Lab 10: Solving a Maze Using Dijkstra

Alex Yazdani
Eric Reed

CS F003C

26 June 2023

Introduction

In this project, a class Graph is defined which uses another class, Vertex to create objects representing a graph, complete with edges defined in an adjacency table. The Dijkstra Algorithm is used within the class to return the shortest path of vertices between any given start and stop vertices. The Graph method Dijkstra_solve() will do so and return a list of the data of these vertices.

Another class, Maze, is defined which creates a grid of nodes based on the height and width inputs and sets up vertical and horizontal walls. The create_solution_path() method tears down parts of the walls in order to create a solution path, and then creates dead-end paths to trick the user / algorithm that is trying to solve the maze. A solution path is built, and calling self.solution_path returns a list of nodes representing the solution from start to finish. A method create_graph() is defined within the Maze class which will instantiate a Graph object and create edges for all neighboring nodes that are not separated by a wall. Using the dijkstra_solve() method on this Graph object will return the same list as the Maze object's self.solution_path property.

During the construction of the maze, 2 different methods (or a combination of the two) can be used. The first is the random method, which keeps track of nodes visited and randomly selects adjacent, unvisited nodes and tears down the wall between the two. The next method is the stack method, which keeps a stack of nodes. The stack is popped and a random, unvisited, adjacent node is selected, the wall is torn down, and this new node is pushed back onto the stack (as well as the current node if it still has adjacent, unvisited nodes). Of course, both methods have more nuance not stated here, but these are the basic ideas of how each method operates. Table 1 below shows the timing data for different sized mazes and different construction methods. Finally, a bias can be set (representing percent likelihood of each method being used) which will produce a combination of both methods.

Data and Analysis

	Size (ms):	5	10	20	40	80	160	320
Stack Method:	**Measure Time:**	0.955	2.663	9.261	31.745	127.795	515.862	2030.379
	Factor Increase:		**2.788482**	**3.477657**	**3.427816**	**4.025673**	**4.036637**	**3.935896**
Random Method:	**Measure Time:**	0.979	3.037	9.267	34.344	129.997	498.011	2036.723
	Factor Increase:		**3.102145**	**3.051366**	**3.706054**	**3.785144**	**3.830942**	**4.089715**
Bias = 0.5:	**Measure Time:**	0.951	2.789	8.806	33.201	124.875	503.475	2018.707
	Factor Increase:		**2.932702**	**3.157404**	**3.77027**	**3.761182**	**4.031832**	**4.009548**

