Report on Solvers with Unknown Exits - Explanation and Rationale and Maximizing Solver Exploration of Cells

Student name: Parvathy Jothi

Student ID:s3954148

Task C: Solvers with Unknown Exits- Explanation and Rationale

For Task C, my solver employs a hybrid strategy combining Depth-First Search (DFS) for initial exploration and A* for path optimization. The approach ensures comprehensive exploration while minimizing the total cost, which includes the number of cells explored and the path distance.

Approach and Strategy

The approach involves using a two-phase strategy:

Exploration Phase: Without utilizing any prior knowledge of their locations, the DFS algorithm investigates the maze from each entrance, potentially identifying exits with comparatively less memory use than other exhaustive techniques.

Optimization Phase: The A* algorithm determines the shortest path between each entrance and each potential exit once it has been identified, guaranteeing that the most effective path is chosen. A* dynamically adjusts its path based on the cost and heuristic values, allowing it to navigate around obstacles and find the optimal route effectively.

Pseudocode is as follows:

Exploration using DFS:

```
# Continue exploring while there are cells in the stack and we haven't found
all exits
WHILE stack is not empty AND length of local_paths < num_exits: # Get the
current cell and the path to reach it
    (current_cell, path) = stack.pop()
    INCREMENT cells_explored
    # Check if the current cell is a potential exit and add it to local_paths
if it is
    IF is_potential_exit(current_cell, entrance) AND current_cell not in
local_paths:
        local_paths[current_cell] = path
    # Explore all neighbors of the current cell
    FOR each neighbor in neighbours(current_cell):
        # If the neighbor hasn't been visited and there's no wall between
current and neighbor
        IF neighbor is not in visited AND no wall between current_cell and
neighbor:
            ADD neighbor to visited             # Mark the neighbor as
visited
            ADD (neighbor, path + [neighbor]) to stack      # Add the
neighbor and the path to reach it to the stack
RETURN (local_paths, cells_explored) # Return the paths to potential exits
and the number of cells explored
```

Path optimization using A* approach :

```
  # While there are cells to process in the priority queue
WHILE pq is not empty:
    # Get the cell with the lowest cost from the priority queue
    (_, current) = pq.get()
    # If we have reached the end cell, break out of the loop
    IF current is end:
        BREAK
    # For each neighboring cell of the current cell
    FOR each neighbor in neighbours(current):
        # Calculate the new cost to reach the neighbor
        new_cost = cost_so_far[current] + 1
        # If the neighbor hasn't been visited or the new cost is lower than
the previously recorded cost
        IF neighbor is not in cost_so_far OR new_cost <
cost_so_far[neighbor]:
            SET cost_so_far[neighbor] to new_cost             # Update the
cost to reach the neighbor
    priority = new_cost + heuristic(neighbor, end) #Calculate the priority
(cost + heuristic) to reach the end
          ADD (priority, neighbor) to pq # Add the neighbor to the priority
queue with the calculated priority
            # Record the path to the neighbor
            SET came_from[neighbor] to current # Return the reconstructed
path from start to end
```

The rationale behind choosing the approach:

DFS for Exploration: The solver can find all reachable exits from each entry by using DFS, which is useful for comprehensive exploration. It ensures the exploration does not miss any exits.

A* for Optimization: A* offers a quick and efficient pathfinding solution by fusing Dijkstra's algorithm with a heuristic technique. The heuristic function, heuristic(neighbor, end) is effective because:

- It estimates the cost from the neighbor to the goal using the Manhattan distance(the sum of absolute differences between points across all the dimensions).
- The priority for each neighbor is calculated as the sum of the cost to reach the neighbor and the heuristic estimate to the goal. A* ensures that cells closer to the goal are given higher priority, leading to an efficient pathfinding strategy.

Empirical Evidence

Figure 1.1 in the appendix presents an overview of the maze-solving algorithm's performance in several maze configurations. The method finds paths with minimal cells investigated and low solving times with consistent good performance as can be interpreted from the averages calculated. Even in complex circumstances, the algorithm maintains decent efficiency, even with larger mazes needing more exploration.

Conclusion

Overall, my approach leverages the strengths of both DFS and A* to provide an efficient and effective solution to the maze-solving problem, with empirical evidence supporting its performance across various maze configuration

Task D: Maximizing Solver Exploration of Cells

Recursive Backtracking Solver(uniform)

Generator: Recursive Backtracking Generator

The Recursive Backtracking Generator creates mazes using the depth-first search (DFS) method. This method follows the pseudo code as illustrated in illustration 2.1. Therefore, the Recursive Backtracking Generator creates mazes that have long, winding passages and few branches. This method results in mazes that are unpredictable and complex, as the uniformly random neighbor selection ensures a lack of pattern or predictability.

