

Lab assignment 3

Solving the Traveling Salesman problem with Genetic Algorithms

Examiner	Email
Ning Xiong	ning.xiong@mdh.se
Students name	Email
Robin Calmegård	rcd10002@student.mdh.se
Dennis Stockhaus	dss10002@student.mdh.se

Abstract

This report is written in relation to Assignment 3 in the course 'DVA427 – Learning Systems' at Mälardalens University. The assignment is to develop an application that applies an optimization algorithm, e.g. Genetic Algorithm (GA) to solve The Traveling Salesman Problem (TSP). TSP is generally described as follows:

Given a list of cities and the distance between each other, which is the shortest route to travel across all the cities? Such that, you visit all the cities once and you start and finish in the same city.

This report will cover the following information: An explanation of the important operation of the employed algorithm to solve the TSP problem, an explanation of the representation of the individuals solution in the algorithm, The equation of the fitness function used, the configuration of parameters in the algorithm, Illustrations of the evolution of the population over iterations, also a showcase of the best result obtained by this algorithm.

Implementation

The software for the assignment is built with C# programming language and uses predefined graphical libraries offered in Microsoft's Integrated Development Environment (IDE) Visual Studio 2015 for the user interface (UI), as well as an external library 'Sparrow Toolkit' which contains a set of Data Visualization controls, charts etc.

User Interface

The user-interface is built to be easy to use, with controls for loading different layouts from file, configuring the parameters of the algorithm as well as collecting the results of computation (Figure 1).

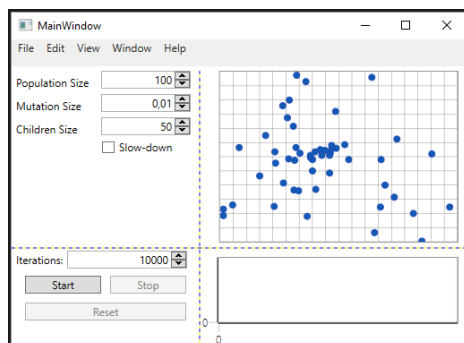


Figure 1. User-Interface, assignment 3.

Genetic Algorithm

```
Initialize first generation with Random DNA

Repeat
  For each Individual in Population
    Compute fitness and store result bound to individuals DNA
  End For each

  Sort Population by fitness
  Reproduce Population
    For each child to be created
      Cross breed ParentA, ParentB
      If (rnd_value == mutate)
        Mutate child
      End For each

  Select Individuals to be replaced by new generation
    Replace Individual from bottom of Population
    For each child in new generation
      n = iteration
      Population[populationSize - n] = new child
    End For each

  Force mutate duplicates
    For each Individual check each Individual
      If (IndividualA DNA == IndividualB DNA)
        Mutate IndividualB
      End For each

Check if (maxIterations reached or User stopped algorithm)
```

Key Operations of the Algorithm

Fitness Computation

The equation to compute the fitness of an individual in the population is the sum of the Euclidean distances of the route specified by the individuals DNA (Figure 2).

$$\left(\sum_{i=0}^{n-2} \text{EuclideanDistance}(\text{Point}[i], \text{Point}[i + 1]) \right) + \text{EuclideanDistance}(\text{Point}[n - 1], \text{Point}[0])$$

Figure 2. Visualization of fitness equation.

Representation of the Individuals solution

The representation of the Individuals solution or “DNA/Chromosomes” is made of an integer array with as many indices as locations in the example “layout”. The values in the DNA then represents an order of which to visit these locations (Figure 3). Also noteworthy is that the first location in the DNA is always the “home” of the individual e.g. the starting and endpoint of the route.

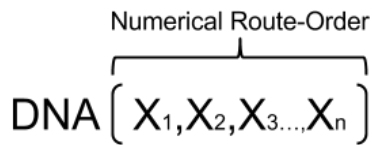


Figure 3. Visualization of Individual DNA.

Reproduction

Selection of Individuals to breed

Selection of Individuals for breeding in the algorithm is fairly straight forward; the population is split into two halves, the top half is the Elite of the population, the other half consist of those who didn't quite make the cut. The Elite is the only individuals allowed to breed and they are also only allowed to breed with other individuals from the Elite half of the population. The selection of breeding partners within the Elite population is done completely random where any Elite may breed with any other Elite any number of times until the quota of offspring's is met.

Cross-Breeding

Breeding is done between two individuals with a 50% uniform crossover-rate, which means that the offspring of two individuals is likely to have equal influences of both parents. This is done without any concern of the rules of the DNA-format which is:

- All locations needs to be visited.
- Locations may only appear once in a route.
- The route should start and finish on the same location.

