



Silesian University of Technology  
Institute of Informatics



Academic Year

Type\*: SSI/NSI/NSM

Subject:

Group

Section

**2018/2019**

**SSI**

**BIAI**

**GKiO4**

**9**

Name:

**Piotr**

Teacher:

**GB**

Surname:

**Paczuła**

## ***Project Report***

Project's subject:

**Program for solving travelling salesman problem  
using evolutionary algorithm**

Date:

dd/mm/yyyy

**09.06.2019**

## Topic:

Implementation of travelling salesman problem solution using evolutionary algorithm. Algorithm should use representation of chromosomes in form of permutation. Should also use operators of crossover and mutation adapted for permutation. Program will be implemented as a web application, so anyone with internet access can try it.

## Functional requirements:

- Program should use evolutionary algorithm in order to solve the travelling salesman problem.
- Program will search for the correct result until user will stop the program.
- Program will not deliver the correct answer to the problem, but will try to return “satisfying” result for the given time as there is no way to generate valid result with genetic algorithm and confirm it without use of other methods (like bulk search).
- User will be able to change both size of generated populations in each generations and rate of mutation.
- User will be able to set both random travel points and choose it manually.
- User will be able to either pause/continue or restart the process.
- User will be able to check current best found distance and average value of fitness function from the last generation.

## Non-functional requirements:

- Application should work on the latest versions of browsers like Chrome and Firefox.
- Internet explorer will not be supported .
- Web application should also work in mobile version.

## Implementation:

Project is implemented in JavaScript. Main libraries that were used are:

- ReactJs – JavaScript library for creating web application’s user interface.
- P5.js – graphic library for managing WebGL in browser in order to draw on HTML5 canvas element. Is used for inserting custom points of travel and also for results presentation.

## 1. Pick DNA:

```
_pickDna=() => {  
  let prob = this._p5.random(0, 1);  
  let index = 0;  
  while (prob > 0 && index < this._population.length) {  
    prob -= this._population[index].probability;  
    index++;  
  }  
  index--;  
  return this._population[index];  
}
```

Function picks one DNA from the whole population. First it generates number value from the range 0-1. Every child of population has its probability which given every child sums up to 1. Function is subtracting child's probabilities from generated probability until it goes below 0. The last subtracted child is then returned.

## 2. Crossover:

```
_crossover=(dnaA, dnaB) => {  
  const { length } = dnaA.genes;  
  const i = Number.parseInt(this._p5.random(length));  
  const j = Number.parseInt(this._p5.random(i + 1, length + 1));  
  const newGenes = dnaA.genes.slice(i, j);  
  dnaB.genes.forEach((b) => {  
    if (!newGenes.includes(b)) {  
      newGenes.push(b);  
    }  
  });  
  const fitness = this._calcFitness(newGenes, this._points);  
  return DnaFactory.get(newGenes, fitness);  
}
```

Crossover function is taking two DNA elements and combining them into a new one.

### 3. Mutation of DNA:

```
_mutateDna = (dna) => {  
  const prob = this._p5.random(1);  
  const { length } = dna.genes;  
  if (prob > this._mutation || length < 2) {  
    return dna;  
  }  
  
  const first = Number.parseInt(this._p5.random(length));  
  let second = Number.parseInt(this._p5.random(length));  
  if (first === second) {  
    second = (second + 1) % length;  
  }  
  const genes = [...dna.genes];  
  const temp = genes[first];  
  genes[first] = genes[second];  
  genes[second] = temp;  
  const fitness = this._calcFitness(genes);  
  return DnaFactory.get(genes, fitness);  
}
```

