Genetic Algorithm Solving Travelling Salesman Problem.

Łukasz Kwiecień

Computer Programming
Semester Project
Teacher: mgr. inż. Wojciech Dudzik

Wydział Automatyki, Elektroniki i Informatyki
Politechnika Śląska
Polska
12.06.2020

Contents

1 Topic

The topic of the project is a program solving the travelling salesman problem using genetic algorithm. Program should also solve the problem using brute-force method.

2 Topic analysis

2.1 Travelling Salesman Problem

According to Wikipedia:

> "The travelling salesman problem asks the following question:"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" "

There are a lot of different routes to choose from, but finding the best one in a efficient way is really difficult. That's because the TSP is classified as **NP-hard** problem in combinatorial optimization and belongs to the class **NP-complete** of combinatorial optimization problems. This means that the TSP has no "quick" solution and the complexity of calculating the solution will increase with the number of cities.

The problem can be solved by trying every possible route to determine the shortest one. However, as the number of cities increases, the correspoding number of possible routes can exceed the capabilities of even the fastest computers. Because the time complexity of the brute-force solution is $O(n!)$ number of the routes to check e.g for 10 cities is bigger than 300,000.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Figure 1: table shows how long we should expect different algorithms to run on modern hardware

As you can see on *figure 1*, by using the brute-force method for even 30 cities we will not obtain the solution during our lives. In order to solve the problem faster it is necessary to use different approach.

2.2 Genetic Algorithm

One of the approaches that allows us to obtain really good solution quite fast is the Genetic Algorithm. Genetic algorithm is a search heuristic that is inspired by Darwin's theory of natural evolution. Genetic algorithms are commonly used to generate high-quality solutions for optimization problems.
Genetic algorithm can be divided into five phases:

1. Initial population - randomly generate the individuals.
2. Fitness function - determine how good is every individual.
3. Selection - select the parents that will create new individuals (children).
4. Crossover - combine previously selected parents and create the children.
5. Mutation - Randomly swap some genes in the chromosome to introduce the diversity in the population.

Process begins with set of individuals called a **Population**. Each individual is a solution to the problem. Individual contains variables known as **Genes**. Genes joined into a string are called a **Chromosome** (solution).

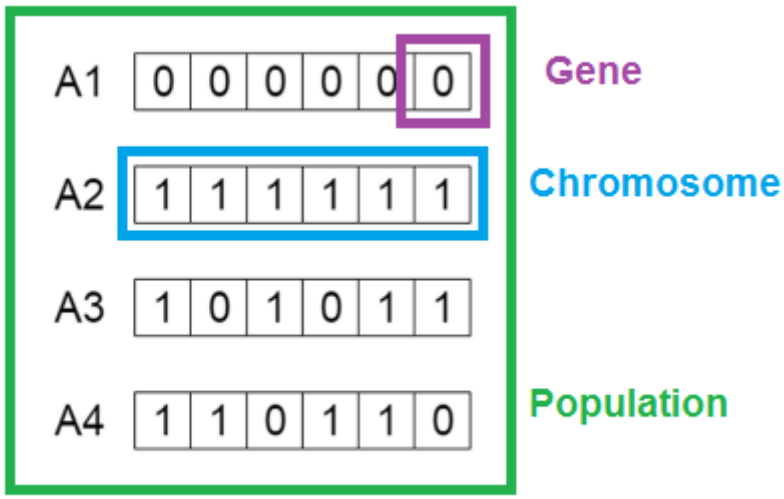

Figure 2: Population, gene and chromosome

2.3 Implementation

The design process began with the decision that program would display the solutions on the screen and allow the user to change the algorithm settings using buttons and sliders. I had to choose a library that would provide the simple graphical user interface. I decided to choose the Texus GUI library which is based on SFML library. TGUI is a really simple and well-documented library, and since I worked with SFML before, I chose this library.

Then I started thinking about the class design of the program. Since I wanted my algorithm to perform well and offer different configurations to solve the problem I decided to implement 3 selection methods and 3 crossover methods. This would allow the user to check whether a combination other than the one he was using would provide a better solution. Because I wanted to use different variants of the selection and crossover operators, and to be able to switch between them during runtime the strategy pattern was the perfect choice for the design pattern.
Preliminary assumptions of the class structure looked like this:

- GUI class
- Brute-Force solution class
- Class representing a City - Gene in terms of Genetic Algorithm
- Class representing a Path - Chromosome
- Class represeting a Generation - Population
- Selection classes - interface and implementations of the strategies
- Crossover classes - interface and implementations of the strategies
- Fitness class
- Mutation class
- Genetic Algorithm class responsible for the GA solution

