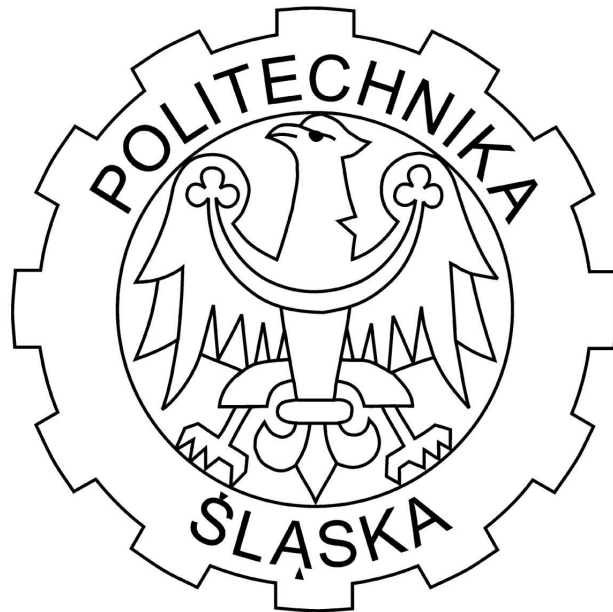


Politechnika Śląska w Gliwicach
Wydział Automatyki, Elektroniki i Informatyki



Biologically Inspired Artificial Intelligence

Topic: Travelling Salesman Problem

skład sekcji	Michał Jakóbczyk Marta Miler
prowadzący	dr inż. Grzegorz Baron
rok akademicki	2017/2018
kierunek	Informatyka
rodzaj studiów	SSI
semestr	6
grupa	GKiO3
data oddania sprawozdania	2018-06-17

1. Topic

Topic of the project is the travelling salesman problem.

2. Problem description

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

3. Used tools and technologies

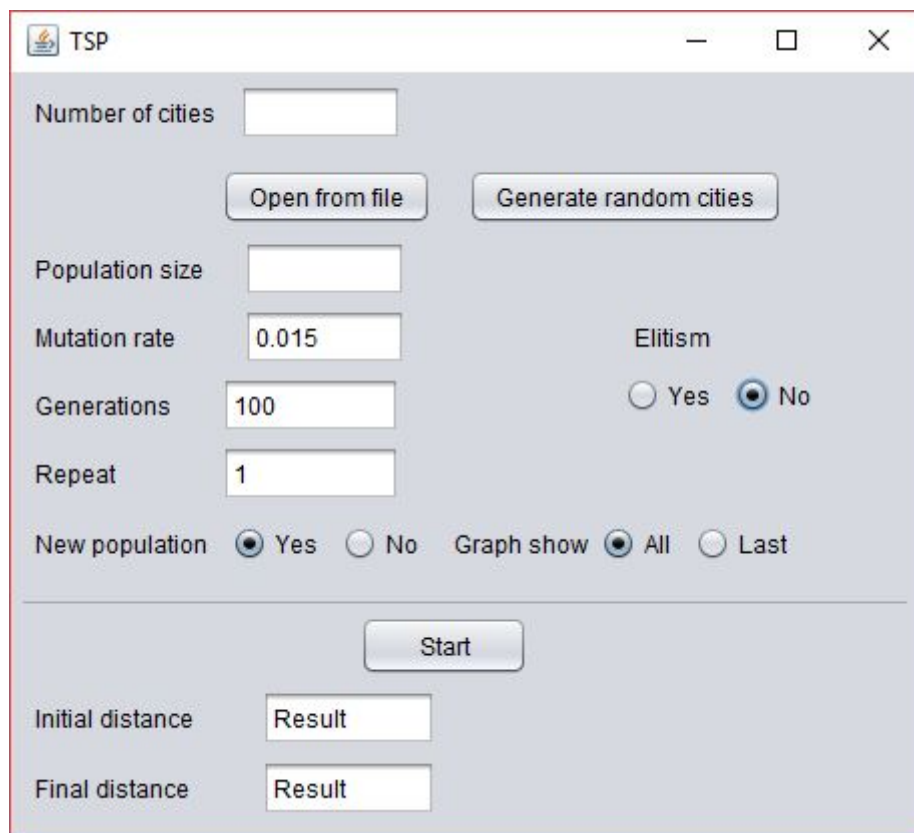
To create application we selected following technologies:

- Java to create algorithm
- Swing to create graphical user interface

We used following tools:

- Netbeans IDE

4. Extern specification



The screenshot shows a Java Swing window titled "TSP". The interface includes several input fields and buttons. At the top, there is a "Number of cities" input field, followed by "Open from file" and "Generate random cities" buttons. Below these are "Population size", "Mutation rate" (set to 0.015), "Generations" (set to 100), and "Repeat" (set to 1) input fields. To the right of the "Mutation rate" and "Generations" fields is the "Elitism" section with "Yes" and "No" radio buttons, where "No" is selected. Below the "Repeat" field are "New population" and "Graph show" sections, each with "Yes" and "No" radio buttons. "Yes" is selected for both. A "Start" button is centered below these sections. At the bottom, there are "Initial distance" and "Final distance" labels, each followed by a "Result" input field.

Photo: Graphical User Interface of the application.

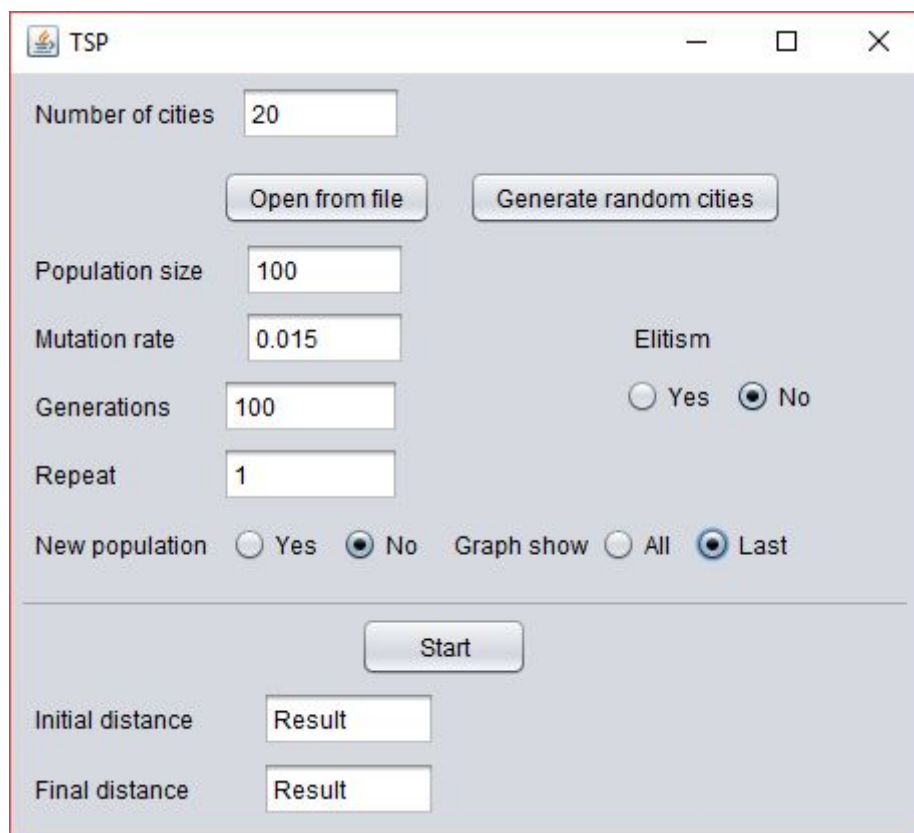
The picture presents program's graphical user interface.

