Maze Generation via Spanning Trees

CS 223

November 19, 2022

1 Introduction

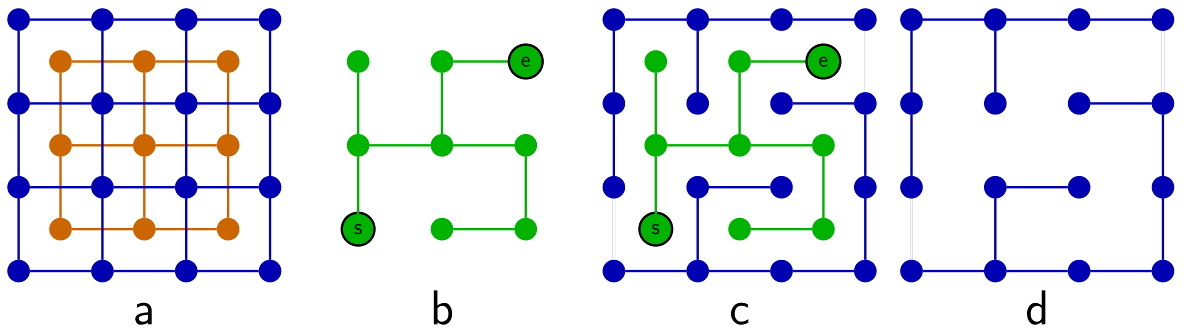

Figure 1: Figure 1: Planar graph with vertices arranged in a grid represent spanning tree and maze. (a) Nested graphs: orange representing possible corridors, blue representing possible walls; (b) Spanning Tree representing actual corridors; (c) Corridors and start/end points inform walls; (d) final maze

For this project, you will create a Java application that creates a maze as illustrated in Figure 1. This is done by considering two graphs, each arranged in a grid and nested inside one another (Figure 1a). Starting with a randomly selected vertex s on the left side the orange graph, we create a spanning tree (ST), and then choose a vertex e on the right side of the graph as an ending point. This results in a spanning tree with named start and end vertices which can be thought of as representing the corridors through the maze (Figure 1b). Given the spanning tree, we can then create a corresponding graph representing the walls of the maze (Figure 1d). Both graphs can be nested inside one another (Figure 1c).

Your program will reside in the `MazeGenerator.java` file and will contain two methods: `generateST` which creates a spanning tree of a given grid-based graph (as in Figure 1b), and `generateMaze` which generates the walls as in Figure 1d). The `MazeGenerator`'s `main` method has been written for you and takes three arguments that can be supplied on the command line or from within your IDE as described below: a width, a height and a filebase name. The `main` method then calls `generateST` to create a spanning tree on a graph of size `width x height`, and then calls `generateMaze` with the spanning tree. The spanning tree and maze are stored in files ending with `-st.txt` and `-maze.txt` respectively.

The `MazeViewer` class provides methods to create scalable vector graphics (svg) files from graph descriptions. Specifically, when given a particular *basename*, the main method of `MazeViewer` will load the two files *basename*`-maze.txt` and *basename*`-st.txt` corresponding to the maze and spanning tree respectively. It will then produce three `svg` files: *basename*`-st.svg` (with a figure similar to Figure 1b); *basename*`-detail.svg` (with a figure similar to Figure 1c); and *basename*`-maze.svg` (with a figure similar to Figure 1d). If you're using IntelliJ to do you programming, you can pass arguments to a main method by right clicking the green arrow in the source code next to the main method and selecting "Modify Run Configuration...". Under the heading "Build and Run", there will be a text box to enter "Program arguments".

For `MazeGenerator.main` modify the run configuration so the program arguments specify the width, height and output file basename. For example, use: `5 4 maze`

For `MazeViewer.main` modify the run configuration so the program arguments specify which maze files to read from input (these should probably agree with the output file you specified above). For example use: `maze`

2 Graph Representation

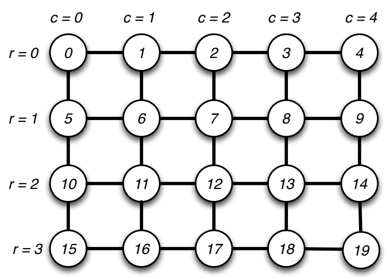

Figure 2: For a $W \times H$ maze (5×4 in this example), the vertex at row r and column c is encoded by the integer $v = r \cdot W + c$.

The `generateST` and `generateMaze` methods both return instances of the `LabeledGridGraph` and `GridGraph` interfaces respectively. Both of these interfaces can be thought of as a "Graph as Adjacency List" representation, although the latter allows arbitrary labels to be associated with vertices.

```
public interface GridGraph {
    public int getHeight();
    public int getWidth();
    public int nVertices();
    public List<Integer> adjacent(int v);
}

public interface LabeledGridGraph extends GridGraph {
    public List<Integer> getVertices(String label);
    public String getLabel(int v);
}
```

Important: the `generateST` method returns a `LabeledGridGraph` instance. This instance, should have two labeled vertices: the “start” vertex should be on the left edge of the grid, and the “end” vertex should be on the right edge of the grid. There should be exactly one start and one end vertex, and no other labels need to be supported for the graph returned by `generateST`. (This fact can simplify your implementation). Finally, note that the list return by `getVertices` should be a “view” of the vertices with this label. Specifically, this means that modifying the list returned by `getVertices` should not alter the labels of the graph.

Given that the vertices of G are laid out in $W \times H$ rectangular grid as shown in Figure 2. The vertex at row `r` and column `c` is encoded by the integer `v` corresponding to its row-major index:

```
v = r*W + c;
```

Thus, given `v` we can recover the grid position `(r,c)` as follows:

```
r = v / W;
c = v % W;
```

Thus, for the spanning tree in Figure 1b, the start vertex is 6 and the end vertex is 2. Further, for the spanning tree graph: `adjacent(6)` will return a list containing 3, while `adjacent(4)` will return a list containing: $1, 3, 5$. For the maze graph (Figure 1d), adjacencies represent walls: so `adjacent(0)` returns the list containing: $1, 4$, and `adjacent(13)` returns $9, 12, 13$. *This row major encoding must be used for your graph; otherwise vertex locations not be interpreted correctly by the test suite or by the MazeViewer.*

3 Creating the Spanning Tree

You can use any approach you’d like to create the spanning tree. Depth first search, Prim’s method or Kruskal’s algorithm are obvious choices.

4 What to submit

Submit your `MazeGenerator.java` file to autolab. You will likely want to implement supporting classes (such as an implementation of the `LabeledGridGraph` interface). These should be included as non-public classes in the `MazeGenerator.java` file.