were 10 years ago and how powerful are they today. It is possible to say computers are more powerful day by day. Maybe it is worth considering if in (near) future there could be a computer so powerful that it will be possible to solve the traveling salesman problem (and many others) just with the exact algorithm without any revolutionary optimize algorithm. The author of this thesis is aware this is only speculation and probably sci-fi and many educated people will not agree with him but in his opinion it could be one of more possible views on the matter.

3 Software for solving traveling salesman problem

3.1 Developed software

The author developed his own software for solving this problem. This software is made in programming language Java in Netbeans 7.2.1.This programming language was chosen because the author has a lot of experiences from many curses from Faculty of management sciences and informatics at University of Zilina e.g. Java – language and development of applications, Data structures 2, Discrete simulation etc. The author programmed mainly in operation system Fedora 18, however, this software works on any other usually used platforms (functionality was tested on Windows 7, Windows 8, Ubuntu etc.) because Java is multi-platform language.

Netbeans

Netbeans (Wiki Ubuntu) is an open-source software development kit. It is a relatively quick development environment with many possibilities for developing programs not just in Java (which is primary target) but also for many others (e.g. PHP, C++ etc.). This is a product of American software company Oracle (earlier known as Sun Microsystems). This development environment observes generally known standards and can be run on every usually used operation system where Java Virtual Machine is run. Netbeans also contains designer of graphical user interface, thus with just little bit of exaggeration it could be said to make a nice user interface with just a “click”.

Java

Java is an object oriented programming language. Java was developed and introduced on 23rd May 1995 by the American company Sun Microsystems, at this time know as Oracle. Java is one of the most popular and used programming language. For its total versatility is used not just for developing applications for desktop computers and notebooks but as well for mobile phones and other mobile devices (Wiki Ubuntu). In the last months and years Java has seen a boom thanks to new platform Android because Java is the main used programming language for this platform.

The basic functions of control are very simple. Users have to choose an input file with input data from computer, all points are automatically drawn according to specified coordinates and it them calculates a matrix of distances. After that users can choose which algorithm will solve traveling salesman problem on input data. When all of steps of the algorithm are done the application opens a tab with graphical representation of a solution where all points and the final tour between them are. In tab “Results” is the final distance, sequence of points, data processing time and algorithm processing time.

The basic functionality of this application is described in the following subchapter and chapter two and three are devoted to implemented algorithms.

3.2 Input data

Input data are really necessary for this thesis. A good source of testing data for all algorithms was found, web page http://www.tsp.gatech.edu/. It is a web page focused on solving the traveling salesman problem. Every researcher around the world can find there “infinite” quantity of information about this problem, they can connect with other researchers, can share their own examples, algorithms and solutions. This source was also chosen because there are not just fictive examples but real examples of real states with an optimal solution. Every researcher can share his/her own best solution. This information will be really useful for the last chapter of this thesis to evaluate powerful of implemented algorithms.

Structure of input data

The structure of input data is changeless. All data which are used in this thesis are from the http://www.tsp.gatech.edu/.

The first number is identification of vertex, the second number is x-coordinate and the third number in line is y-coordinate of vertex.

Under this paragraph is shown an example of input data. This example is from site of vertexes from Finland with 10 639 vertexes.

```
5883 62466.6667 29050.0000
5884 62466.6667 29200.0000
5885 62466.6667 29250.0000
5886 62466.6667 29916.6667
5887 62466.6667 30000.0000
5888 62466.6667 30100.0000
5889 62466.6667 31116.6667
5890 62483.3333 21266.6667
5891 62483.3333 21350.0000
5892 62483.3333 21483.3333
5893 62483.3333 21733.3333
5894 62483.3333 21933.3333
5895 62483.3333 22016.6667
5896 62483.3333 22050.0000
5897 62483.3333 22300.0000
```

3.3 Structure of data in application and drawing of vertexes and solution

Data are loaded into memory – to the objects (*Point*). All of these objects have 5 parameters:

- ID – identification of the *Point*
- X – x-coordinate
- Y – y-coordinate

- dX – regulated x-coordinated
- dY – regulated y-coordinate

Every *Point* has regulated drawing coordinates. The reason is simple – as can seen on the example of input data the range of coordinates can be really huge and the drawing could be very complicated (drawing area starts with coordinates (0,0) so *Point* with coordinates around e.g. 5000 could be problem to draw and too uncomfortable for evaluation etc.). This is the reason why regulated drawing coordinates were used which are adjusted comparatively to the size of the drawing area.

Drawing in Java is not as easy as many peoples think and the same concerns the work with many others graphical components. For the drawing in Java's component *Jpanel was used*. There was a little problem with sending data for drawing. The option with sending data with static variable was chosen– coordinates and final solution.

Points are drawn with a method drawCircle with coordinates dX and dY. The final solution is drawn with method drawLine with dX and dY coordinates of *Points*. The final sequence of *Points* contains all Points in a system so to make a complete tour is necessary to make one more way, from the last Point in sequence to the first Point of sequence.

3.4 Matrix of destinations

For computing and work with a matrix of destinations so-called Euclid's distance is used. It means it does not matter in which units there is input data (or what is the map scale) because just ways in the same units are compared.