First, we need to generate cities or open them from file. To generate random cities, we need to fill "Number of cities" field with chosen amount of cities and click on the "Generate random cities" button. To open them from file, we need to click on the "Open from file button" and then select file from dialog.

After choosing cities, we need to fill rest of fields:

- Population size - amount of population (Population means random series of cities)
- Mutation rate - factor that determines how often mutation will appear
- Generations - number of repetitions of the genetic algorithm on the population
- Repeat - number of repetitions of the program
- New population
 - Yes - generate new population each time,
 - No - generate the population once
- Graph show
 - All - show graphs of each repetition
 - Last - show only the last graph

After filling the fields, we need to click "Start" button. Application will start the algorithm and display results. For example:



The screenshot shows a graphical user interface for a Traveling Salesman Problem (TSP) application. The window is titled "TSP" and contains several input fields and buttons. The fields are: "Number of cities" (20), "Population size" (100), "Mutation rate" (0.015), "Generations" (100), "Repeat" (1), "Initial distance" (Result), and "Final distance" (Result). There are two buttons: "Open from file" and "Generate random cities". There are also two radio buttons for "Elitism" (Yes and No) and two radio buttons for "Graph show" (All and Last). A "Start" button is located at the bottom. The "No" radio button for "Elitism" and the "Last" radio button for "Graph show" are selected.

Photo: Application with all the brackets filled with exemplary values.

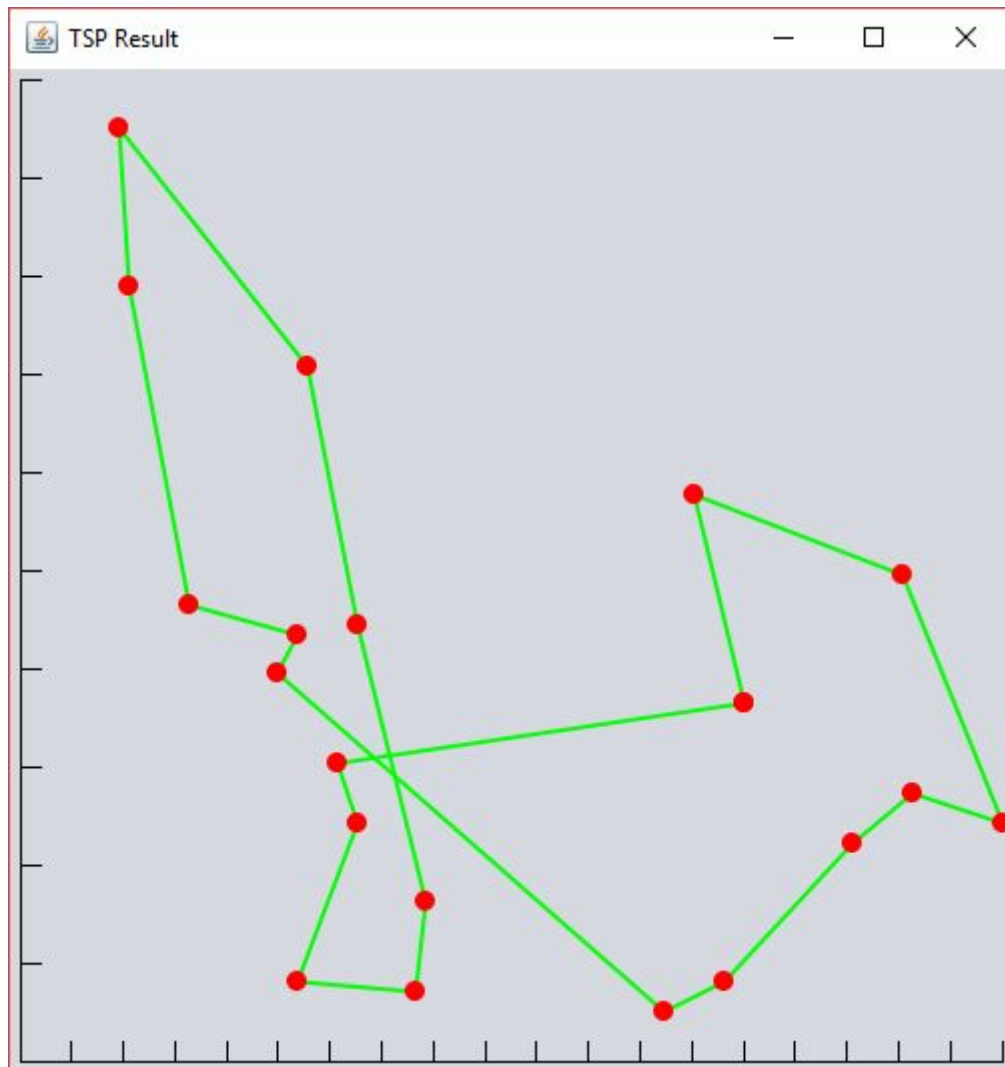


Photo: Output of the application showing results on the graph.

5. Intern specification

Domain:

1. City

Holds information about location's coordinates.

- fields
int x, y; - coordinates of the City
- methods
public double measureDistance(City city); - measures distance between the city that calls the method and the one passed as an argument

2. Population

Holds a population of possible tours between cities.

- fields
ArrayList<SingleTour> tours - list of all generated SingleTours
- methods

SingleTour getTheShortest() - get the shortest generated SingleTour among all the generated SingleTours contained in collection

3. SingleTour

Represents a haul to be gone between two places.

- fields
ArrayList<City> tour - list of Cities to be visited
int distance - total distance measured while travelling
- methods
void generateSingleTour(TravelList travelList) - add every city from the TravelList passed as an argument to the tour list, lastly shuffle obtained order of the list in order to randomize results
int getDistance() - count total distance between all cities
void addCityToSingleTour(City city, int index) - add City on the position
void swapCities(int pos1, int pos2) - swap two Cities on their positions

4. TravelList

Holds a list of cities to visit.

- fields
ArrayList<City> travelList - collection that holds list of cities that are going to be visited while processing the algorithm
- methods
void addToTravelList(City city) - add another City to the collection
void clearTravelList() - remove all Cities from the collection
City getCity(int index) - get City upon a specific index
int citiesNumber() - get total amount of Cities present in the collection

Controller:

1. GeneticAlgorithm

Implementation of genetic algorithm.

- fields
TravelList travelList - list of cities to visit
double mutationRate - mutation rate parameter passed in GUI
boolean elitism - elitism parameter passed in GUI
 - methods
Population evolvePopulation(Population population) - method that holds all operations which are evolving population
SingleTour parentSelection(Population population) - method that selects parent randomly
void mutate(SingleTour tour) - method that mutates SingleTour (randomly swaps two cities)
SingleTour crossover(SingleTour parent1, SingleTour parent2) - methods that does crossover (selects subset from the first parent and then adds the subset to the offspring, any missing values are then added to the offspring from the second parent)
- Algorithm
1. Create new temporary population

2. Check elitism
 - a. if elitism is enabled, add to population the shortest tour from old population
 - b. if elitism is disabled, continue
3. Ordered crossover
 - a. Selecting parents
 - i. Create new population (size is random number between 2 and original population size)
 - ii. Select random tour
 - iii. Get the tour with the shortest distance and save it as a parent
 - b. Create child
 - i. Select randomly the start and the end of subset from the first parent
 - ii. Save the cities from the subset in child's tour on the same positions
 - iii. Analyse the second parent and add missing values in order that they are found
 - c. Add child to new temporary population
4. Swap mutation
 - a. Check if randomly selected number is lower than mutation rate
 - b. If it is, do swap mutation
 - i. Swap two random cities
 - c. If not, do nothing
5. Return temporary population

2. TheTravellingSalesmanProblem

Initialises genetic algorithm, creates population, repeats algorithm depending on the number of generations and displays results.

