Maze Algorithms

Maanvi Nunna
Department of Computer Science
Computer Science Engineering
PES University

Niharika G
Department of Computer Science
Computer Science Engineering
PES University

Omkar Metri
Department of Computer Science
Computer Science Engineering
PES University

***Abstract*—We start by generating mazes using different approaches, where we are differentiating on various properties like simply connected graphs, number of dead ends, convolution, distribution of valency and complexity. The mazes generated are random and unique every time you run the program. The goal of this project is to understand the graph data structures at a deeper level while enjoying generating mazes with all the different algorithms out there. There is no universally ideal algorithm to generate mazes, hence we will explore various techniques.With the maze generated by the above algorithms, we want to delve deeper into graph searching and traversal and hence would like to challenge ourselves to implement maze solving algorithms**

I. Introduction

There are two types of mazes : perfect mazes and braided mazes. In perfect mazes there is only one path from every cell to the other cell, whereas in braided mazes you can have many solutions from one cell to the other, or sometimes even none. In case of perfect mazes you can choose any two cells as the start and end points and be sure that there will be a solution between those two points. Texture refers to the style of the passages of the maze such as how long they tend to be or which direction do they lead to . When algorithms tend to produce a maze of a certain texture it means they are biased, and here we try to avoid bias by using random walk. Bias is the characteristic that makes pathways in maze go in one direction more than the others. And lastly an uniform algorithm generates all mazes with equal probability. Here, we try to generate perfect mazes with very subtle bias and explore the various characteristics of the texture and other properties of the algorithms.

II. Maze Generation Algorithms

A. *Backtracking*

This algorithm is a randomized version of the depth-first search algorithm. Frequently implemented with a stack, this approach is one of the simplest ways to generate a maze using a computer. Starting from a random cell, the computer then selects a random neighbouring cell that has not yet been visited. The computer removes the wall between the two cells and marks the new cell as visited, and adds it to the stack to facilitate backtracking. The computer continues this process, with a cell that has no unvisited neighbours being considered a dead-end. When at a dead-end it backtracks through the path until it reaches a cell with an unvisited neighbour, continuing the path generation by visiting this new, unvisited cell (creating a new junction). This process continues until every cell has been visited, causing the computer to backtrack all the way back to the beginning cell. We can be sure every cell is visited.As we know that this algorithm involves deep recursion which may cause stack overflow issues on some computer architectures. The algorithm can be rearranged into a loop by storing backtracking information in the maze itself. This also provides a quick way to display a solution, by starting at any given point and backtracking to the beginning.Mazes generated with a depth-first search have a low branching factor and contain many long corridors, because the algorithm explores as far as possible along each branch before backtracking.Generating mazes using Depth First Search ends up becoming on order of time complexity a function of the number of nodes (cells) and of the number of edges for a given maze.Since for the number of cells visited each edge has to be considered, the runtime ends up becoming O(|V|+|E|)

B. *Prims*

The algorithm works by filling in a grid with "wall" tiles and opening one starting cell at random. This starting cell basically becomes a root node from which paths begin to branch. For each surrounding wall of the open cell which comprise the "frontier", the algorithm makes the decision to, if the tile on the other side of the wall has not been explored and is not part of the maze, knock down the wall and essentially open that tile as well as the tile where the wall was, connecting the two nodes. The walls of this new node then become part of the frontier. Intuitively, the random selection from the frontier at each iteration provides branching, and the removal from the frontier only when the cell cannot be operated on because the opposite cell has already been explored ensures sufficient backtracking to explore the entire maze. The typical features of these mazes are that they have strong radial texture centered on the starting cell. These mazes tend to have more dead ends than other algorithms, and shorter paths.This algorithm implementation ends up becoming on order of $O(|V|^2)$ since for each cell frontier cells must be generated and added to the maze.

C. *Eller's*

Ellers maze generation algorithm works by considering one row at a time, hence it is not memory expensive at all and it doesn't have the obvious biases that other efficient algorithms have. It builds up sets to keep track of which cells are reachable from which other cells. This algorithm is a

combination of two other algorithms Sidewinder and Kruskal. As we go through every row, mark each cell with a distinct number if not market yet. In short the algorithm says, in every row until the last but one, randomly link adjacent cells as long as they don't share a set. Sharing a set means that linked cells will fall into the same set. Next step is, choose at least one cell from each set to carve a passage towards the next southward row. Then advance to the next row and repeat the process. When you reach the last row, link all the cells as long as they belong to different sets. The reason we can not merge cells of the same set is that it would create loops which is not desirable in a perfect maze. Also, why we need to carve a passage down from at least one cell in every set is that if we don't do it would lead to isolation. The time complexity of this algorithm is linear i.e., O(|V|) since it processes one row at a time.

