Maze Generation Algorithms

Nathaniel Graham and Kelly Olivier

Focus of project:

Analyzing various algorithms to generate perfect and complete mazes, the speed at which these mazes are generated, and the style of maze that the algorithms generate.

We focused on three algorithms:

- Recursive Backtracking Algorithm
- Kruskal's Algorithm
- Prim's Algorithm

The first thing to notice is that these three algorithms are generally used to compute a minimum spanning tree. The properties of a minimum spanning tree are such that

1. Every node has a path to every other node
2. There are no cycles in the tree

These are the same properties of a "perfect maze," or a maze which has no divisions (no part of the maze is completely separated from another) and which has one and only one solution between two arbitrary locations in the maze. As such, the algorithms for a minimum spanning tree can be applied to the generation of a randomized maze.

Summary of Algorithms:

- **Recursive Backtracking Algorithm**
    - The recursive backtracking algorithm is effectively a depth-first search on a grid of unconnected grid nodes. Starting from the first node (which, for simplicity, we always assigned to location (1, 1)), the search will mark itself as visited, pick a neighbor, connect to it if it isn't already visited, then do this process again for that node. This results in a perfect maze that has a bias towards long weaving pathways with few dead ends.
- **Kruskal's Algorithm**
    - Kruskal's Algorithm treats every group of nodes as belonging to a set. At the beginning of the algorithm, every individual node belongs to its own set. At each step in the algorithm, an edge between two nodes is selected, and if the two sets are disjoint, meaning that they have no intersection, then they are merged on that edge, creating one larger set. The result of this is, after many iterations, there is only one final set, which represents every node, connected together in a spanning tree with no cycles.
- **Prim's Algorithm**
    - Prim's Algorithm uses the idea of a "frontier node" or any node which is a candidate for the next connection. (1, 1) is set as the first and only existing

frontier node, and therefore by “random selection” it is selected as the chosen frontier. It’s marked as visited, then all four of its neighbors are considered. All unvisited neighbors are marked as frontier nodes, then the node connects itself to one random visited neighbor. This algorithm continues until there are no more frontier nodes.

Output:

All png files generated by the generation algorithms are saved into the mazes/ folder. The name of the algorithm used to produce the png, as well as the size and duration of the algorithm are part of the file name. For example, a 25 x 25 pixel Kruskal’s algorithm file which took 0.001234 seconds to execute will be saved to *Kruskal-25x25-0.001234s.png.*

Analysis & Test Cases

For the analysis of the speed and effectiveness of the maze generation algorithms, we used mazes of size 25 by 25, as they are large enough to not be trivial, yet small enough that we can run them quickly and easily and without the need to store large png files. We ran all of our algorithms (written in python) on Nathaniel’s Ubuntu computer, the stats of which can be seen in slide 2 of our powerpoint.

The proof of concept that the mazes were perfect and complete, as well as analysis of the solution patterns was completed using a depth first search (Recursive Backtracking algorithm) and A* search. The RB algorithm chose a random direction at each junction, and therefore each time it was run on the same maze would produce a different “thought process” but which always resulted in the same final solution. The A* search was conducted with a Manhattan Distance added to the distance traveled as the heuristic and the open node with the smallest sum is chosen. If multiple open nodes have the same sum, then the first one encountered is chosen. As such, the A* search will have exactly the same “thought process” every single time as well as the same solution.

See attached excel file: Maze-speed-analysis.xlsx for the table and calculations.

Challenges

We had few challenges with the actual implementation of the maze creation and solution algorithms. We ran into issues mostly with finding the most efficient data structures to use for each of the generations (we weren’t focusing on the solutions as more than a check of functionality, so not so much on them). For Kruskal’s Nathaniel ended up using a disjoint set to implement it efficiently, whereas I used a set to contain the frontier nodes for Prim’s. We also spent some extra time to add colors into the mazes for generation, so that they would be more understandable in the animated output. We had trouble getting the leading color that shows the current path to persist long enough to be truly visible, and we ended up extending the length of the colored “head” of the maze generation. This was especially a problem with recursive backtracker as it moves very fast on a single path for extended periods of time.

Future Expansion

This project has a lot of possible artistic applications, like giving Kruskal's a pre-created seed with some connections already made to create desired patterns, or writing, in the maze. Those patterns would go through the part of the maze given by the seed since Kruskal's essentially creates a minimal spanning tree. Furthermore, we would want to test how different start and end points would compare, since we used a set start of the upper left corner, and an end of the bottom right. This would be especially interesting on Prim's since the solution of Prim's mazes seem to always have a nearly diagonal solution. They don't stray far off that diagonal, and changing the starting and ending points to be maybe the upper left and bottom left corners could create interesting changes in the mazes' layouts. The other natural expansions on a maze creation focused project would be to add more creation or solution algorithms and look for the best matches between creation and solution algorithms with a focus on speed of solution. We could also look at further solution or creation algorithms to further expand our data set. Finally, we also could have expanded to creating non-square mazes, since our current program takes a single size input and makes a square maze with sides the length of the closest odd number by adding one to any even number that is inputted.