Genetic Shortest Path Finder

Manual

University of Fribourg, Switzerland
Supervised by
Eduardo Sanchez, eduardo.sanchez [at] epfl.ch

Lorenzo Clementi
lorenzo.clementi [at] gmail.com

Dominik Zindel
dominik [at] zindel.org

July 4, 2007

Contents

1 Introduction **1**

2 User Guide **2**
2.1 Requirements 2
2.2 Definition of the Graph 2
2.3 Application 2
2.3.1 Parameters 3
2.3.2 Possible Actions 4
2.3.3 Graphical Display 5
2.3.4 Statistical Information 5

3 Algorithm **8**
3.1 Dijkstra Shortest Path Algorithm 8
3.2 Genetic Shortest Path Algorithm 8
3.2.1 Initial Population 8
3.2.2 Crossover 8
3.2.3 Reproduction 10
3.3 Genetic Shortest Path Algorithm Without Constraints 10
3.3.1 Initial Population 10
3.3.2 Crossover 11
3.3.3 Reproduction 11

4 Conclusion **12**

List of Figures

1 Graph tab to draw a graph. 3
2 Parameters to the algorithm. 4
3 Possible action. 5
4 Graphical display. 6
5 Textual information about the progress. 7
6 Flow diagram for the Genetic Shortest Path Algorithm. 9

1 Introduction

The genetic shortest path finder has been developed during the summer term 2007 at the University of Fribourg in the frame of the lecture 'Systèmes bio-inspirés' (bio-inspired systems). The aim of the project was to define and implement a genetic algorithm allowing to find the shortest path in a graph and to compare it to the Dijkstra algorithm which is known to produce optimal results.

In this document first a user guide explaining how to use the program shall be given. Second, the algorithm defined is explained. Last, the project is evaluated in a short conclusion. It is explicitly not the aim of this document to provide a complete report on the project. Being targeted to non-scientific readers, this document has to be seen as a user manual rather than a report.

All documents including the source code can be found on `http://bioinsp.zindel.org`.

2 User Guide

In this section the use of the application is explained. As the application reads in an XML file containing the definition of the graph, this file has first to be created (confer to Section 2.2 for more details how to define an XML). Then, the application can be started.

2.1 Requirements

The application has been successfully tested on Microsoft Windows XP, Mac OS X and Ubuntu Linux. The best results are achieved when using Linux.

To be able to define a graph (cf. Section 2.2) you need a web browser (e.g. Firefox). To start the application, Java 1.5 or a newer version has to be installed.

2.2 Definition of the Graph

Before you can start to discover the program, you have to define a graph[1]. This definition has to be in an XML file in which also the position of each node has to be indicated.

For this project, an editor for the XML file has been created. The editor allows to easily define the graph without bothering about the XML syntax and can be found on `http://bioinsp.zindel.org`. You can log in with the username `testuser` and the password `testuser`. Once logged in follow the following steps:

1. Create a graph (not necessary if you only want to modify an existing one).
2. Create all vertices (can be seen as cities). The values `x-coordinate` and `y-coordinate` represent the coordinates of the node where the point (0,0) is in the top left corner.
3. Create all edges (can be seen as streets between the cities). Edges connect two vertices (which have to be defined previously). Besides an ID (name) they do have a weight. The weight represents the costs to take this edge (price for the train ticket) and might be the distance. You do not have to enter any unit, just numbers.
4. Once you have defined all vertices and edges, you can create the XML file by using the function `XML-Export`. Download the file using the link and save it on your computer (it is recommended to save it in the folder `maps` of the application folder).

2.3 Application

The application should be more or less self-explaining. However, a short overview shall be given here. To compute the shortest path you have to complete the following steps:

1. Define a graph and download the XML file (cf. Section 2.2).

[1]The application already comes with three example graphs so that you can immediately try out the application.

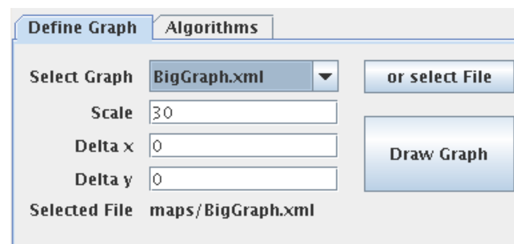

Figure 1: Graph tab to draw a graph.