- fields
 - TravellList travellList - list of cities to visit
 - Population pop - created population of algorithm
 - String initialDistance - the first distance of created travellList
- methods
 - void generateRandomCities(int citiesNumber) - generates random cities
 - boolean initTravellListFromFile(String file) - opens cities from file
 - int getSize() - gets size of travellList
 - SingleTour initAlgorithm(int populationSize, double mutation, int generations, boolean newPopulation, int repeat, int index, String fileName, JTextField result, JTextField initial, boolean elitism) - initialises algorithm, transfers parameters to GeneticAlgorithm class
 - void clear() - clears travellList
 - void registerLogResult(String fileName, int index, SingleTour result) - saves results in file
 - registerEachGeneration(String fileName, SingleTour shortest, int index) - saves results from each generation in file

6. Results

We decided to proceed our investigations while testing our application designed in two different ways.

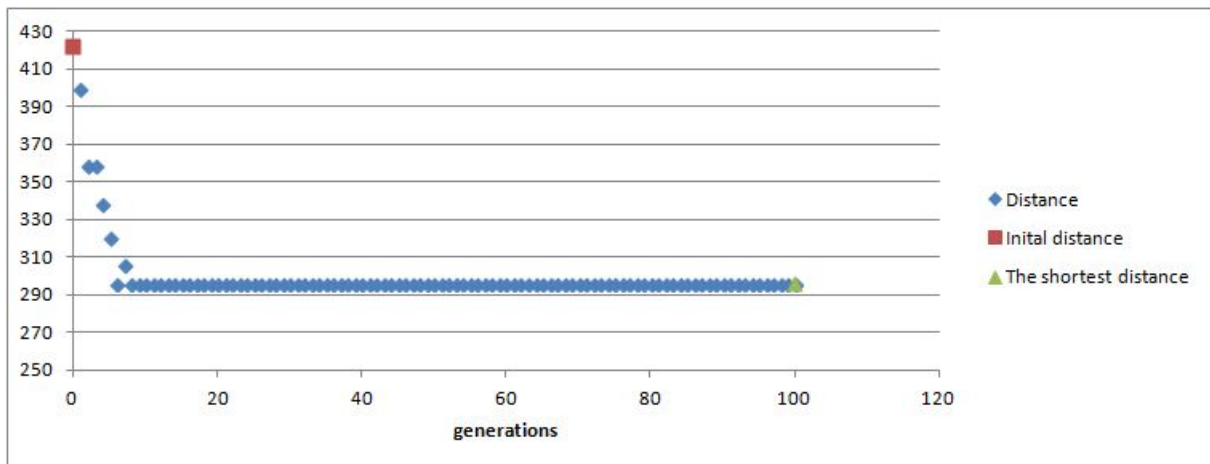
First, we were testing algorithm without using elitism option.

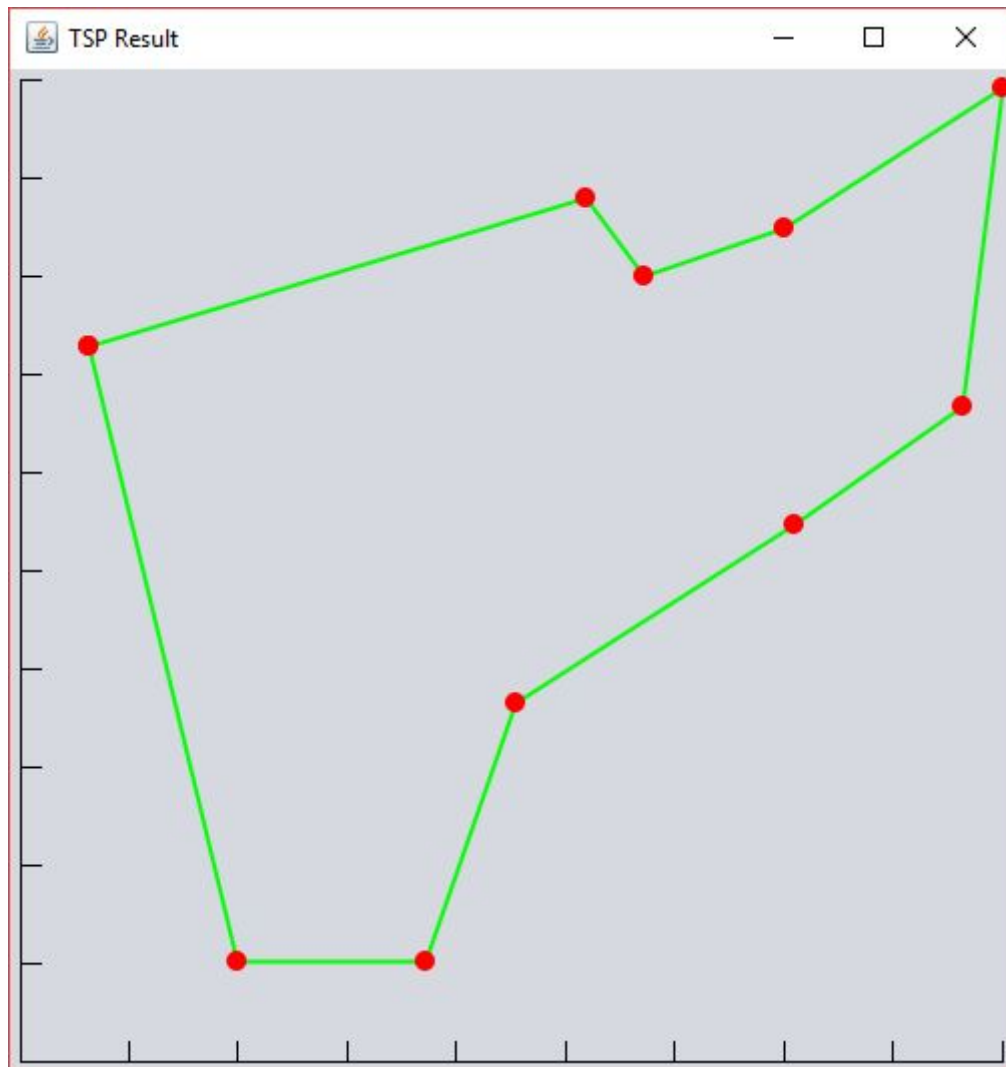
Second and last, we tested these cases with an addition of elitism.

Then we repeated the algorithm for every set of values 50 times and gathered them altogether. It means that for every single case of given values (for example: **Cities: 10, Population: 100, Generations: 100, Mutation rate: 0.015**) we picked the best results and checked how big are the output results between every start-end cycle of the application.