Table 1: Data from Test Program

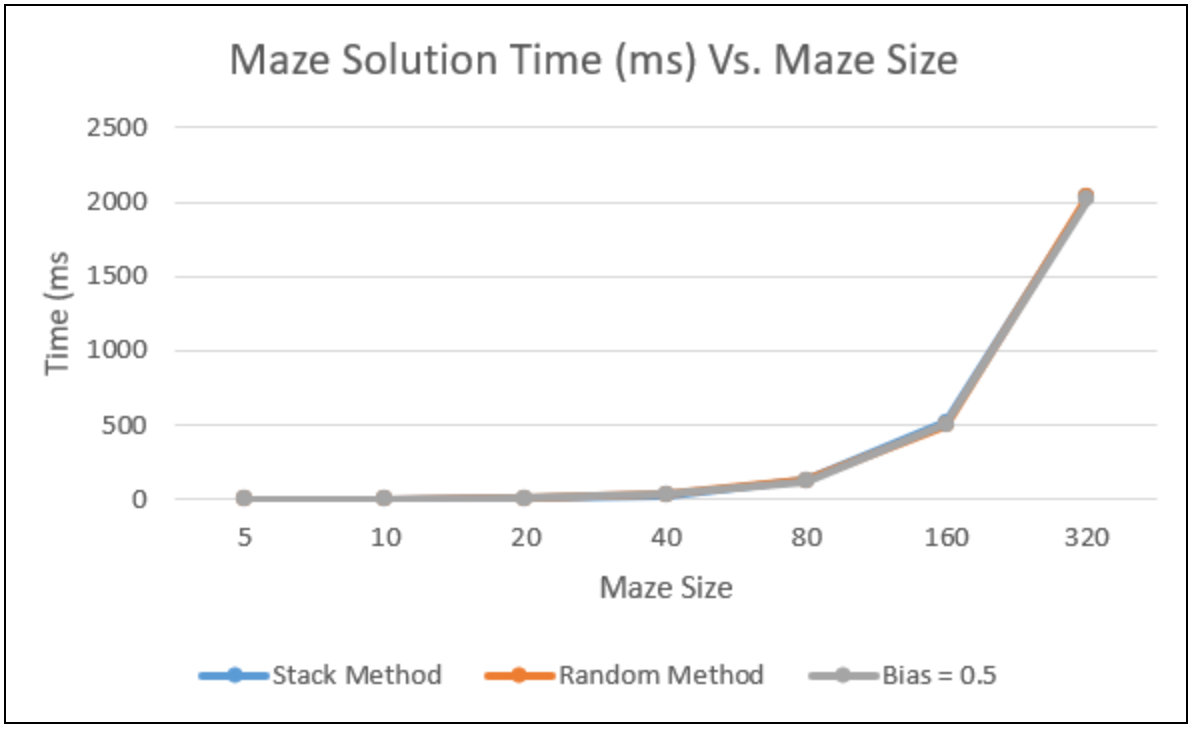

Figure 1: Plot of Test Program Data for Each Maze-Builder Method

Within the test program, a variable “size” is set which controls both the width and the height of the maze. There is a quadratic relationship between the value of size and the number of nodes in the maze. This is because for every increase in size, there is the same increase in both the number of nodes in the x-axis as well as the y-axis. In the Dijkstra Algorithm, each node in the partially processed queue is looped through, and for each node, each edge is looped through as well. In this specific case, the number of edges will be at most 4, and is not heavily related to the size of the maze, so can be ignored when considering the time complexity. Nodes on the border will have 2-3 edges, but this becomes negligible as size increases. Instead, there is a direct linear relationship between the number of nodes in the maze and the number of iterations of the for loop that runs through each element of the partially processed queue. Therefore, there is a quadratic relationship between size and the time to run through each node, meaning the time complexity is O(N^2) where N represents size. Empirically, this is seen in Table 1. Specifically, looking at the factor increases between size=160 and size=320, each method results in a value of right around 4. This is expected from quadratic time complexity. With smaller sample sizes, the factor increases are lower than 4. This is because the test algorithm actually sets the height to be size+5, while the width is simply size. This is negligible when comparing mazes 160x165 vs. 320x325, but cannot be ignored with smaller mazes, such as 5x10 vs. 10x15.

From Figure 1, there appears to be no difference in solution time for the maze using any of three methods. This remains true for every maze size; the shape of the curve is the same for the stack/random methods, as well as the bias of 0.5. Looking at the actual mazes and solutions that are produced, the stack method produces a very complex solution with simple distractor paths, while the random method produces a rather simple, straight solution with more complex distractor paths. These can be seen below in Figures 2 and 3.