Main settings | View | Matrix | Results

x\y	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0.0	74.53531	4109.913	3047.9956	2266.9111	973.53827	4190.101	3301.8933	4757.7417	3044.347	3094.9775	3986.2612	5092.5044	6406.149	5903.3887
2	74.53531	0.0	4069.704	2999.49	2213.5935	900.61664	4137.397	3238.5266	4701.3584	2969.8943	3020.6694	3913.829	5023.748	6336.6587	5828.856
3	4109.913	4069.704	0.0	1172.3667	1972.942	3496.2278	891.005	1814.9076	1415.4904	3672.7976	3777.1604	2995.6453	2876.1465	3903.8013	5055.8276
4	3047.9956	2999.49	1172.3667	0.0	816.667	2350.0	1172.1306	994.70886	1796.6792	2648.3745	2755.0457	2315.2878	2719.885	3972.3835	4546.8247
5	2266.9111	2213.5935	1972.942	816.667	0.0	1533.333	1923.9003	1188.0197	2497.221	2208.38	2311.8052	2324.0896	3087.8342	4400.104	4557.2905
6	973.53827	900.61664	3496.2278	2350.0	1533.333	0.0	3416.8296	2409.7605	3935.3806	2113.7769	2175.1118	3013.535	4141.289	5448.662	4954.908
7	4190.101	4137.397	891.005	1172.1306	1923.9003	3416.8296	0.0	1233.3336	650.8537	3086.3499	3184.38	2202.2715	1986.7605	3063.53	4179.746
8	3301.8933	3238.5266	1814.9076	994.70886	1188.0197	2409.7605	1233.3336	0.0	1586.4878	1876.7579	1978.5651	1320.7742	1899.8527	3213.1753	3555.6294
9	4757.7417	4701.3584	1415.4904	1796.6792	2497.221	3935.3806	650.8537	1586.4878	0.0	3285.3628	3373.507	2177.2173	1576.2988	2490.665	3883.655
10	3044.347	2969.8943	3672.7976	2648.3745	2208.38	2113.7769	3086.3499	1876.7579	3285.3628	0.0	106.71884	1359.7394	2673.8447	3821.5696	2863.808
11	3094.9775	3020.6694	3777.1604	2755.0457	2311.8052	2175.1118	3184.38	1978.5651	3373.507	106.71884	0.0	1411.953	2723.6104	3850.6023	2824.103
12	3986.2612	3913.829	2995.6453	2315.2878	2324.0896	3013.535	2202.2715	1320.7742	2177.2173	1359.7394	1411.953	0.0	1314.2377	2510.0571	2250.2468
13	5092.5044	5023.748	2876.1465	2719.885	3087.8342	4141.289	1986.7605	1899.8527	1576.2988	2673.8447	2723.6104	1314.2377	0.0	1321.7631	2330.2966
14	6406.149	6336.6587	3903.8013	3972.3835	4400.104	5448.662	3063.53	3213.1753	2490.665	3821.5696	3850.6023	2510.0571	1321.7631	0.0	2350.1606
15	5903.3887	5828.856	5055.8276	4546.8247	4557.2905	4954.908	4179.746	3555.6294	3883.655	2863.808	2824.103	2250.2468	2330.2966	2350.1606	0.0
16	8436.198	8368.359	5441.2266	5801.9395	6341.092	7490.327	4733.714	5175.288	4088.058	5889.374	5914.6235	4583.0903	3350.1663	2073.3833	3950.0352
17	6962.778	6889.8677	4989.601	4883.3623	5174.3228	5989.273	4116.363	4005.1355	3600.3472	4089.1465	4086.7332	2980.119	2171.1504	1202.9152	1739.8113
18	6693.6123	6619.584	5150.2163	4886.1772	5070.7227	5724.121	4260.575	3946.3416	3817.8672	3722.1946	3704.2024	2777.139	2274.07	1669.9232	1108.0516
19	6575.9233	6501.495	5315.726	4959.0825	5074.391	5614.713	4425.086	3992.1096	4029.3362	3559.6902	3530.2659	2752.8262	2457.6982	2040.8546	772.082
20	8009.387	7937.7617	5594.542	5695.2227	6093.554	7039.2754	4775.458	4906.062	4179.746	5216.695	5221.457	4030.5085	3007.03	1724.9331	2879.8633
21	7398.4624	7324.427	5726.22	5536.1006	5754.029	6428.979	4843.4614	4614.7495	4355.7	4421.197	4400.6157	3474.4153	2866.2961	1998.4175	1701.1515
22	7266.049	7191.688	5809.84	5544.929	5710.72	6302.327	4920.893	4598.577	4467.8223	4255.952	4227.455	3401.5796	2934.1355	2212.0344	1449.9512
23	7424.9526	7350.714	5856.587	5633.718	5826.8296	6458.3755	4970.77	4700.012	4496.6904	4427.565	4402.3096	3530.3843	2987.3098	2172.3735	1649.9427
24	9639.2705	9570.382	6674.6826	7044.2446	7572.2446	8685.173	5976.9224	6399.847	5330.4673	7000.2583	7016.2505	5733.26	4546.8247	3237.8794	4779.7026
25	9229.437	9159.225	6464.691	6739.7666	7220.4385	8266.617	5718.6343	6036.969	5083.3335	6513.1074	6523.1426	5281.677	4152.644	2830.9314	4196.725
26	8320.273	8247.71	6060.162	6110.2837	6470.5537	7347.2036	5227.332	5287.354	4644.74	5454.714	5450.408	4334.1665	3399.3477	2164.0383	2930.918
27	9300.611	9230.459	6521.9287	6804.328	7288.2134	8338.148	5779.5127	6105.234	5143.0903	6587.067	6597.242	5354.385	4222.2954	2900.709	4269.823
28	8102.4688	8028.837	6164.257	6090.179	6373.034	7130.062	5300.969	5208.8867	4760.544	5156.0386	5140.2554	4141.6914	3375.4436	2284.4236	2469.8179
29	7798.6973	7724.4736	6162.5483	5975.7305	6186.06	6831.794	5281.052	5051.39	4787.1035	4801.3403	4775.619	3896.8513	3305.8994	2396.639	2009.9938

FIGURE 1: Matrix of destinations

User chooses a file with input data which are loaded to the memory and the matrix is computed. Computing of distances is really simple because coordinates are known. It helps remember some of very basic mathematics – Pythagoras's sentence

$$c^2 = a^2 + b^2$$

In this case the distances are computed between two *Points* with coordinates

$$distance(i, j) = (Xi - Xj)^2 + (Yi - Yj)^2$$

where (dXi, Dyi) are coordinates of the first Point and (dXj, dYj) are coordinates of second Point.

The author of this thesis considered whether it would be better to compute a matrix of destinations (even with more claims for memory) before at the importing data or if would be better to compute every destination only when any algorithm will ask. In the author's opinion, current computers do not lack enough memory so there will not be any problem with computing the whole matrix before and saving it in memory. In case a user wants to solve just one problem it is possible that options with computing destinations just on demand should be better, however, in this thesis the assumption is to use input data again and again with different setting and see and compare different solutions.

3.5 Choosing algorithm

After all the aforesaid processes an user can chooses any algorithm for solving this problem.

There are six algorithms to choose from “Algorithms” tab. The user just selects one of them from “combo box” and selects ID of *Point* which will be marked like starting *Point*.

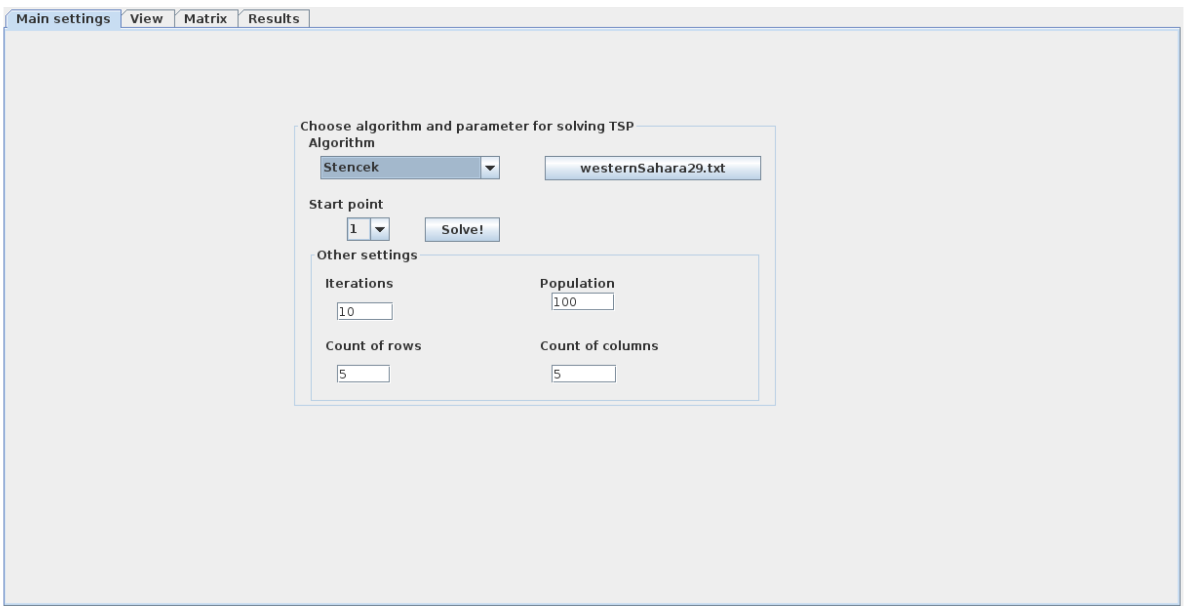

FIGURE 2: Choosing algorithm

After this all steps of algorithm are shown tab “View” with a graphical representation of the final solution and in tab “Results” every other information about the result of algorithm is shown:

- distance of final solution
- sequence of *Points*
- count of *Points*
- time of computing matrix of destinations
- time of importing input data
- time of computing algorithm.