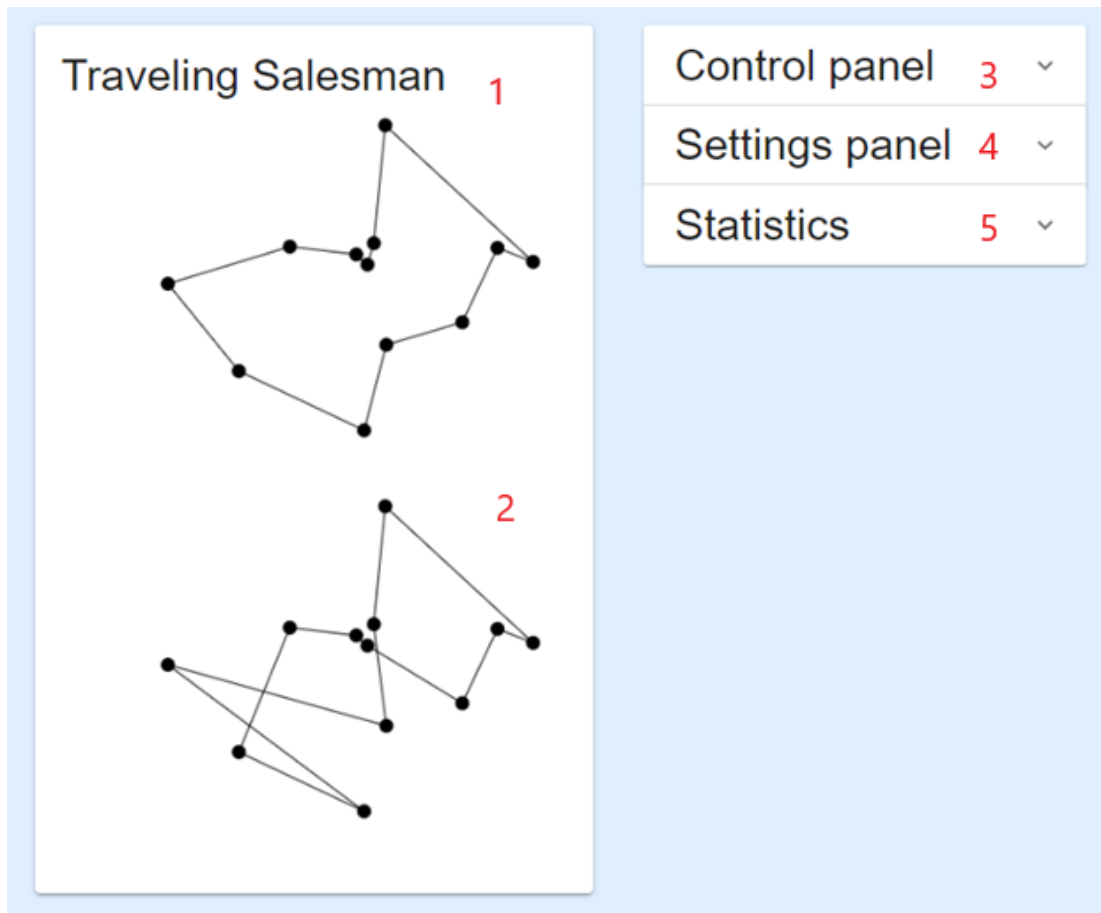
If generated random value is smaller than set mutation rate then random two genes will have its places switched.

### 4. Fitness function:

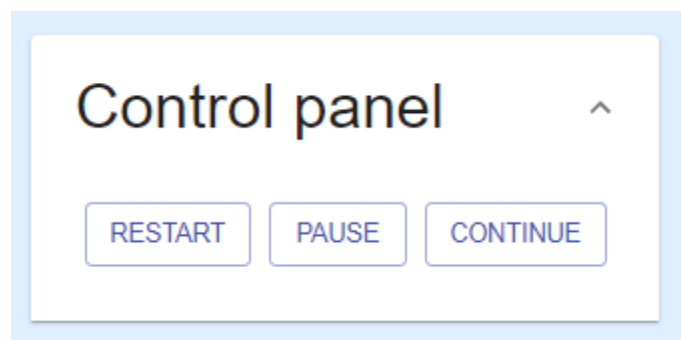
```
_calcFitness=(array) => {  
  const distance = calculateDistance(array, this._points);  
  return (1 / distance) ** 2;  
}
```

Function calculates fitness with given distance of full travel. The value is taken to power of 2 to increase the importance of better children.