To correct any flaws in the offspring's DNA a list is created before the breeding-process that contains all available locations, these are then matched with the new chromosome to be added and "popped" from the available-list, afterwards a correction is made checking for any duplicates in the DNA, if found – that chromosome is then replaced with the first available location in the available-list.

Mutation

In a genetic-algorithm mutation is used as an additional way to achieve diversity and explore new solutions in the population, without it the population will basically not evolve at all.

There are three different kinds of mutation-functions defined in the implementation; Chromosome-Swap, Chromosome-Shift and Chromosome-Reverse-Range. Possibly the most common function is the Chromosome-Swap where a chromosomes index simply is swapped with another (Figure 5). Although the swap function may solve more complex problems eventually we are impatient and want good results faster, that's where the Chromosome-Shift and Chromosome-Reverse-Range comes in handy.

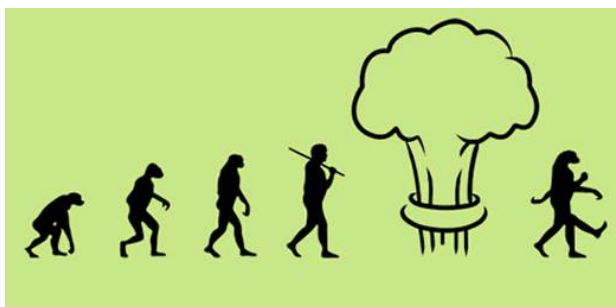


Figure 4. Just a funny picture of mutation. (<http://themetapicture.com/media/picture.jpg>)

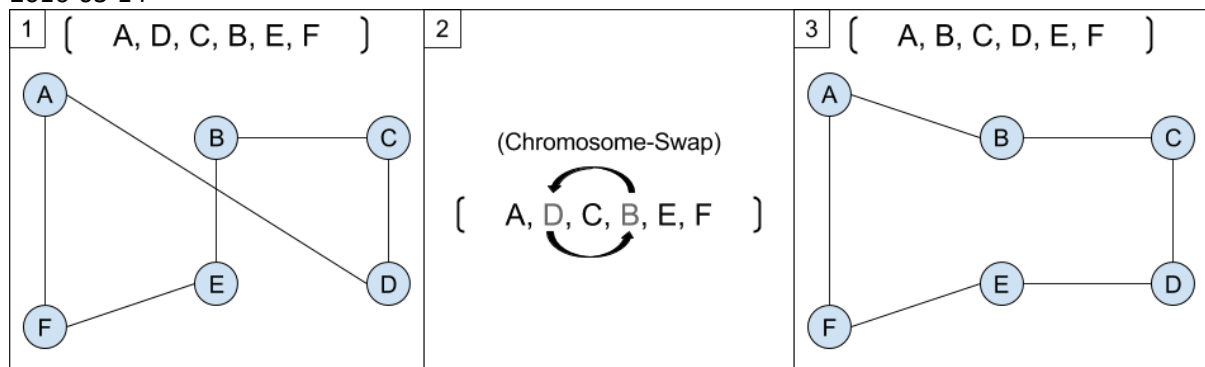


Figure 5. Visualization of Chromosome-Swap mutation.

The Chromosome-Shift mutation can solve a complex problem in a single mutation, for example: a route as AFBCDE \rightarrow ABCDEF (Figure 6), to solve this we need to “move” the first item to the last position, also moving all other items one index to the left. This cannot be achieved by using a single swap.

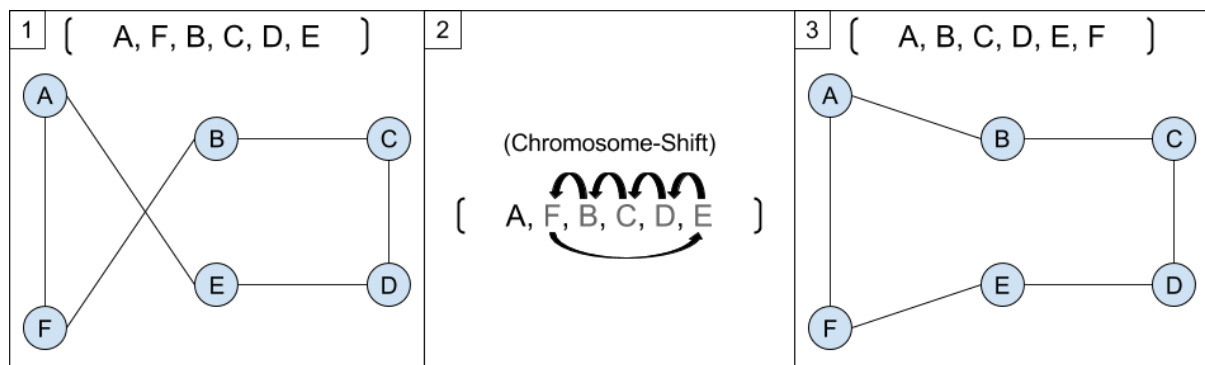


Figure 6. Visualization of Chromosome-Shift mutation.