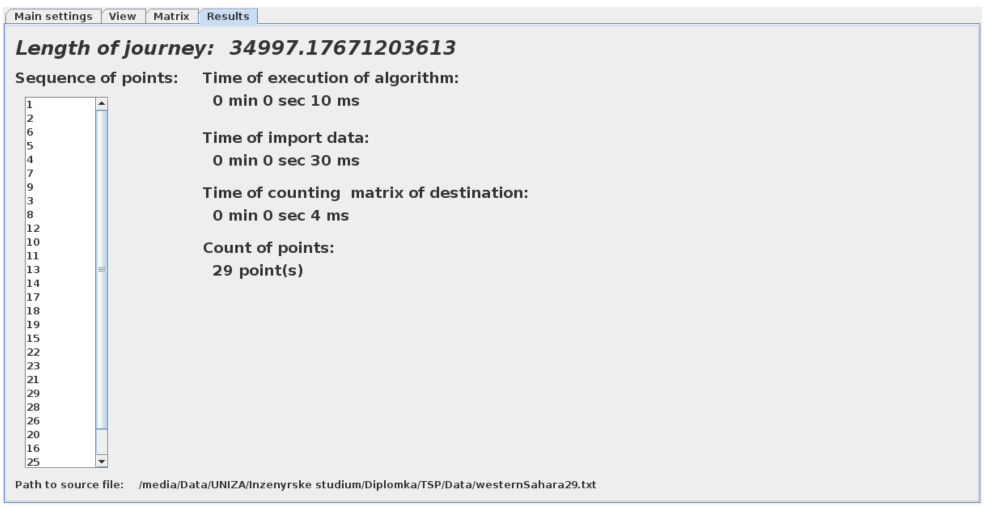

FIGURE 3: Results

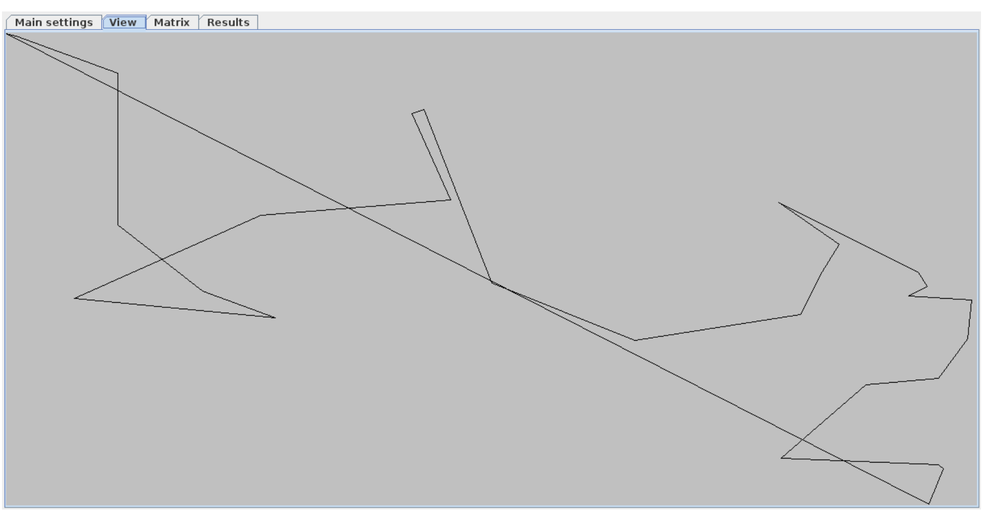

FIGURE 4: Graphical output

4 Implementation of some known solutions

4.1 Chosen sources

This chapter is focused on some of the known algorithms and testing them on chosen examples. All algorithms will be tested with five different input data (for all algorithms will be used same same data) which are attached to this thesis with every source codes:

- Djibuti38.txt – state Djibouti with 38 vertexes
- PMA343.txt – fictitious set with 343 vertexes
- Luxembourg980.txt – Luxembourg with 980 vertexes
- Nicaragua3496.txt – state Nicaragua with 3 496 vertexes
- Canada4663.txt – state Canada with 4 663 vertexes.

The author of this thesis had good reason for this selection. It can be seen the count of vertexes in input data is rising. The author of this thesis hopes it will be interesting to see how implemented algorithms can solve the traveling salesman problem with count of vertexes from a few tens to a few thousands. The author of this thesis thinks that demonstrate performance of algorithms on input data with little bit less than 5 000 vertexes is enough. A bigger problem could need more space in operation memory and not each computer would it sufficed.

In testing of algorithms could be probably occur problem because this is just heuristic algorithms and final solution can be strongly affect with option of first point. This was reason why I decided four different starting vertexes for each experiment and count just the best solution. The chosen vertexes are:

- first vertex of set
- vertex on the border between first and second thirds of set

- vertex on the border between second and third thirds of set
- last vertex of set.

It is little bit more objective then just searching from first vertex or just random vertex. For every experiment is attached time of solving.

4.2 Nearest neighbor

This method was mentioned before in first chapter. It is the basic and probably the simplest method and I think first which I learned on optimization courses in school. This algorithm is very simple, very fast and very easy for implementation. Quality of solution is not so good.

The algorithm starts in starting vertex and looks for the nearest non-visited vertex until all vertexes are visited. It known more way how implement this algorithm in source code, my way is attached under this paragraph as pseudo-code.

```
minimalDistance = Float.MAX_VALUE;
nextPoint;
actualPoint = firstPoint;
matrixOfDistance[][];
solution[];
totalDistance = 0;
for (i = 0, i < sizeOfPointSet, i++) {
    removePointFromFinding(actualPoint)
    for (j = 0, j < sizeOfPointSet, j++) {
      if (minimalDistance >
            matrixOfDistance[actualPoint][j]) {
            minimalDistance =
                  matrixOfDistance[actualPoint][j];
            nextPoint = j;
      }
      solution.add(actualPoint);
      totalDistance +=
```

```
			matrixOfDistance[actualPoint][nextPoint];
		if(i == sizeOfPointSet - 1) {
			totalDistance +=
				matrixOfDistance[actualPoint][firstPoint];
		}
		actualPoint = nextPoint;
		minimalDistance = Float.MAX_VALUE;
	}
}
```

At the beginning are initialized all variables – minimal destination (is set on very big number, bigger then the biggest possible way on matrix of destinations), next point (vertex where will lead next tour), actual point (at the beginning is set on starting point), matrix of destinations, solutions (sequence of vertexes), total distance (set on 0).

The next step is going throw matrix and searching for shortest way from actual vertex to every other non-visited vertex. When is the shortest way known, it is saved as actual shortest possible way, this vertex is saved as next and searching is continued to the end of array of vertexes.

When algorithm is at the end of array, actual vertex is added to the solution, actual vertex is excluded from vertexes which can be visited in next round (distance from this vertex to every other vertex is set to maximal possible value), as actual vertex is set next vertex, minimal distance is initialized on maximal possible value. If it is last round of algorithm will be added one more way – from last vertex of sequence to the first and algorithm is done.

TABLE 2: The results of Nearest neighbor algorithm

	Start point	Length of tour	Duration
Djibuti38.txt	38	7941	0.0s
PMA343.txt	230	1836	0.01s
Luxembourg980.txt	327	14 590	0.035s
Nicaragua3496.txt	2334	119 866	0.527s
Canada4663.txt	4663	1 633 565	1.05s

4.3 2-approximation algorithm

The basic assumption for solving traveling salesman problem is this algorithm is the metric case of traveling salesman problem. This heuristic solves problem of traveling salesman problem in polynomial time (Algorithms).

Procedure:

1. Construct minimal spanning tree
2. Apply "deep search first algorithm"
3. Reduce ways – skip all duplicity vertexes, which originate from "deep search first" algorithm.

Minimal spanning tree

In theory of graphs minimal spanning tree of the coherent graph G is its sub-graph on set of all vertexes which is tree. This tree has to contain all vertexes of the original graph G and contains minimal possible count of edges just like every two vertexes have to be connected with way (edge is not the same as way! - way means sequence of edges). Minimal spanning tree does not contain any circle. The word *minimal* means minimal possible sum values of edges used in minimal spanning tree.