## Manual instruction:

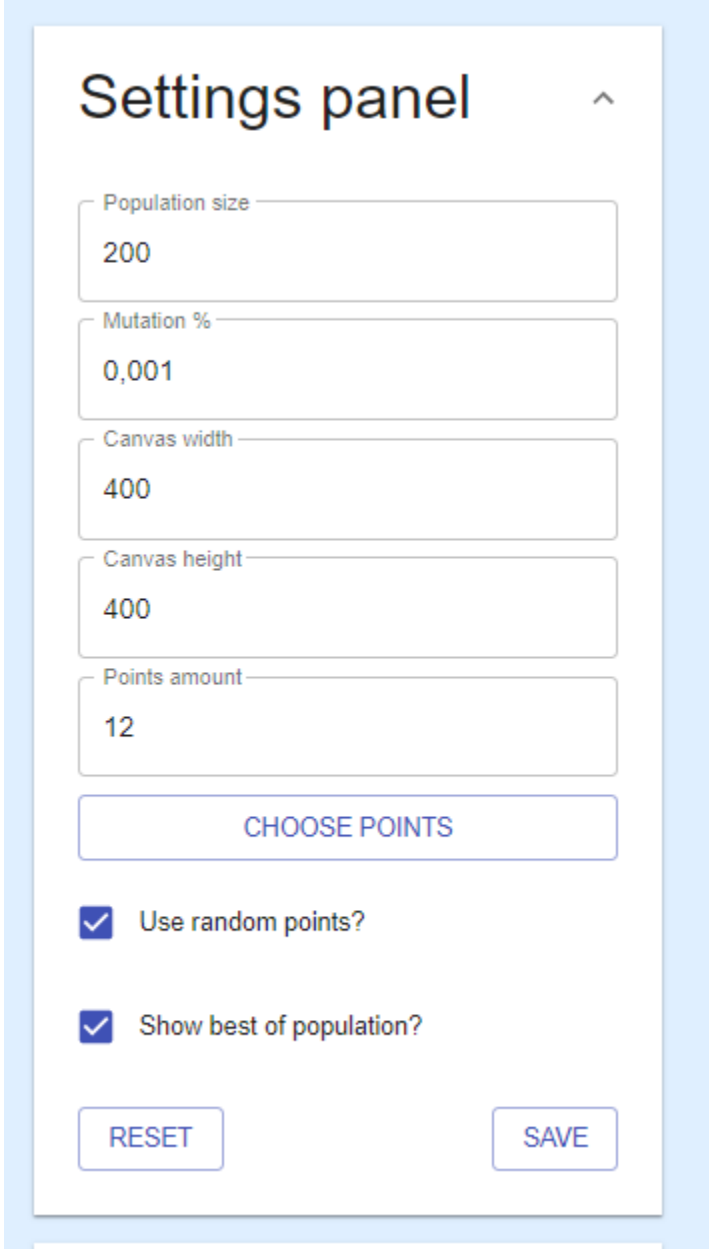


1. Shows best result generated so far
2. Shows best result of current generation
3. Control panel



- Restart – starts the process over,
- Pause – stops searching for results,
- Continue – continues searching for result.

#### 4. Settings panel

The image shows a 'Settings panel' with a title bar at the top right containing an upward-pointing chevron. Below the title are five input fields, each with a label and a value: 'Population size' (200), 'Mutation %' (0,001), 'Canvas width' (400), 'Canvas height' (400), and 'Points amount' (12). Below these fields is a button labeled 'CHOOSE POINTS'. Underneath the button are two checkboxes, both of which are checked: 'Use random points?' and 'Show best of population?'. At the bottom of the panel are two buttons: 'RESET' on the left and 'SAVE' on the right.

Settings panel ^

Population size  
200

Mutation %  
0,001

Canvas width  
400

Canvas height  
400

Points amount  
12

CHOOSE POINTS

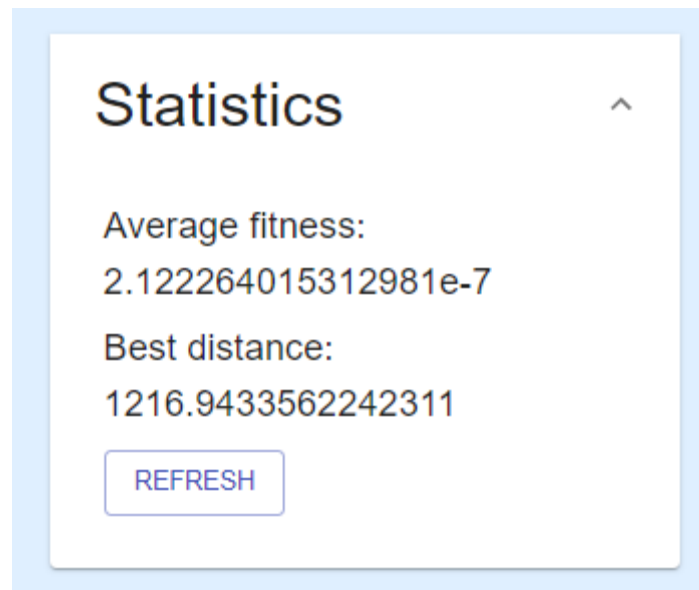
☒ Use random points?

☒ Show best of population?

RESET SAVE

Allows to set population size, mutation rate, canvas width and height. User can choose points himself or set them randomly using “CHOOSE POINTS” button or “Use random points?” checkbox respectively. After applying each change user has to save them using “SAVE” button. “RESET” button restores default settings.

