Maze Generation using Graph Traversal

Nathan Hunt, nsh3vq@virginia.edu, 4/30/2014

Project Description

I created a program that can generate mazes using Depth-First Search. Each maze is represented as a grid of cells, each of which has four walls. Through the course of the algorithm, each cell is visited, removing walls until a maze is created which has a path to every possible cell. The starting and ending locations are randomly chosen cells on the left and right edges, respectively.

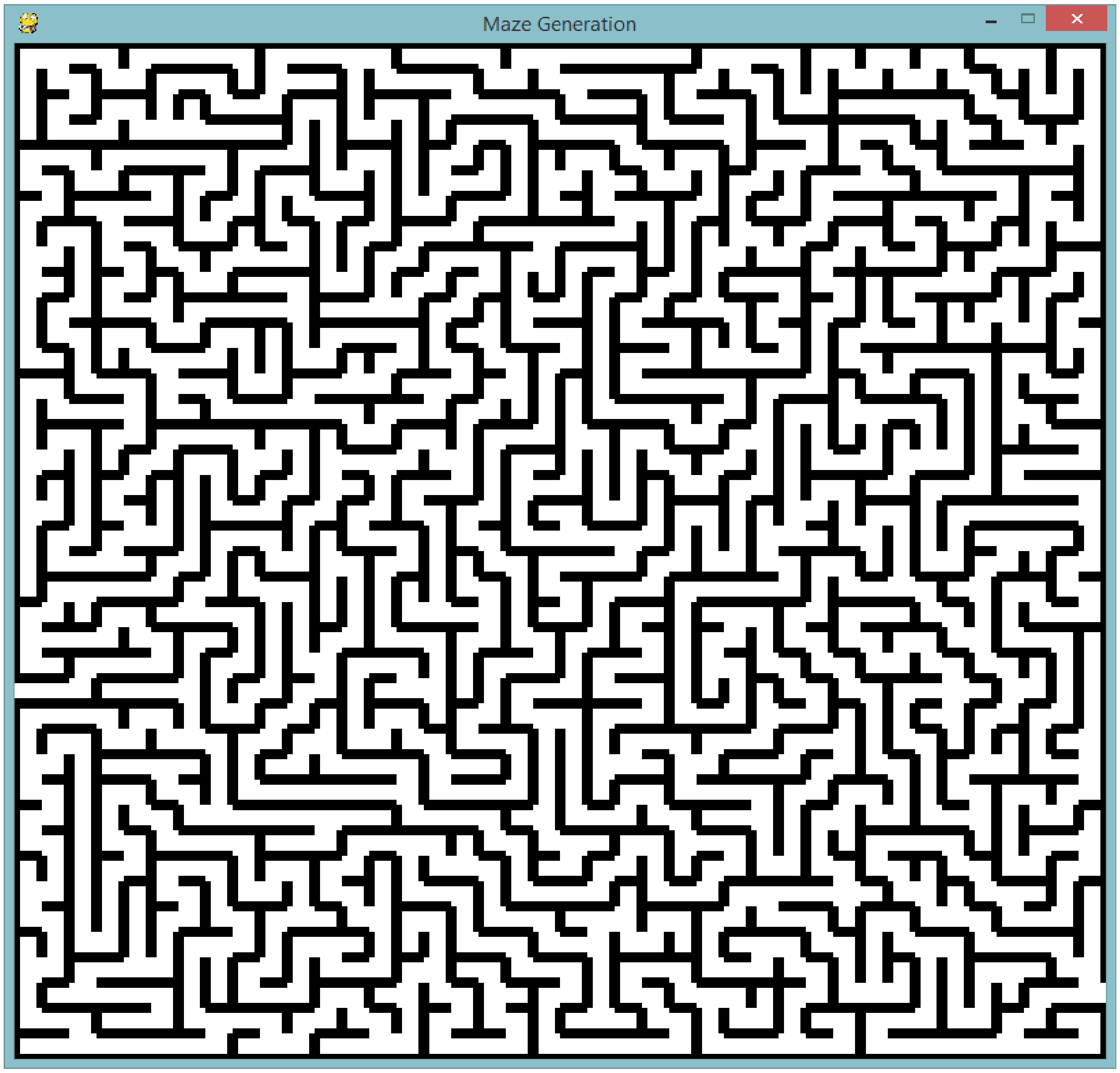


Illustration 1: A 40x40 cell maze

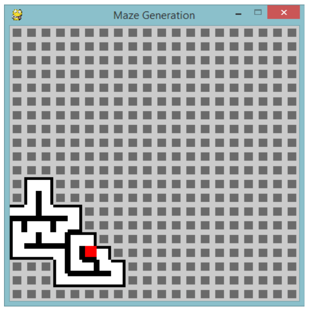
Approach & Solution

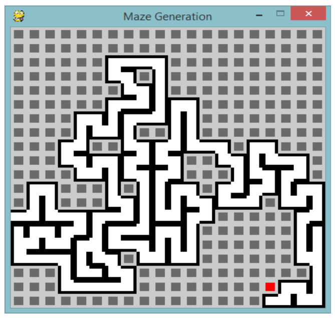
The maze can be thought of as a graph, with the cells as nodes and edges being drawn between all adjacent cells that do not have a wall between them. Because of this, a graph traversal algorithm such as Depth-First Search or Breadth-First Search is an appropriate way to generate a maze. Depth-First was chosen for generation, because it tends to create long, winding paths, whereas Breadth-First Search leads to trivially solvable mazes.

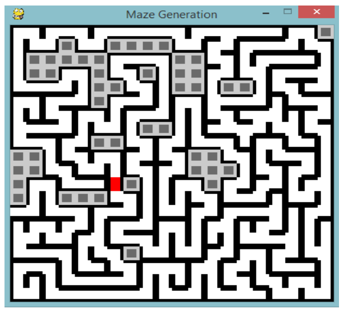
Each maze is represented in the program as a collection (called a 'grid') of cells, each of which has four walls. At first, each cell has all four of its walls. The generation algorithm uses a modified, randomized version of Depth-First Search implemented with backtracking. A description of the algorithm follows; the full source is available in the function *dfs()*.