For construction minimal spanning tree Kruskal's algorithm was used.

Kruskal's algorithm

This algorithm is used to search minimal spanning tree. The algorithm was developed and presented by Joseph Kruskal in 1956 (Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein).

The principle of the Kruskal's algorithm is relatively simple. The first step is to sort every edges in graph by its value (in case of traveling salesman problem – length) from smallest to largest. The second step is to add sorted edges to the graph there must not be any circles. It follows this algorithm a new graph will contain *count of vertexes -1* edges. The example of Kruskal algorithm is under this paragraph.

```
SpanningTree kruskalAlgorithm(Graph g, weights w)
   SpanningTree tree
   for Node : n in graph do
     makeSet(n)
   List edges = sort(graph.getEdges(), w)
   for Edge e in edges do
     if findSet(e.start) != findSet(e.end)
          tree.add(e)
          union(e.start, e.end)
          if tree.edgeCount() == graph.nodeCount() - 1
               break
return tree
```

Really necessary is to ensure the new graph is acyclic. There is used data structure

“Disjoint set” which keeps information about vertexes. This structure uses two basic operations:

- union – unite two components
- find – find relationship to any component

Time complexity is in case of use sorting algorithm based on comparing $O(|H|*\log|H|)$. In case the edges are pre-sorted of if it is possible to use some algorithm with linear complexity for sorting could be complexity of this algorithm $O(|H|*\alpha(|H|))$, where α is inverse function to Ackermann's function.

Ackermann's function

This function is an example of function which is recursive and at the same is not primitively recursive. The value of this function grows fast even for very small numbers (e.g 4, 5, 6, ...) and it is practically impossible (or very difficult) compute it. E.g. A(4) is so large number then count of its numerals is bigger than count of atoms in known universe. Ackermann's function grows over all possible bounds and cannot be limited with any known and used function.

Because this function grows extremely fast its inverse grow extremely slow. Whereas the Ackermann's function is even for A(4) practically unimaginative $A^{-1}(n)$ is for every possible and imaginable values *n*. For every practical purpose this inverse function can be deemed as constant function. This inverse function is usually used to analyze the complexity of some algorithms e.g. Kruskal's algorithm (Paul E. Black).

Disjoint-set

This data structure is usually used to creating system of non-overlapping sub-sets. For implementation are used 2 basic operations:

- union
- find.

This data structure found its use to solve problems where two vertexes are in the same sub-set (component of the context) of the graph G which is used in the Kruskal's algorithm for creation minimal spanning tree and many others.

For representation is usually used an n-ary tree where root is representative. Assuming combines two components with *union* operation shorter sub-tree is hanged under representative next tree. In case of equality of lengths does not matter which tree will be hanged. In case looking for the representative with operation *find* algorithm goes up from given vertex to the root of the tree which is the searched representative.

This algorithm can be optimized with hang browsing vertexes after operation *find* under the representative – way will be shorter. After this update should be complexity little bit worst but it happens only once and in next searches will be this algorithm faster than before. These operations have complexity $O(\alpha(n))$ where α is inverse function to the Ackermann's function.

Deep search first

This algorithm is usually known as DSF algorithm. It is used for browsing in graphs and has wide use. Its principles are used for detecting count of components, topological sorting or detecting circles in graph.

Algorithm chooses random vertex and marks it is as open. After that algorithm processes this vertex and calls itself on every still undiscovered offspring of this vertex. After return from recursion this algorithm marks itself as closed. This way every branch in graph will be browsed to maximal depth.

It is possible to approach neighbors of given vertex in constant time complexity of this algorithm is $O(|U|+|H|)$ because this algorithm goes throw every vertex and edge right at once.

TABLE 3: The results of 2-approximation algorithm

	Start point	Length of tour	Duration
Djibuti38.txt	14	8552	0.004s
PMA343.txt	1	1809	0.098s
Luxembourg980.txt	980	17 452	0.223s
Nicaragua3496.txt	1166	144 521	4.573s
Canada4663.txt	4663	1 758 540	11.228s

4.4 Christofides algorithm

As in previous algorithm also in this algorithm the basic assumption is it must be the metric case of traveling salesman problem. This heuristic can solve traveling salesman problem with solution does not worst then1.5 multiple of the optimal solution (Algorithms).

The procedure of solving traveling salesman problem with Christofides algorithm:

1. Construct minimal spanning tree
2. Find all vertexes which have odd count of adjacent edges
3. Construct minimal perfect matching on complete graph from vertexes from step two
4. Add edges from minimum weight perfect matching to minimal spanning tree
5. Find the Eulerian path.

For find a minimal spanning tree was used the same procedure as in previous algorithm.

Find vertexes with odd count of edges was not so hard. There are many possible way to find these vertexes but important is only result. In this thesis is used simple going throw *algorithm* which checks every vertex in array and reckons count of edges for all vertexes.

In application is not necessary to construct complete graph between vertexes with odd count of edges. The indexes of these vertexes are just saved in array and every distance could be simple get from matrix of destinations.

Paring in graph

Paring in graph is in theory of graphs called sub-graph of graph G where any two edges can have no common vertex.

Maximal matching in graph

Maximal matching in graph G is matching which contains maximal possible count of edges and preserves condition any two edges can have no common vertex.

Perfect matching in graph

Perfect matching is graph is special case of paring which contains every vertexes from graph G and preserves condition any two edges can have no common vertex. From this definition is obvious perfect matching can be found only on graphs with even count of vertexes and then perfect matching is simultaneously maximal matching.

Find maximal or perfect matching is basically very simple problem but find minimum weight perfect matching is very complicated problem.

First attempt to find minimum perfect matching was procedure which tried out every possible permutation pairs of vertexes. This is probably the easiest procedure and implementation was not difficult at all. But there was huge problem with time consumption. This could solve traveling salesman problem with seven vertexes in a few second. The problem with 29 vertexes was solving for more than 4 minutes! So this way of finding minimum perfect matching was rejected. The better algorithm could be the Edmonds algorithm which can find minimum weight perfect matching in polynomial time.

Edmonds algorithm

Know Edmond's blossom algorithm from 1965 can solve problem of minimum weight perfect matching in polynomial time. Direct implementation of this algorithm has complexity $O(n^2 m)$ where *n* is count of vertexes and *m* is count of edges in graph. Lawler and Gabow gained time of solving $O(n^3)$ which after them Galil, Micali and Gabow improved on $O(nm \log n)$. Next value of improvement was $O(n(m \log \log \log_{max(m/n,2)} n + n \log n))$ by Gabow, Galil and Spencer. In this time is the best and the most know possible time complexity (related *n* to *m*) $O(n(m + \log n))$ by Gabow (Benova, I. 2010).

Computer implementation of blossom algorithm started with *Bloosom I* by Edmonds, Johnson and Lockhart. It was followed by *Blossom II* and *Blossom III*. *Blossom IV* by Cook and Rohe was the fastest implementation of blossom algorithm for many years. The newest *Blossom V* by Vladimir Kolmogorov from 2009.

Implementation of this algorithm is extremely difficult problem. This was the topic for master thesis on our faculty a few years before so this suggests how demanding task it could be. Exist many sources from where could be borrowed implementation of this algorithm but for this thesis Edmonds algorithm is not mandatory so here could be used simpler heuristic which does not find minimum weight perfect matching but finds perfect matching with relatively satisfactory value of solution in satisfactory time. Every edges are sorted in dynamic array by value of length from shortest to longest. The heuristic goes throw array edge by edge and selects edges until finds *count of vertexes / 2* edges. In selection of edges is checked if some of vertexes was used if not edge is added to array of perfect matching else the edge is deleted from array and the heuristic continues to next edge. Final solution of this heuristic is not probably optimal but for simple demonstration of Christofides algorithm is sufficient.