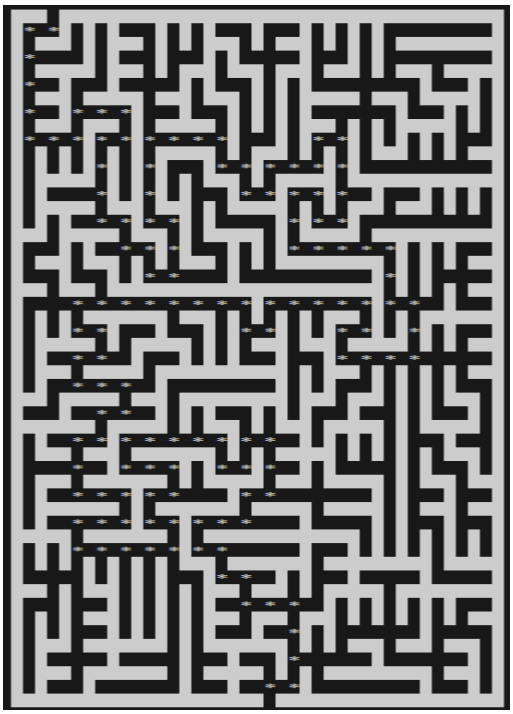

Figure 2: Size 20 Maze, Stack Method

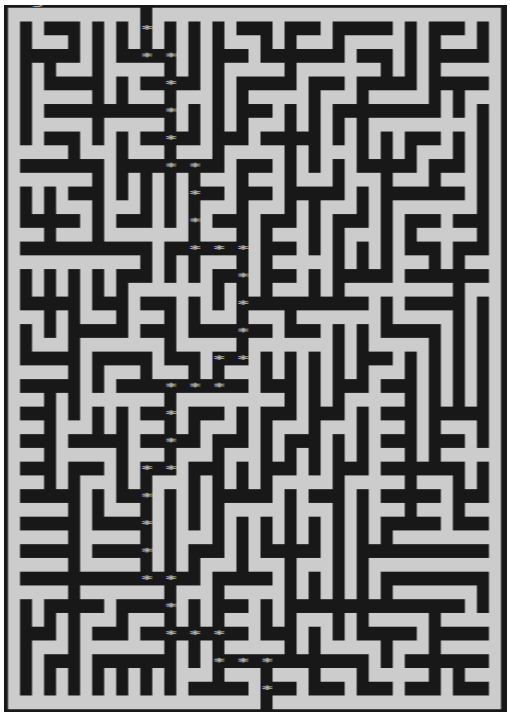

Figure 3: Size 20 Maze, Random Method

Using the Bias method, a combination of the previous two methods can be used (with a small change to the provided code. A low bias value will generate a mostly random maze, while a high value will generate a mostly stack-like maze. A bias value of 0.5 will generate the maze using each method 50% of the time. This behavior is shown in Figures 4-6 below:

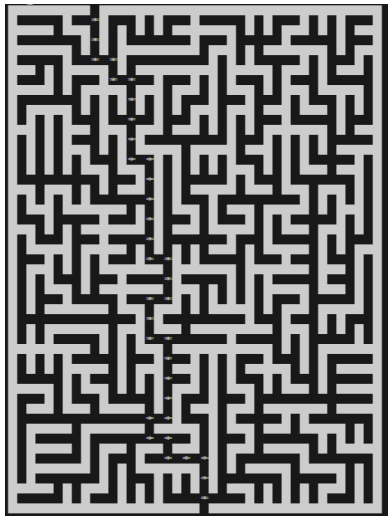

Figure 4: Bias = 0.05

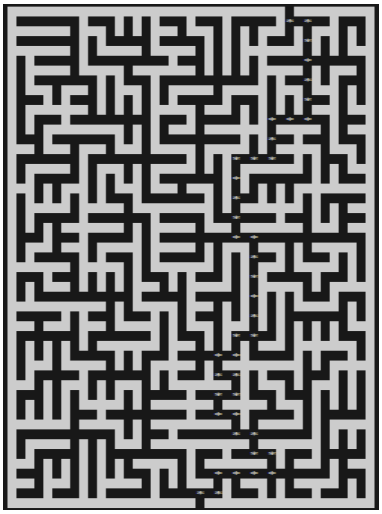

Figure 5: Bias = 0.50

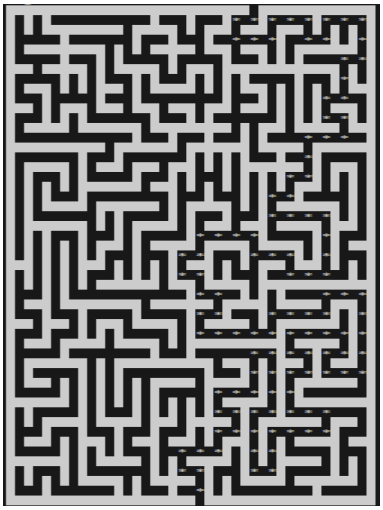

Figure 6: Bias = 0.95