Algorithmic Maze Solving

Kalab Depew  
 *Southern Illinois University Edwardsville*  
Department of Computer Science  
 Edwardsville, IL  
 kadepew@siue.edu  
  
Gwyneth Cross  
*Southern Illinois University Edwardsville*  
Department of Computer Science  
Edwardsville, IL  
gwcross@siue.edu

*Abstract*—We constructed a maze generation algorithm and used Dijkstra’s and Path Counting algorithms to solve the randomly generated mazes. We also used standard depth-first and breadth-first searches as a benchmark. We discuss the benefits and drawbacks of each algorithmic approach using three metrics: execution time, nodes explored, and the length of path found. The results show that Dijkstra’s performs the best when finding the shortest path and has acceptable time complexity. Path Counting finds approximately the shortest path, but has very high time complexity.

# Big Problem

Pathfinding algorithms are important in various real-world applications, from robotics to navigational systems. This project intends to compare three different algorithms when applying them to our own randomly generated mazes. ASCII mazes are a useful simulacrum of real-world pathfinding scenarios because finding the best path requires applying the same algorithmic strategies. Additionally, we are able to manually determine the size and shape of each maze, making it easier to experimentally determine the algorithms’ performance. We will test two algorithms presented in two sub-areas of CS 456: Dijkstra’s (Greedy) and Path Counting (Dynamic programming). Our goal is to find which algorithm is the most efficient at finding a path through each maze using execution time, nodes explored, and length of path.

# Dataset

To create an easily modifiable set of randomly generated mazes, we coded our own maze generator. A user specifies as input to the function N rows and M columns. In our experiments, we generated mazes from sizes 10x10 to 100x100. However, the algorithm can create any size of NxM maze. Our algorithm works by creating a maze that is entirely filled with the unicode character Full Block. Then, it picks a random space on the top row to be the starting point and carves a path in a random direction from that point. If the maze generator runs into a dead end, it picks a random other space that it carved and attempts to continue randomly carving paths from there. The possible directions it can carve are limited to places that 1) have not already been carved, and 2) walls that do not already have carved spaces next to them. The second condition is necessary so that the maze is not filled with paths that are multiple blocks wide. The maze generator keeps track of any dead ends that it runs into. The finish is selected from 3 randomly chosen dead ends, prioritizing the one furthest away from the start.

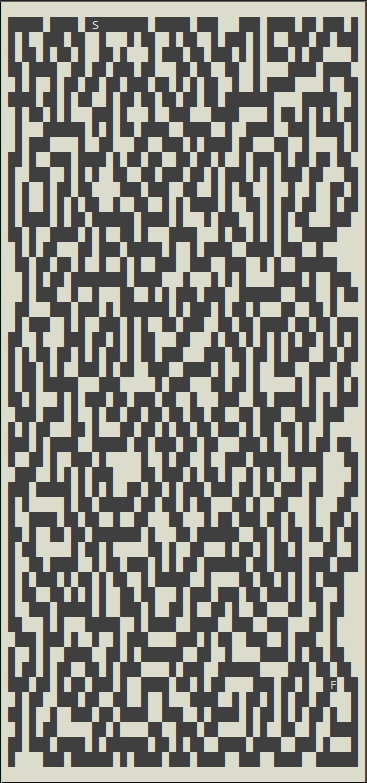
1. Example Maze Images

In the output printed to the terminal, the start is marked with an “S” and the finish is marked with an “F”. The black tiles are paths and the white tiles are walls.

