Exploring Depth-First Search for Maze Generation and A* for Maze Solving

Nick Abegg nicholas.abegg@stvincent.edu
Jake Buhite jake.buhite@stvincent.edu

I. Problem Summary

Maze generation is often used in fields such as robotics and entertainment. Often it is not convenient or reasonable to generate mazes manually due to the time or tedium of generating mazes that vary in design and size. Having a method of generating mazes automatically allows for not only diversity amongst generated mazes but also the ability to do so in significantly less time. We utilized a randomized iterative depth-first search algorithm to automatically generate mazes. The simplicity of the algorithm allows for fast generation while still producing diverse results.

II. Literature Overview

A. Randomized Depth-First Search

Randomized depth-first search has been utilized before when generating mazes automatically. Research has demonstrated that the algorithm is capable of guaranteeing complete maze traversal. This ensures that none of the generated mazes are unsolvable and that they all have a solution. This is due to the nature of depth-first search and that it explores every node in the generated maze [1].

Pseudocode Recursive Depth-First Search [1]

```
function GENERATE-MAZE(cell):
    MARK-CELL-VISITED(cell)
    neighbors =
    UNVISITED-NEIGHBORS(cell)
    if neighbors is not empty:
        neighbor = RAND-NEIGHBOR(neighbors)
        PUSH-STACK(neighbor)
        REMOVE-WALL(cell, neighbor)
        cell = neighbor
        GENERATE-MAZE(cell)
    else:
        previousCell = POP-STACK()
        cell = previousCell
        GENERATE-MAZE(cell)
```

Note that while this algorithm utilizes recursion, our maze generator algorithm will implement an iterative form of depth-first search in which no recursion is needed.

B. Real World Application in Video Games

A real-world application of automated maze generation is in video game autonomous character path-finding. Video game Artificial Intelligence requires path-finding for navigating non-player characters throughout a game world. Calculating the best path for the character to take within reasonable computation time under restricted resources of memory and CPU is rather challenging. Path-finding problems in video games can often be represented, explored, and solved as mazes. Therefore investigating automated maze generation techniques such as Iterative Depth-First Search can result in improvements to video game artificial intelligence [2].

III. Algorithm Description

The two algorithms used for maze generation and solving include iterative depth-first search and A*, respectively.[1] These algorithms consist of significant differences in their behaviors, as iterative depth-first search is a form of uninformed search, while A* falls under the category of informed search [3]. As a result, each algorithm is expected to display different traversal behavior and varying runtime and memory performance compared to the other.

A. Maze Generator: Iterative Depth-First Search

1) Algorithm Description: Iterative depth-first search is a search algorithm that utilizes randomization and a stack to traverse the depth of the maze [1]. For each iteration, the popped cell is used to determine which neighbors around the

[1]Code for both the algorithms and the data collection process can be found here: https://github.com/jakebuhite/maze-generator-solver.

cell have not been explored. It randomly selects one of these neighbors and removes the wall between the current cell and the chosen neighbor. The chosen neighbor will then be pushed to the stack, popped in the next iteration, and used to determine which of its own neighbors have not been explored. If a popped cell's neighbors have all been explored, the search algorithm backtracks until it finds a cell that has an unexplored neighbor. This continues until all of the nodes have been explored.

2) Pseudocode Iterative Depth-First Search:

```
function GENERATE-MAZE(maze):
    Initialize an empty stack
    MARK-CELL-VISITED(maze[start])
    PUSH-STACK(maze[start])
    while stack is not empty:
        current = POP-STACK()
        nbr = RAND-NEIGHBOR(current)
        if nbr is not None:
            PUSH-STACK(current)
            REMOVE-WALL(current, nbr)
            MARK-CELL-VISITED(maze[nbr])
            PUSH-STACK(nbr)
```

3) Data Analysis: One of our experiments consists of the generation of mazes with various dimensions. The expected runtime complexity of depth-first search is O(b^d), where b represents the number of edges that leave a node [3] and d represents the maximum depth in which the algorithm will travel until it reaches a cell that has no unexplored neighbors. For space complexity, iterative depth-first search has an expected upper bound of O(bd). In this case, the branching factor of each node is dependent on the number of neighboring cells that have not been visited at each cell in the maze, while d increases as the maze size increases. As shown in Fig. 2, the maze generator appears to have an exponential runtime. Similarly, Fig. 4 appears to demonstrate exponential space complexity as the maze sizes increase. Therefore, our findings support the expected asymptotic runtime but do not support the asymptotic space complexity for iterative depth-first search.