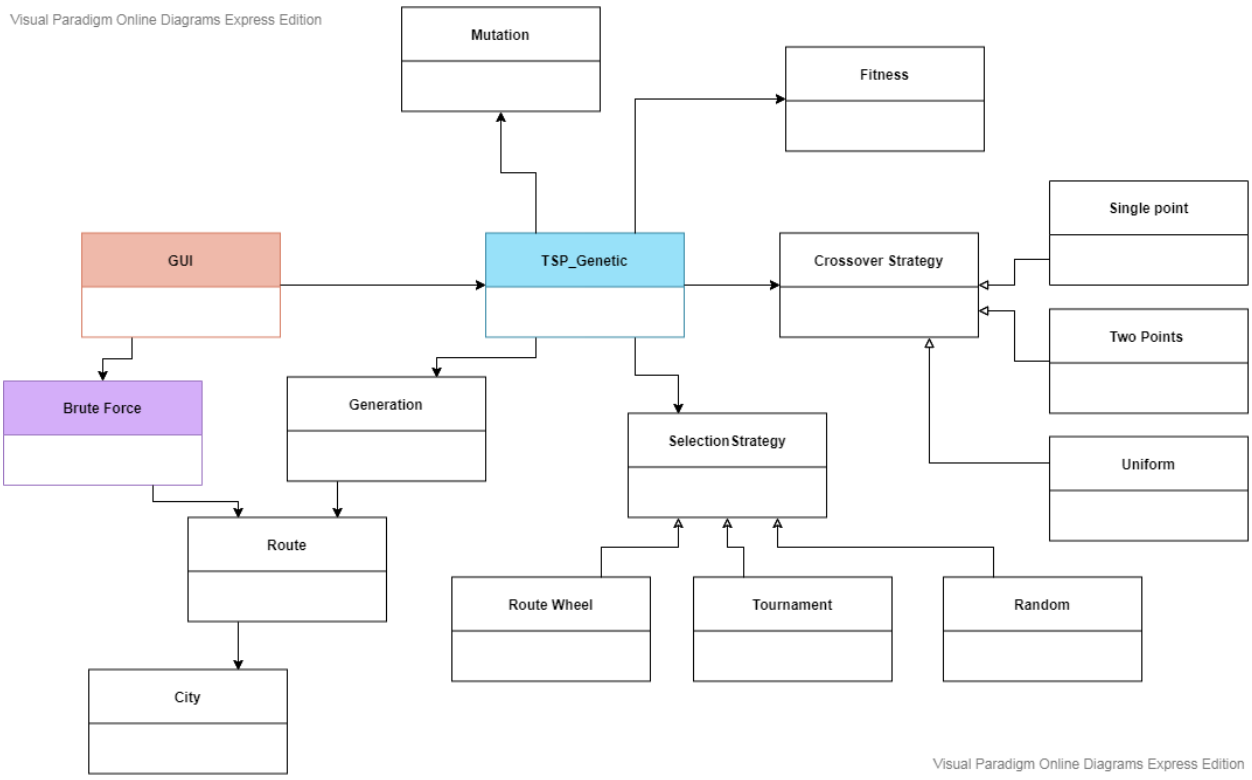

Figure 3: Preliminary class diagram

3 External specification

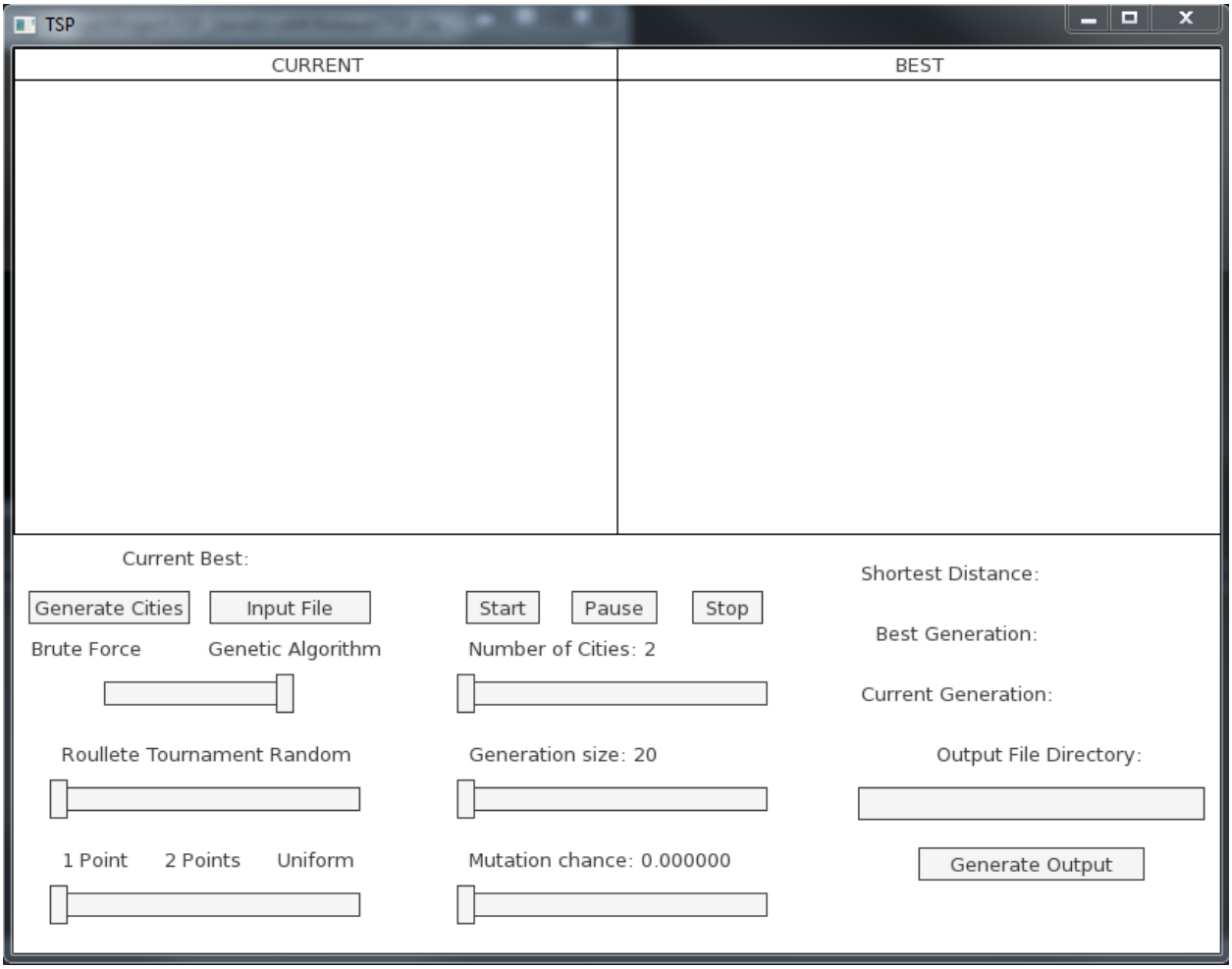

Figure 4: Program interface

As you can see on *figure 4*, program can be divided into two sections. The upper section contains 2 windows which display the solutions. The bottom section contains program settings and buttons to control it.

(*Figure 5*)Cities are represented by black dots, the solution (path) is represented by the black line connecting the dots.