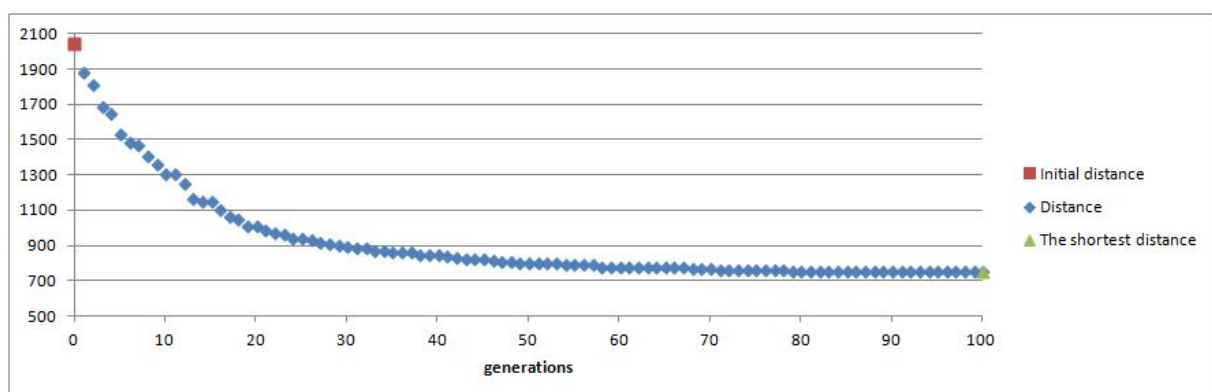
> Generations without elitism

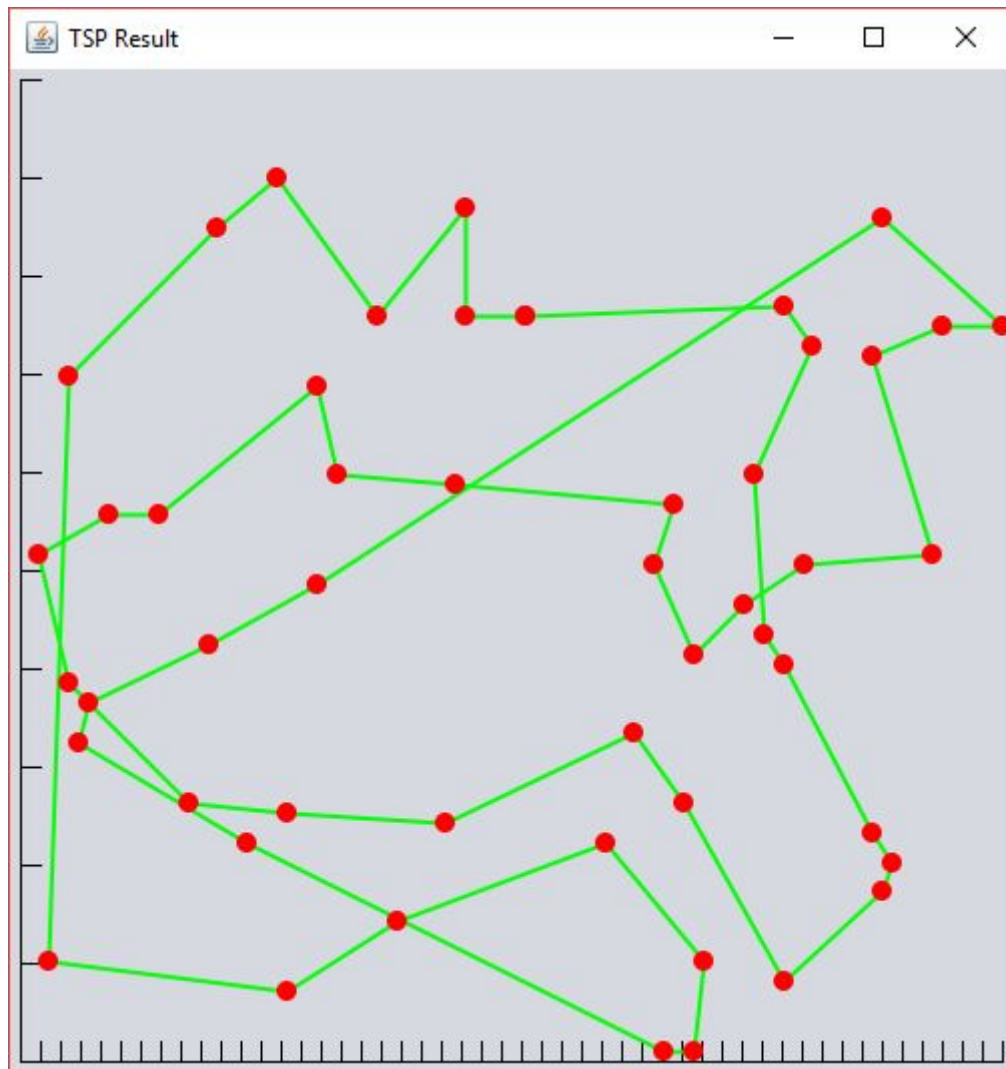
1. Cities: 10, Population: 100, Generations: 100, Mutation rate: 0.015