Figure 1: 10x10 Maze



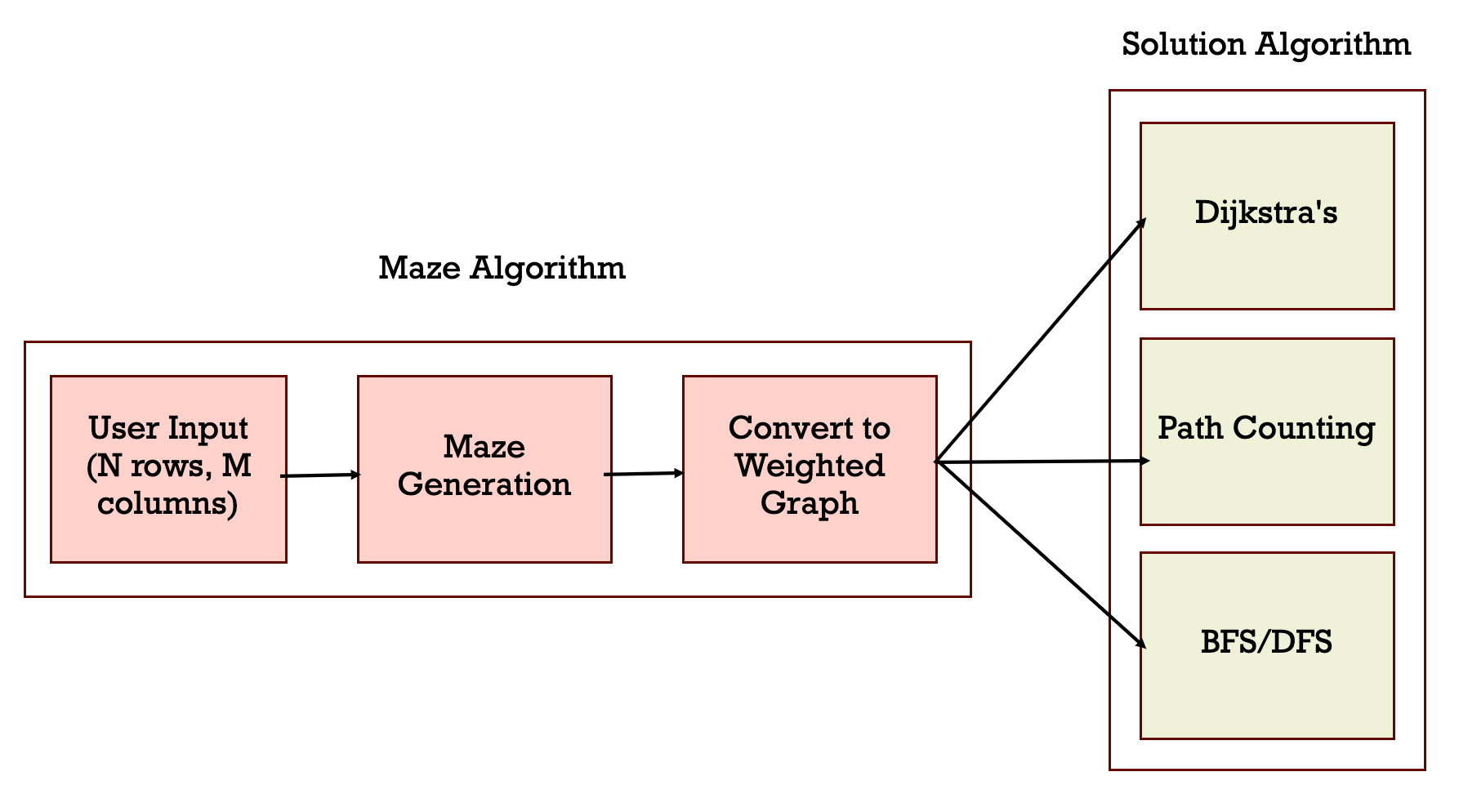
Figure 2: 50x50 Maze



# Methodology

We implemented maze-solving functions using Dijkstra’s, Path Counting, Breadth-first search, and Depth-first search. Additionally, we implemented code to convert each maze to a weighted graph. Each empty space of the maze is a node, while each path has a weight equal to its length in characters. We compared each algorithm using three different metrics: 1) length of path found through the maze, 2) nodes visited by the algorithm, and 3) number of seconds to reach a solution. For our specific maze code, see the dataset section. This code ensures that there is always at least one path from the entrance of the maze to the exit. Metric 1 demonstrates whether the algorithm found the shortest path to the solution. Metric 2 measures the “luckiness” of the algorithm: was it able to find a solution without searching all possible paths? Finally, Metric 3 measures time efficiency.