Minimal spanning tree is represented by array so operation adding new edges from previous heuristic is not difficult. After this operation is next step to find Eulerian path.

Eulerian path

In the theory of graphs is as Eulerian path called path in graph which contains each edge right once, in layman's terms if is possible draw this graph in one move. In 1736 (Swiss mathematician and physician) Leonhard Euler instituted this term. This term was allegedly arise when he tried to solve famous problem “Seven bridges of Königsberg” - Königsberg is town on river Pregola which creates two islands (can be seen at figure 5). The islands were joined with rest of town with seven bridges. The question is “is possible go throw all of bridges and get yourself back to the start point and use each bridge right once?”. Leonhard Euler was first man which proved that it is impossible because in appropriate graph does not exist the *Eulerian path*.

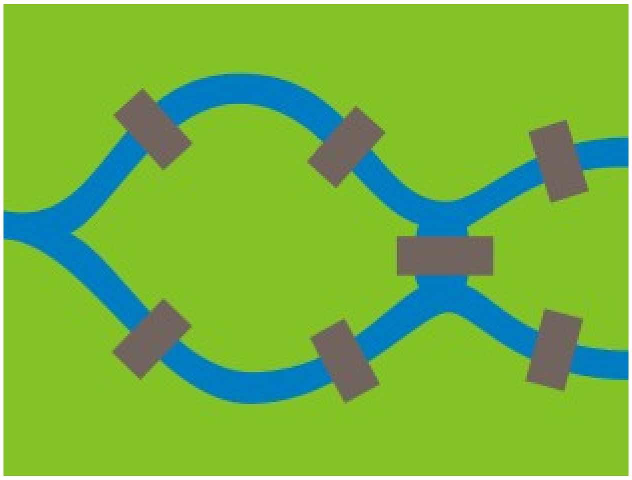

FIGURE 5: Seven bridges of Königsberg

If in a graph exists the Eulerian path the graph is called Eulerian graph.

Everybody knows old game where is drawn “house” with one draw (move). But probably nobody knows that main target of this game is to find *Eulerian path*.

In a graph exists Eulerian path if:

- graph is unoriented, coherent and each vertex has even count of edges (Eulerian path starts from any of them)
- graph is unoriented, coherent and contains of right two vertexes which have odd count of edges (Eulerian path starts in one of them)
- graph is unoriented, coherent and is possible spread it on edge disjoint cycles
- graph is oriented and each vertex has equal count of input and output edges.

Is is so obvious that in our case after construction of minimal spanning tree and perfect matching result graph will be Eulerian graph. After construction of minimal spanning tree left even count of vertexes with odd cont of edges eventually this count could be null. And after construction perfect matching will be added right one edge to each vertex with odd count of edges. This procedure secure that each vertex has even count of edges.

Eulerian graphs have very useful property – when is some circle deleted from Eulerian graph rest sub-graph will be still Eulerian if this graph is still coherent. In case the graph is not coherent anymore and is fallen apart on two or more fragments it is violates condition for Eulerian graphs but for purpose of this algorithm is enough that fragments is still Eulerians graphs.

In this thesis is used *deep search first* algorithm for searching sub-circles and ultimately Eulerians path. This algorithm was used in previous 2-approximation algorithm but there was used for searching on vertexes and in this algorithm for searching on edges. Even operations for going throw were regulated (see below) but main point of deep search first is still the same. The example of *deep search first* algorithm is attached under this paragraph.

```
stack.push(first);
visitedEdge[first] = true;
while (!stack.isEmpty){
	actual = stack.peek();
	next = findNextEdge(actual);
	if(next != null) {
		visitedEdge[next] = true;
		stack.push(next);
	} else {
		stack.pop();
		eulerPatjAdd(actual);
	}
}
```

The principle of deep search first algorithm itself is simple. This algorithm uses data structure *Stack*. The main point of this structure is storing records in order how they were added. There is only one rule for getting records out – **last in, first out** (LIFO). This algorithm start from starting point, selects first edge which contains starting point and adds it to the stack. For this edge and each other is set which vertex is input and which is output. Actual edge is marked as visited. Next is selected edge which contains the same vertex as was output vertex previous edge, this edge is added to the stack and marked as next. In this way algorithm continues until does not exist any non-visited edge where is possible to go. After that edges in stack are selected one by one and algorithm searches for non-visited edges. If algorithm does not find any non-visited edge then continues in standard deep first search. If algorithm push out every edges from stack it means each edge was visited and algorithm terminates.

The solution = the Eulerian path saves edges in order how they were pushed from stack. This Eulerian path is just in reverse order. Now it is necessary just this order reverse and get solution of Christofides algorithm from starting point.

TABLE 4: The results of Christofides algorithm

	Starting point	Length of tour	Duration
Djibuti38.txt	26	6770	0.024s
PMA343.txt	1	1671	0.206s
Luxembourg980.txt	326	13 754	9.964s
Nicaragua3496.txt	1	115 010	50min 42s
Canada4663.txt	1	1 512 161	17min 27s

4.5 Genetic algorithm

The last usually used algorithm implemented in this thesis is a simple genetic algorithm. As previous algorithms also this one is just a heuristic algorithm so optimal solution is possible but not guaranteed.

Generally genetic algorithms are relatively popular way how to find solution for many different complex solution. Genetic algorithms are usually fast and powerful but with more complicated implementation. A genetic algorithm tries to simulate genetic principles of evolution to find solution of a problem. Different genetic algorithms are able to solve even really complicated problems (Genetic algorithm).

The basic idea and procedure of genetic algorithm:

1. Initialization – made first population which is usually generated randomly. This population can have any size – from a few to millions
2. Evaluation – each population is evaluated = there is computed so-called fitness function of given solution. The purpose of this function is to find out to how extent this solution fulfills given requirements. This requirement can have different form – the fastest computing as possible, the best solution as possible etc.
3. Selection – the purpose is to improve fitness value of population. So it is important to select just population which is the right pattern for find the best solution => principle of evolution, only the strongest individuals can life. There are many methods of selection but the basic idea is still the same – selection of the best candidates for making the best possible future generation
4. Crossover – this operation makes new population by making hybrid of two selected populations – they can be called *parents*. The basic idea is to combine the best attributes of each parent.
5. Mutation – this operation makes the procedure of making new generation little bit random. It is important for possible improvement.
6. Repeat! - generate new generation and continue from step two.

End of the algorithm

There could be many reasons to end this algorithm. It depends on type of problem which algorithm solves and how good solution is expected. Genetic algorithm usually ends:

- after given count of iterations
- after given time of solving
- after achieve given solution.

The solution of genetic algorithm heavily depends on defined of:

- count of populations
- count of iterations.

Count of iterations means how many times is algorithm repeated. First run has as input parameter random value but every next starts from the best founded solution from previous iteration.

Count of populations means the size of arrays for development stages of algorithm. The more populations means more space to algorithm for developing better solution.

In this thesis user can choose this values. The choice of this values is really important not just because of quality of solution but even because of computing time. When is chosen less count of population probably the algorithm will not have enough time for developing and the solving will not have enough time for mutations and finding for satisfied solution. But on the other hand if is chosen larger count of populations the algorithm can solve the problem very long time. As well the count of iterations has to be chosen reasonably. When problem has tens or hundreds vertexes the time of solving is not so big even with thousands of population and iterations. When a problem has thousands of vertexes and thousands of populations and iterations it could take really long time.

As mentioned before the quality of solution heavily depends on chosen value of populations and iterations. So because of the same conditions for solving all testing data were chosen values one hundred iterations and three hundreds populations. First try was made with much bigger values but the time of solving was more than hour on the problem with 343 vertexes. It means to solve problem with 4663 vertexes (Canada) should take very long time. But for needs of this thesis and demonstration of this algorithm is enough use just smaller values of populations and iterations and could be interesting to see how this algorithm can solve large problems just with “a few” iterations and populations.