The starting cell is chosen first, a random cell on the west edge of the grid. It is marked as discovered, and the west wall of the cell is removed. Then, the cell is pushed onto a stack twice. The main loop of the algorithm operates as long as this stack is not empty. It pops off the top two cells of the stack, the first being the new cell and the second being the previous cell, from which the new one branches. If the new cell is not marked as discovered, it is marked as such, and then the wall between it and the previous cell is removed. Afterwards, all of the cell's neighbors are fetched and placed onto the stack in random order, with the current cell placed on the stack in between each neighbor.

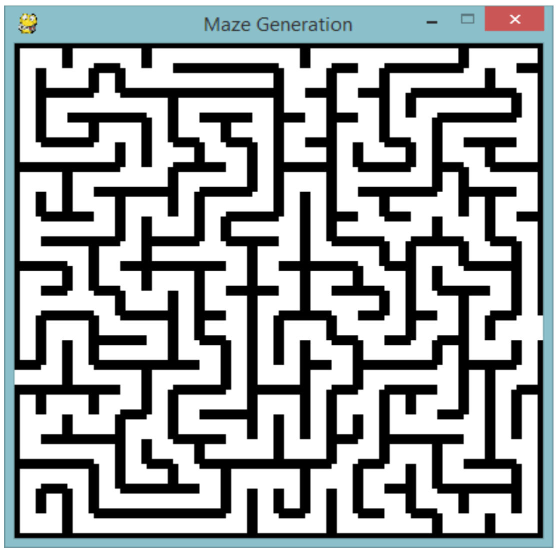
This double pushing is necessary to maintain a record of which cell a new path should use as its starting point. New paths are started whenever the current path hits a 'dead end' where the current cell has no unvisited neighbors. The algorithm pops cells off the stack until it finds an unvisited one, and the cell before it is always a previously-visited neighbor of the cell, so it can branch. If this was not done, then each new path generation would be a separate compartment of the maze, and the maze would most likely not be solvable because the compartments around the entrance and around the exit would not connect.