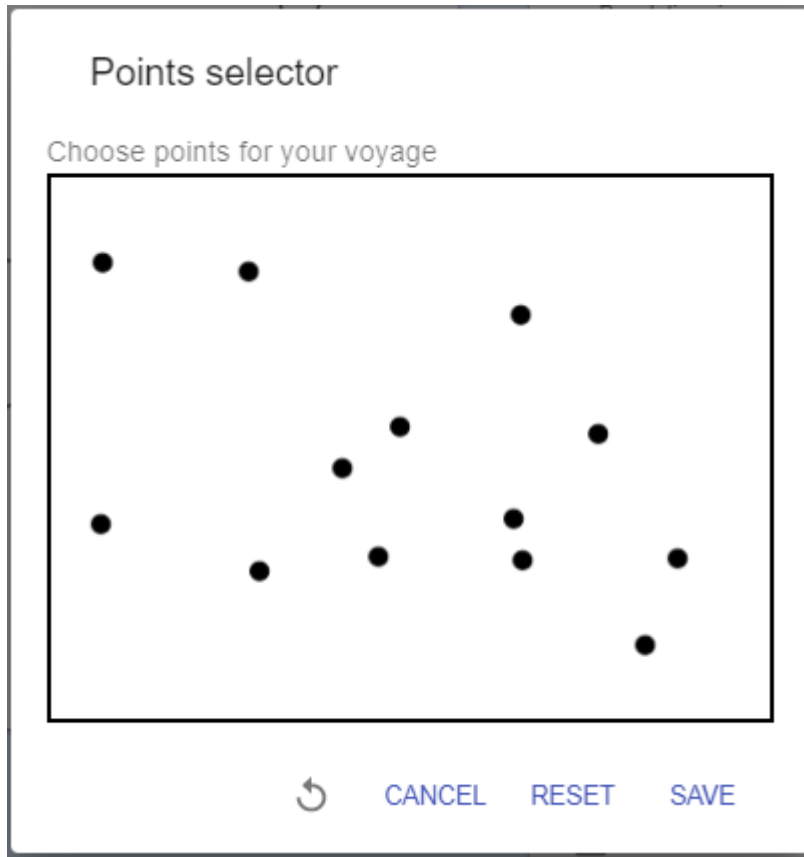
## 5. Statistics panel



After clicking “REFRESH” button user will get last average fitness of population from last generation and current best distance calculated.