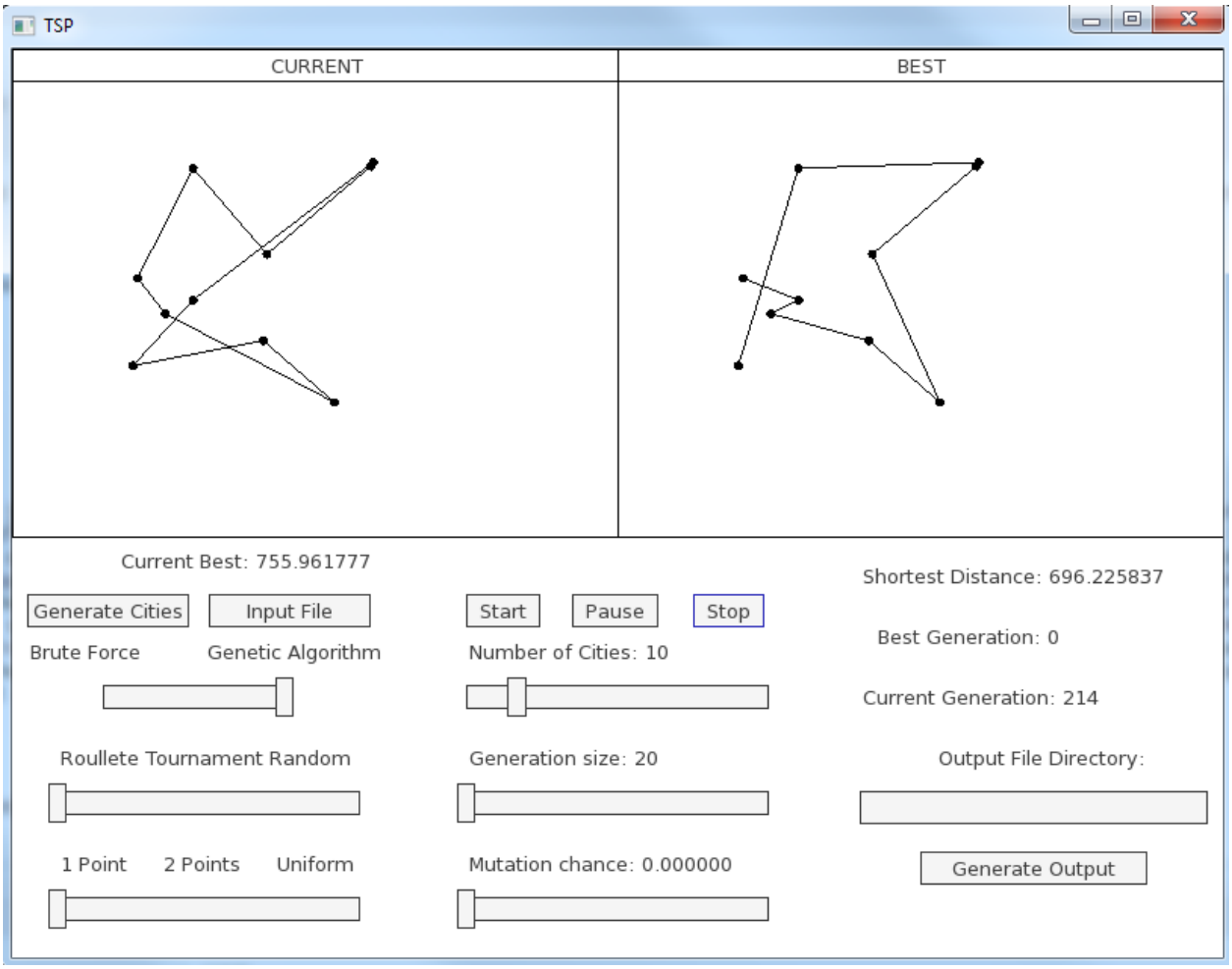

Figure 5: Graphical representation of the solution

3.1 Basic use

3.1.1 Brute-Force

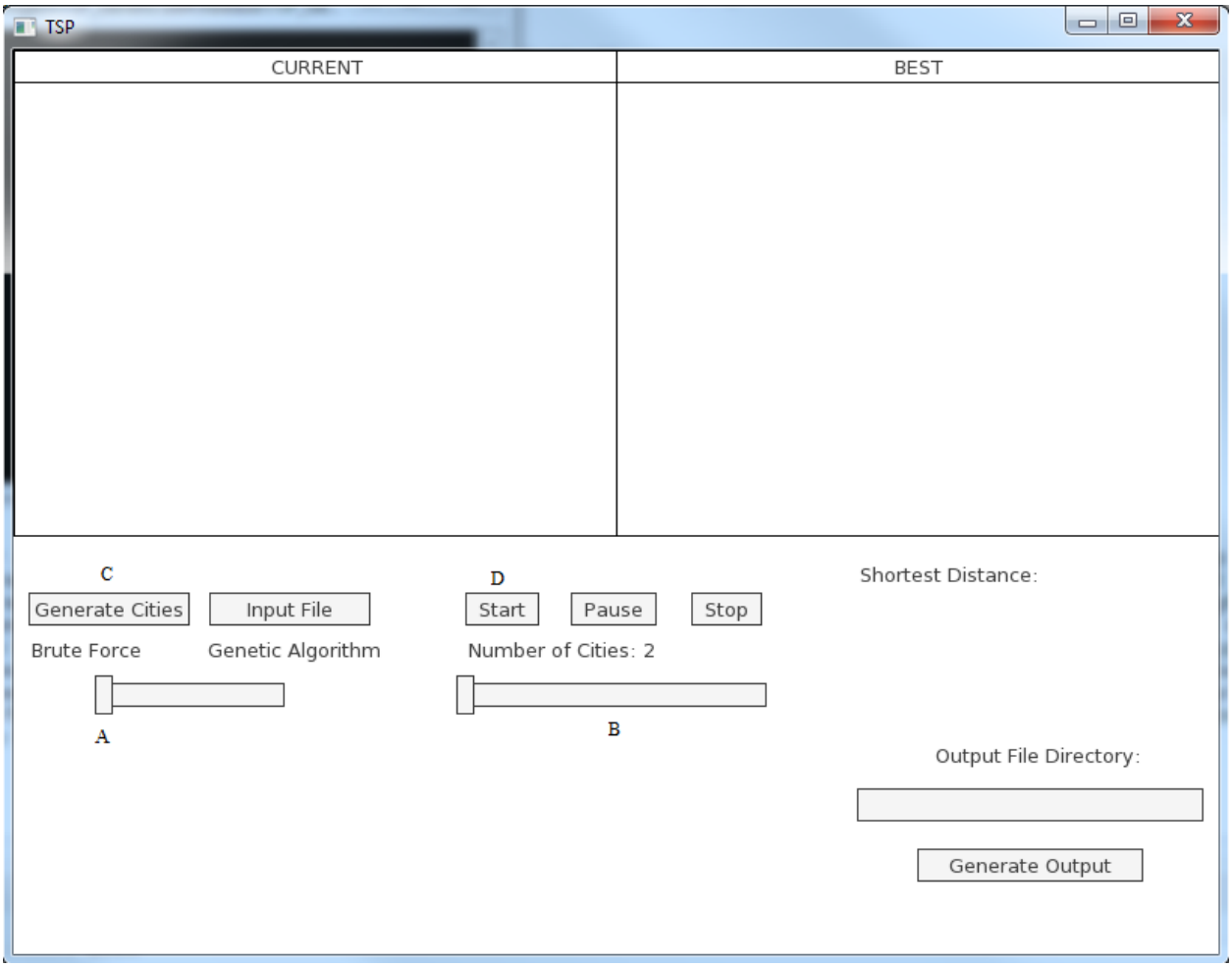

Figure 6: Brute Force

When you want to solve the TSP using brute-force method the type slider **(A)** has to be switched to the brute force side. Then you can choose the number of cities that will be used. In the brute-force case you can choose from 2 to 10 cities, which is due to the time complexity of this solution. After that you can generate new citites **(C)** or start generating the solution **(D)**. Generating cities is not necessary because they will be generated after clicking the start button **(D)** but only if there was no cities generated before, otherwise to change cities, use the **(C)** button before clicking the start **(D)**. Without generating cities, TSP will be solved for the same set of cities. Program displays the first and the best path obtained by the brute-force method as well as the shortest distance.