D. *Recursive Division*

The recursive division algorithm treats the maze as a shape whose components are nearly identical to the whole maze, in other words the sub-mazes are nearly the same as the outer maze. Until now we looked at algorithms that carve passages starting with a grid made of solid walls for every cell, whereas recursive division starts with an open space and adds walls till a maze has been produces. This algorithm works by dividing an open space into two sub spaces by adding a wall between them and the way these two sub spaces will remain connected is that one part of the wall previously added is removed creating a passage between the otherwise completely disconnected sub spaces. From there on, this algorithm works by repeating the above process on each subspace recursively until the passages created are of desired size. The biases of recursive division include the boxiness characteristic in the texture of the maze due to repeated subdivision of the maze. While creating a single passage between two areas after placing a wall, there were regions of the maze that are enclosed with very few passages leading out. This creates a bottleneck where any path from one side of the maze to the other, must go through this path. One practical application of recursive division is that it can be used to generate rooms or open areas or a real life example like a parking lot. The way to generate an open space wider than one or two cells would be to stop the recursion at the desired size. The time complexity of this algorithm is also O(|V|) because throughout the program it needs to keep track of a stack whose maximum size if the size of a row.

III. Maze Solving Algorithms

There are a number of different maze solving algorithms, i.e, automated methods for the solving of mazes.

Mazes containing no loops are known as "simply connected", or "perfect" mazes, and are equivalent to a tree in graph theory. Thus many maze solving algorithms are closely related to graph theory. Intuitively, if one pulled and stretched out the paths in the maze in the proper way, the result could be made to resemble a tree. Here, we have implemented two solving algorithms, i.e, BFS and A*.

A. *Breadth First Search*

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

It uses the opposite strategy as depth-first search, which instead explores the highest-depth nodes first before being forced to backtrack and expand shallower nodes.Breadth first traversal is accomplished by enqueueing each level of a tree sequentially as the root of any subtree is encountered.

There are 2 cases in the iterative algorithm:

Root case: The traversal queue is initially empty so the root node must be added before the general case.

General case: Process any items in the queue, while also expanding their children. Stop if the queue is empty. The general case will halt after processing the bottom level as leaf nodes have no children.

Time and space complexity The time complexity can be expressed as O(|V|+|E|), since every vertex and every edge will be explored in the worst case. |V| is the number of vertices and |E| is the number of edges in the graph. Note that O(|E|) may vary between O(1) and O($|V|^2$.), depending on how sparse the input graph is. When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as O(|V|), where |V| is the cardinality of the set of vertices. This is in addition to the space required for the graph itself, which may vary depending on the graph representation used by an implementation of the algorithm. When working with graphs that are too large to store explicitly (or infinite), it is more practical to describe the complexity of breadth-first search in different terms: to find the nodes that are at distance d from the start node (measured in number of edge traversals), BFS takes O(bd + 1) time and memory, where b is the "branching factor" of the graph (the average out-degree).

B. *A**

A* algorithm is widely used in pathfinding and graph traversal, which is the process of finding a path between multiple points, called "nodes". It enjoys widespread use due to its performance and accuracy. However, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

f(n)=g(n)+h(n)

where n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal.

A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended.The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

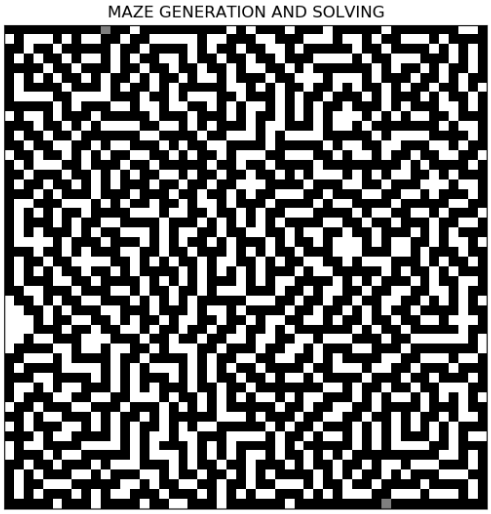
Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the cost of the shortest path, since h at the goal is zero in an admissible heuristic.

