Introduction to Algorithms, Data Structures, and Problem Solving

DA4002 (HT11) Halmstad University

Assignment for Project 2: Positive-Weighted Shortest-Path (Dijkstra's Algorithm)

roland.philippsen@hh.se

October 13, 2011

1 Introduction

The second project of the ITADS course focuses on more independency for the creation of an application. It is a rather direct continuation of exercise 6. The starting point is a skelettal implementation of a graph, some example graph definition files, and some example code fragments. All example graphs are planar. The task is to create an application that performs optimal path planning on directed graphs. Each edge has a strictly positive cost, and an optimal path is one which minimizes the sum of edge costs.

Similarly to the first project, students work in teams of two, and they have to hand in **all source code** along with **a report** which documents their work and the results they obtain. The deadline for handing in the source code and the report is **Friday, October 21, 2011, at 18h00**.

Teams who miss the deadline will receive a penalty by lowering their grade by one. In case of exonerating circumstances, such as sickness certified by a medical doctor, a deadline extension will be granted. Participants must notify the lecturer of such circumstances as soon as possible when they arise.

2 Starting Point

The source archive for the second project contains the following files:

Vertex.java contains a bare-bones `Vertex` class, which simply contains a list of `Edge` objects that point to adjacent neighbors. Over the course of the project, fields need to be added here in order to support the planning algorithm.

Edge.java contains a bare-bones `Edge` class, which contains a reference to the `Vertex` which is at the end of the edge. The edge class will also need to be extended with fields, for instance the cost of traversing an edge.

Graph.java is a skeleton `Graph` class, which is incomplete except for the `load` method which parses adjaceny list files. The methods used by the `load` method have an empty body with comments describing what needs to be filled in. In particular, the provided `Graph` class contains no field for storing vertices. This‘ is one of the first things that needs to be added in order to turn the `Graph` class into something usable. Several possibilities exist for storing the vertices, from bare arrays to more flexible containers [4, 1, 3, 6, 2, 5].

PriorityQueueExample.java contains code which illustrates two different ways of using the `PriorityQueue` shipped with Java's Collection library. Such a priority queue [10] is necessary in order to implement Dijkstra's algorithm for planning the shortest path through a positive-weighted graph.

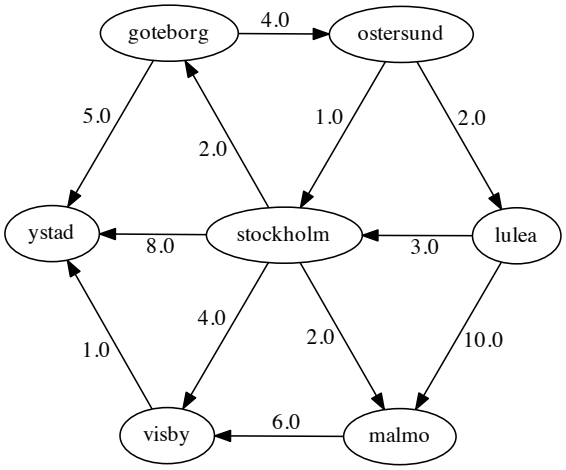

Figure 1: Graphical representation of the graph defined in the `cities.adj` file.

cities.adj is a text file in adjacency-list format containing a simple example graph (see figure 1). Each line of the file contains two strings and a number. The two strings are a source and destination city, and the number is the cost of traveling from the source to the destination. The `cities.adj` file defines the same graph which has been used during exercise 6.

zig-zag.adj contains a graph with 510 weighted edges. It is based on a grid example that is illustrated in `zig-zag.png` (see also figure 2). This graph forms a zig-zag curve that is several vertices wide. Some of the edges which lie close to the border have elevated costs, but most of them have a cost of 1 or $\sqrt{2}$.

maze.adj is also based on a grid, but it is a more complex maze and contains 85226 edges (see `maze.png` and figure 2). This graph represents a maze, with corridors that are several vertices wide. Edge costs increase in the vicinity of walls.

terrain.adj is also based on a grid example, with 317588 edges (see `terrain.png` and figure 2). This graph represents an outdoors environment with a road (low edge costs), a river (very high edge costs), some grassland (low edge cost), and some forest (moderate edge costs).

zig-zag.grid, maze.grid, and terrain.grid are meant to serve as input files for one of the bonus tasks. They are textual representations of the images in `zig-zag.adj`, `maze.adj`, and `terrain.adj`. Each line of a `.grid` file corresponds to a row of pixels in the images, but the axis has been inverted and obstacles are handled specially: white pixels are represented with a grid value of 1, obstacles are stored as the character "#", and darker tones of grey receive higher grid values.