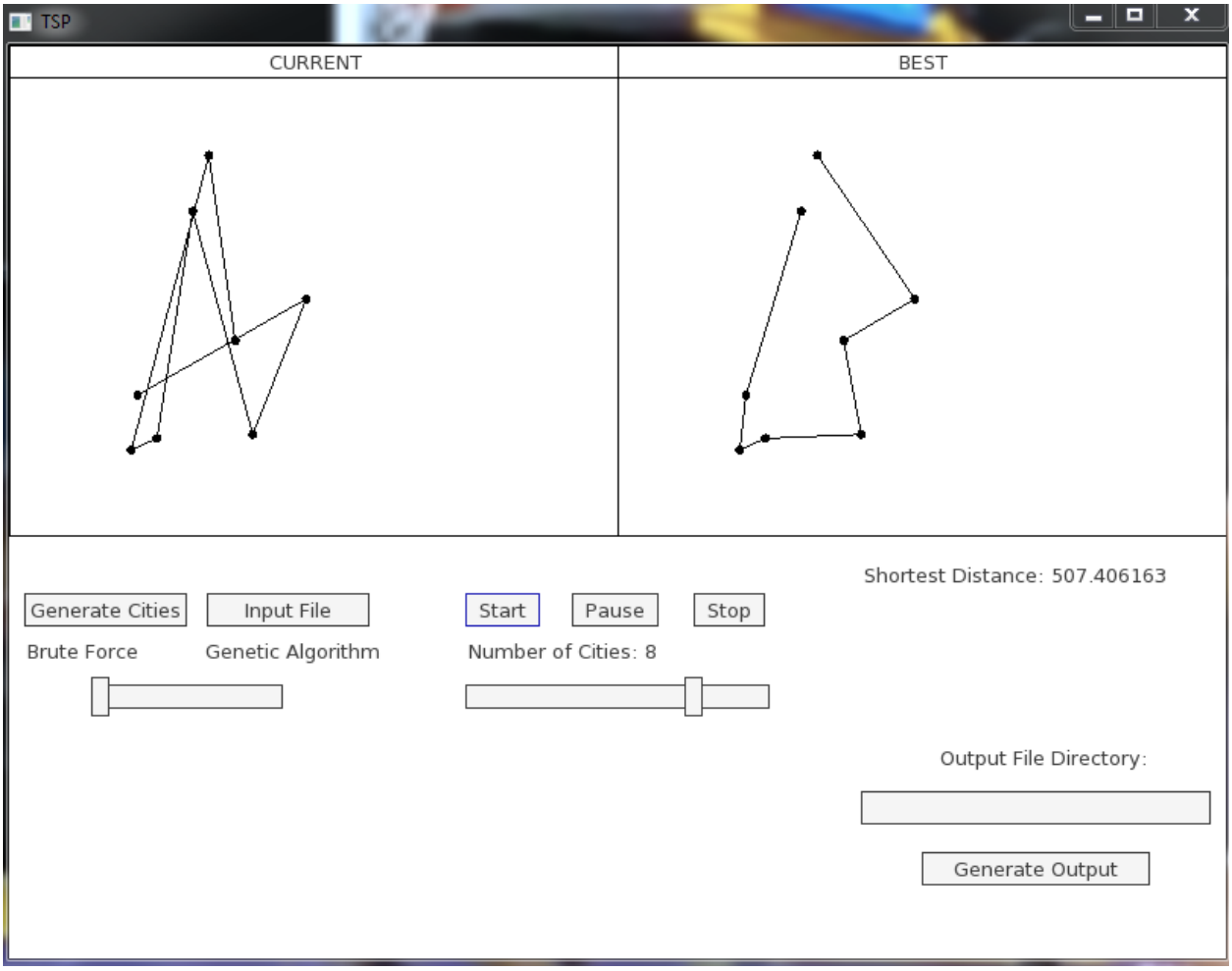

Figure 7: Brute Force solution

3.1.2 Genetic Algorithm

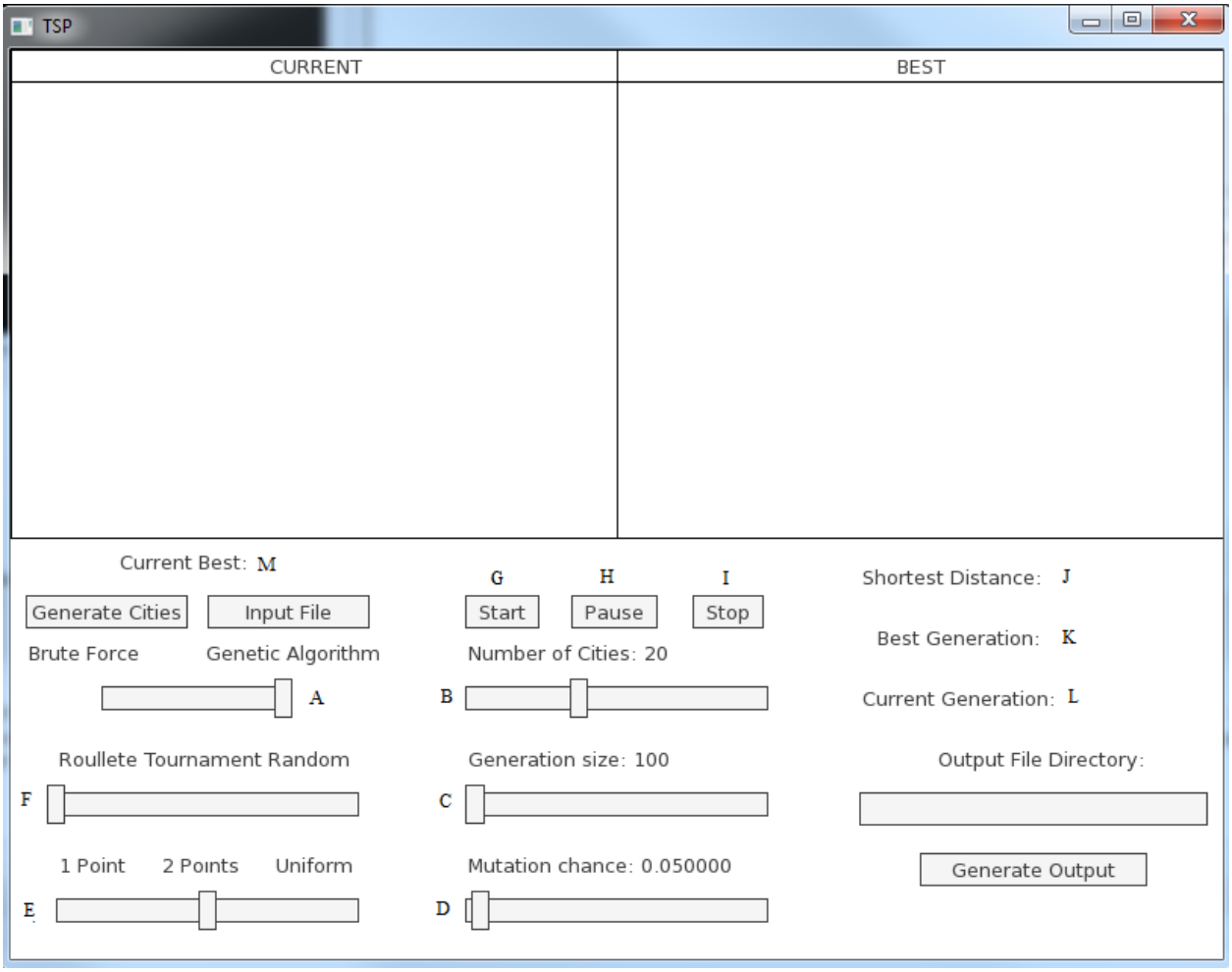

Figure 8: Genetic Algorithm

To solve the TSP using genetic algorithm the type slider **(A)** has to be switched to the GA side. The number of settings here is greater than for the brute-force method. Let's explain their meaning:

B) **The number of cities.**

- You can choose from 2 up to 50 cities in this method.

C) **Size of the generation.**

- Here you can decide how many individuals (chromosomes/paths) will be created in one generation.
- You can choose from 20 up to 2500.
- The program may run slower for large generations with a large amount of cities.

D) **Mutation chance.**

- Here you can set the probability that the chromosome will be mutated.

E) **The crossover method.**

- 1 point - Random crossover point is selected and the tails (cities after that point) of its two parents paths are swapped to get new paths (children).
- 2 points - Two random crossover points are selected and the cities between this points of its two parents paths are swapped to get new paths (children).
- Uniform - In this crossover method each gene (city) is selected randomly from one of the corresponding genes of the parent chromosomes (paths).

F) **The selection method.**

- Roullete - picks two chromosomes with a probability which is proportional to its fitness.
- Tournament - picks the two fittest individuals from a randomly selected pool of four.
- Random - randomly picks 2 chromosomes from the generation.

G) **Start button.**

- Runs the algorithm for the same cities.
- You can run the algorithm with the same settings for the same cities multiple times.

H) **Pause button.**

- You can pause the algorithm whenever you want.
- E.g pause and generate the output file for the current solution.

I) **Stop button.**

- When you are satisfied with the solution you can stop the algorithm.
- To change the method to the brute-force you have to stop the algorithm.
- To start the algorithm with changed settings you have to stop and restart the algorithm.
- To generate new cities you have to stop the algorithm.

J) **Shortest distance.**

- The distance of the best solution ever.

K) **Best generation.**

- Number of the best generation.

L) **Current generation.**

- Number of the current generation.

M) **Current best.**

- The distance of the best solution from the current generation.