In this algorithm is not necessary to set (for computing) starting point. Operations crossover and mutation would probably change it. Because of this reason the starting point is set before start of algorithm but there is not computing with it and after algorithm is done a simple procedure processes a final tour and adjusts it form starting point to other. For use of this algorithm is starting point for each testing input data set vertex number one.

It should be borne in mind even when this genetic algorithm runs twice with the same data and with the same conditions the results will probably not be the same. It is because in this algorithm works a certain degree of randomness – mutation and crossover.

TABLE 5: The results of Genetic algorithm

	Length of tour	Time
Djibuti38.txt	7 337	3s
PMA343.txt	24 120	3 min 9s
Luxembourg980.txt	265 992	24min 9s
Nicaragua3496.txt	4 910 493	308min 11s
Canada4663.txt	11 868 417	518min 39s

4.6 Exact algorithm

As mentioned before the optimization algorithm which can solve traveling salesman problem in polynomial time does not exist but there is a way how to get optimal solution. This way is “exact algorithm”. This algorithm goes through all possible tours in graph. From table one is obvious the count of this tour can be very large even in relatively small problem. This way can be acceptable for very small problem, for larger absolutely not!

It can be really interesting to see how much time take solving a problem with exact algorithm compare with previous heuristics – time and quality of solution.

E.g. there is the problem with 14 vertexes. To look for solution with exact algorithm means go throw 14! possible tours, for interest it is $87 * 10^9$ tours. Here is little point of interest – the tour 1-2-3-4-5-6-7-8-9-10-11-12-13-14-1 is basically the same as tour 2-3-4-5-6-7-8-9-10-11-12-13-14-1-2 etc. So here is possible cut little bit of complexity from *N!* with simple adjustment to *N-1!*. It is not so big help but in larger problems could be helpful. This adjustment consist of choose starting vertex, set it as first of tour and go throw *cont of vertexes – 1* possibilities. It is true if a problem has six vertexes it is not so big different to compute 5! or 6! possibilities. But if the problem has 14 vertexes the different of time can be known.

Below is a few testing examples for demonstrate time and quality of solution of implemented algorithms.

For algorithms Nearest neighbor, 2-approximation and Christofides was set vertex number one as start of the tour. For genetic algorithm was not set starting point (see chapter 3.4) and value of population and iterations were set on the five times respectively fifty times of number of vertexes.

TABLE 6: The results of Exact and other implemented algorithms

	Near. neighbor		**2-approxim.**		**Christofides**		**Genetic**		**Exact**	
8	358.24	0.0s	358.24	0.0s	358.24	0.275s	312.26	0.404s	312.26	0.087s
10	409.53	0.0s	409.53	0.0s	409.53	0.002s	351.44	0.301s	351.44	0.047s
12	435.3	0.0s	435.3	0.0s	435.3	0.001s	361.24	0.464s	361.24	5.39s
14	445.85	0.0s	392.48	0.0s	401.9	0.001s	366.12	0.731s	366.12	15min
16	448.57	0.066s	488.6	0.172s	392.95	0.003s	367.25	0.987s	?	> 36h

In table six can be seen the time consumption of exact algorithm is huge. The test with 16 vertexes with exact algorithm was not done because even after 36 hours the algorithm did not find solution (according the calculations it could take more then 53 hours). On the contrary by surprise the genetic algorithm because the values of the solutions look quite good against the other heuristics and exact algorithm. It is possible to see that the genetic algorithm is very powerful for solve smaller problems. It is really interesting to see how the heuristics can keep up with exact algorithms in smaller problem in terms of quality of solutions. But here is impossible compare exact algorithm with the others in sense of time consumption. The time claims of exact algorithm are really unacceptable.

5 Stencek algorithm

An inspiration for this algorithm (for the needs of this thesis it can be called *Stencek algorithm*) was the genetic algorithm from the previous chapter. The main point of *Stencek algorithm* is to cut the set of points to smaller pieces. From previous result can be seen that smaller problems can be solved a relatively effectively and in relatively short time.

Steps of the Stencek algorithm:

- cut set of points on smaller pieces (chess field style)
- find minimal spanning tree in each piece of original set
    - find way with deep search first algorithm
    - check if the longest way in tour is longer than the way between first and last vertex, if it is then delete the longest way and add the way between the first and last vertex to the tour
- join every pieces of original set with genetic algorithm.

As everybody can see this is not an optimization algorithm, but it can save much computing time and get solid result.

The sense of cut of set of vertexes is obvious – solving of smaller problems is much easier and takes much less time.

Find of minimal spanning tree was used even in previous algorithms and the results of this heuristic is quiet good and this way of solve can has good potential. Even big potential if this method is upgraded with some another heuristic.

Probably the best choice how to find the possible solution of the connection between fragments of original set should be genetic algorithm – of course after a necessary upgrade. It can be seen on testing examples that the genetic algorithm can effectively solve problem with tens or a few hundred vertexes. If a problem will be cut on ten times ten fragments i.e. one hundred fragments the genetic algorithm should be able to find very good solution in a few tens of seconds or maximum of a few minutes. This solution can be even optimal or very close to optimal – of course it does not mean optimal solution of the whole problem but "just" problem of looking for the best possible connection between fragments. This statement implies from previous experiments on test example on *Djibuti38.txt* where this genetic algorithm found solution which was approximately 122% of optimal solution in a few seconds with relatively small numbers of populations and iterations. It is obvious if in this problem would use a bigger number of this values the solution can be much closer to optimal. Of course the bigger number of this value means longer time of solving but it is still really remarkable. This algorithm should be very effective and fast if the count of fragments will be a few tens or maximum of a few hundreds.

Minimal spanning tree

This method was used in previous chapter in implementation of 2-approximation algorithm. In this case was used Kruskal's algorithm as before. The main reason is relatively easy implementation and really short time of computing.

Find the tour in each fragment

- Deep search first
    - This method is very fast and effective and was used in previous chapter.
    - Improving heuristic:
        - in many fragments the longest way between vertexes in tour was longer than the possible way between first and last point of the tour. If this case

happened the longest way is deleted and the way between first and last vertex is added to the tour.

In the sense of saving computing time for each fragment the distance is counted immediately after creation and in procedure of connecting all of fragments only the length of way of connections is counted. This is one of the differences between the original genetic algorithm and this one.

The fitness function was counted in original genetic algorithm as *1 / length of whole tour*. It means the most useful tour is the tour with the lowest value. In this case fitness function is counted similarly but with a difference that it is here divided by the *length of connection between fragments*.

Connecting fragments

For this operation many algorithms can be used. Probably the best should be Edmond's algorithm, mainly because this algorithm can find the minimal weight perfect matching in graph in a very short time. But in this case it would be probably very difficult to secure to prevent the sub-cycles. This algorithm was not used because the implementation is extremely difficult.

The author of thesis decided to use the genetic algorithm. It was necessary to adjust this algorithm.

Adjustments against the original genetic algorithm

- **Generation of random tou**r
    - the original algorithm was generated just with a simple shuffle of array with

vertexes. In this case it is necessary to secure not to do any sub-cycles. Here effective algorithm was used which works infallibly and does not need any control. The principle of this algorithm is not so difficult (see illustration number six). One vertex is chosen randomly, this vertex is excluded from the next search with its “neighbor” from the same fragment and this vertex is connected with a randomly chosen vertex from other fragments and in this way the algorithm continues until there are not any usable vertexes in the array. When there are not any usable vertexes in array and still some vertexes are not connected, the unused vertexes are added to the new array in the order how they were excluded from searching and the algorithm continues just like in previously until all vertexes have a connection. This algorithm surely generates a tour without sub-cycles.