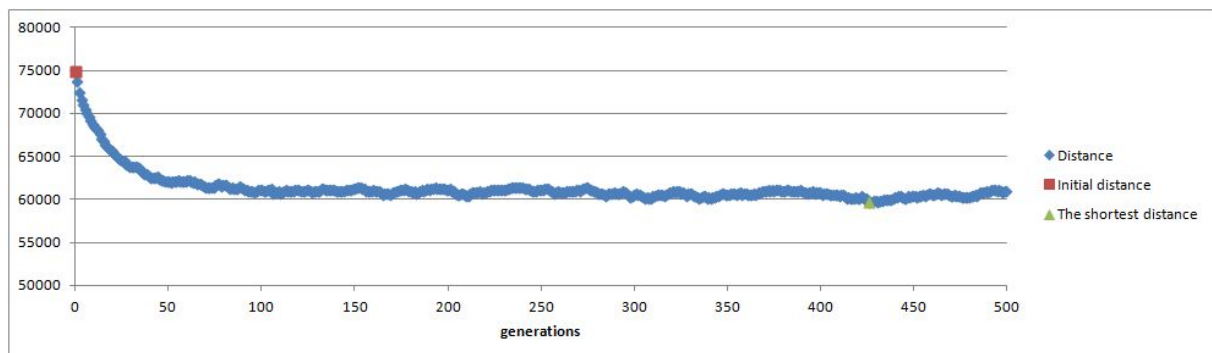


2. Cities: 50, Population: 500, Generations: 100, Mutation Rate: 0.015

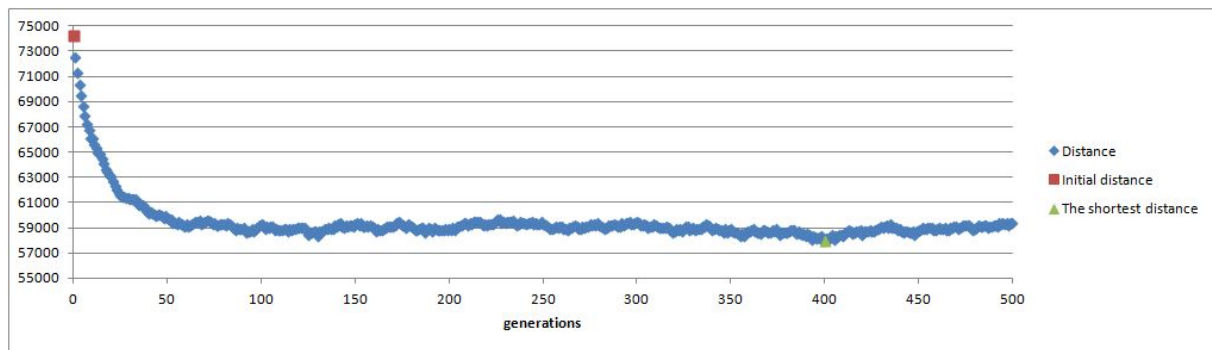




3. Cities: 1500, Population: 500, Generations: 500, Mutation Rate: 0.015

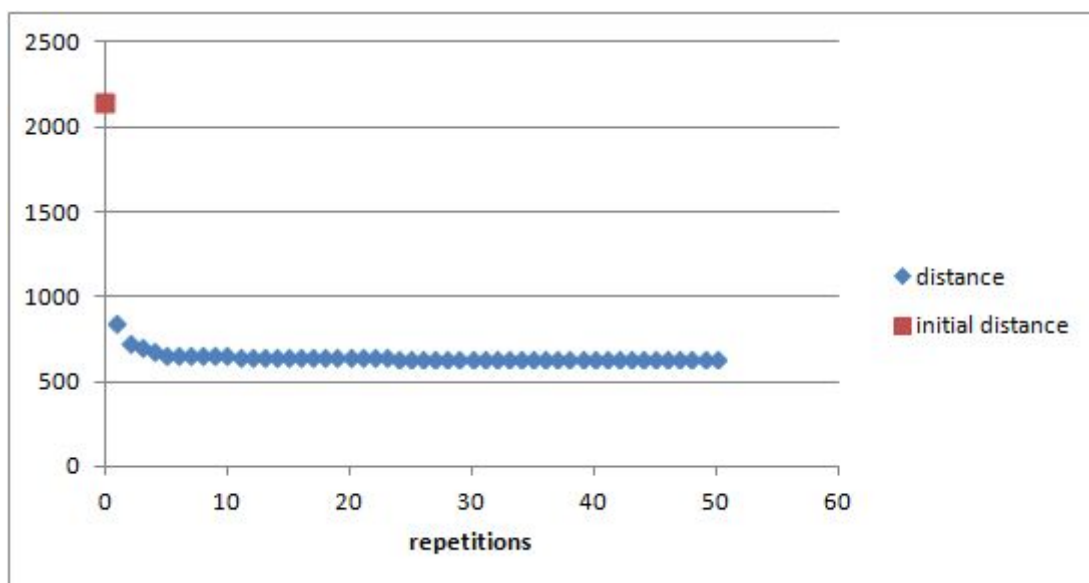


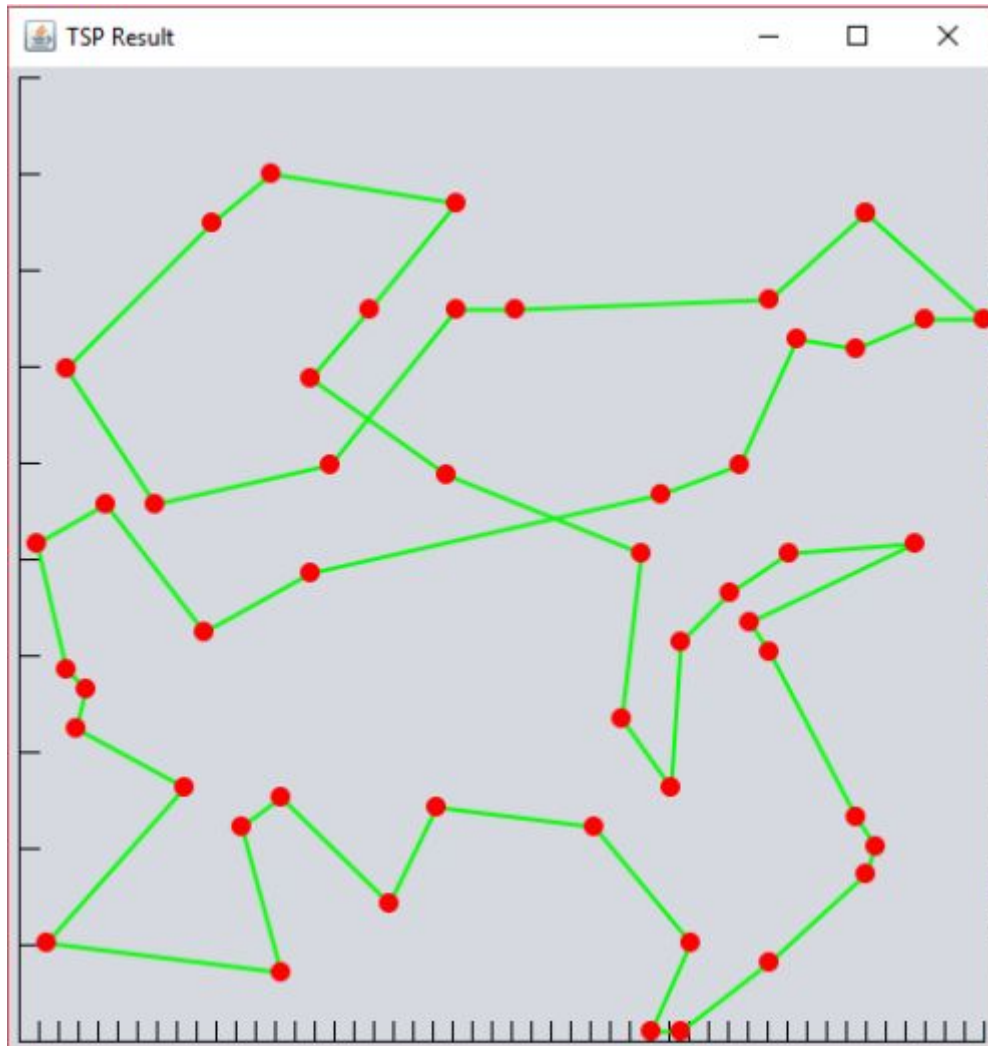
4. Cities: 1500, Population: 10000, Generations: 500, Mutation Rate: 0.015