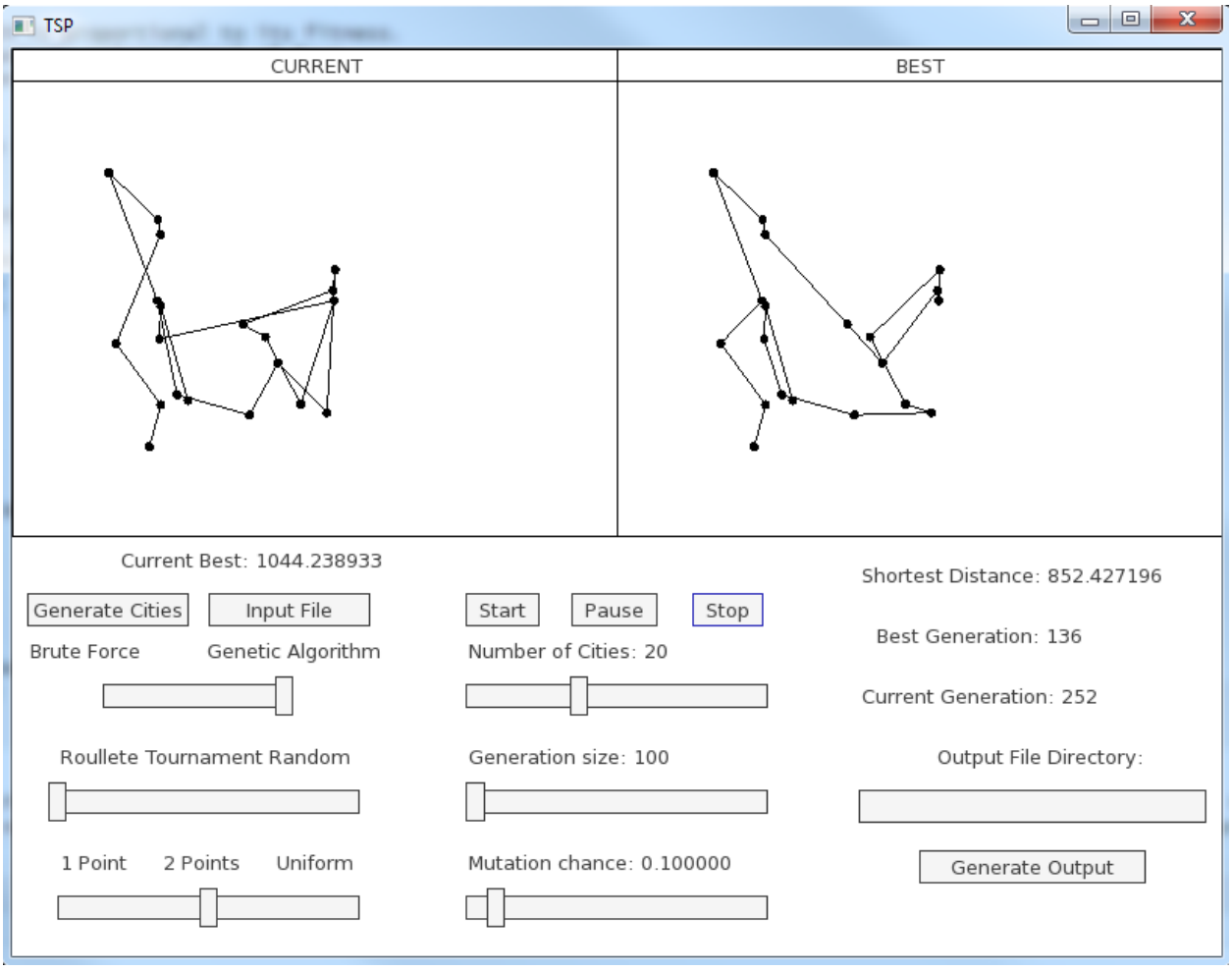

Figure 9: Genetic Algorithm solution

3.2 Input File

Program supports reading the input data from the file. It can be **.txt** or **.csv** file.

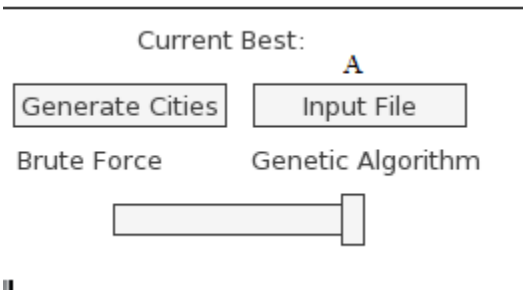

Figure 10: Input button

To open the load file menu you have to click the input file button **(A)**.

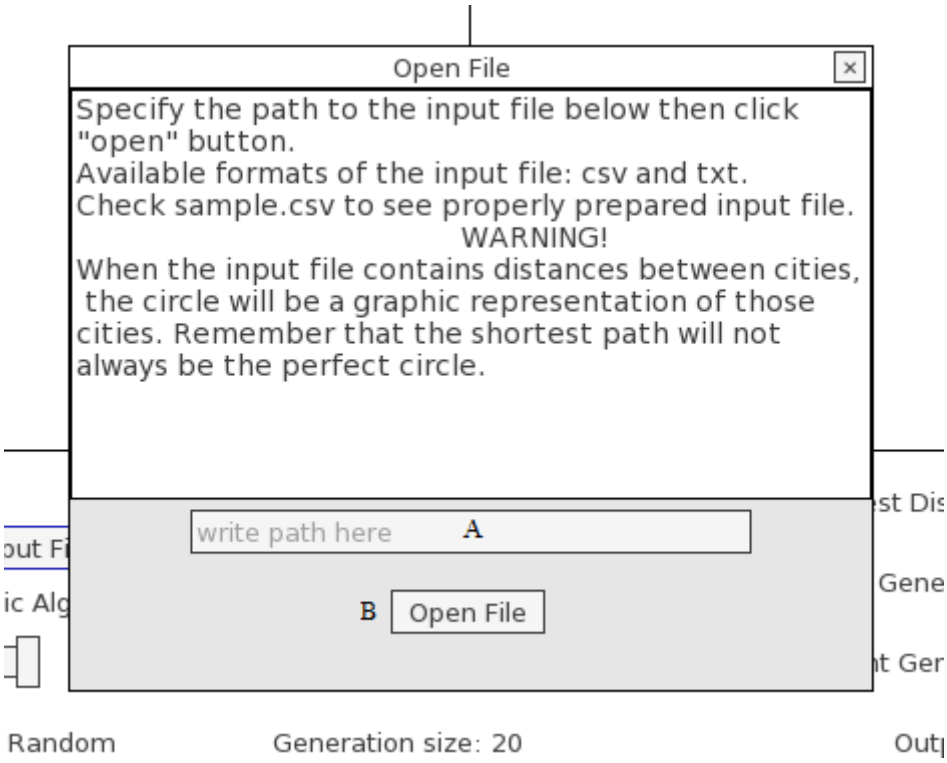

Figure 11: Open file menu

You have to provide the exact path to the file into the box **(A)** and the click the button **(B)**.
If the menu disappears, the cities have been successfully loaded.
If the input file is not correctly formatted or there is some invalid data, program will display a messagebox with the information about the problem.

There are two types of the input files.

Figure 12: Example of the error with the input file

3.2.1 Distances

First one is the “DIST” type.

```
DIST;
6;
0.0;8.25;13.93;9.22;5.1;12.17;
8.25;0.0;11.4;4.12;7.62;11.66;
13.93;11.4;0.0;15.52;8.94;3.16;
9.22;4.12;15.52;0.0;10.63;15.65;
5.1;7.62;8.94;10.63;0.0;7.07;
12.17;11.66;3.16;15.65;7.07;0.0;
```

Above you can see a sample input file of type “DIST”. Each line ends with a semicolon.