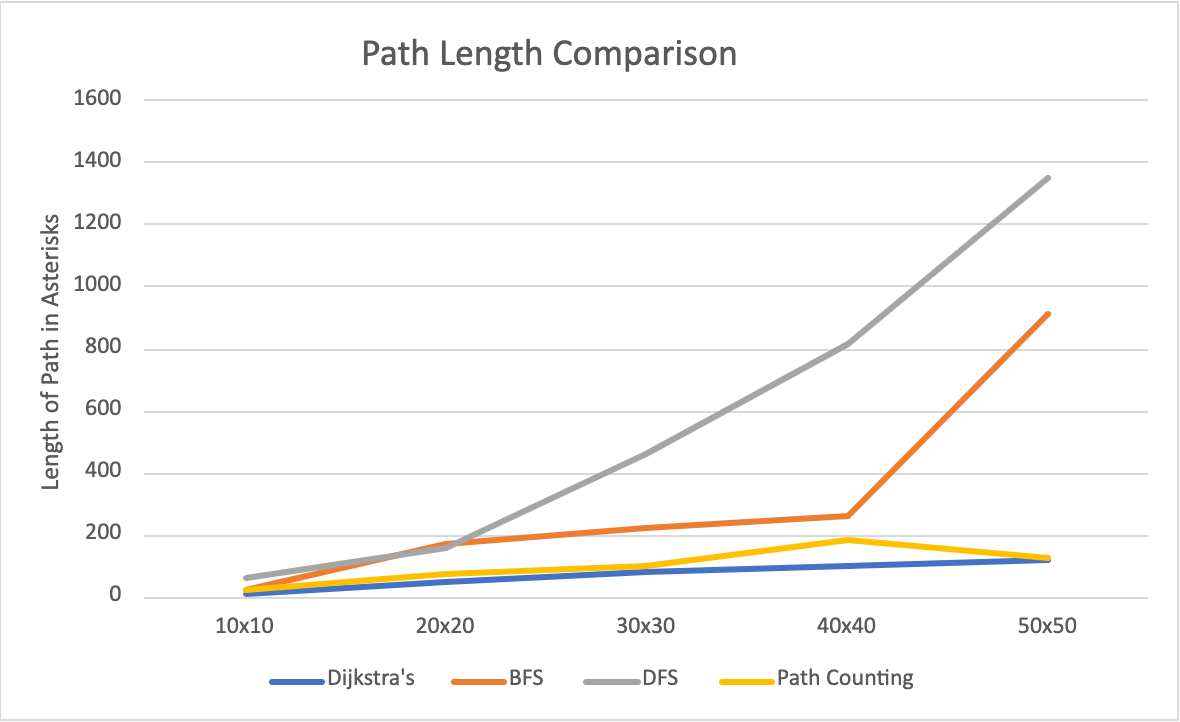
1. Block Diagram



# Results and discussion

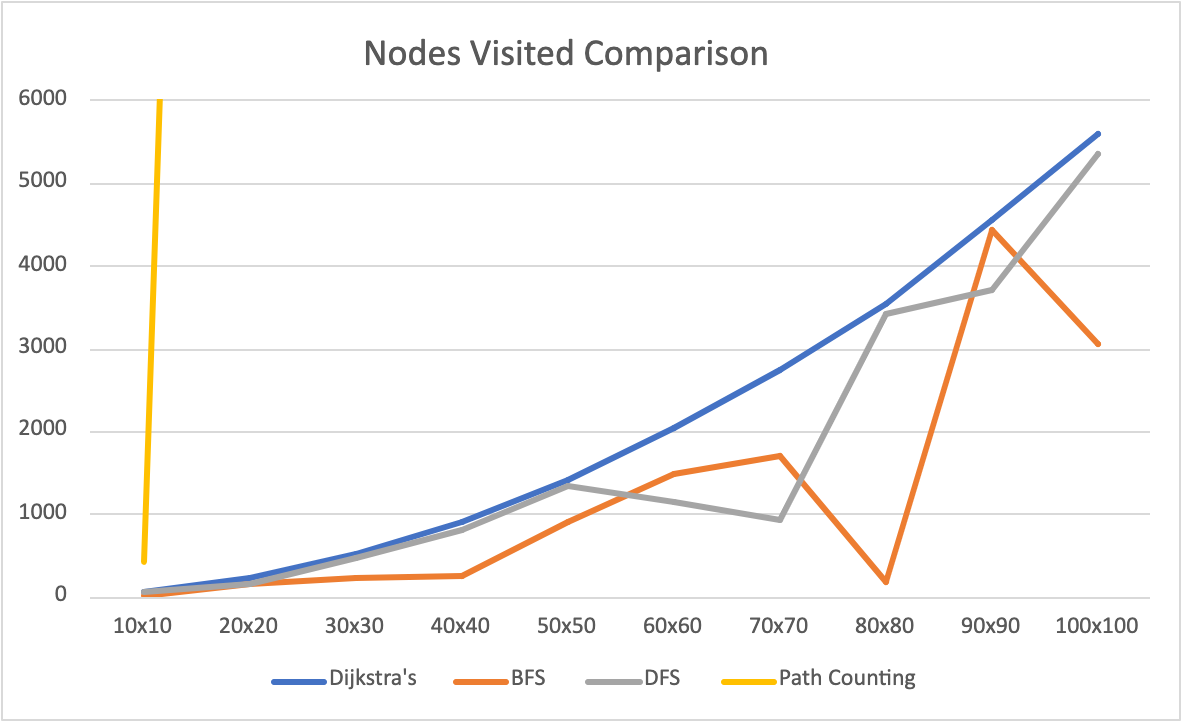
1. Experiment 1: Length of Path through the Maze

For breadth first search and depth first search, the number of nodes visited is equal to the length of the path. This is because these search techniques do not output a specific path, but rather are completed when they find the end node. Dijkstra’s and Path Counting, however, can find shorter paths; Dijkstra’s is guaranteed to find the shortest path.