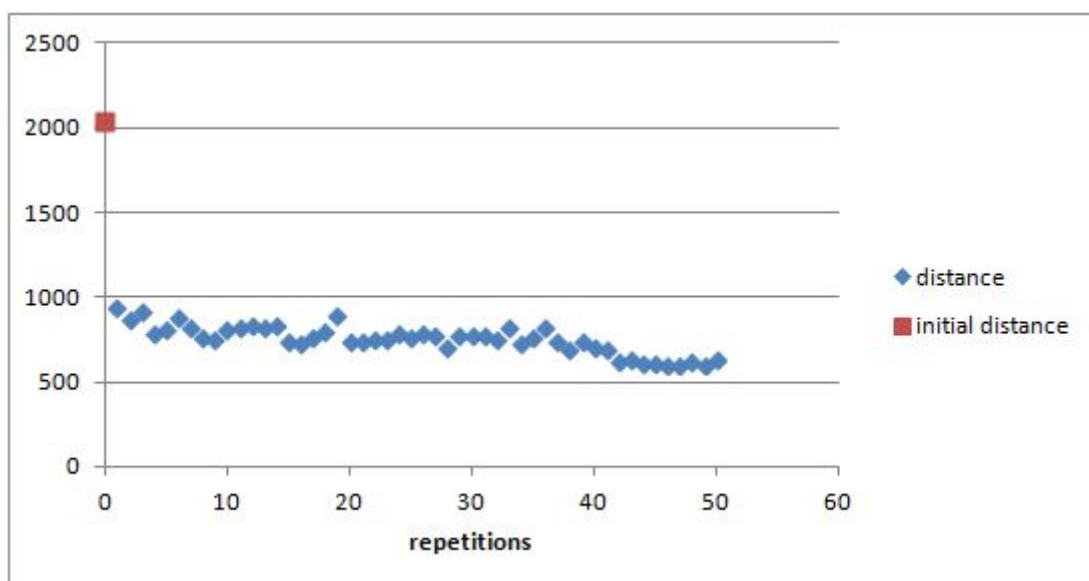
> Repetitions without elitism

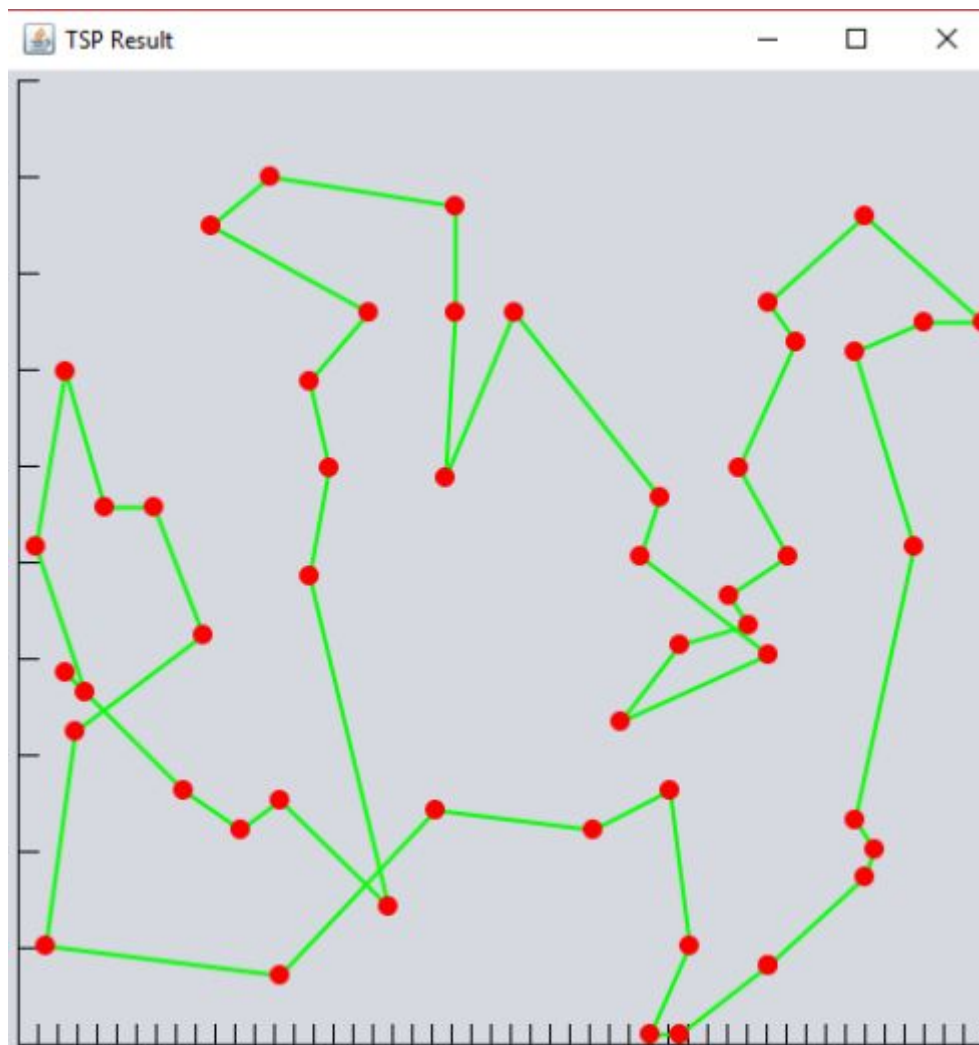
1. Cities: 50, Population: 50, Generations: 100, Mutation Rate: 0.015





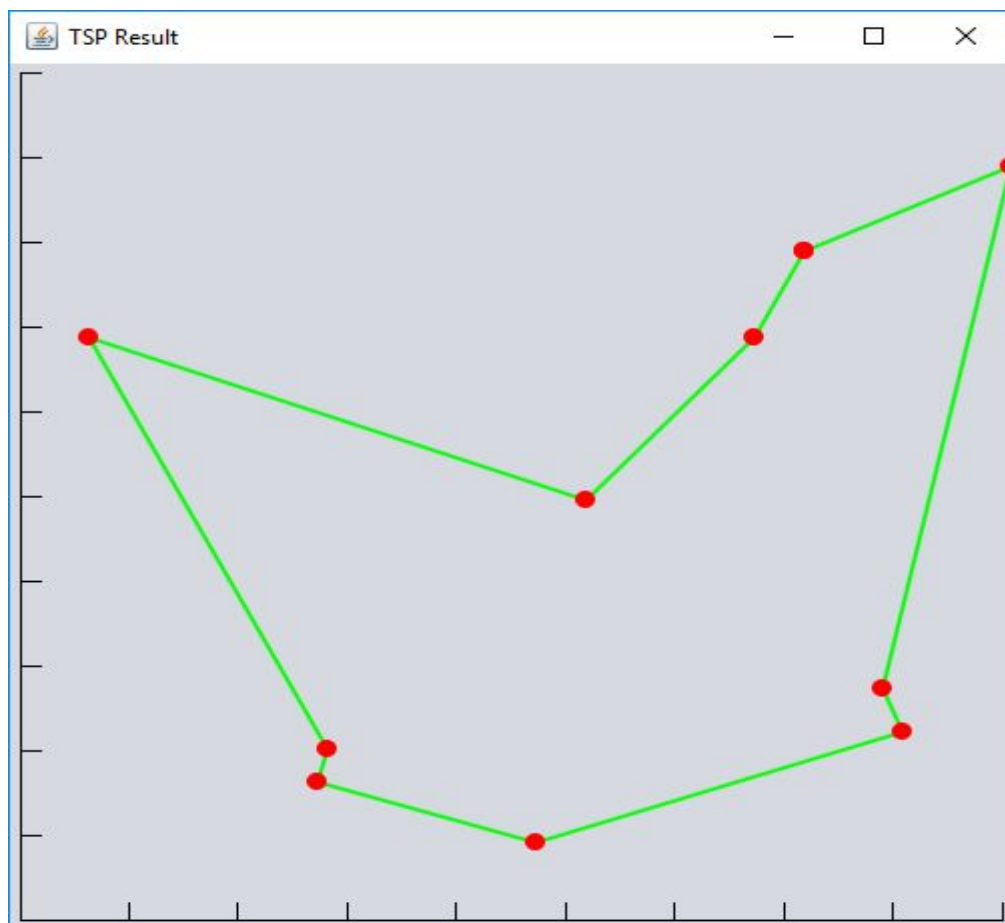
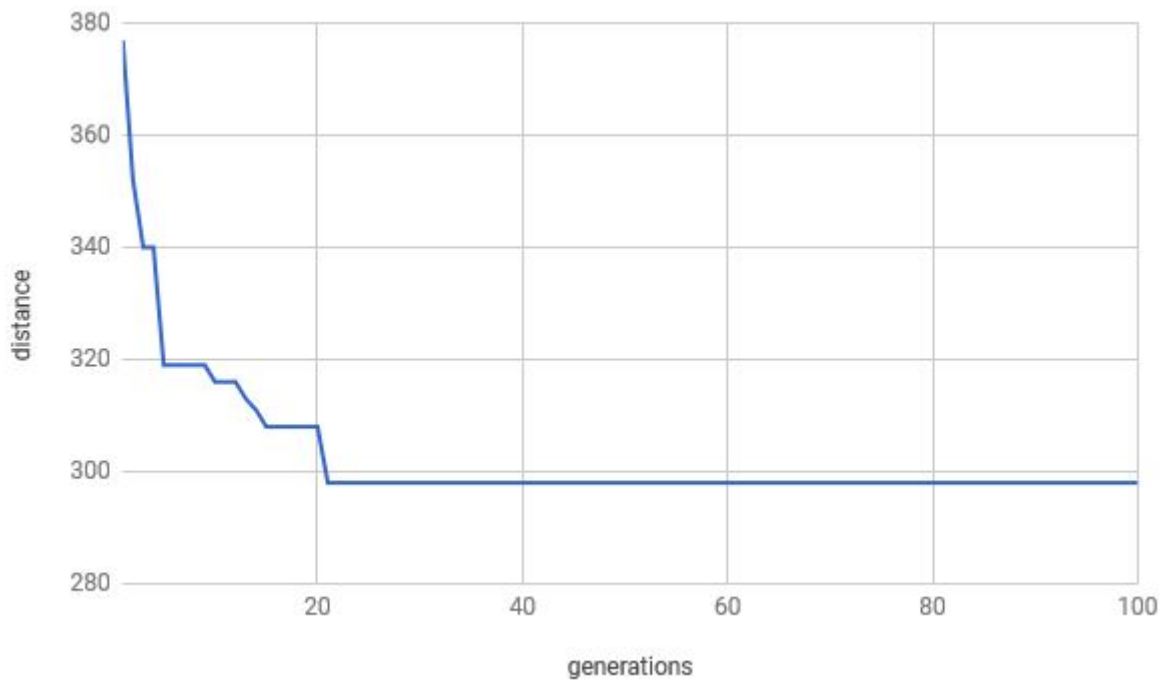
2. Cities: 50, Population: 50, Generations: 100, Mutation Rate: 0.045



