Cole Erickson, 18nde1, nde1@williams.edu
Anna Neufeld, 18acn2, acn2@williams.edu
CS 136 Final Project

Implementing Randomized Prim's Algorithm for Maze Generation

Introduction

We sought to algorithmically create two-dimensional mazes with solutions that are not immediately apparent to humans. The algorithm we implemented to achieve this goal was the Randomized Prim's Algorithm (RPA). RPA generates a spanning tree for the graph of the grid of the maze. This spanning tree can be viewed as a minimum spanning tree for a grid graph where the edges are assigned random weights. [3]

Any two points in a maze produced by a spanning tree maze-generation algorithm can be reached by a unique shortest path. A maze produced by such an algorithm has no cycles. These qualities produce a typical maze.

RPA works by first creating a grid with every node "closed"—that is, there are no open spaces in the maze. A node is selected to be the start node and "opened", adding it to the maze. The four edges extending from the node are added to a set referred to as the "frontier."

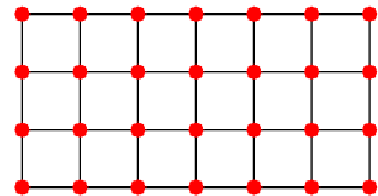

Figure 1. A grid graph.

Once these initial steps are completed, an edge is selected at random and removed from the frontier. The frontier edge and the nodes it connects to are then examined to determine how the maze should be expanded:

- If one of the nodes that it connects to is closed, both the frontier edge and the adjacent closed node are opened, adding a small path to the maze. The edges surrounding the newly opened node are added to the frontier list if those edges connect to closed nodes.
- If both of the nodes connected to the frontier edge are already opened, the frontier edge stays closed.
- Because of the way the frontier is maintained, it is impossible that a frontier edge will be attached to two closed nodes.

This process ends with the frontier empty, all nodes marked open, and only some of the edges marked open, as seen in Figure 2. The underlying graph at the end is a spanning tree for the corresponding grid graph. (A grid graph is depicted in Figure 1.)

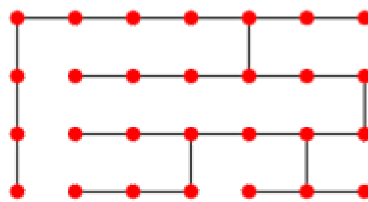

Figure 2. One potential result of RPA.

Methodology

To explore the problem of maze generation with computer science, we implemented RPA in the C++ programming language. C++ was chosen due to our familiarity with existing graphics tools in the language. The structure of our implementation is mostly defined by two classes: an application class responsible for providing the graphical view of the maze (MazeApp.cpp) and a maze class containing both the implementation of RPA and the underlying data representation of the maze (Maze.cpp). Existing code from Morgan McGuire was also included in our project for producing the graphics (App.cpp). [6]

One important choice in structuring the implementation was choosing how to represent the maze in code. Instead of opting for classes representing nodes and edges, we used a flattened two-dimensional array of Booleans to represent the graph. This allows for a simple conversion from the underlying model of the maze to the actual graphical depiction of the maze, since each element of the array corresponds to a pixel in the resultant image of the maze. The value of the Boolean determines whether that space in the maze is open or closed, and thus whether it appears black or white in the image. Figure 3 demonstrates how the nodes and edges used by RPA correspond to spaces in the array used by our implementation.

Closed	Closed	Closed	Closed	Closed	Closed	Closed
Closed	Node	Edge?	Node	Edge?	Node	Closed
Closed	Edge?	Closed	Edge?	Closed	Edge?	Closed
Closed	Node	Edge?	Node	Edge?	Node	Closed
Closed	Edge?	Closed	Edge?	Closed	Edge?	Closed
Closed	Node	Edge?	Node	Edge?	Node	Closed
Closed	Closed	Closed	Closed	Closed	Closed	Closed

Figure 3. A demonstration of how the underlying array maps to a graph. The border spaces and the spaces between edges are invariantly closed. At the end of RPA's execution, all of the nodes will be opened, but only some of the edges will.

We opted to implement the frontier set using a dynamic array of coordinate pairs. Each pair in the frontier array corresponds to the locations of one of the frontier edges in the grid.

The rest of the algorithm follows the definition from the introduction quite closely, with special considerations for the single-pixel borders of the maze and the spaces in the maze that are guaranteed to be closed.