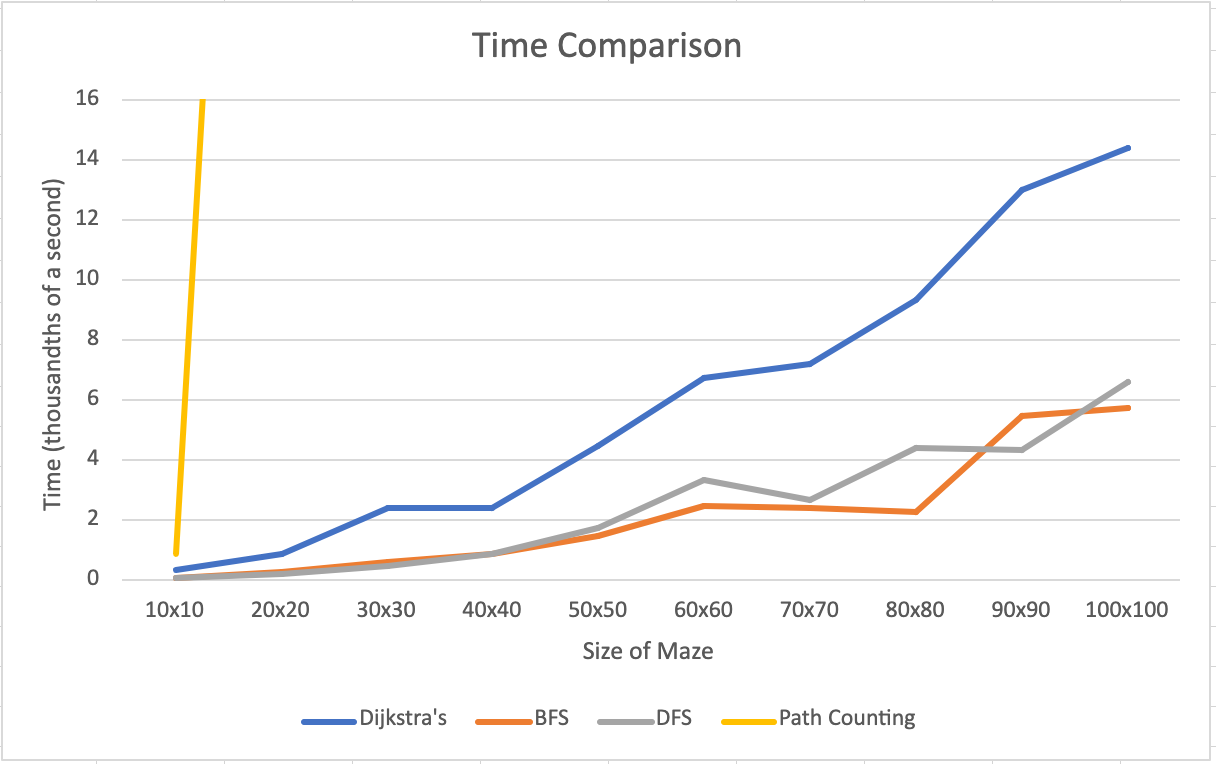
1. Experiment 2: Nodes Visited

Experiment 2 is similar to Experiment 1. However, both Dijkstra’s and Path Counting need to visit many nodes before outputting their results, while BFS and DFS can get “lucky” and find the solution without visiting every node. Path Counting in particular visits an exponential number of nodes, leading to a nearly vertical nodes visited curve on the scale of the graph.



1. Experiment 3: Number of Seconds to reach a Solution

We used the high\_resolution\_timer class from chrono to time the execution of each function. Because BFS and DFS can get “lucky” and find the path to the end in a shorter amount of time, they perform the best for this metric. Path counting performs the worst by far as it is a recursive algorithm and the number of paths it must visit for each maze grows exponentially as the size of the maze increases, along with the time the algorithm takes to run.



1. Discussion

Dijkstra’s algorithm is the most consistent with respect to all three metrics, and performs the best with respect to path length. Breadth First Search consistently produces the longest path, though it performs well with respect to the other two metrics. Finally, Path Counting produces a decent estimate of the shortest path, but its exponential time cost means it is the worst performing by far on 2 out of 3 metrics.

# Gaant Chart

Our final Gantt chart is located here: [Cross Depew CS 456 Project Gantt Chart.xlsx.](https://siuecougars-my.sharepoint.com/:x:/r/personal/gwcross_siue_edu/Documents/Cross%20Depew%20CS%20456%20Project%20Gantt%20Chart.xlsx?d=w6a6cda94467943279fda886e5dd299fa&csf=1&web=1&e=qI9s81) We initially wanted to implement a third maze-solving algorithm from the Divide and Conquer section of the class so that we could compare and contrast all three strategies for this application. However, we determined that none of the algorithms we learned in class were suitable for maze solving. We completed all of our other objectives.

References

1. Quiñones, Rubi. *Lecture 5: Activity Selection Problems* [PowerPoint slides]. Department of Computer Science, Southern Illinois University Edwardsville.
2. Quiñones, Rubi. *Lecture 12: Fibonacci and Path Counting* [PowerPoint slides]. Department of Computer Science, Southern Illinois University Edwardsville.
3. Appendix

I think the lectures could be taught at a bit of a slower pace. I had a hard time trying to keep up with hand writing notes.