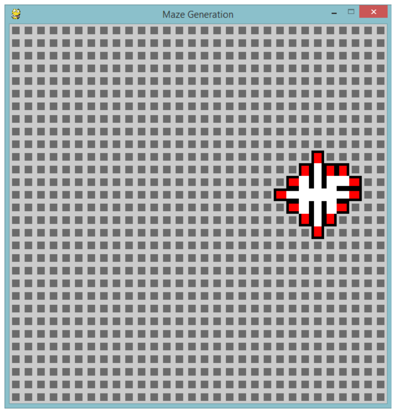
The red dot illustrates the current cell in the algorithm's progression

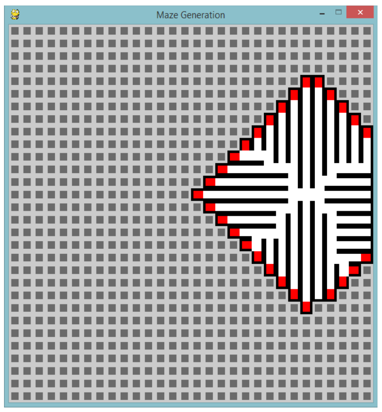


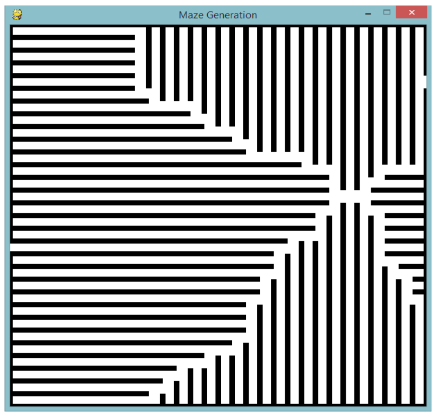
Surprises & Obstacles

Graphics were a major challenge on this project, as I have not worked extensively with graphics software before. One time sink was simply learning how to jump through the hoops necessary to make what I had represented in code appear on screen.

As a fun aside, I implemented a “maze generation” version of Breadth-First Search as well. The resulting maze is laughably easy, but the stark contrast with DFS is very noticeable. The difference in the algorithms is very slight, the biggest difference being the use of a queue instead of a stack. The difference in results, however, demonstrates very well the parallel nature of BFS. The algorithm is therefore able to complete in a far faster time than DFS, because it can investigate several paths at once. Of course, because the branches never get farther from each other than one block ahead, they cannot create the twists and turns characteristic of a maze.







Lessons Learned

The Depth-First approach is useful to generate mazes, but seems generally inefficient for finding specific cells: if I was to write an algorithm to solve mazes rather than generate them, Breadth-First search would be a much better choice. One major drawback the approach has is that it must go all the way down a path until finding a dead end until it can back out and try something new. Breadth-First search, on the other hand, parallelizes: it can search many paths simultaneously, leading to the solution being found much more quickly.

In short, the project demonstrates the different usefulnesses of DFS and BFS: DFS creates excellent mazes, but cannot solve them well, while BFS solves mazes well, but cannot create them well.

Future Development

The ideas here can easily be generalized. While 3-D mazes were not implemented here due to limitations of the graphics software, the change to the underlying code representing the mazes would be simple: rather than four walls, each cell needs six, along with the ability to detect neighbors in the new dimension. Higher dimensions can be added in a similar fashion. The code used to run the DFS is agnostic to the number of walls of each cell, and would require no change at all. This applies to the BFS algorithm as well. This could also be used to make mazes using shapes like hexagons instead of squares.

Other graph traversal algorithms can be used to make mazes as well: Prim's and Kruskal's Minimum

Spanning Tree algorithms are two examples that would generate challenging mazes.

Finally, a solver can be implemented using Breadth-First Search. As stated before, the parallel nature of it makes it ideal; while one path may lead to a dead end, other paths are still being searched so that the search does not get bogged down in a fruitless endeavor like DFS would.