Although the algorithm does not specify a particular endpoint for the maze, to achieve our goal of creating a solvable maze, we needed to specify an endpoint. To make the maze more difficult, we wanted a long path to the endpoint. We chose to select our endpoint by picking the last point added to the maze and marking the space red.

Results

Images produced by our implementation are included below. Note that the solutions to mazes generated by this algorithm tend to follow a slightly meandering but mostly linear path to the destination point. This is a result of the way the algorithm generates the maze, with the maze expanding essentially radially from the starting point. [3] Other algorithms avoid this kind of bias, such as Wilson's Algorithm. [7][9] Thus, other algorithms might be better for generating difficult mazes than the one implemented in this report.

It is worth noting that another algorithm which avoids this bias also has the name of "randomized Prim's algorithm" and that the algorithm described in this document is sometimes referred to as "random traversal." [1][2]

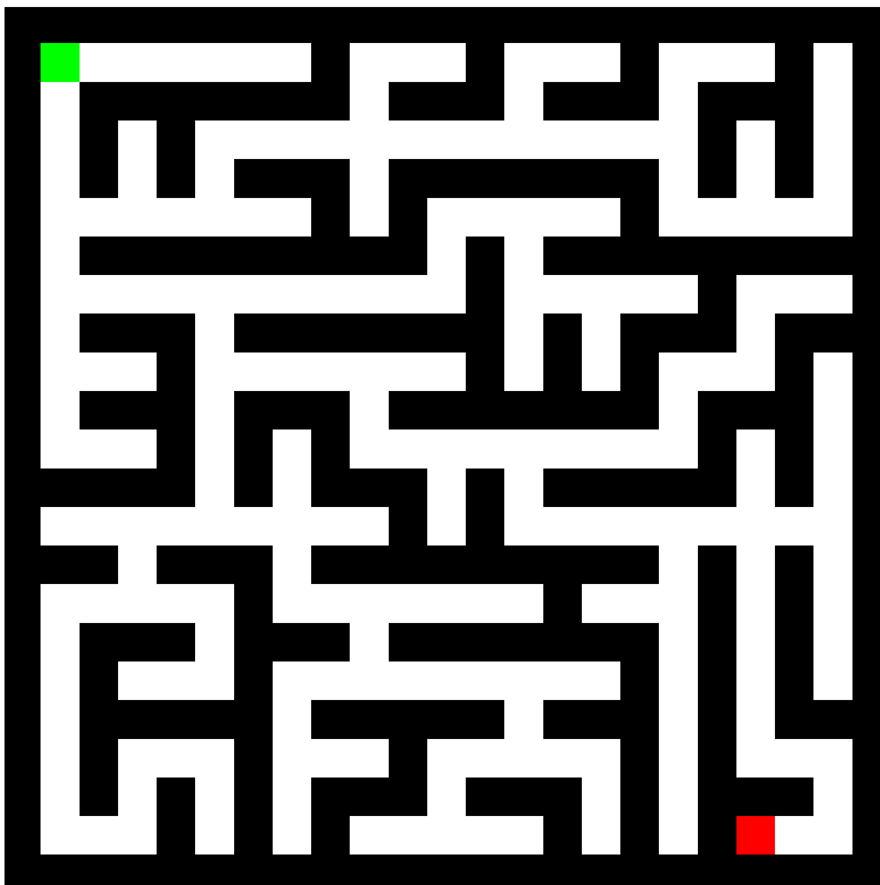

Figure 4. A typical maze generated by our implementation of RPA. The green cell is the starting point and the red cell is the exit.