2. Download the application from `http://bioinsp.zindel.org` and unzip the file.
3. Start the application. In order to do so enter the folder and double click geneticpath.bat on Windows or execute `./geneticpath.sh` in a terminal when using Mac or Linux.
4. In the left tab (cf. Figure 1) choose a graph. The files in the folder `maps` are shown in the dropdown menu, others can be chosen using the button `or select file`. You can indicate a zoom factor and an intendation. Using the parameters `Delta x` and `Delta y` you can define how many pixels the graph should be moved right and down, respectively. Once you have chosen all parameters, click on the button `Draw Graph` to draw the graph.
5. Select the right tab and choose all parameters (confer to Section 2.3.1 for more details).
6. Click on the two nodes between which you would like to compute the shortest path to select them.
7. Click on the button `Start Algorithms` to start the execution of the algorithms. Please confer to Section 2.3.2 for more details about the possible actions.
8. The progress during the execution is displayed in two ways: in the graphical panel (cf. Section 2.3.3) and in the panel `Algorithms` (cf. Section 2.3.4).
9. The algorithm automatically stops when the indicated number of generations are completed but you can also stop it earlier (cf. Section 2.3.2).

2.3.1 Parameters

The genetic algorithm defined for this project (see Section 3 for a description of the algorithm) takes several parameters as input. The following parameters can be defined in the graphical interface (cf Figure 2, note that a short description is also displayed when you move the mouse pointer over the name of the parameter in the graphical interface):

Max # Edges Init Sol Maximum number of edges that an invidual in the initial population may have.

Figure 2: Parameters to the algorithm.

Init Population Size The size (i.e. number of individuals) of the initial population (randomly created).

Size of Population The final size of the population. If the population exceeds this size, individuals are removed from it.

Min Difference The minimum difference between the two parents to allow for crossover. If the difference between the two parents is smaller than the given value, a reproduction is performed. This parameter only matters when only valid genomes are considered.

# of Generations The number of generations to be created. If the given value is 0, the algorithm runs 'forever'.

Generation Delay (s) The break between to generations in seconds. Allows to better follow the progress of the algorithm.

Crossover Probability The probability of a crossover (number from 0 to 1). If there is no crossover, a reproduction is done.

Consider Only Valid Genomes This option allows to indicate whether only valid genomes have to be considered. If checked, a genome has to be a valid path from the start to the end. Otherwise, all paths (which do not necessarily have to go from the start to the end) are taken into consideration.

2.3.2 Possible Actions

Beside by choosing the parameters (cf. Section 2.3.1) the user can influence the application through several possible actions (cf. Figure 3):

Reset Selection The selection of the start and end nodes is reset.

Start Algorithms Starts the genetic algorithm as well as the Dijkstra algorithm.

Figure 3: Possible action.

Reset Everything Reset the selected nodes and all information in the algorithms-panel (cf. Section 2.3.4). This button has to be pressed once the algorithm has completed before it can be executed again.

Pause/Continue Algorithm Pressing the button `Pause Algorithm` interrupts the execution of the algorithm. The execution can then be continued by clicking on the button `Continue Algorithm`.

Stop Algorithm Definitely stops the execution of the algorithm. The same actions happen as when the given number of generations had been achieved.

2.3.3 Graphical Display

The progress of the (genetic) algorithm can be observed in the drawing of the graph (cf. Figure 4). The colors of the edge have the following meaning:

Green The corresponding edge does not belong to any path.

Red The corresponding edge belongs to the optimal solution found by the Dijkstra algorithm.

Blue The corresponding edge belongs to the best solution found by the genetic algorithm so fare.

Cyan The corresponding edge belongs to the first child of the current generation of the genetic algorithm.

Yellow The corresponding edge belongs to the second child of the current generation of the genetic algorithm.

If a line is dashed this means that several paths go over this edge. If the dashes of one color are longer than those of another color, a path goes several times over this edge.

2.3.4 Statistical Information

In addition to the graphical display (cf. Section 2.3.3) the progress of the (genetic) algorithm can also be followed over a textual output (see Figure 5 for an example). The following information is indicated:

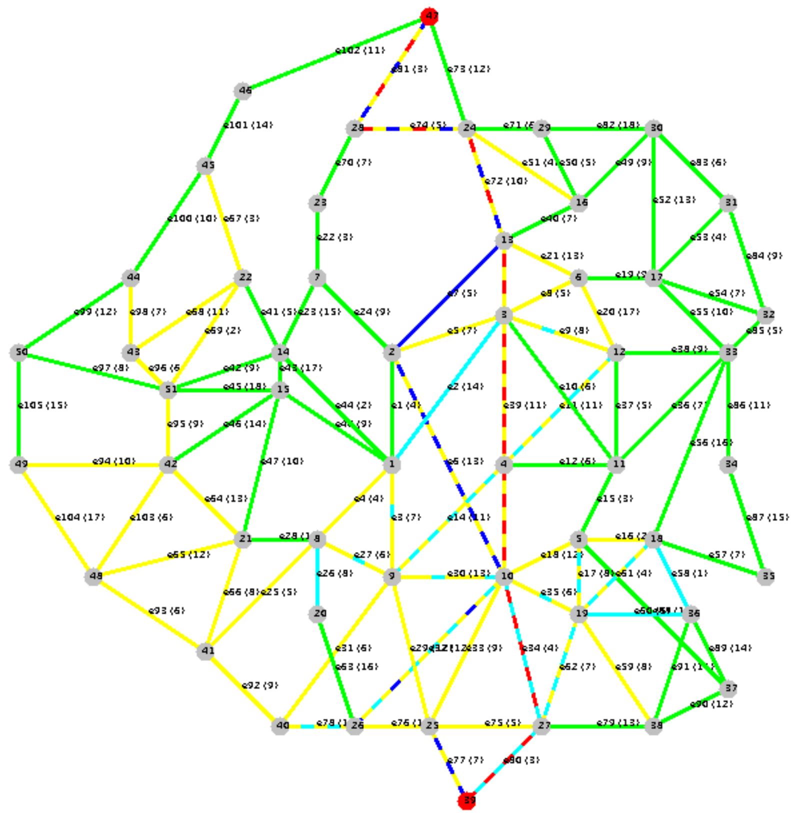

Figure 4: Graphical display.

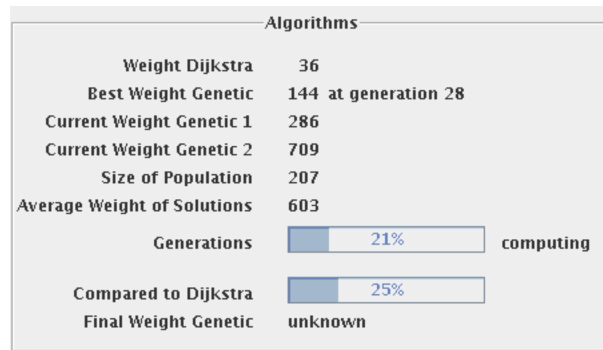

Figure 5: Textual information about the progress.

Weight Dijkstra The weight of the shortest path found by the Dijkstra algorithm.

Best Weight Genetic The weight of the shortest valid path found so far by the genetic algorithm.

Current Weight Genetic 1 The weight of the first child of the current generation of the genetic algorithm.

Current Weight Genetic 2 The weight of the second child of the current generation of the genetic algorithm.

Size of Population The current size of the population, i.e. the number of individuals living at the moment.

Average Weight of Solutions The average weight of all solutions in the current population.

Generations This progress bar displays the percentage of generations that has already been computed. When the mouse is moved over the bar, the number of the current generation is shown.

Compared to Dijkstra Shows how good the shortest valid path found so far by the genetic algorithm is compared to the solution found by the Dijkstra algorithm. 1% means that the genetic solution is 100 times heavier than the Dijkstra solution, 100% means that the genetic solution is equally good as the Dijkstra solution.

Final Weight Genetic The weight of the final solution found by the genetic algorithm, i.e. the best solution found by the genetic algorithm during the entire process.

3 Algorithm

The application implements three different algorithms:

1. Dijkstra Shortest Path Algorithm.
2. Genetic Shortest Path Algorithm.
3. Genetic Shortest Path Algorithm Without Constraints.

3.1 Dijkstra Shortest Path Algorithm

The Dijkstra Shortest Path algorithm always produces the best solution, i.e. the shortest path between the source and the destination vertices. The path computed by the Dijkstra algorithm is highlighted in red; the solutions produced by the genetic algorithms are compared to Dijkstra's result in order to evaluate how close to best possible solution they are.

3.2 Genetic Shortest Path Algorithm

The Genetic Shortest Path Algorithm looks for the shortest path between the source vertex and the destination vertex. Figure 6 shows the structure of the genetic algorithm.