B. Maze Solver: A*

1) Algorithm Description: The A* search algorithm is a complete and optimal algorithm that finds the shortest path in a maze from the starting cell to the goal cell [3]. One dictionary manages the path costs (*g(n)*) of each cell, while another manages the sum of heuristic costs with path costs, or *f(n)*. Both are initialized with a value of infinity for each cell. Beginning with the starting cell, the search algorithm retrieves the cell with the lowest *f(n)* from the priority queue. If the current cell is the goal cell, the loop terminates. Otherwise, it checks all possible directions to determine which neighbors of the current cell are valid paths. For each valid path, it calculates its new *f(n)* by adding its path cost to the newly calculated heuristic cost. The heuristic used for calculating heuristic cost is Manhattan distance, which is the sum of the number of horizontal and vertical movements needed to reach the goal cell [3]. If this value is less than its previous value, the neighbor's *g(n)* and *f(n)* are updated, and the neighbor is added to the priority queue. Furthermore, the neighbor and its parent cell are added to a dictionary responsible for managing the parent cells of each cell. The loop continues until the frontier is not empty or the goal node is found. Finally, the algorithm traverses the parent-child dictionary to visited and mark each cell of the solution path in the maze.

2) Pseudocode A Search [4]:*

```
function SOLVE-MAZE(maze):
    gn, fn = {cell: INF for cell in maze}
    gn[start] = 0
    fn[start] = HEUR(start, goal)
    frontier = PRIORITY-QUEUE()
    frontier.INSERT((fn[start], start))
    path = {}
    while frontier is not empty
        current = frontier.POP().second
        if current equals goal: break
        for dir in POSSIBLE-DIRECTIONS()
            dx = current.x + dir.x
            dy = current.y + dir.y
            nbr = Cell(dx, dy)
            updatedG = gn[current] + 1
            if updatedG < gn[nbr]
                gn[nbr] = updatedG
                fn[nbr] = updatedG
                + HEUR(nbr, goal)
                frontier.INSERT((fn[nbr],
                nbr))
                path[nbr] = current
```

```
cell = goal
while cell is not start
    MARK-CELL-VISITED(cell)
    cell = path[cell]
MARK-CELL-VISITED(start)
```

3) Data Analysis: Like iterative depth-first search, our experiment for A* search includes collecting the runtime and memory usage of maze solving with increasing maze sizes. The expected runtime and space complexity of A* search is O(b^d), where b represents the possible directions that can be traversed (up to 4) and d represents the number of steps required to reach the goal cell. As shown in Fig. 4 and 5, the runtime and memory usage of A* search appear to grow exponentially as the maze size increases. Therefore, our findings support the expected asymptotic runtime and space complexity of A* search.

C. Example Generated & Solved Maze

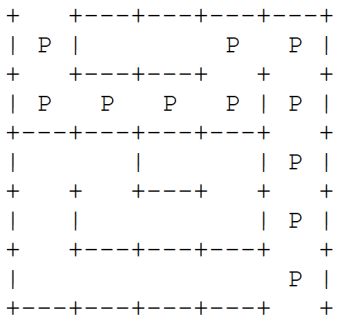

Fig. 1. 5x5 Generated and Solved Maze

Fig. 1 displays a simple 5x5 maze generated by iterative depth-first search and solved by A*. Notice each square of the maze represents one cell in the 5x5. The solution path, found by A*, is marked by the P's. The start of the maze is the top left cell, and the ending or "goal" is the bottom right cell.

IV. Testing & Data

We conducted two separate experiments with different dimensions tested within each of the experiments. Each experiment iterated through dimensions as follows:

- 5x5 - 100x100 increment x and y by 5
- 100x100 - 1000x1000 increment x and y by 100
- 1000x1000 - 10000x10000 increment x and y by 1000

This resulted in a total of 40 test for each experiment. In our test we measured the both the runtime & memory usage for both the maze generator and the maze solver.

A. Maze Generator Data

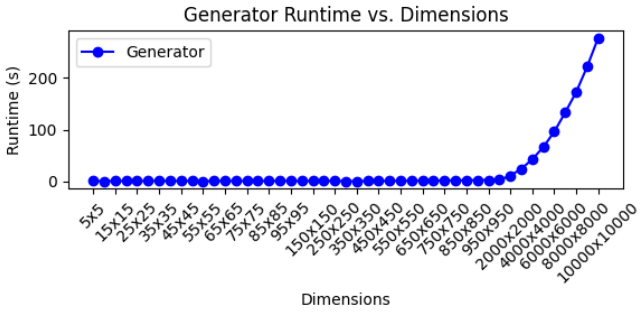

Fig. 2. Runtime of the Maze Generator using Iterative Depth-First Search

Here, in Fig. 2 we observe a exponential runtime which is expected for Iterative Depth First Search.

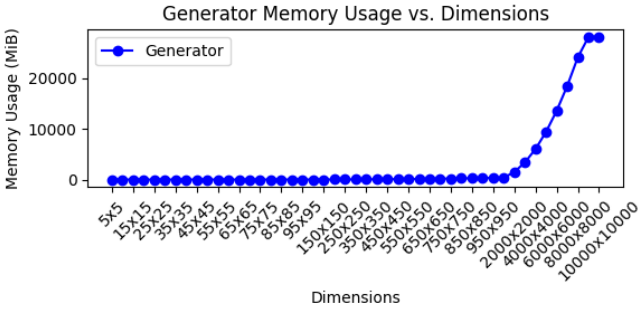

Fig. 3. Space Complexity of the Maze Generator

Notice there seems to be a peak in memory usage at 9000x9000 and 10000x10000. It is highly possible that the higher dimensions have reached a particular threshold wherein our computer's available memory is maximized. This leads to the plateauing that is shown on the far right of Fig. 3.