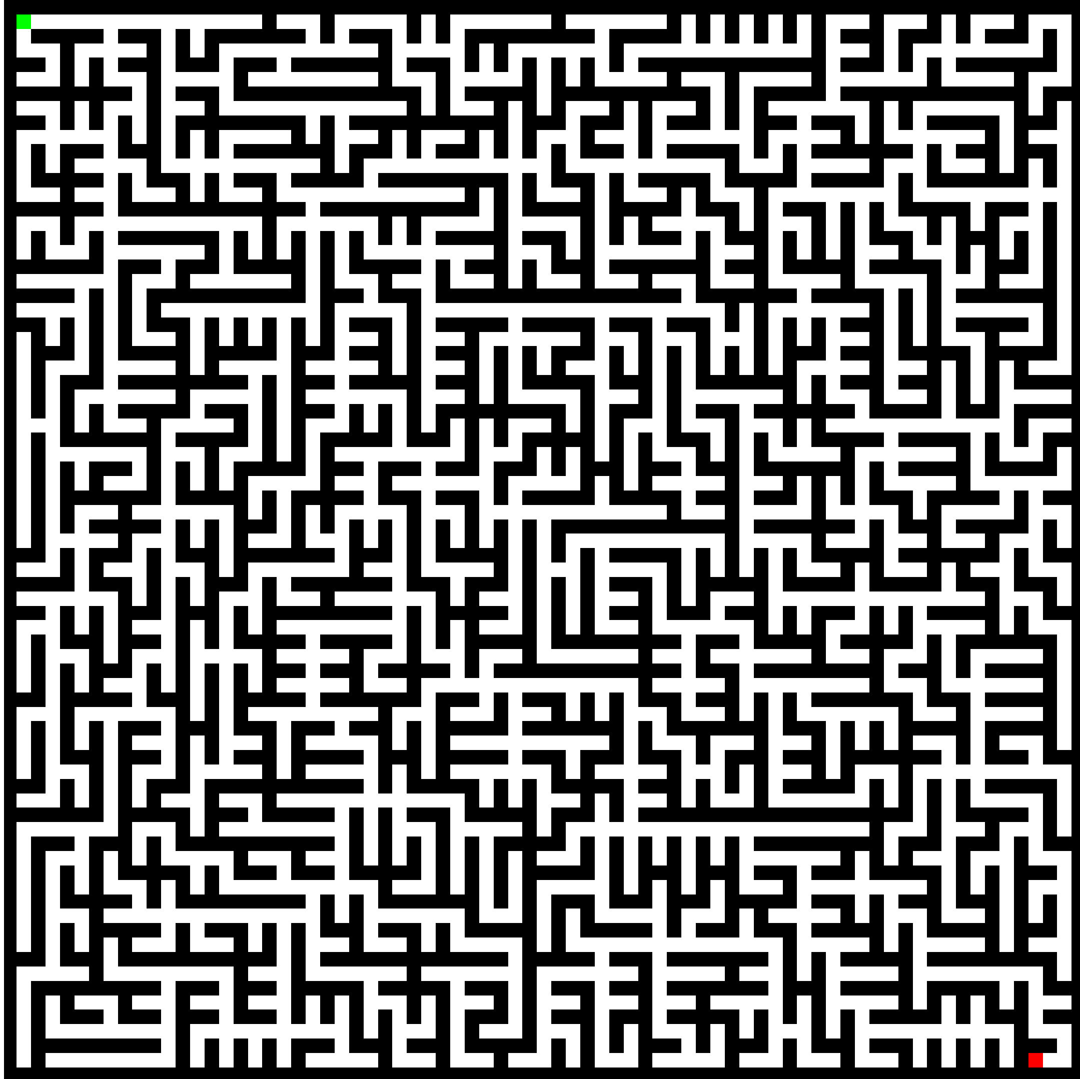

Figure 5. A larger maze.

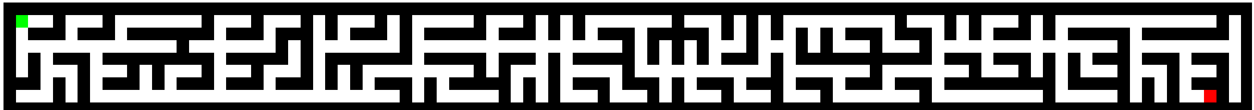

Figure 6. Our implementation can also generate non-square mazes.