The first line of the file must contain the information about the input type. In this case - “DIST”.

The next line contains the number of cities. It has to be an integer and has to be greater than 2 and less than 50.

Then there should be a matrix containing distances between the cities. Each line of the file is one row of the matrix. Distances have to be positive doubles, separated by a semicolon, with at least one digit after the decimal.

0 - is incorrect

0.0 - is correct

Cities will lie on the circle, it is for just a graphical representation because in this case there are no x and y coordinates.

3.2.2 Coordinates

The second type is the “COORD” type.

```
COORD;
6;
104.478;144.455;
133.21;50.6821;
179.856;73.738;
120.529;144.506;
124.429;117.287;
177.98;211.181;
```

Above you can see a sample input file of type “COORD”. As for the “DIST” type, the first two lines have to contain the informations about the type and the number of cities.
Rest of the file contains x and y coordinates of each city. Each line represents one city.

Distances have to be positive doubles, separated by a semicolon, with at least one digit after the decimal. Moreover there is an extra condition for this type of input.

x coordinate has to be less than 400

y coordinate has to be less than 300

3.3 Output file

The program can generate an output file that is a summary of the solution.

```
Generation Number: 5
Shortest distance: 391.138
Shortest Path: 3->0->4->1->2->5->3
Number of Cities: 6
Cities:
FORMAT:  x ; y ;

104.478 ; 144.455 ;
133.21 ; 50.6821 ;
179.856 ; 73.738 ;
120.529 ; 144.506 ;
124.429 ; 117.287 ;
177.98 ; 211.181 ;

DISTANCES BETWEEN CITIES:

 0.0000 ; 97.1236 ; 102.5914 ; 16.0000 ; 33.0151 ; 98.4124 ;
 97.1236 ; 0.0000 ; 51.4296 ; 93.7710 ; 66.4831 ; 165.9397 ;
 102.5914 ; 51.4296 ; 0.0000 ; 91.5478 ; 69.8140 ; 137.0036 ;
 16.0000 ; 93.7710 ; 91.5478 ; 0.0000 ; 27.1662 ; 87.2067 ;
 33.0151 ; 66.4831 ; 69.8140 ; 27.1662 ; 0.0000 ; 107.0420 ;
 98.4124 ; 165.9397 ; 137.0036 ; 87.2067 ; 107.0420 ; 0.0000 ;
```

Sample output file for the genetic algorithm solution.

```
  Brute Force solution
Shortest distance: 391.138
Shortest Path: 0->4->1->2->5->3->0
Number of Cities: 6
Cities:
FORMAT:  x ; y ;

104.478 ; 144.455 ;
133.21 ; 50.6821 ;
179.856 ; 73.738 ;
120.529 ; 144.506 ;
124.429 ; 117.287 ;
177.98 ; 211.181 ;

DISTANCES BETWEEN CITIES:

 0.0000 ; 97.1236 ; 102.5914 ; 16.0000 ; 33.0151 ; 98.4124 ;
 97.1236 ; 0.0000 ; 51.4296 ; 93.7710 ; 66.4831 ; 165.9397 ;
 102.5914 ; 51.4296 ; 0.0000 ; 91.5478 ; 69.8140 ; 137.0036 ;
 16.0000 ; 93.7710 ; 91.5478 ; 0.0000 ; 27.1662 ; 87.2067 ;
 33.0151 ; 66.4831 ; 69.8140 ; 27.1662 ; 0.0000 ; 107.0420 ;
 98.4124 ; 165.9397 ; 137.0036 ; 87.2067 ; 107.0420 ; 0.0000 ;
```

Sample output file for the brute-force solution.

As input file you can use both city coordinates and the distance between them.
A generated solution is needed to generate the output file.
To get the output file you have to provide path to the directory where the file should be generated and then click the button **(A)**.

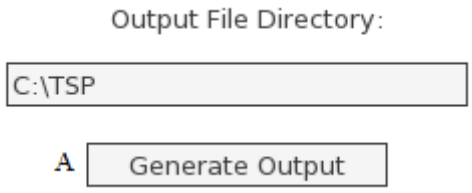

Figure 13: Output file

If the file is correctly generated, program will display the box with information about that.

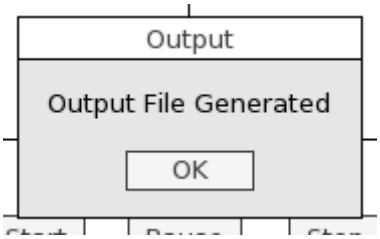

Figure 14: File was generated

If there are problems, e.g path is invalid program will display a messagebox with the information about the problem.

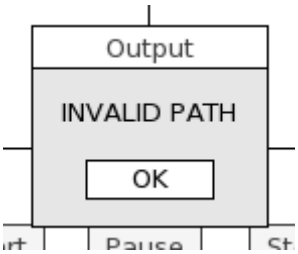

Figure 15: Example of the error with the output file

4 Internal specification

4.1 Classes

Informations about the classes, its methods and members can be found in the documentation under the link :

https://lukiop7.github.io/TSP-Gen/

Or in the attached documentation.pdf file.

4.2 Data Structures and Algorithms

4.2.1 Algorithms

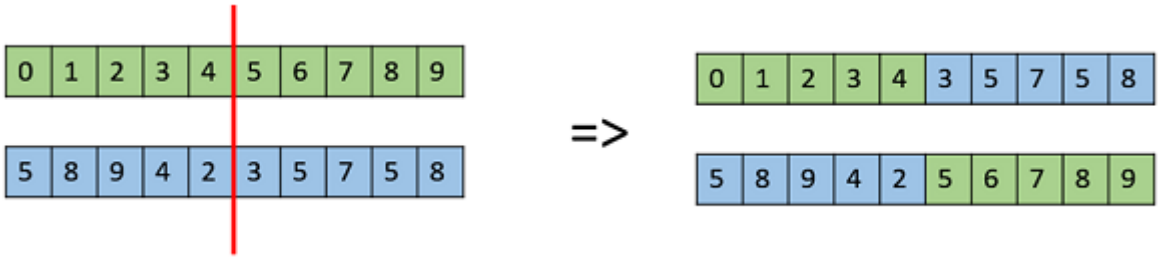

Figure 16: Generalization of the single point crossover

4.2.1.1 Single Point crossover In my case it was a bit tricky to implement this one, I had to create the algorithm which would check if the city can be added or it is already present in the path. When the city is already in the path it should find the one that can be added.
Algorithm checks if the city from the second parent is already present in the half from the first parent. If not it adds the city, otherwise iterate over the second parent's path and adds the first city than can be added to the child.

Implementation can be found in "Single_Point.cpp" file - singleCross function.