B. Maze Solver Data

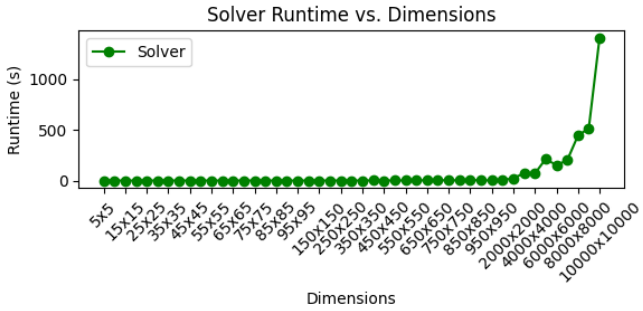

Fig. 4. Runtime of the Maze Solver using A*

The runtime in 4 is clearly exponential. A* is

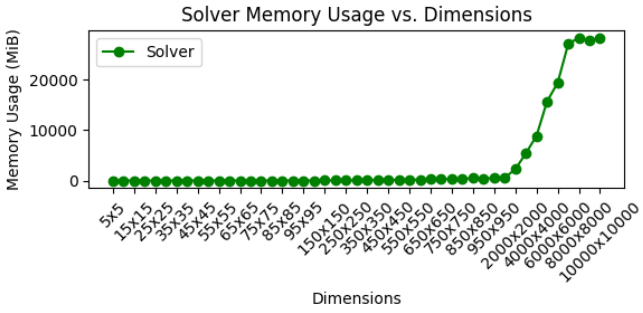

Fig. 5. Space Complexity of the Maze Solver

There also seems to be a similar trend to iterative depth-first search for memory usage in terms of the plateauing that occurs.

V. Conclusion

Automated maze generation using the randomized iterative depth-first search algorithm provides a simple solution to the problem of manually creating mazes of varying sizes. The ability to generate mazes automatically not only saves time but also ensures the creation of solvable and unique mazes. The A* search algorithm, while more complex than depth-first search, provides a complete and optimal methodology for obtaining the path from the start cell to the end cell of the maze.

The experiments conducted for the maze generator and solver support the expected asymptotic runtime and space complexity of these algorithms in all cases but the space complexity of the maze generator. Our findings demonstrate exponential runtime and space complexity for both maze generation and solving. While this is expected of A* search, iterative depth-first search is expected to have a linear space complexity. The reason for this significant increase in memory usage may be a result of the tool used to obtain memory usage metrics. It may also be possible that Python allocates additional resources to ensure that larger and more populated data structures have enough memory to maintain data. Additional exploration can include exploring the potential in SMA* and other search algorithms for solving larger mazes more efficiently.

The collection of data during the experiments conducted led to the discovery of variations in algorithmic performance between the two computers used for data collection. While it has not been determined whether this was a result of the environment chosen (Python) or the utilities used to collect data, future analysis should examine the effects of hardware on the runtime of the maze generator and solver. To maintain the integrity of our data, only data collected from one of the computers was used for analysis.

One potential area that could use additional exploration is the heuristic used in the A* search algorithm of the maze solver. While our current implementation of A* utilizes Manhattan distance, other heuristics may provide more efficiency and better performance when searching for the optimal path.

Another potential area that could be examined more in-depth is the types of mazes generated and solved. Our current implementation of maze generation and solving only supports rectangular mazes. However, adding support for circular or multidimensional mazes may provide more insight into the performance of these algorithms when used in the context of more complex maze designs.

Overall, this exploration provides further insight into the expectations of the performance and usability of iterative depth-first search for maze generation and A* search for maze solving in fields such as robotics, puzzles, and video games. Our findings provide a foundation for supporting further exploration of other algorithms for generating and solving mazes. Further investigation is necessary for a more comprehensive understanding of utilizing hardware of varying specifications, different search algorithms for generation and solving, efficient heuristics for maze-solving and the utilization of these search algorithms in other types of mazes.

REFERENCES

[1] S. H. Shah, J. M. Mohite, A. G. Musale, and J. L. Borade, "Survey paper on maze generation algorithms for puzzle solving games," *International Journal of Scientific & Engineering Research*, vol. 8, no. 2, p. 4, Feb. 2017.

[2] N. H. Barnouti, S. S. M. Al-Dabbagh, and M. A. S. Naser, "Pathfinding in strategy games and maze solving using a* search algorithm," *Journal of Computer and Communications*, vol. 04, no. 11, pp. 15–25, Jan. 2016.

[3] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson, 2009.

[4] X. Cui and H. Shi, "A*-based pathfinding in modern computer games," *International Journal of Computer Science and Network Security*, vol. 11, no. 01, pp. 125–130, Jan. 2011.