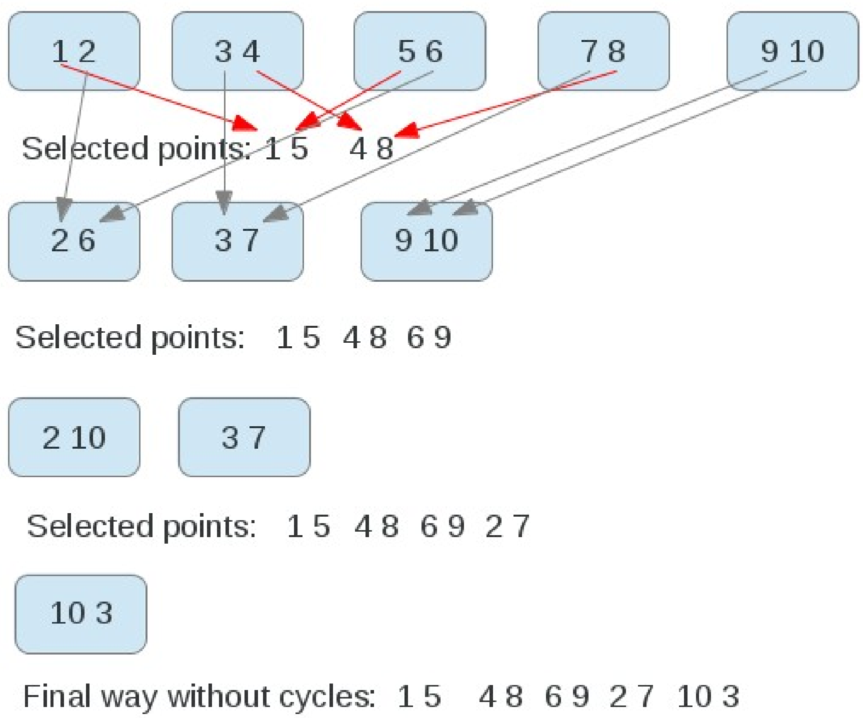

FIGURE 6: Generation of the random way without sub-cycles - red arrows mean random choose of vertexes, gray mean sorted of pairs of vertexes

- **Mutation**
  - in the original genetic algorithm mutation works on the base of randomly chosen vertexes which switch their places in the array. In this case it is impossible because it will probably create sub-cycles. For this reason the way of switching vertexes is used just in a random fragment (see Figure 7).

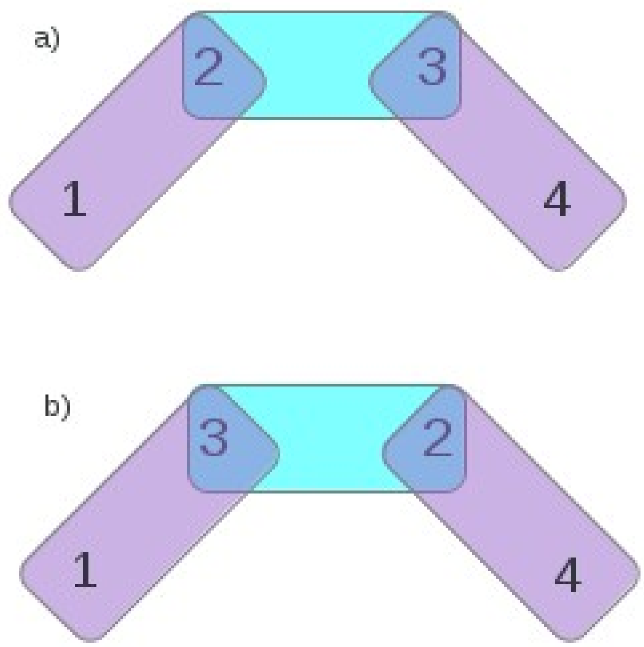

FIGURE 7: Mutation - a) before mutation, b) after mutation, blue is the fragment, purple is the edge

- **Crossover**
  - in the original genetic algorithms were joined in two good enough ways (cut the first and complete it with the second), however, it is not possible in this case. This procedure had to be much adjusted. The main point is quite similar the procedure of generating a random way. The edges from the *good enough tour* are added in first step and to them is added the rest of vertexes according to the rules from generating of the random tour (see Figure 8).

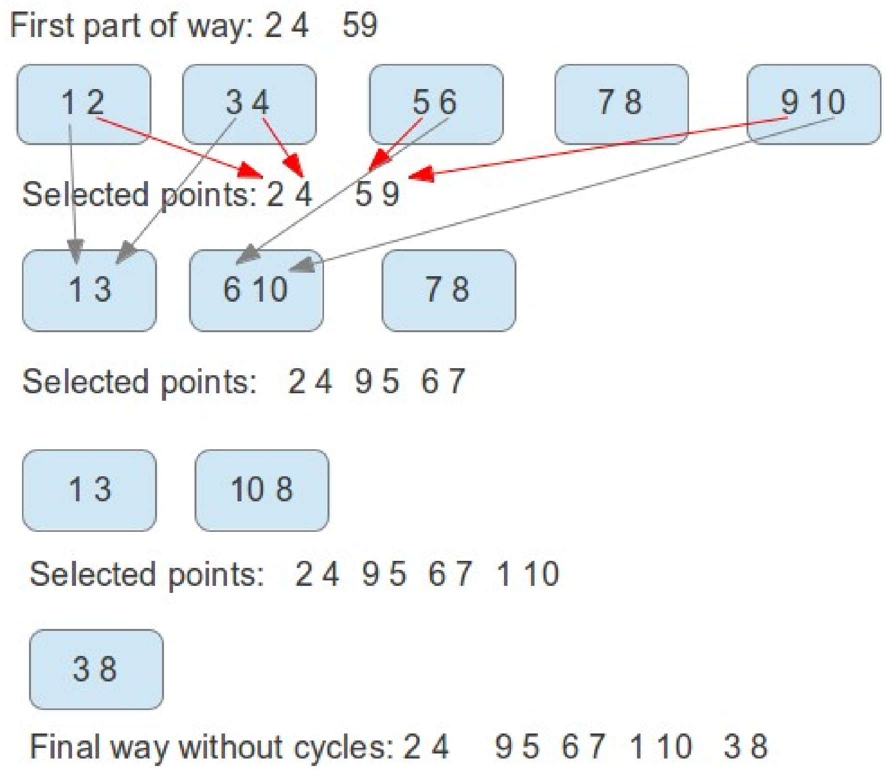

FIGURE 8: Crossover - red arrows mean choose of the vertex from original tour, gray arrows mean new arrangement of pairs

All of this operation costs some computing time but this time is still a bearable.

The real benefit and the power of genetic algorithm was known mainly in the problems with maximum of tens of vertexes because in the designed software there is the condition of maximal and minimal count of fragments (minimal is two time two and maximal is twenty times twenty). In maximal case the theoretical count of vertexes is 800 (400 fragments with 2 vertexes). This count is extremely large and to find an acceptable solution is necessary in order to do many iterations with many populations and it could take very long time. The recommendation is to choose the count of fragments sensitively by the distribution of the vertexes.