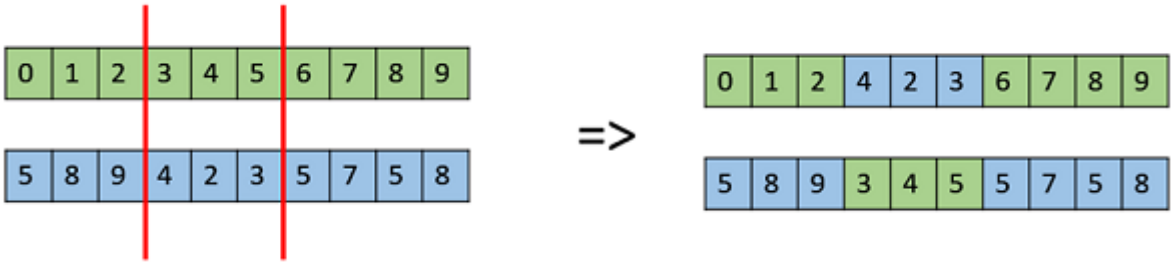

Figure 17: Generalization of the two points crossover

4.2.1.2 Two points crossover In this crossover method I implemented algorithm based on the one from the single point crossover. The difference in this case is that it has to iterate over the two segments of the second parent which are not used.

Implementation can be found in "Two_Points.cpp" file - twoCross function.

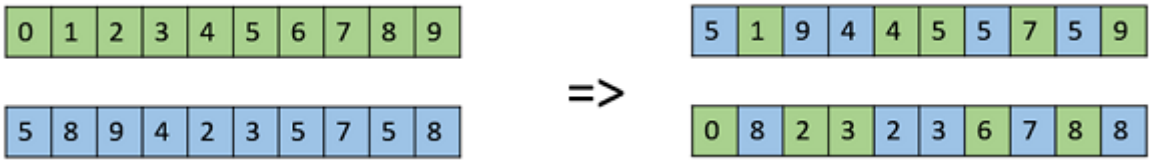

Figure 18: Generalization of the uniform crossover

4.2.1.3 Uniform crossover The hardest part was to perform the crossover in efficient way. My implementation uses a set which stores unused cities.If the city of the selected parent is already present in the child, the algorithm checks if it is possible to add the city of the other parent. If the city cannot be added because it is already present too, algorithm takes city from the set which contains the cities from the other parent that were not used.

Implementation can be found in "Uniform.cpp" file - uniformCross function.

4.2.1.4 Brute-Force Algorithm I had to implement the brute-force algorithm.

Pseudocode:

- get an initial path; call it T
- best_path <- T
- best_distance <- distance(T)
- while there are more permutations of T do the following
  - generate a new permutation of T
  - if distance(T) < best_distance then
  - best_path <- T
  - best_distance <- distance(T)

And the implementation:

```
    std::vector<int> index;
    for (int i = 0; i < cities_amount; i++)
        index.push_back(i);
    int ind = cities_amount;
    for (int i = 0; i < ind; i++)
    {
        index[i] = i;
    }
    int cnt = -1;
    do
    {
        Path tmp(ind);
        for (int i = 0; i < ind; i++)
        {
            tmp.addCity(cities->at(index[i]));
        }

        paths.push_back(tmp);
        cnt++;
        if (paths[cnt].getDistance() < best)
        {
            best = paths[cnt].getDistance();
            best_path = std::make_shared<Path>(paths[cnt]);
            index_best = cnt;
        }
    }
    while (std::next_permutation(index.begin(), index.end()));
```

4.2.2 Data Structures

I did not use any complex data structures in my program. I decided not to use lists, trees or graphs. Below are some of the STL containers I have used in my program.

> std::vector< City > cities; E.g Path.h

> std::vector< Path > paths; E.g Generation.h

Path and Cities are stored in the std::vector.

> std::map<double, int> sort_map; from createCities(std::vector<std::vector< double » input) in GUI.cpp

std::map is used when the cities form the input file of type “DIST” are created.

4.3 Object Techniques

4.3.1 Exceptions

Exceptions are thrown by many functions in the program. Moreover almost every exception that is thrown in the program is a custom exception. They can be found in “exceptions.h” file.
Example of the usage:

```
try
            {
                if (type->getValue() == 0)
                    FileIO::bruteFile(output_path->getText(), brute->getBestPath(), cities);
                else
                    FileIO::geneticFile(output_path->getText(), genetic->getCntbest(), genetic->getBest
            }
            catch (FilePath& e)
            {
                output_error->setText(e.what());
                output_error->setEnabled(true);
                output_error->setVisible(true);
                return;
            }
            catch (FileOpen& e)
            {
                output_error->setText(e.what());
                output_error->setEnabled(true);
                output_error->setVisible(true);
                return;
            }
            catch (std::exception& e)
            {
                output_error->setText(e.what());
                output_error->setEnabled(true);
                output_error->setVisible(true);
                return;
            }
```

Example of the exception class:

```
class FilePathExt : public std::exception
{
public:

    /**
     * @brief   Gets the message
     *
     * @returns Null if it fails, else a pointer to a const char.
     */

    const char* what() const throw () override
    {
        return "INVALID PATH OR EXTENSION";
    }
};
```

4.3.2 Templates

Templates are used only in the "helpers.h" file, in functions that allows us to check if the vector contains the given value e.g in the given range. Templates allows to use the functions for different data types.

Example:

```
    template <class T>
    bool contains(const std::vector<T>& vec, const int border, const T& value)
    {
        return std::find(vec.begin(), vec.end() - (vec.size() - border - 1), value) != vec.end() - (vec
            - 1);
    }
```

4.3.3 STL containers

STL containers are used many times in the code. I use them to store Cities, Paths etc.
Vectors, maps and sets are used.

Examples in the Data Structures paragraph.

4.3.4 STL algorithms and iterators

Functions from the algorithms library are often used. For example in the fitness function for sorting the entire generation. As well as in the previously mentioned contains function - std::find. They can be found in many places in the code, as well as the iterators which are often used with those functions.

Example from the fitness function:

```
std::sort(generation->getPaths().begin(), generation->getPaths().end(), std::greater<Path>());
```

4.3.5 Smart Pointers

Smart pointers are used many times in the project. They are the only pointers that were used. Strategy pattern was implemented with the usage of the smart pointers, as well as the GUI.

Example from the Genetic class:

```
    /** @brief  Pointer to the crossover strategy interface*/
    std::unique_ptr<Crossover> crossover_strategy;
    /** @brief  Pointer to the selection strategy interface */
    std::unique_ptr<Selection> selection_strategy;
    /** @brief  Pointer to the mutation object*/
    std::unique_ptr<Mutation> mutation_strategy;
    /** @brief  Pointer to the fitness object*/
    std::unique_ptr<Fitness> fitness_strategy;
    /** @brief  Pointer to the generation object */
    std::unique_ptr<Generation> generation;
    /** @brief  Pointer to the best ever path */
    std::shared_ptr<Path> best_path;
    /** @brief  Pointer to the best path from the current generation */
    std::shared_ptr<Path> current_best_path;
    /** @brief  Pointer to the vector with cities */
    std::shared_ptr<std::vector<City>> cities;
```

4.3.6 Regular expressions

Regular expressions are used in file processing. The input data and paths to the files are validated using regular expressions. The can be found in the FileIO namespace.

Example:

```
std::string checkFile(std::string path)
{
    std::regex check(R"(^(?:[\w]:|\\)(\\[A-Z a-z_\s0-9.-]+)+\.(txt|csv)$)");
    if (!path.empty())
    {
        if (std::regex_match(path, check))
        {
            std::ifstream sample(path);
            std::string line;
            std::getline(sample, line);
            sample.close();
            return line;
        }
        throw FilePathExt();
    }

    throw FileOpen();
}
```

4.4 General scheme of the program operation

After launching the program displays the GUI and waits for the user's action. When the button is clicked is GUI calls appropriate functions from the other classes, for example the start button calls the constructors of the BruteForce or Genetic classes with the parameters set by the user. Then the solution is generated and displayed on the screen. Appropriate functions are called from the classes responsible for them.