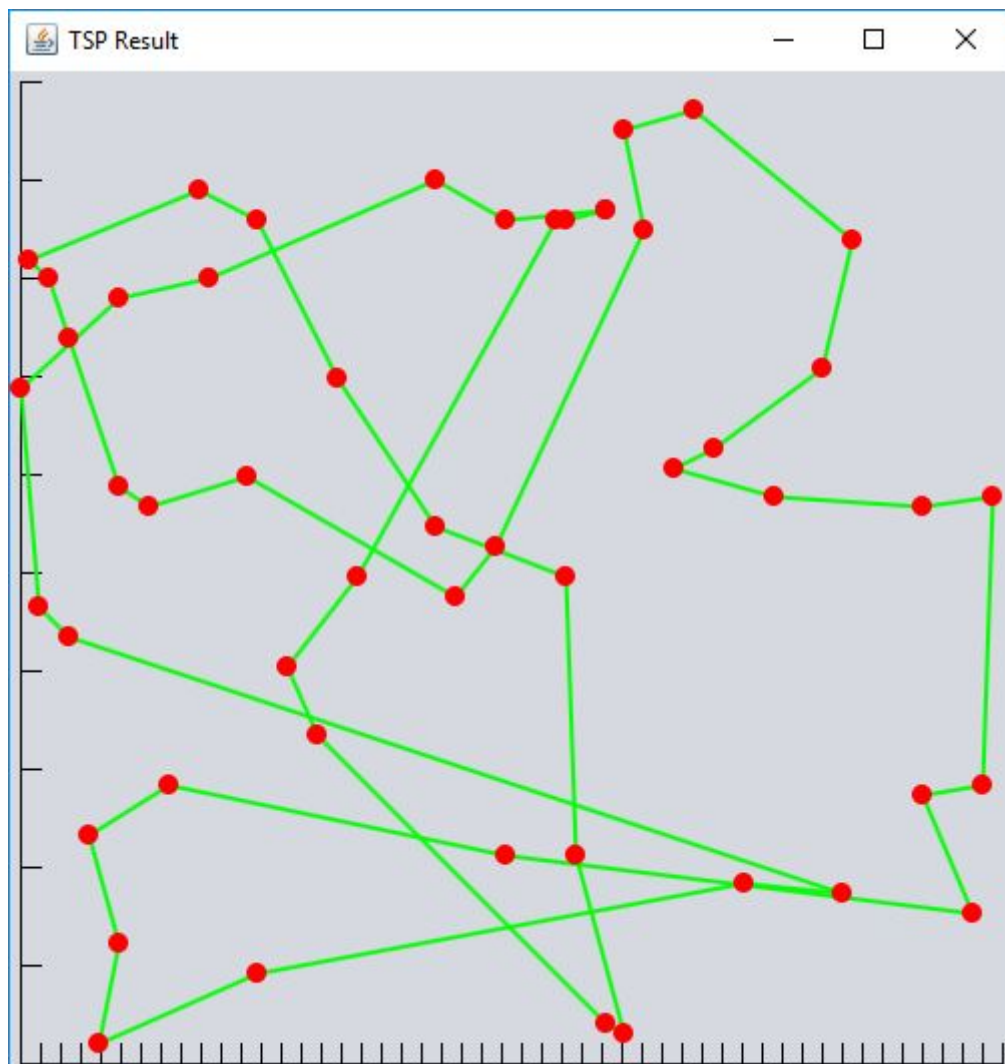
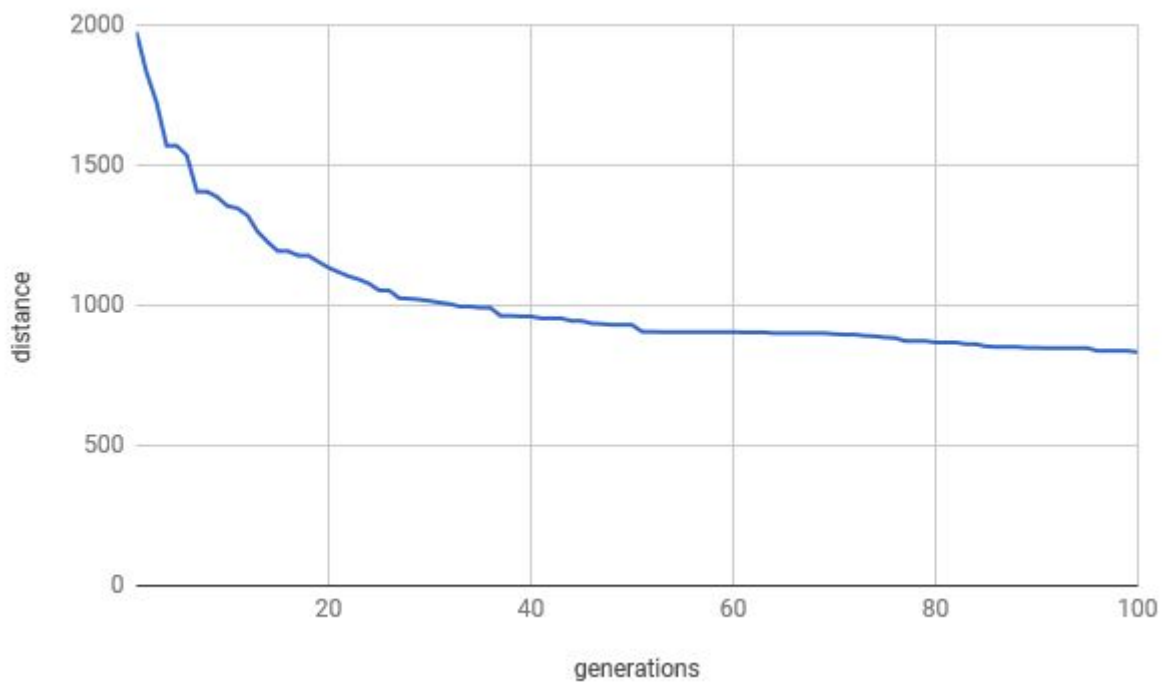