3 Mandatory Tasks

- Extend the provided `Graph` class such that it actually creates and stores the vertices it reads from the text file. This implies extending the `Vertex` class such that it stores its name as well. Test your result by writing an application that reads `cities.adj` and prints the resulting graph on the console. Your output should look similar to the example given on the left in figure 3.

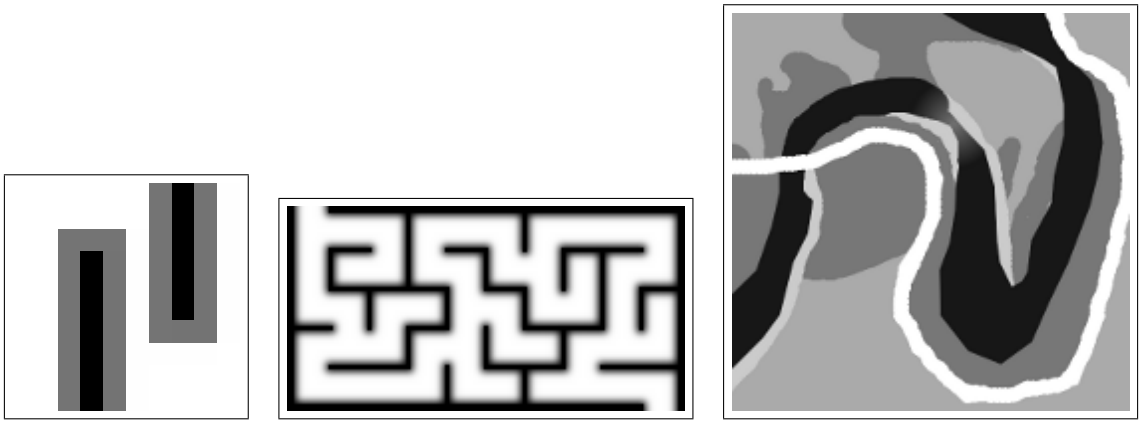

Figure 2: Greyscale representations of the three grid-based graph examples `zig-zag.adj`, `maze.adj`, and `terrain.adj`. In the images, white signifies easily traversable regions (edge costs = 1), obstacles are black (non-traversable regions: no vertices or edges exist in those areas). Darker areas correspond to areas with higher edge costs. Note that the files `zig-zag.grid`, `maze.grid`, and `terrain.grid` contain the same data, but in a textual representation.

```
neighbors of ostersund:
    lulea
    stockholm
neighbors of stockholm:
    goteborg
    malmo
    ystad
    visby
neighbors of goteborg:
    ostersund
    ystad
neighbors of visby:
    ystad
neighbors of ystad:
neighbors of lulea:
    stockholm
    malmo
neighbors of malmo:
    visby
```

```
neighbors of ostersund:
    lulea (cost: 2.0)
    stockholm (cost: 1.0)
neighbors of stockholm:
    goteborg (cost: 2.0)
    malmo (cost: 2.0)
    ystad (cost: 8.0)
    visby (cost: 4.0)
neighbors of goteborg:
    ostersund (cost: 4.0)
    ystad (cost: 5.0)
neighbors of visby:
    ystad (cost: 1.0)
neighbors of ystad:
neighbors of lulea:
    stockholm (cost: 3.0)
    malmo (cost: 10.0)
neighbors of malmo:
    visby (cost: 6.0)
```

Figure 3: Example output for printing the graph defined in `cities.adj`. On the left, before edge cost parsing. On the right, after adding edge cost support to the `Edge` and `Graph` classes.

```
ostersund [v: 0.0] --> lulea [c: 2.0]   stockholm [c: 1.0]
stockholm [v: 0.0] --> goteborg [c: 2.0]   malmo [c: 2.0]   ystad [c: 8.0]   visby [c: 4.0]
goteborg [v: 0.0] --> ostersund [c: 4.0]   ystad [c: 5.0]
visby [v: 0.0] --> ystad [c: 1.0]
ystad [v: 0.0] -->
lulea [v: 0.0] --> stockholm [c: 3.0]   malmo [c: 10.0]
malmo [v: 0.0] --> visby [c: 6.0]
```

Figure 4: Example output for printing the graph defined in `cities.adj`. Here, each vertex also has a value assigned to it. After running Dijkstra's algorithm, the values will be non-zero.