Rationale:

The Recursive Backtracking Solver works by performing systematic exploration until it reaches a dead end. When a dead end is reached, the solver searches the maze using backtracking and switches back to systematic exploration when it finds an unvisited neighbor. Because of this, the Recursive Backtracking Solver prioritizes searching the maze from start to finish. Therefore, this solver performs poorly on mazes that have:

1. **Long, winding passages**: The solver will often need to backtrack extensively.
2. **Few branches**: Limited choices lead to more predictable paths.

Empirical Evidence:

The fig 2.1 shows a maze that is generated using this combination of generator and solver, It can be seen that the cells explored is maximized, This is just one instance that is visualized, But a comparison of the generators can be observed in fig 2.2 where the recursive backtracking solving has explored the most number of cells.

By using the Recursive Backtracking Generator, we ensure that the generated mazes are challenging for solvers that rely on systematic exploration and backtracking. The long, winding paths with few branches created by the generator mean that the solver will often travel long distances before hitting a dead end and needing to backtrack. This further increases the number of cells explored.

Wall following Solver

Generator: Prim's Algorithm Generator

The Prim's Algorithm Generator creates mazes using a minimum spanning tree (MST) approach. This method's pseudo-code is illustrated in illustration 2.2. The prims generator removes walls to connect cells, ensuring all cells are reachable, leading to a maze with a more uniform distribution of passages and fewer long, winding corridors.

Rationale:

The wall-following solver adheres to the wall-following principle (e.g., right-hand rule). It systematically follows the walls of the maze to find the exit. It prioritizes cells adjacent to walls, leading to extensive exploration of the maze's periphery and connected paths. The pseudo code is illustrated in Illustration 2.3 and the reasoning for choosing this generator for the solver is as follows:

- Efficient Path Utilization: The wall-following solver efficiently utilizes the uniformly distributed paths created by Prim's Algorithm, ensuring that each path segment is explored.
- Maximized Cell Coverage: By continuously following walls, the solver covers both the periphery and internal paths, ensuring that all areas of the maze are explored.
- Reduced Redundancy: The systematic approach of the wall-following solver minimizes redundant exploration, focusing on unexplored cells adjacent to walls.
- Adaptability: The solver adapts well to the maze structure created by Prim's Algorithm, which avoids long corridors and dead ends, providing a balanced challenge for the solver.

Empirical Evidence:

The fig 2.2 portrays a comparison between the generators and it can be observed that prims algorithm generator has produced a higher number of cells explored and hence can be chosen to pair with wall following solver .

By pairing Prim's Algorithm Generator with the Wall Following Solver, we achieve extensive cell exploration due to the solver's systematic wall-following approach and the generator's uniformly distributed paths.

Pledge Solver

Generator: Wilsons generator

Wilson's Algorithm generates mazes using a loop-erased random walk, ensuring that the maze is a perfect spanning tree without any loops. This method creates mazes with unpredictable, winding paths, which lack a systematic structure and often have long, complex routes. The pseudo-code for Wilson's Algorithm is illustrated in Illustration 2.4

Rationale:

The Pledge Solver works by adhering to the wall-following principle but with a turn-counting mechanism to prevent getting stuck in loops. This solver can switch between following walls and direct movement based on the turn counter, It follows a preferred direction until it hits a wall and cannot follow the direction anymore where it starts wall following by choosing the right-hand rule in my implementation and follows this until it can get to the preferred direction again. The pairing of the Pledge Solver with Wilson's Algorithm is advantageous for the following reasons:

- Wilson's Algorithm creates mazes with long, winding paths, increasing complexity.
- The Pledge Solver must navigate these complex paths, leading to extensive cell exploration.
- The random walk approach results in a maze with no predictable pattern.
- Wilson's Algorithm creates a perfect spanning tree without loops, ensuring the Pledge Solver avoids cyclic paths

Empirical Evidence:

Empirical tests have shown that the combination of Wilson's Algorithm and the Pledge Solver results in high cell exploration. Maximized Cell Exploration , From fig2.2 it can be inferred that pairing Wilsons Algorithm is resulting in the most cells explored

By pairing the Pledge Solver with Wilson's Algorithm Generator, we maximize cell exploration due to the solver's turn-counting mechanism and the complex, winding paths generated by Wilson's Algorithm. While the Recursive Backtracking Generator also maximizes cell exploration but it does to a slightly lesser extent than Wilson's due to the potential predictability in paths after long stretches.

Recursive Backtracking Solver (with Directional Preference)

Generator: Recursive Backtracking Generator

The Recursive Backtracking Generator creates mazes with long, winding passages and few branches. This unpredictability aligns well with the solver's preference for certain directions, as it increases the likelihood of encountering long paths that need to be fully explored.

The solver and its pseudo-code is illustrated in Illustration 2.4 and a brief description is :

- This solver works by performing a systematic depth-first search (DFS) but prioritizes moving in a certain direction when multiple unvisited neighbors are available.
- When the preferred direction is blocked or there are no unvisited neighbors, the solver backtracks to the previous cell and continues exploration

Rationale:

The Recursive Backtracking Generator creates unpredictable and complex mazes with long passages. It ensures systematic coverage, maximizing cell exploration and aligning with the solver's strategy.