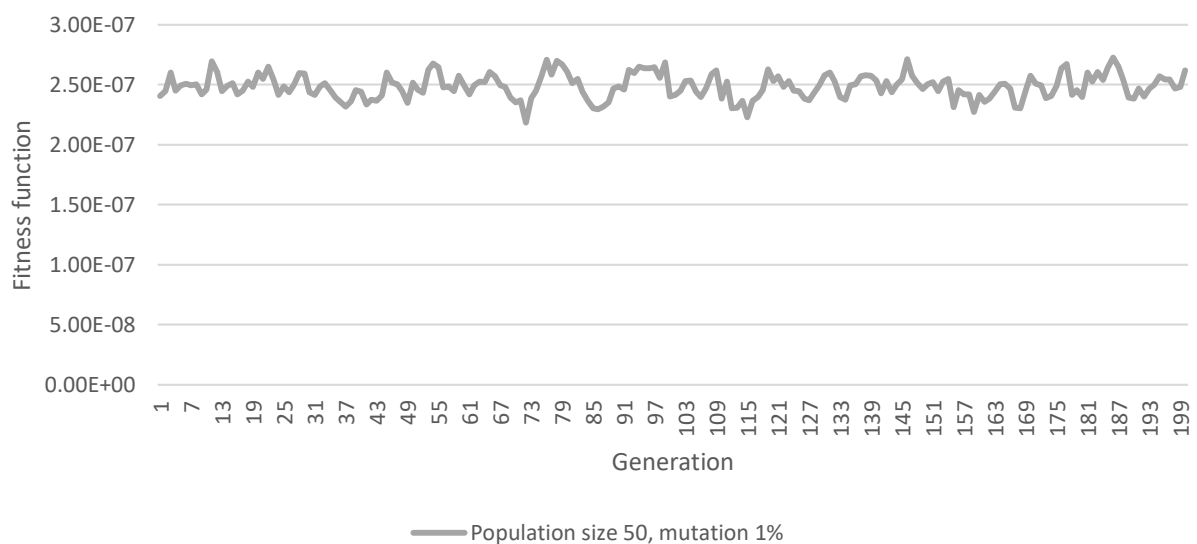
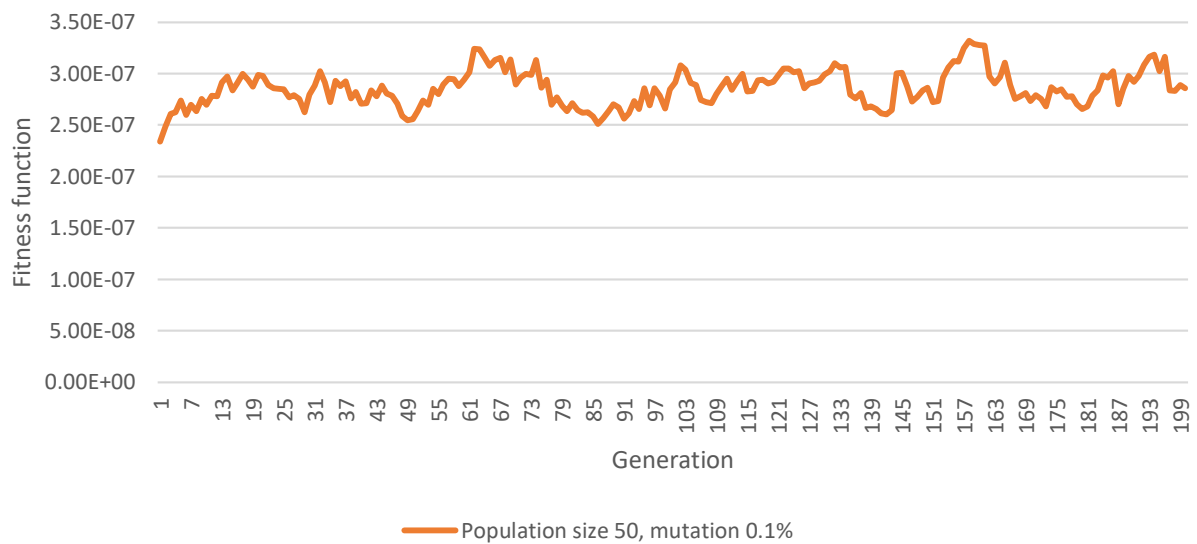
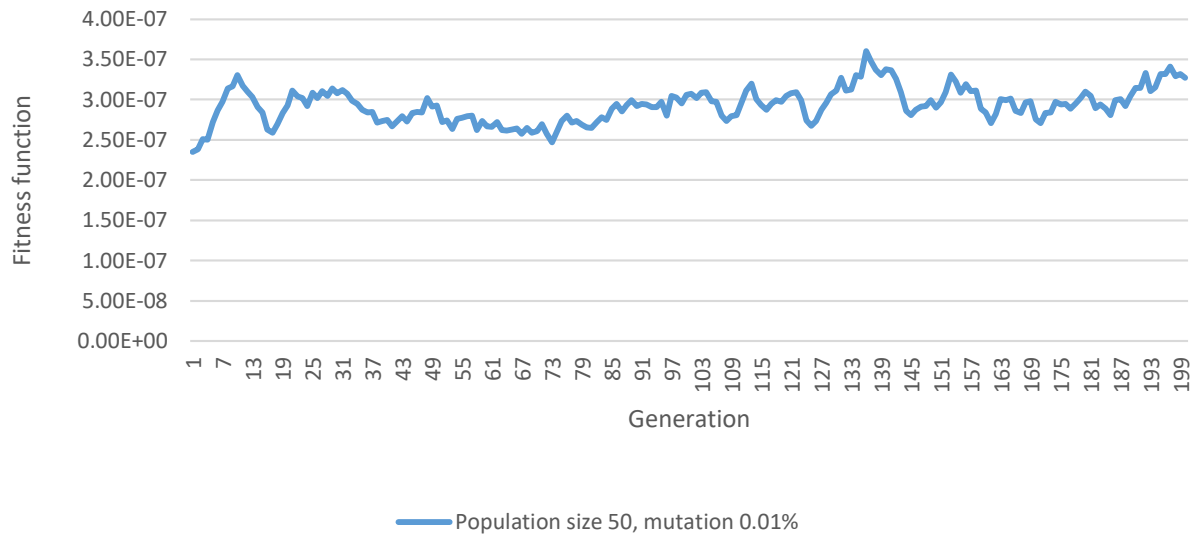
## Results:

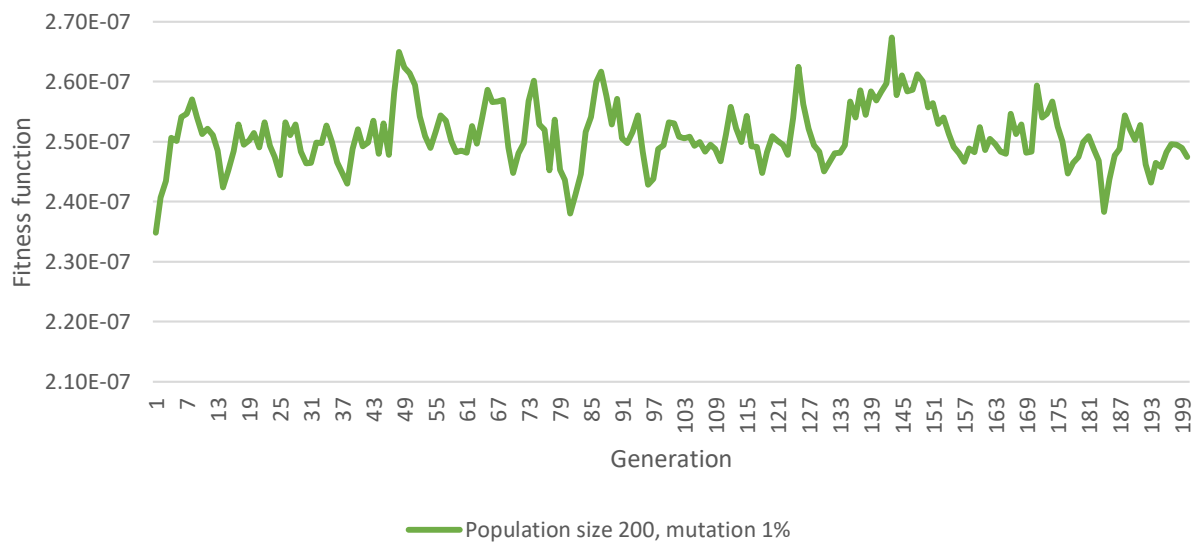
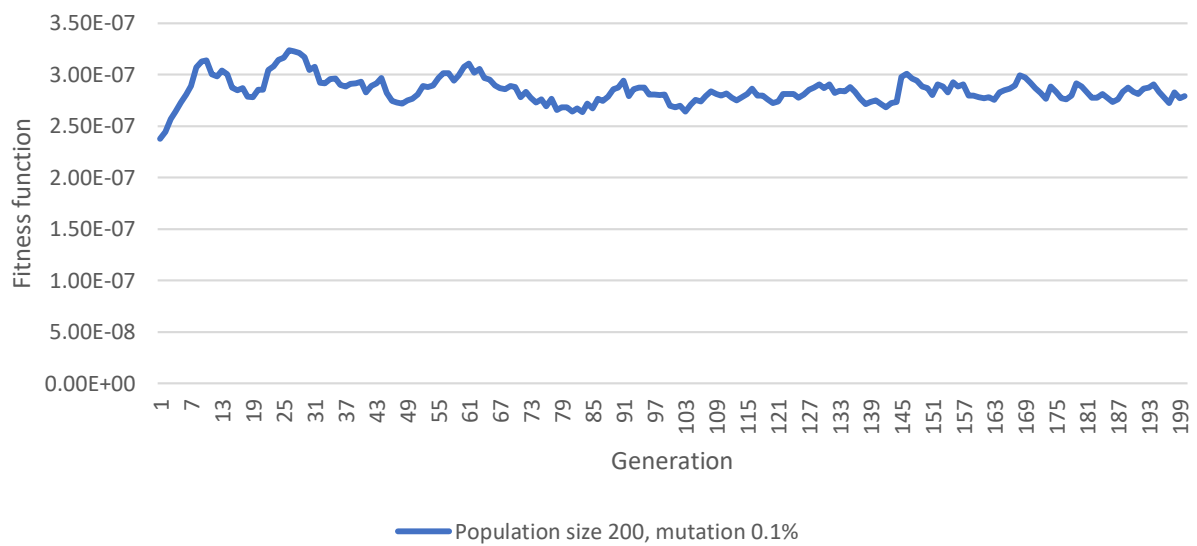
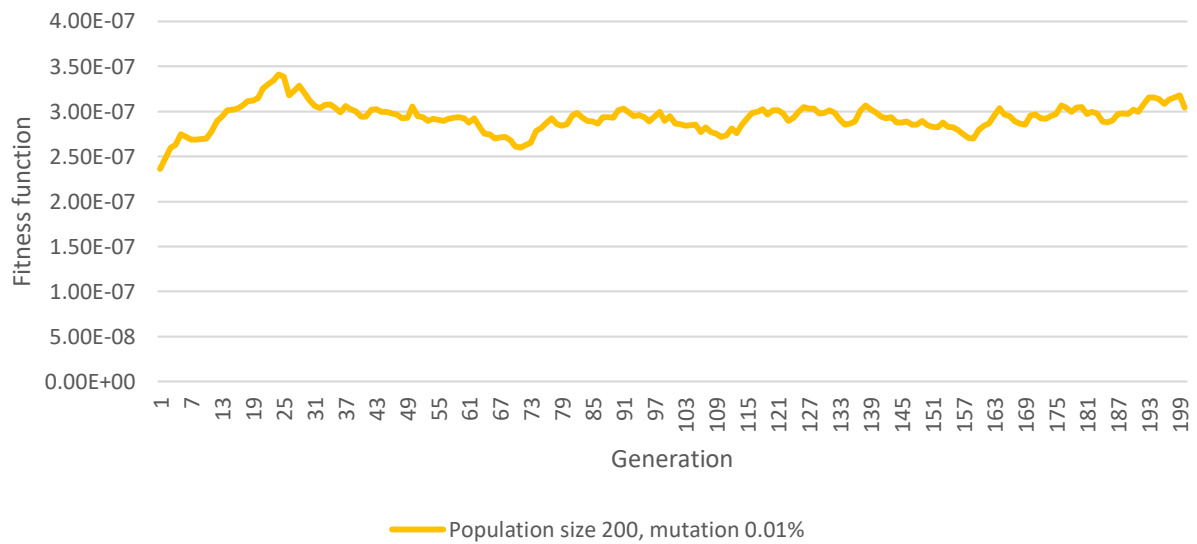
All of the following results were obtained from the same traveling salesman problem scenario with 13 points:

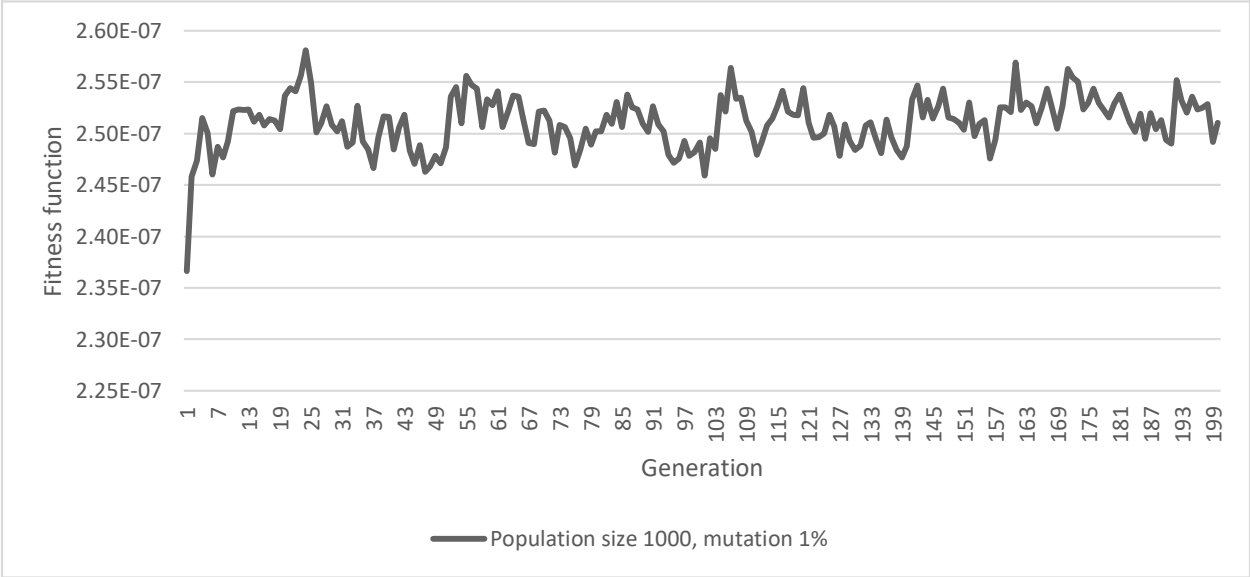
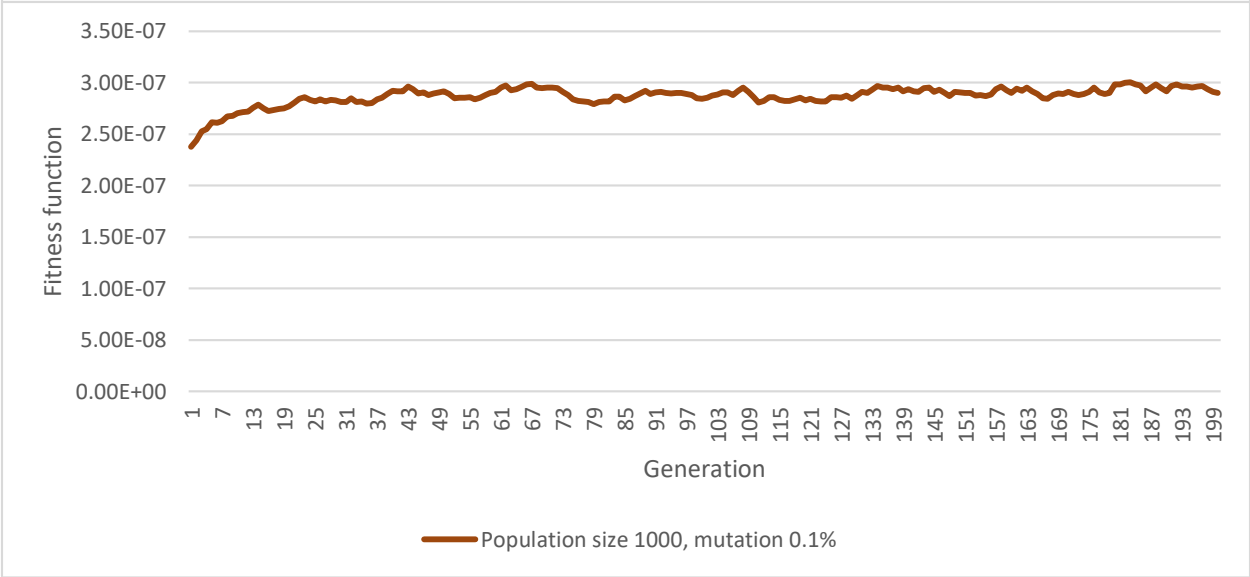
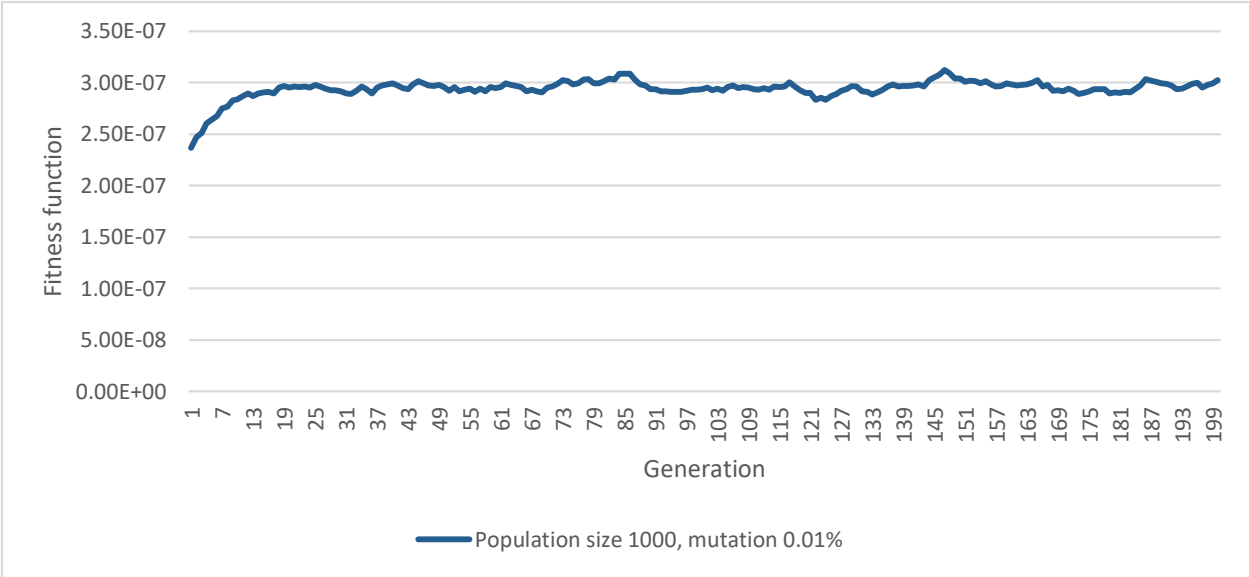


Average fitness function value is quickly peaking its limit after around 20<sup>th</sup> generation in all following scenarios. After reaching its peak it is no longer making visible progress in case of average value. According to the following examples the best mutation rate among them is the lowest one which is 0.01%. For bigger values mutation is visibly too high which causes populations to change too much. Heredity is really diminishing it those cases. This effect can be seen especially in examples with population of 200 and 1000. Also average fitness function is better seen for bigger sizes of population. Population of 50 is clearly too small to acquire good testing results.









## Conclusions:

Using evolutionary algorithm to solve traveling salesman problem is good solution for getting good results quickly. Its huge drawback is that we can never get the correct answer, or at least we can never be sure if given result is the best one possible. This implementation has two very important parameters which are population size and mutation rate. Setting one of them to values too big or too small will cause this implementation to be highly ineffective. As shown in results above setting mutation rate above 1% will create highly random values in generated populations. I have learned that there are plenty of other possibilities to use genetic algorithms and implementing it might not be as complex as I thought before. I am almost certain that I will use this knowledge in the near future.

Working application can be found here: <https://salesman-biai.herokuapp.com/>