Much like the shift function the Chromosome-Reverse-Range mutation may solve a more complex problem in a single mutation, for example: ABFEDCGH \rightarrow ABCDEFGH (Figure 7), those crossed lines seen in the figure means that we aren’t using the fastest route, to resolve this we simply need to avoid crossed lines. This is something that either the swap or the shift function can solve in a single mutation. The chromosomes FEDC will need to be visited in the reverse order \rightarrow CDEF to achieve this adjustment, and the reverse-range function does just that.

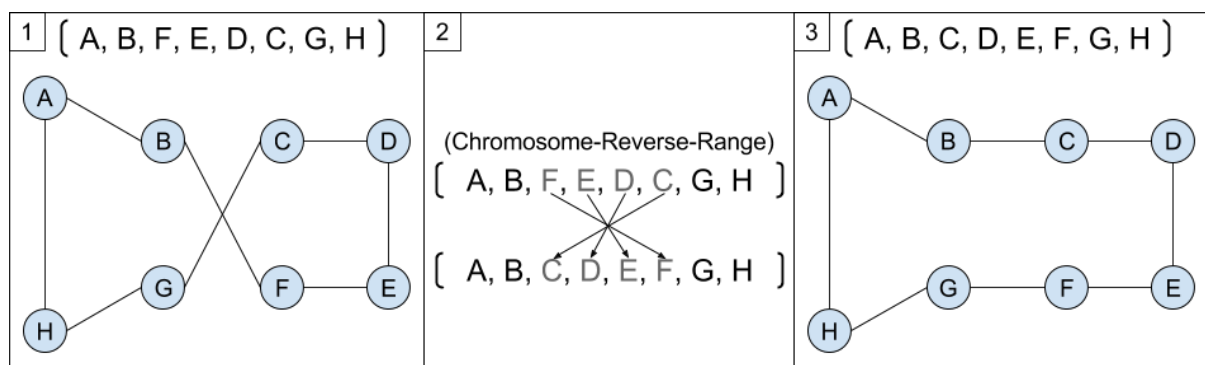


Figure 7. Visualization of Chromosome-Reverse-Range mutation

Another key-operation in this algorithm is the Force-mutation concept. In the search of the most optimized route through all locations diversity among the population is important, therefore we have deployed a function that simply forces duplicates in the population to mutate.

Results

Performance over generations

The table (Table 1) as well as the figure (Figure 8) below shows the evolution of population over generations. The configuration is as follows:

Populations-size: 100
Reproduction-volume: 50
Mutation-chance: 0.01
Crossover-rate: 0.5

In the figure and table you can clearly see that the performance of the population getting better over time. The tables illustrate both the fitness value/distance getting better/lower as well as clearly showing improved routes in the charts.