> Generations with elitism

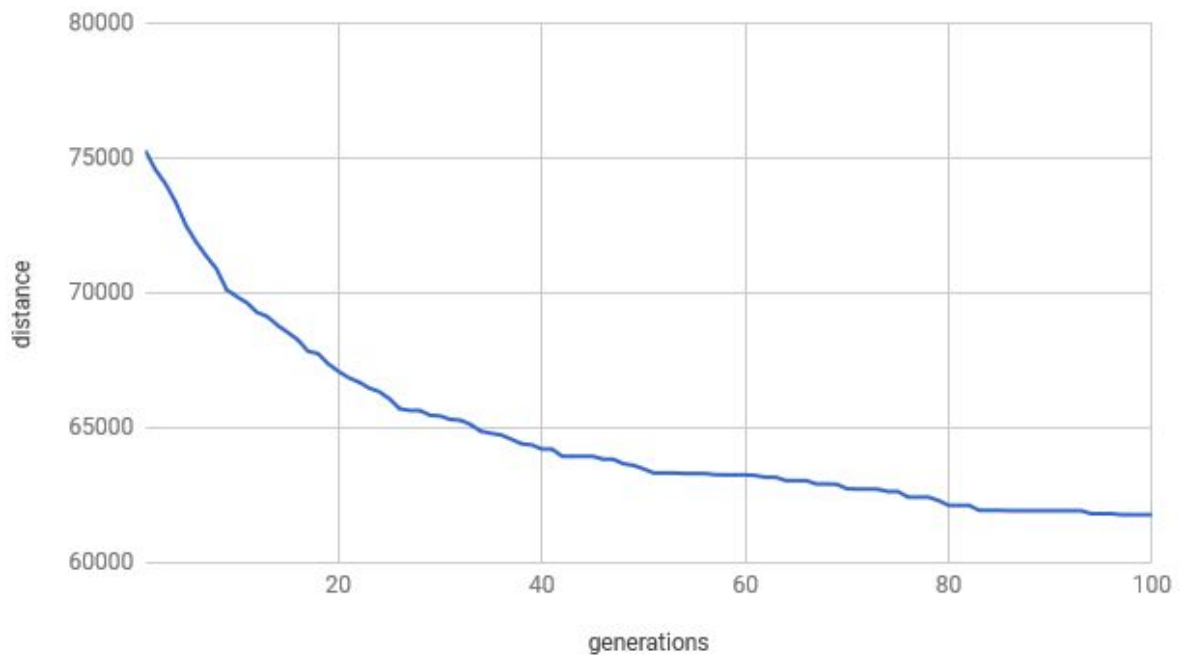
1. Cities: 10, Population: 100, Generations: 100, Mutation rate: 0.015



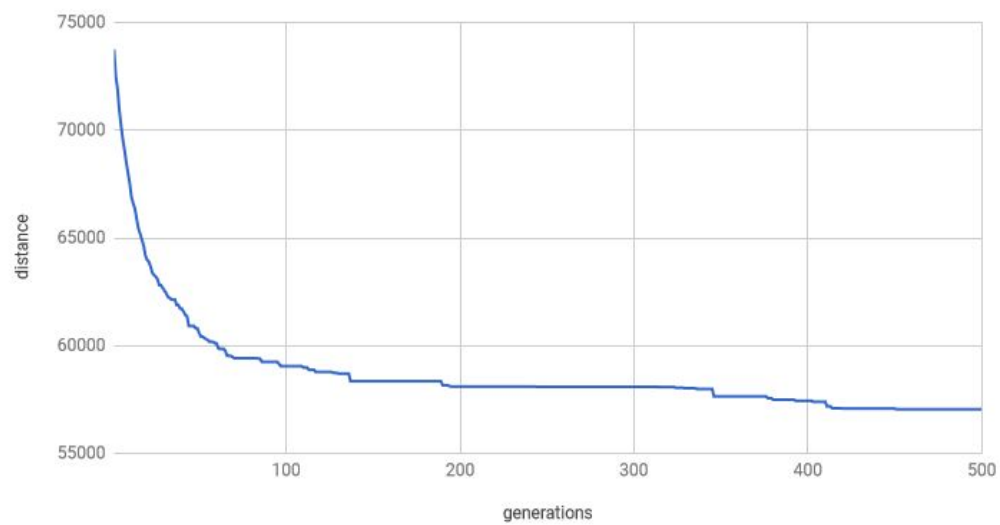
2. Cities: 50, Population: 500, Generations: 100, Mutation rate: 0.015



3. Cities: 1500, Population: 500, Generations: 100, Mutation rate: 0.015



4. Cities: 1500, Population: 10000, Generations: 500, Mutation rate: 0.015



7. Summary

To sum up, we performed many different experiments in order to get results for given conditions. We divided our tests into 3 different sections.

The first section was **running application once without using elitism**. It means that results came from a single run of our algorithm without rerunning it couple of times. What we got was a diagram showing how the distance changed when processing through next generations. Red square points the initial distance that was measured before starting application. In all situations it was of course the worst results that we wanted to improve.

We tested all the cases using mutation rate equal to 0.015. This is what we stucked to in future experiments as well. Next, we decided to pick variety of numbers sets. The first experiment was as few as 10 cities, population of 100 single tours and processing through 100 generations. The output was not surprising, because among first 10 generations we got the best possible result. Entire distance was not long enough to show it smoothly on the diagram, which is the reason why we can see definite points. Later on the shortest distance did not improve at all. It turns out that for a few cities (10 or 15) there is no point processing through so many generations. There are hardly any tours to count, so we got the best result pretty soon.

Next experiment consisted of 50 cities, 500 populations and once again 100 generations. In this case we were obliged to wait a bit longer in order to get satisfying results. Actually the last increase in performance proceeded in between generations 70 - 90. Simply saying, we might expect even better results if we ran the application for more generations. However, taking into the consideration mutation rate at some point we might come back to longer distance, because this is what sometimes may happen to our population. In addition, the difference between distances getting the ones on the border of the improvement is so low that this is not worth waiting longer for this result. Even though it might seem complicated on the diagram, this one turns out to be the most efficient way to travel across all the cities.

Both experiments before took us as long as 1 up to 2 seconds waiting for the result. The amount of time that we have to spend on getting the output was so short that we decided to run the application with much beginner input. We prepared 2 sets of numbers which differed with population only.

The first huge experiment consisted of 1500 cities, 500 population and 500 generations. This was very demanding to our computers, because it took as long as 2,5 hours for getting the output. It took more or less 10-20 seconds for each generation to be count. We checked this by checking the text file where each distance was preceded by number of generation where it was generated. As we can see on the diagram, this time it was pretty clear to spit that the mutation rate started it work. There were a lot of points where the last distance started to getting worse instead of getting shorter. For example, between generations 140 - 170 we can see that the distance started increasing its value.

The second huge experiment gave similar results. However, the major difference was that peaks between values were much beginner in comparison to the previous case. All in all at the end the final value was not the best one found through all generations.

In next step we decided to rerun application couple of times in order to compare final results. We performed repetitions for small data (10 and 50 cities). Running application for the same sets of values gave as variety of results. It was surprised to see that in the second case we had to get through all the repetitions to eventually see the best value. It turns out that the output is not always the same even for the same data. As a conclusion we may say that if the hardware allows us to run application fast enough it is a good idea to perform many experiments to see if the generations will differ between each others while giving them a chance to grow from scratch a few times.

Last, we redesigned our application to support elitism. It turned out that it gave us huge benefits to our final results. We no longer had any peaks in the latest distance, because there was almost no way to occur. The only possible way to ruin the best result was no mutate our member of population and as a result assign to it much worse conditions. Fortunately this did not happen in any of our experiments, starting from low population and proceeding to huge experiments. We did the same with elitism as we did without it while doing our tests. We had no doubt that this time algorithm will do much better and we were right. Diagrams clearly present just getting better while getting through all the generations. When taking into consideration the biggest experiment consisting of 10000 population with 500 generations it took way too long to get these results. Starting from generation 150 up to generation 500 the increase was not as clear as the beginning with a tremendous cost of time. We assume that if we decided to pick even more generations it may get yet slightly better but not as much as we would expect and surely not as much as if it was worth it all.

As a conclusion, it turned out that elitism is very useful tool while performing genetic algorithms. It allows us to save the best member of population and make it even better. When running with hardly any data it does not bring any benefits at all because we will be able to grab sufficient information with almost no cost of the time. The problem comes when we decide to run application with a huge population of so many cities living in a lot of generations. At that point spending more time is not worth picking slightly any increase of performance.

Working on genetic algorithm is definitely not an easy job. We tried our best to solve the travelling salesman problem and in our opinion we got good enough results. Genetic algorithms are not supposed to give the best solutions globally, but the solutions close to the best ones in most cases are satisfying to the user. Our algorithm proved that we should take several experiments to see if the data that we get could be even better.

8. Repository

<https://github.com/martmil403/BIAl-tsp>

9. Literature

- https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_travelling_salesman_problem.htm