The time complexity of A* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: $O(b^d)$, where b is the branching factor (the average number of successors per state). This assumes that a goal state exists at all, and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate.

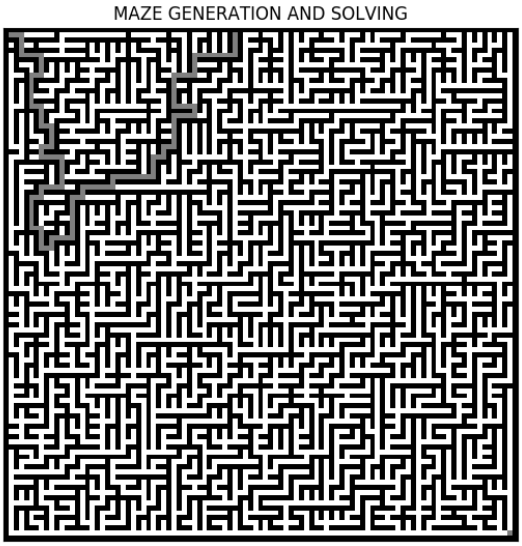
A* is commonly used for the common pathfinding problem in applications such as games, but was originally designed as a general graph traversal algorithm.

IV. Visualizations

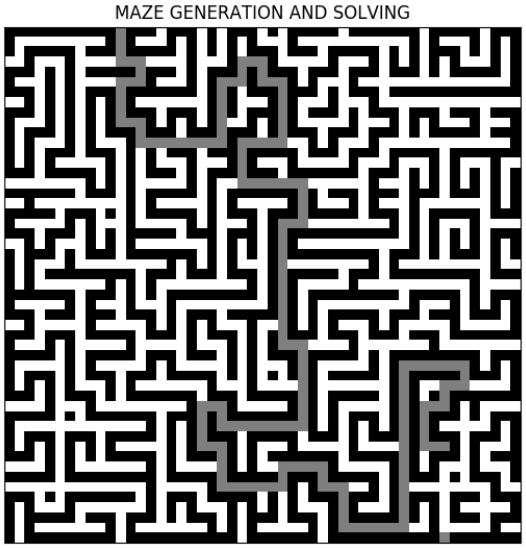
A. Recursive Backtracking



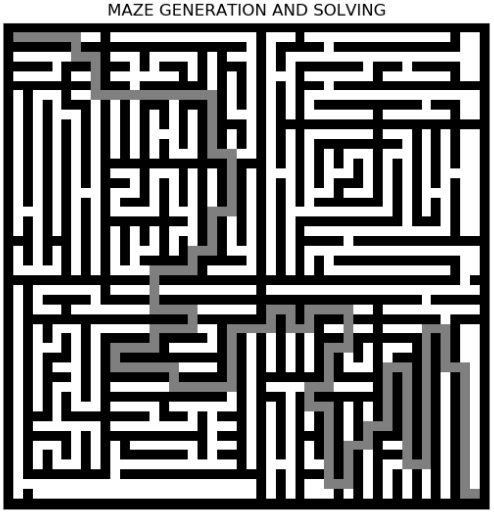
B. Ellers



C. Prims



D. Recursive Division



V. Results

A. Analysis of Maze Generation Algorithms

Algorithm	Time Complexity
Backtracking	O (|V|+|E|)
Prims	O ($\|V\|^2$)
Ellers	O (|V|)
Recursive Division	O (|V|)

B. Analysis of Maze Solving Algorithms

Algorithm	Time Complexity
BFS	O (|V|+|E|)
A*	Heuristic

VI. Conclusion

Therefore, here ends our quest to understand a few interesting maze generation and maze solving algorithms. In case of maze generation we have looked at Recursive backtracking algorithm, Prims, Ellers and Recursive division algorithms. Each algorithm has its features and complexities and based on the application where it is desired to be used the respective algorithm can be employed. We have also looked at a couple of maze solving algorithms like Bread First Search and A* algorithm. Generating mazes and solving them is like any other algorithm driven procedure, however one thing that distinguishes mazes from the rest is that you have the liberty to implement the algorithm the way you like, there is no right or wrong way as long as the requirements of the application are met.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.
[2] http://www.astrolog.org/labyrnth/algrithm.htm
[3] http://weblog.jamisbuck.org/under-the-hood/
[4] https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
[5] https://www.geeksforgeeks.org/a-search-algorithm/