3.2.1 Initial Population

At the beginning, an initial population of size s_0 is created by the following procedure:

repeat
  Pick a random edge leaving from the source vertex
  Add this edge to the current solution
  if the newly added edge reaches the destination vertex **then**
    Add this solution to the population and start over
  end if
  if the solution has reached the maximum length **then**
    Stop and start over
  end if
until the population has reached the required size

Once the initial population has reached the required size (chosen by the user), the evolutionary part of the algorithm is started. According to the probabilities p_t and p_r, at each iteration the crossover or the reproduction is executed.

3.2.2 Crossover

The crossover procedure works as follows:

Randomly pick a solution from the current population; this solution is called the *father*
Randomly split the solution into two sub-paths: the edge where the path is splitted is called the *splitting point*
Search within the population for another solution – different from the previous one – containing the same *splitting point* edge. This solution is called the *mother*

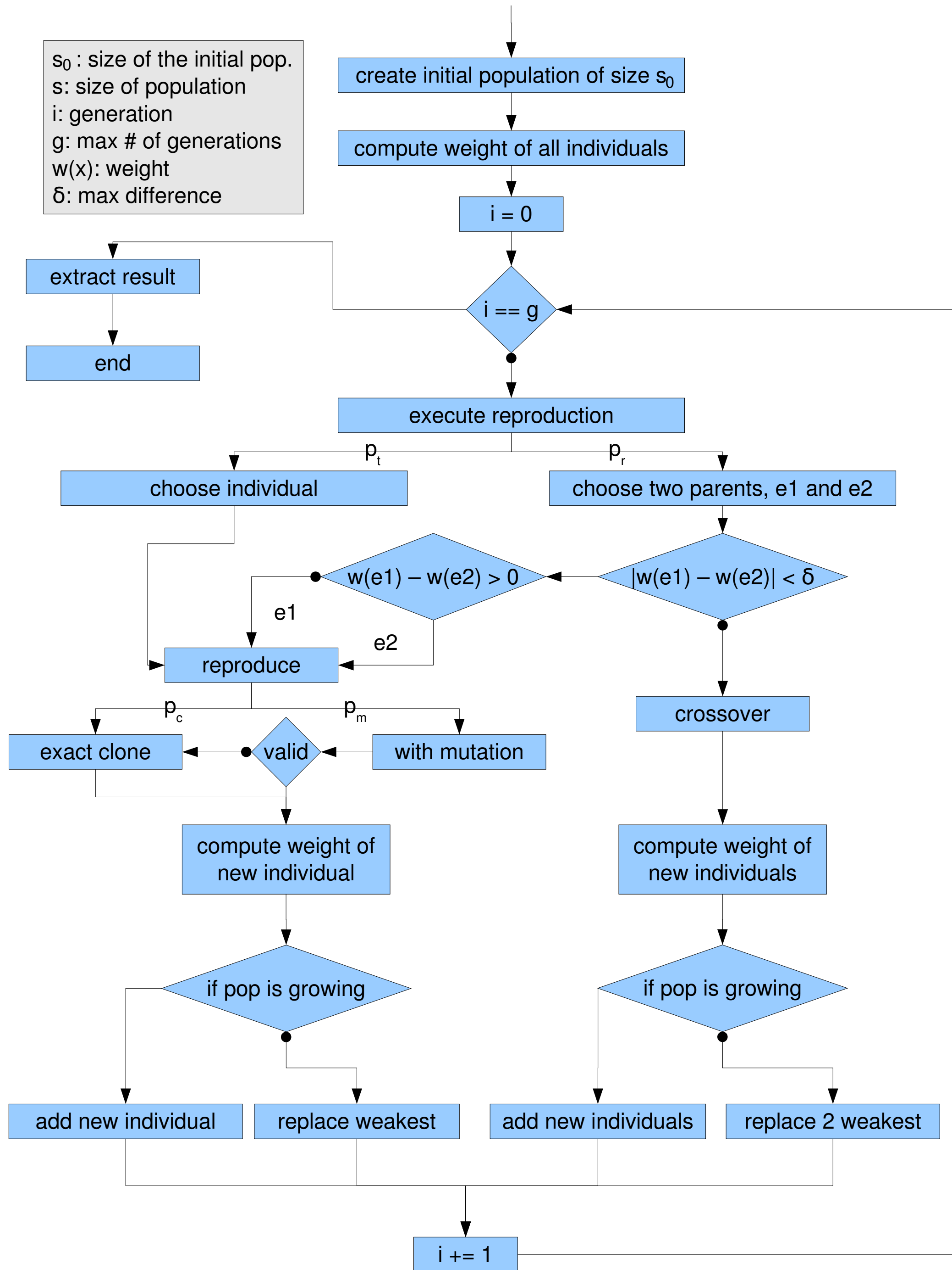

Figure 6: Flow diagram for the Genetic Shortest Path Algorithm.