The way the program works can be generalized into phases:

- Display the GUI
- Wait for the user
- Pass the settings to the constructors
- Perform the algorithm
  - Brute-Force
    - Generate all possible paths and determine the best one
    - Display the solution
  - Genetic Algorithm - While is not paused or stopped
    - Perform selection using the strategy chosen by the user
    - Perform crossover using the strategy chosen by the user
    - Perform mutation
    - Calculate fitness
    - Check the best path
    - Display the solution

4.5 Class Diagram

Class diagram can also be found in the attached diagram.png file.

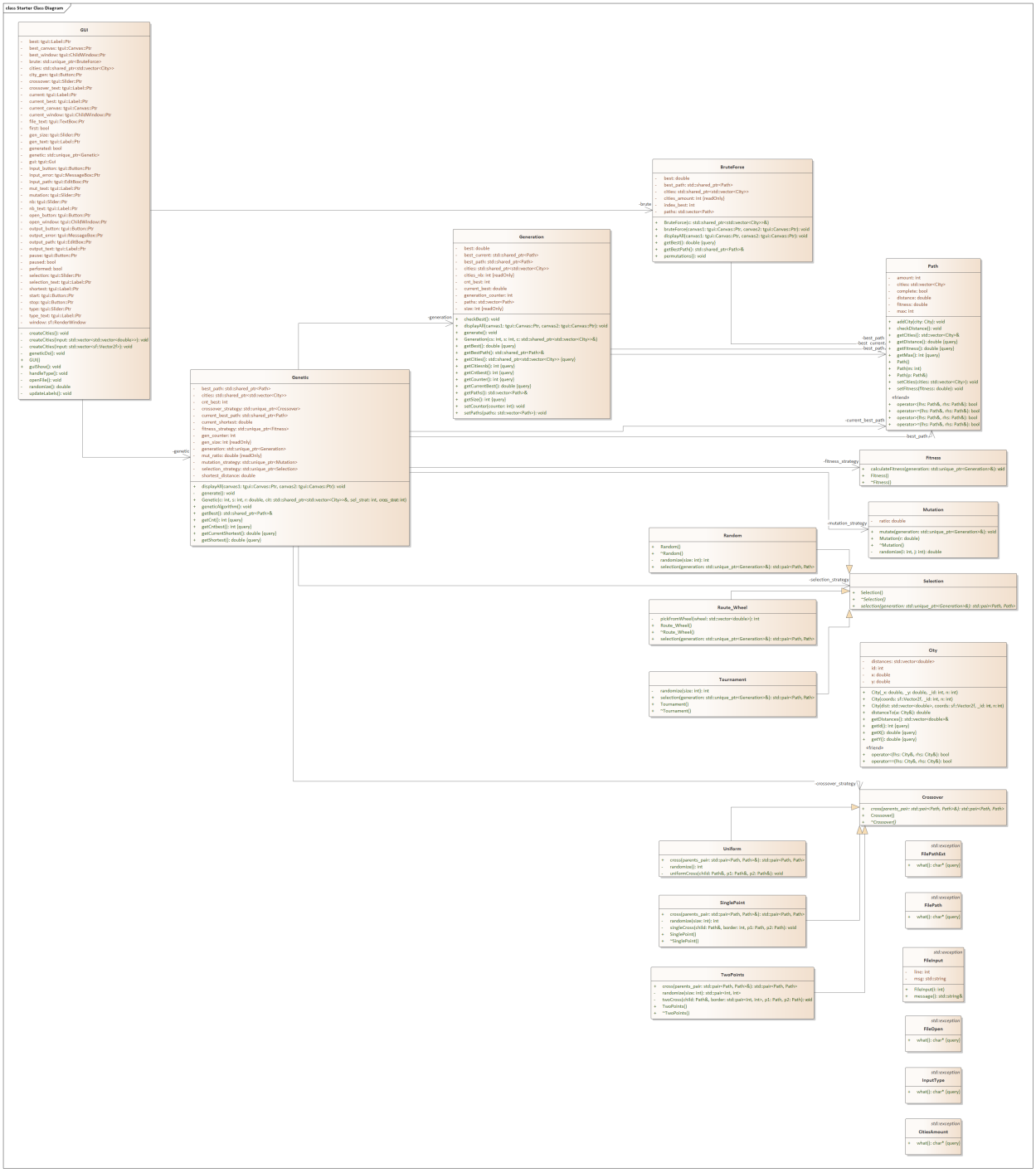

Figure 19: Class Diagram

5 Testing and debugging

Program was tested using different input files and clicking the buttons in many different orders. All bugs found during the testing were fixed. Most of them are minor problems, but there were 3 bugs that caused the program to give wrong solutions or crash.

1. After some time I discovered that my function that calculates the distance to a given city (using Pythagorean theorem) is not giving correct results. It was fixed by a small modification in the code in the City.cpp file.
2. It was possible to change the type slider to the Brute-Force solution during the genetic algorithm. After doing that the program was crashing. It was fixed by disabling the possibility to move the slider when the algorithm is running.
3. Even if the smart pointers are used in the program, it had memory leaks. It was caused by the cyclic references. I discovered it when I closed the program and the error "Program not responding" appeared. It was fixed by changing the vector of pointers to the City objects to the vector of the City objects in the Path class.

The program was also tested for memory leaks and and none were found:

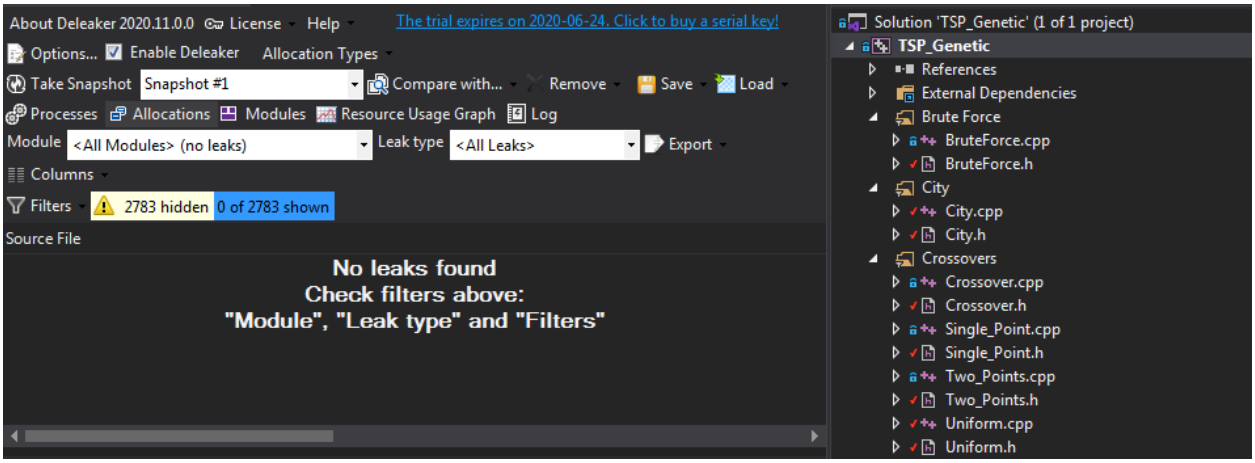

Figure 20: No memory leaks

Hidden leaks are the leaks caused by the SFML library connected with the render window. They are hidden because that leaks are created by known leaks function.

Genetic Algortihm after the testing works the best with tournament selection and two points crossover methods.