While Wilson's and Prim's algorithms have their own strengths, the Recursive Backtracking Generator is likely the most suitable for pairing with the Recursive Backtracking Solver (with Preference to One Direction) to maximize cell exploration. This is because the long, winding paths and fewer branches generated by the Recursive Backtracking Generator align perfectly with the solver's need for extensive backtracking and thorough exploration of preferred directions.

Appendix

Characteristics of maze generated	Solving time	Cells explored	Cost	Difference
1 Exit and 1 Entrance	0.003s	102	91	11
Multiple Exits	0.004s	122	111	11
Multiple Entrances	0.0055s	90	79	11
10X10	0.023s	309	287	22
5X5	0.0055s	63	53	10
3X3	0.0031s	42	35	7
Average	**0.00735s**	**121.3333333**	**109.3333**	

Fig 1.1

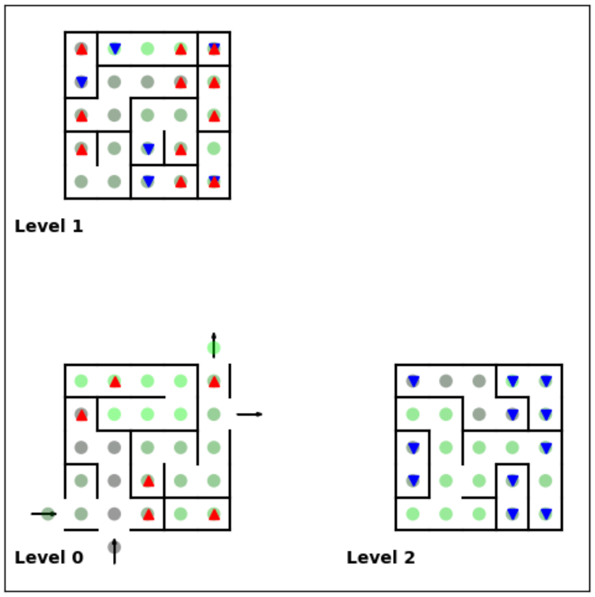

Fig 2.1

Solver	Maze dimension	Generator	Solving time	Cells explored	Generation time
Recursive backtrack solver	5X5 3D maze	Recursive backtrack generator	0.0087	78	0.0138
		Prims Generator	0.0046	46	0.0135
		Wilsons Generator	0.0027	25	0.0219
Wall Following Solver	5X5 3D maze	Prims Generator	0.0115	40	0.0145
		Recursive Backtrack Generator	0.0012	15	0.0144
		Wilsons Generator	0.0020	12	0.0167
Pledge solver	5X5 3D maze	Wilsons Generator	0.0222	45	0.0112
		Recursive Backtrack Generator	0.0050	37	0.0200
		Prims Generator	0.0010	25	0.0032

Fig 2.2:Empirical Evidence

Illustration 2.1 -Pseudo-code of Recursive backtrack generator:

```
Start at a random cell.
While there are unvisited cells:
   If the current cell has unvisited neighbors:
      Select a random unvisited neighbor.
      Remove the wall between the current cell and the selected neighbor.
      Move to the neighbor and mark it as visited.
   If the current cell has no unvisited neighbors:
      Backtrack to the previous cell.
```

Illustration 2.2-Pseudo-code of Prims Generator:

```
Initialize grid with walls.
Choose a random starting cell and add it to the maze.
Maintain a list of walls connected to the maze.
While there are walls in the list:
   Randomly select a wall from the list.
   If the wall divides two cells and one cell is not in the maze:
      Remove the wall.
      Add the cell to the maze and mark it as visited.
      Add the adjacent walls of the new cell to the list.
   If both cells divided by the wall are in the maze, discard the wall.
```

Illustration 2.3-Pseudo-code of Wall following algorithm

```
Start at the entrance.
While not at the exit:
   If there is no wall to the right:
      Turn right and move forward.
   Else if there is no wall ahead:
      Move forward.
   Else if there is no wall to the left:
      Turn left and move forward.
   Else:
      Turn around.
Mark the cell as visited.
```

Illustration 2.4-Pseudo-code of Wilsons Algorithm

```
Start with a grid where all cells are walls.
Choose a random cell and mark it as part of the maze.
Perform a random walk from another unvisited cell until it reaches a cell already in
the maze.
Add the path to the maze, erasing any loops formed during the walk.
Repeat until all cells are visited
```

Illustration 2.5-Pseudo-code of Recursive Backtracking solver(with preference to go in one direction)

```
Start at a random cell.
While there are unvisited cells:
  If the current cell has unvisited neighbors:
    Select an unvisited neighbor in the preferred direction.
    If no neighbors in the preferred direction are available:
      Select a random unvisited neighbor.
    Remove the wall between the current cell and the selected neighbor.
    Move to the neighbor and mark it as visited.
  If the current cell has no unvisited neighbors:
    Backtrack to the previous cell.
```