if the father's and the mother's weights differ of more than the value of $delta$ **then**
  Choose the best one between them and apply the reproduction operation on it
else
  Split the second solution into two sub-paths
  Generate two new paths by combining the four sub-paths obtained by splitting the two randomly chosen solutions
end if

3.2.3 Reproduction

The reproduction procedure works as follows:

```
Randomly pick a solution from the current population
Randomly pick two edges from the solution and swap their position
if the newly created solution is a valid path from the source to the destination then
  Return this path
else
  Return an exact copy (a clone) of the randomly chosen solution
end if
```

If the population is growing, that is to say, if the current population size is smaller than the maximum population size, the solution(s) returned by the reproduction or the crossover operators are simply added to the population. Otherwise, if the population has already reached its maximum size, the new solution(s) substitute(s) the worst solution(s) in the population.

If the maximum number of generations has been reached, the algorithm stops and returns the best solution found so far. Otherwise, the number of generations is incremented by one and the algorithm's evolutionary part is executed again.

3.3 Genetic Shortest Path Algorithm Without Constraints

This algorithm is a variant of the genetic algorithm described in the previous paragraph. As a matter of fact, their structures are very similar; this version, however, allows the existence of invalid solutions in the population and does not consider the *delta* value. Concretely, the differences between the two algorithms are located in the generation of the initial population, the implementation of the crossover and the reproduction operators, as shown by the code snippets below:

3.3.1 Initial Population

Contrary to the version with constraints, the members of the individual population do not have to be valid solutions. The generation of the initial population is split into two parts: half of the population is created by starting from the source vertex while the other half of the population is created by starting from the destination vertex.

```
repeat
  Pick a random edge leaving from the source vertex
  Add this edge to the current solution
```

if the solution has reached the maximum length **then**
Stop and start over
end if
until the population has reached half of the required size
repeat
Pick a random edge leaving from the destination vertex
Add this edge to the current solution
if the solution has reached the maximum length **then**
Stop and start over
end if
until the population has reached the required size

3.3.2 Crossover

Randomly pick a solution from the current population
Randomly split the solution into two sub-paths: the edge where the path is splitted is called the *splitting point*
Search within the population for another solution – different from the previous one – containing the same *splitting point* edge
Split the second solution into two sub-paths
Generate two new paths by combining the four sub-paths obtained by splitting the two randomly chosen solutions

Differently from the implementation of the crossover operation given above, the *delta* value is ignored here.

3.3.3 Reproduction

Randomly pick a solution from the current population
Randomly pick two edges from the solution and swap their position
Return the newly created path

Notice that no verification is done on the validity of the newly created path. The fact that the population may contain invalid solutions implies a slight modification to the genetic algorithm: when the maximum number of generations is reached an additional test has to be done: the path to be returned as a result must be the shortest **valid** path within the population.

4 Conclusion

The application that has been developed implements two genetic algorithms that solve the shortest path problem between two vertices in a weighted, non-oriented graph.

Even on the small graphs[2] used to test the application, we noticed that the best solution is rarely found. However, in most cases, the genetic algorithms find a *satisfying* solution.

Both the genetic algorithms can be tuned by modifying several parameters: the quality of the solution dramatically depends on these values. Unfortunately, there exists no precise method to determine the best set of parameters: the algorithms' tuning is based on a *try and fail* strategy.

By running several tests, we pointed out a weakness of the Genetic Shortest Path Algorithm Without Constraints. Since this algorithm allows the existence of invalid solutions, the population my contain very short solutions[3]. Obviously, such solutions have a very small weight and thus their number increases in every following generation. This behaviour quickly produces a population containing only invalid paths.

This project allowed us to get acquaint with the specification and the implementation of a genetic algorithm. Our application could be improved, for example by defining a better strategy to determine the parameters' values. The fitness function for the Genetic Shortest Path Algorithm Without Constraints should be modified, so that the too short paths do not produce a degenerated population. Despite these weaknesses, we believe that this application constitutes a valid proof of concept and it shows that genetic algorithms can be used to solve the shortest path problem in a graph.

The authors have very well appreciated the project. It allowed them to learn a lot about genetic algorithms, an exciting area of computer science.

[2] 51 vertices and about 100 edges

[3] Solutions composed by a too small number of edges to build a path connecting the given vertices. This is a consequence of the crossover operation.