TABLE 7: The results of Stencek's algorithm

	Length of tour	Time
Djibuti38.txt	9756	19.868s
PMA343.txt	3130	30.867s
Luxembourg980.txt	20 822	28.491s
Nicaragua3496.txt	168 968	29.314s
Canada4663.txt	1 894 201	2min 31s

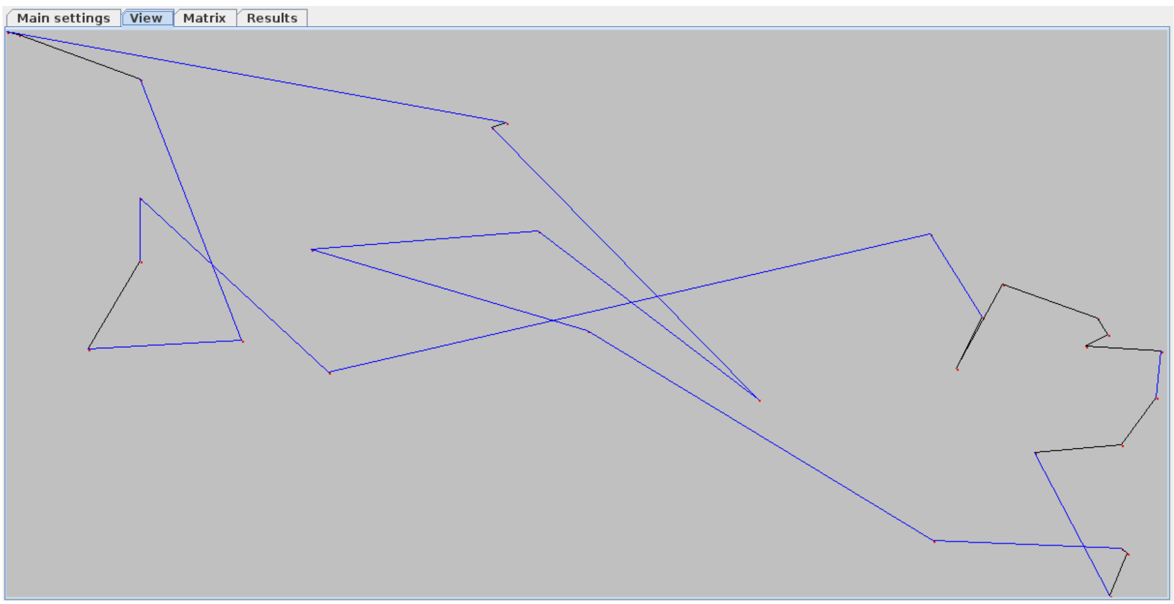

FIGURE 9: Graphical output of Stencek's algorithm. Blue color means connections between fragments, black means way inside each fragment. This example is problem with 29 vertexes and it is divided into five times five fragments. In some fragments there is only one vertex and in some there is even no vertex.

6 Conclusion

This thesis showed how complicated *Traveling salesman problem can be*. There are many ways which can more or less effectively solve this problem; however, none of them can surely find the optimal solution.

Every algorithm was tested on notebook *Lenovo ThinkPad Edge E520* with parameters:

- processor: Intel® Pentium(R) CPU B940 @ 2.00GHz × 2
- operation memory: 4 GB, 1333MHz
- operation system: Fedora 18, 64bit

The solving was assigned to JVM (Java Virtual Machine) 1536MB of memory. The memory consumption was of course different for each algorithm. This memory consumption was not so huge and there is no problem to solve a problem with thousands of vertexes. It is not probably surprise the least of memory was taken by *Nearest neighbor algorithm*. Right after it was *2-approximation* and *Christofides algorithms*. The *genetic* and *Stencek's* were little somewhat more demanding mainly because for solving them it was necessary to use more complicated data structures and a great deal of temporary data. It is unnecessary to write about memory consumption of *Exact algorithm* because this algorithm could solve the problems with just a few vertexes.

Exact algorithm was shown mainly for demonstration of how extremely time demanding it is for current powerful computers. This algorithm is totally inappropriate for use in real situations.

The algorithm **Nearest neighbor** was a pleasant surprise. Despite the very simple process and implementation the solutions were not so bad. In some cases this algorithms could give much better solutions than much more sophisticated algorithms. A huge advantage is as well the minimal memory and time consumption. It can be used like starting heuristic or with some sophisticated heuristic in real situations.

2-approximation algorithm was a huge disappointment. The implementation was much more demanding than *Nearest neighbor algorithm* but the solution was not so good. However, this algorithm has a great advantage because the user can be sure the solution will be maximum of 200% in comparison with the optimal solution. This algorithm is not recommended for use in real problems but as starting heuristic it can be really beneficial.

Christofides algorithm was the most pleasant surprise. This algorithm could find a very good solution in very short time even with handicap no minimal weight perfect matching but just with perfect matching with *good* value of weight. With implemented Edmond's algorithm it can give really good solutions. The way of *good weight perfect matching* slows down the speed of this algorithm because the simple sorting of array is used there. Edmond's algorithm is able to solve it much better and much faster, however, for the purpose of this thesis it is definitely a sufficient solution.

The hugest disappointment was **genetic algorithm**. According to the articles and other materials much better solutions and much smaller time consumption were expected. This algorithm is really able to find solution very close to optimal; however, it costs a very long time because it is necessary to set the values of populations and iterations on very large numbers which means large memory consumption as well. It should not be forgotten that it was just a basic implementation of the genetic algorithm. There are much more possible ways how to make a better genetic algorithm. This algorithm can be really beneficial for solving real problems.

The **Stencek's algorithm** was light disappointment. The solutions of the tested examples were not so bad in comparison with other algorithms but much better solutions were expected here. This algorithm can be improved with some other algorithm e.g. find the optimal way in each fragment or choose some better way to divide the set of vertexes. It is obvious this algorithm cannot find (now even not in future with additional improvements) the optimal solution of the traveling salesman problem. However, this algorithm is able to find a relatively sufficient solution of this problem.

TABLE 8: The summary of solutions

	Djibuti38.txt	PMA343.txt	Luxembourg980.txt	Nicaragua3496.txt	Canada4663.txt
NS	7941	1836	14 590	119 866	1 633 565
APROX	8552	1809	17 452	144 521	1 758 540
CHRISTO	6770	1671	13 754	115 010	1 512 161
GENET	7 337	24 120	265 992	4 910 493	11 868 417
ŠTENCEK	9756	3130	20 822	168 968	1 894 201
OPTIMUM	6656	1368	11 340	96 132	1 290 319

References

Algorithms, www.algoritmy.net/article/5407/Obchodni-cestujici, 4. 3. 2013

Benova, I. 2010, The minimum – weight perfect matching in huge graph, Zilina

Engineering, The Hong Kong Polytechnic University , Hung Hom, Kowloon, Hong Kong

Gatech, www.tsp.gatech.edu/, 1. 2. 2013

Genetic algorithm, http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3, 7. 4. 2013

Gmu, iris.gmu.edu/~khoffman/papers/trav_salesman.html

Integration, Modeling and Genetic Algorithm, Department of Industrial and Systems

Kylie Bryant, 2000, Genetic algorithms and the TSP, Harvey Mudd College

Laporte, Gilbert, 2006, A short history of traveling salesman problem, http://neumann.hec.ca/chairedistributique/common/laporte-short.pdf, 2. 3. 2013

Michael Hahsler & Kurt Hornik, TSP - Infrastructure for the Traveling Salesperson Problem

Paluch, S. 2008, Algorithmic theory of graphs, Zilina

Paul E. Black, "Ackermann's function", in *Dictionary of Algorithms and Data Structures*, *U.S. National Institute of Standards and Technology* , 7. 4. 2013

Ping Ji & William Ho, The Traveling Salesman and the Quadratic Assignment Problems:

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein, Introduction to Algorithms

Plesník, J. 1983, Graph algorithms, Veda, Bratislava

Second Edition, The MJT Press, Cambridge, Massachusetts

Shen Lin, 1965,Computer Solutions of the Traveling Salesman Problem, Manuscript

Ubuntu, www.ubuntu.com

Wiki Ubuntu, http://wiki.ubuntu.cz/java, 4. 4. 2013

Wiki Ubuntu, http://wiki.ubuntu.cz/programování/netbeans, 4. 4. 2013