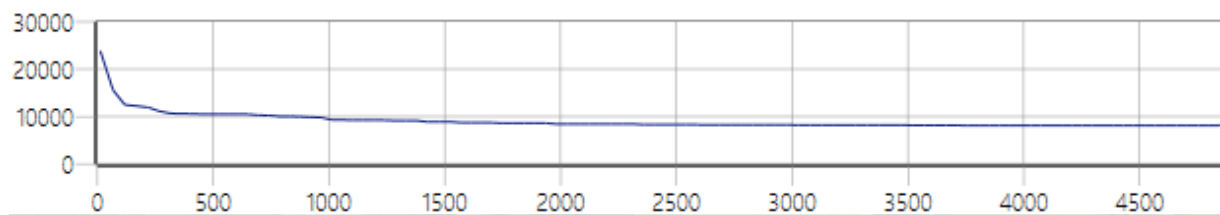
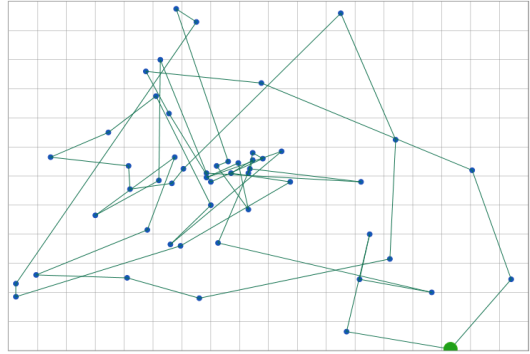
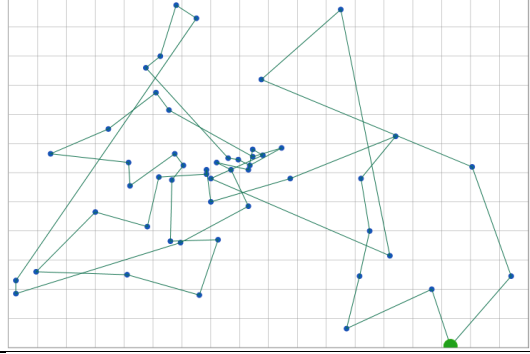
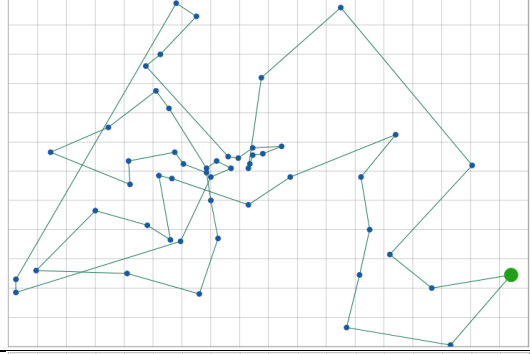
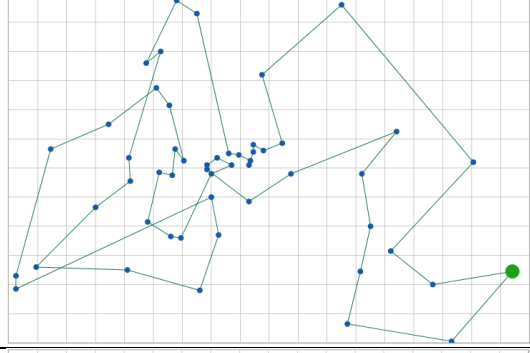
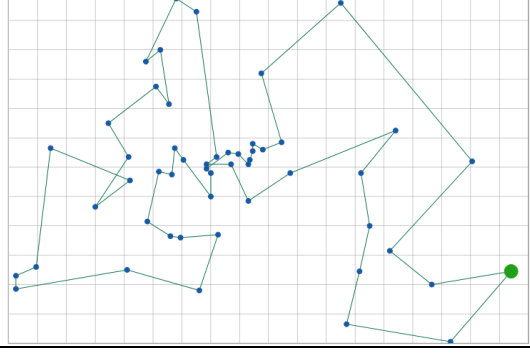


Figure 8. Evolution of fitness over generations.

Generation	Chart of Route (Best Individual)	Fitness value, distance (best individual)
12		23896,6252224232

67		15601,9795807474
118		12502,3272784567
307		10747,0674856611
917		9883,14766372731
1424		8906,78865741004

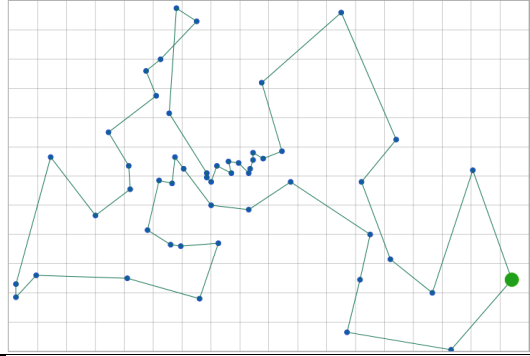
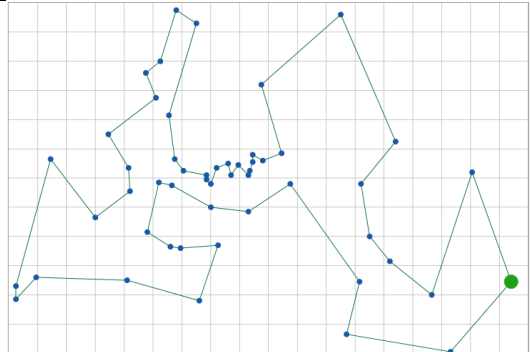
2496		8305,18586400055
4576		8101,63437625181

Table 1. Table illustrating performance over generations; generation: route: fitness/distance.