- Extend the `Edge` class and the relevant methods in `Graph` such that edge costs get correctly parsed and stored. Update the test application, which should now produce output similar to the one on the right in figure 3.
- Extend the `Vertex` class and the relevant methods in `Graph` such that you can store the cumulated path costs. Update the test application and its output (a more compact format is now appropriate) accoring to the example in figure 4.
- Read the information about Dijkstra's algorithm in chapter 14.3 of the course book [7] and on Wikipedia [9]. Then implement the algorithm in new method of the `Graph` class. Some **important remarks** need to be taken into account:
  - You will need a priority queue (read the provided `PriorityQueueExample.java` file for an example).
  - You will need to initialize the vertex values to "infinity." For floating-point numbers, a useful substitute for infinity is `Double.MAX_VALUE` (which is larger than any value we would expect to encounter in a normal run of Dijkstra's algorithm).
  - You will need to further extend the `Vertex` class so that it stores a backpointer to its predecessor in the optimal path. The Wikipedia pseudocode calls this "`previous`," the course book calls it "`prev`."
  - The pseudocode given on Wikipedia relies on a priority queue which allows to re-order an item already on the queue. This can be avoided with the technique presented in the course book: simply create a new queue entry instead of re-ordering the old one. But this means ensuring that when you later take the old one off the queue, you have to check whether it is still relevant. This is explained at the beginning of page 494 in the course book.
- Write an application which takes three arguments: the name of an `.adj` file to use as graph definition, the name of the start vertex, and the name of the goal vertex. The application then builds the graph, applies Dijkstra's algorithm, and prints out the shortest path from start to goal. Note that the vertex names in the grid-based `.adj` files are based on the grid-coordinates of the cells they are based on. Thus, you will get a path in grid coordinates. Document you application, including how to run it and the paths it computes through the zig-zag (from lower-left to upper-right) and through the maze environment (from upper-left to lower-right).

4 Bonus Tasks

- Write a method which reads a `.grid` file and creates a graph from it. In order to convert a cell's value to vertices and edges, the following rules should be applied:
  - If a cell is marked as an obstacle using a "#" character in the `.grid` file, no vertex should be created. This of course also means that there will be no edges going towards or away from that cell.
  - Non-obstacle cells should be connected to their four direct neighbors using edges that have a cost equal to the cell's value. Direct neighbors are the ones which lie to the north, east, south, or west of a cell.
  - Non-obstacle cells should also be connected to their four diagonal neighbors, but with an edge cost that is $\sqrt{2}$ times the cell's value.

  In order to preserve the grid structure in the `Graph` class, it is most appropriate to store the `Vertex` instances in a two-dimenionsional array. But it is also possible to keep a name-based lookup, with names that are generated from the cell indices. Test your implementation by loading the provided `.grid` file and comparing them to the corresponding `.adj` files.

- Write a grid path planning application which takes the name of the grid file as argument, along with the coordinates of the start and goal location. It can also be nice to produce graphical output to show the cost map and the optimal path from start to goal. A simplified "graphical" output could be based on characters output on the console, similar to the `Mansion` class from exercise5. *(Hint: saving the console output to a text file and viewing it in Firefox allows you to zoom out until the characters become so small that it looks almost like a greyscale image).*

- Extend your grid path planner with heuristic search: implement A* search based on the description on Wikipedia [8] or other resources you can find. Make sure you test that the paths computed using A* are the same as the ones computed with Dijkstra's algorithm. Compare the number of vertex expansions between A* and Dijkstra.

References

[1] java.util. ArrayList<E>. `http://download.oracle.com/javase/1.5.0/docs/api/java/util/ArrayList.html`, October 2011.

[2] java.util. HashMap<K,V>. `http://download.oracle.com/javase/1.5.0/docs/api/java/util/HashMap.html`, October 2011.

[3] java.util. HashSet<E>. `http://download.oracle.com/javase/1.5.0/docs/api/java/util/HashSet.html`, October 2011.

[4] java.util. LinkedList<E>. `http://download.oracle.com/javase/1.5.0/docs/api/java/util/LinkedList.html`, October 2011.

[5] java.util. TreeMap<K,V>. `http://download.oracle.com/javase/1.5.0/docs/api/java/util/TreeMap.html`, October 2011.

[6] java.util. TreeSet<E>. `http://download.oracle.com/javase/1.5.0/docs/api/java/util/TreeSet.html`, October 2011.

[7] Mark Allen Weiss. *Data Structures and Problem Solving using Java.* Addison Wesley, 3rd edition edition, 2005. ISBN 0-321-31255-4.

[8] Wikipedia. A* search algorithm. `http://en.wikipedia.org/wiki/A*`, October 2011.

[9] Wikipedia. Dijkstra's algorithm. `http://en.wikipedia.org/wiki/Dijkstra's_algorithm`, October 2011.

[10] Wikipedia. Priority queue. `http://en.wikipedia.org/wiki/Priority_queue`, October 2011.