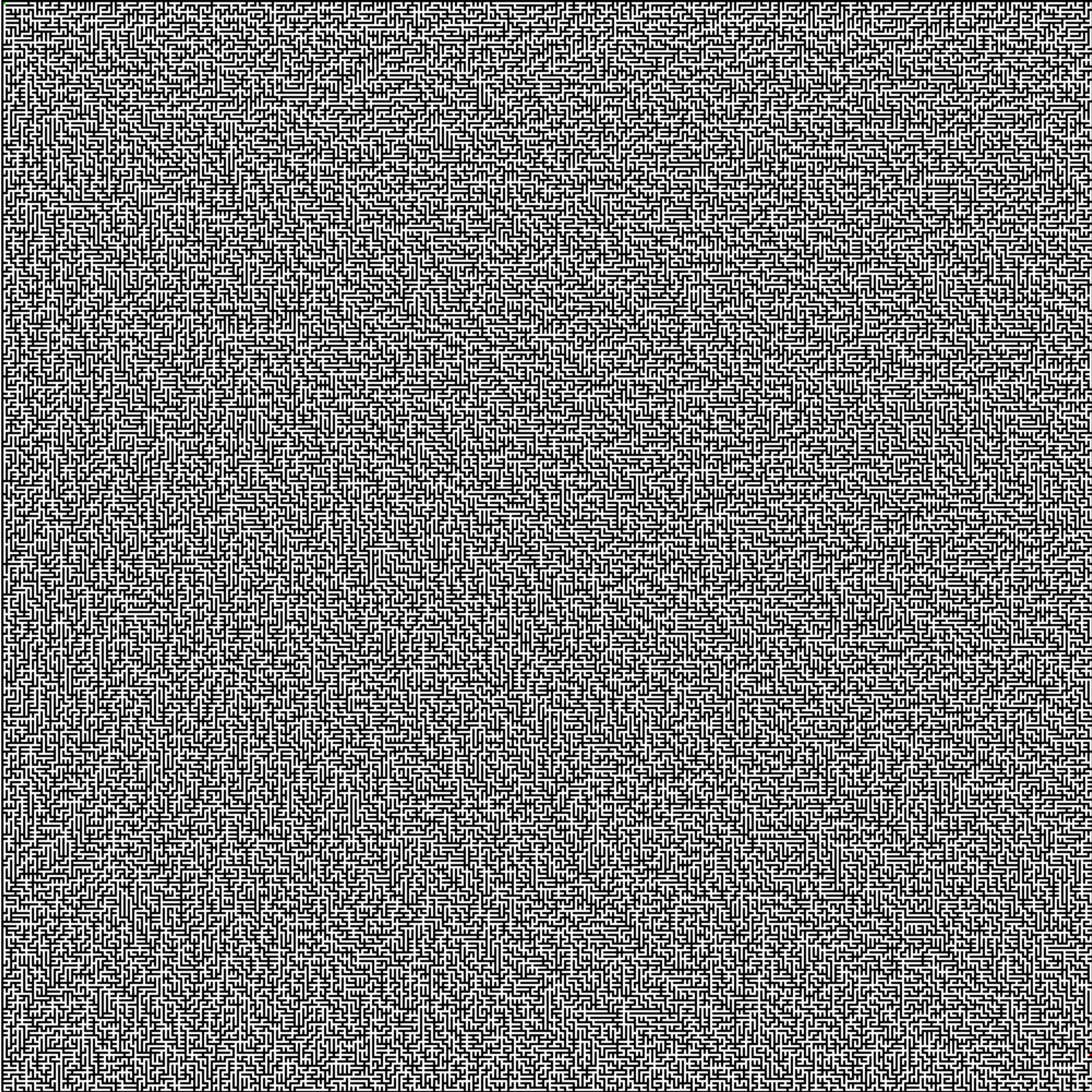

Figure 7. A very large maze.

References

1. Bostock, M. Prim's Algorithm. Retrieved December 2, 2014, from Williams College: http://bl.ocks.org/mbostock/11159599.

2. Bostock, M. Random Traversal. Retrieved December 2, 2014, from Williams College: http://bl.ocks.org/mbostock/70a28267db0354261476.

3. Buck, J. Maze Generation: Prim's Algorithm. Retrieved December 2, 2014, from Williams College: http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm.

4. Buck, J. Prim's algorithm (slide 89). Retrieved December 2, 2014, from Williams College: http://www.jamisbuck.org/presentations/rubyconf2011/index.html#prims.

5. Maze generation algorithm, 2014. Retrieved December 2, 2014, from Williams College: http://en.wikipedia.org/wiki/Maze_generation_algorithm.

6. McGuire, M. Data Structures & Advanced Programming. Retrieved December 2, 2014, from Williams College: http://cs.williams.edu/~morgan/cs136/schedule.html.

7. Spanning tree, 2014. Retrieved December 2, 2014, from Williams College: https://en.wikipedia.org/wiki/Spanning_tree.

8. Thomas, S. Maze Generation and Solution using Randomized Prim's Algorithm. Retrieved December 2, 2014, from Williams College: http://terpconnect.umd.edu/~sthomas2/prims-algorithm.html.

9. Wilson, D. Generating Random Spanning Trees More Quickly than the Cover Time. Retrieved December 2, 2014, from Williams College: https://www.cs.cmu.edu/~15859n/RelatedWork/RandomTrees-Wilson.pdf.

Appendix

Figure 8. A certain Williams College professor's surname blends into the center of this maze. However, the graph represented in this maze is, as a result, neither connected nor acyclic.