Best Results

The figure below (Figure 9) shows the best result obtained by this algorithm. The configuration is as follows:

Populations-size: 80
Reproduction-volume: 75
Mutation-chance: 0.35
Crossover-rate: 0.5
number of Generations: 10000

Achieved fitness-value/distance: 7342.105394305000

Some observations during testing of the algorithm led to a couple thoughts of how it might be improved, the biggest problem seem to be a declining of performance improvement after an arbitrary number of generations, well this could be viewed as expected as the number of “better” solutions becomes fewer as the population improves. The problem actually lies in the population after a while “specializing” in a very specific type of route, and that makes it very difficult for new better solutions to arise. The population is too good to be able to “start” new ideas from scratch and usually just dismiss any attempt to think “outside the box”. A possible solution would be to let multiple populations run at once and within some interval reset the worst of the two populations, this would give new ideas a chance grow freely and could improve the performance in the long run, well, it’s just a thought.

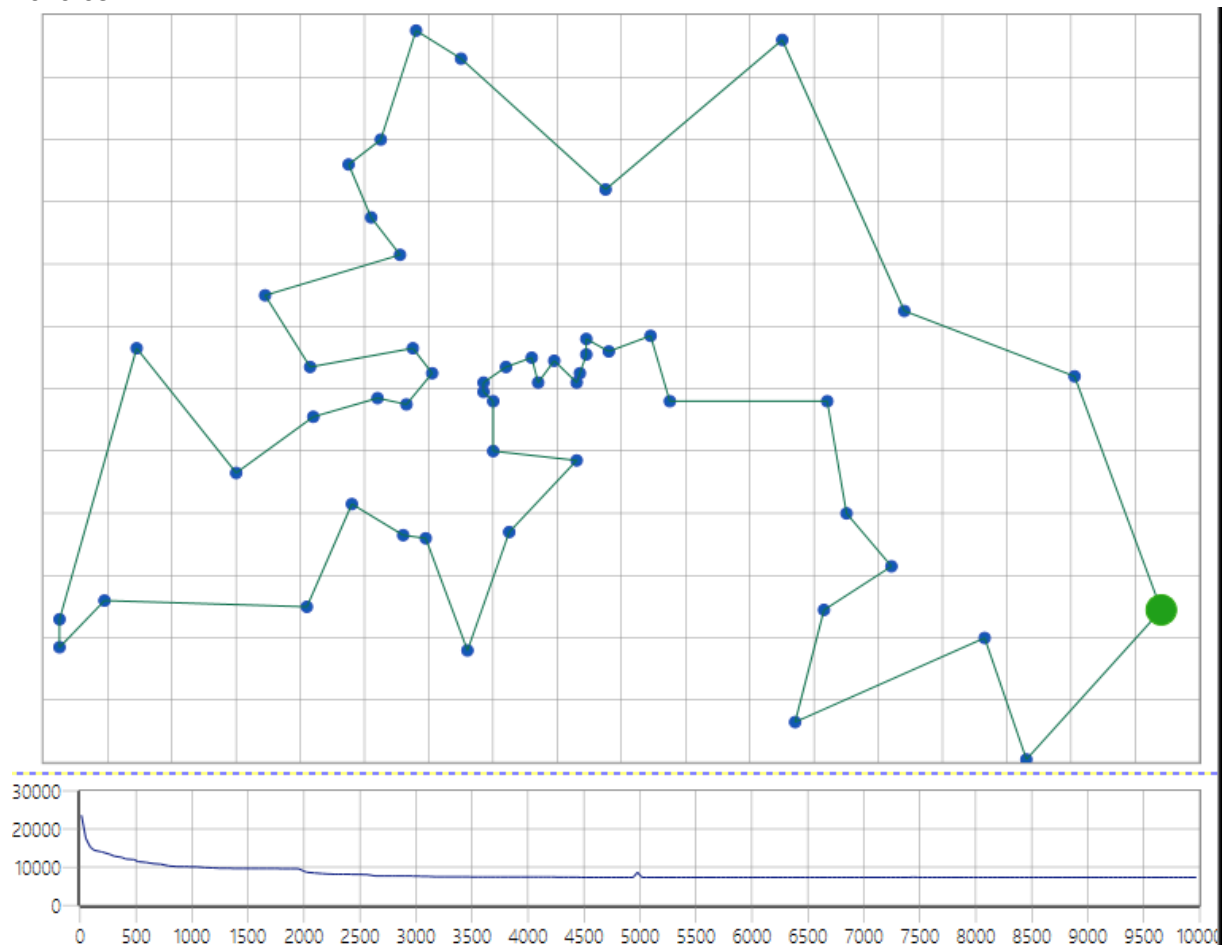


Figure 9. Illustration of "best yet" individual during extensive testing.