AI Project Report 2

Instructor: Dr. Ram Prasad Padhy

Artificial Intelligence (CS2012)

Date: April 14, 2023

Team Members :

CS20B1006 SRAVANTH CHOWDARY POTLURI
CS20B1044 AVINASH R CHANGRANI

Maze Solver Using AI

1 Introduction

The previous report informed regarding mazes, their analogies in real life and the techniques we can use to solve them which can also be used to solve similar problems such as map navigation using techniques both exhaustive and brute force such as BFS And DFS And also AI Heuristic Techniques such as A* Algorithm. In this report we aim to give a comprehensive report on the different Techniques for maze solving that we have used and a comparative study between these different techniques

2 How do we solve Mazes using AI

2.1 Maze Generation

To Solve Mazes we first need some mazes, so that's what we did, we generated some mazes, using the combination of a random function and some constraints we were able to ensure that we generated mazes that were solvable after which we finally saved them as CSV files of a total of 30 different mazes

2.2 Maze Solving

We have used three different algorithms to solve the 30 different mazes we generated - Breadth First Search (BFS), Depth First Search (DFS), and A* algorithm with both Manhattan and Euclidean heuristics.

```python
start = time.time()
for maze in range(30):
    # initialize the grid array full of zeros (blocked)
    num_rows = 41
    num_columns = num_rows
```

```
    grid = np.zeros((num_rows, num_columns))
    print(f"Generating Maze {maze}/{30}...", end=" ")
    done = False # loop until done
    # initialize curr_pos variable. Its the starting point for the algorithm
    curr_pos = (0, 0)
    pos_visited = []
    pos_visited.append(curr_pos)
    back_tracks = 0
    # define start and goal
    grid[0, 0] = 2 # (0,0)
    grid[-1, -1] = 3 # (num_rows-1, num_rows-1)

    while not done:
        # feed the algorithm the last updated position and the grid
        grid, curr_pos, back_tracks, done = generate_move(grid, curr_pos, pos_visited, back_tracks)
        if curr_pos not in pos_visited:
            pos_visited.append(curr_pos)
    # export maze to .csv file
    with open(f"mazes_input/maze_{maze}.csv", "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerows(grid)
        print(f"{time.time()-start:.3f} s")

print(f"--- finished {time.time()-start:.3f} s---")
```

A Python Code Snippet for generation of Mazes

2.2.1 Breadth First Search

BFS is a brute force algorithm that explores all possible paths from the start node to the end node, expanding nodes level by level. It guarantees the shortest path to the goal node but can be computationally expensive for larger and more complex mazes.

```
def solution(self, grid):
        # while the openlist is not empty
        while True:
            # get the first node in the openlist
            curr = self.openlist.pop(0)
            # get the position of the node
            x,y = curr.x, curr.y
            # get the neighbors of the node
```

```
            neighbors = [(x+1,y), (x-1,y), (x,y+1), (x,y-1)]
            # for each neighbor
            for neighbor in neighbors:
                # if the neighbor is in the grid and is not blocked
                if (is_in_grid(neighbor, self.dim) and
                 ↪  (grid[neighbor[0], neighbor[1]] in [1, 3])):
                    # create a new node and add it to the openlist
                    next_node = bfs_Node(neighbor, curr)
                    self.openlist.append(next_node)
                    # if the neighbor is the goal, then we are done
                    if (neighbor == self.goal):
                        self.closedlist.append(next_node)
                        return self.get_path(next_node)
            # if the node is not the goal, then add it to the
             ↪  closedlist (FIFO)
            self.closedlist.append(curr)
            # mark the node as explored in the grid
            explored = [(node.x, node.y) for node in self.closedlist]
            for pos in explored:
                grid[pos[0], pos[1]] = 4
```

A Python Code Snippet to Solve Mazes using BFS Algorithm

2.2.2 Depth First Search

DFS, on the other hand, is another brute force algorithm that traverses the maze depth-first. It explores each path as far as possible before backtracking to explore other paths. It is not guaranteed to find the shortest path but can be more efficient for larger and more complex mazes.

```
def solution(self, grid):
        # while the openlist is not empty
        while True:
            # get the first node in the openlist
            curr = self.openlist.pop(0)
            # get the position of the node
            x,y = curr.x, curr.y
            # get the neighbors of the node
            neighbors = [(x+1,y), (x-1,y), (x,y+1), (x,y-1)]
            # for each neighbor
            for neighbor in neighbors:
                # if the neighbor is in the grid and is not blocked
                if (is_in_grid(neighbor, self.dim) and
                 ↪  (grid[neighbor[0], neighbor[1]] in [1, 3])):
```

```python
                # create a new node and add it to the openlist (at the beginning) (LIFO)
                next_node = dfs_Node(neighbor, curr)
                self.openlist.insert(0, next_node)
                # if the neighbor is the goal, then we are done
                if (neighbor == self.goal):
                    self.closedlist.append(next_node)
                    return self.get_path(next_node)
        # if the node is not the goal, then add it to the closedlist
        self.closedlist.append(curr)
        # mark the node as explored in the grid
        explored = [(node.x, node.y) for node in self.closedlist]
        for pos in explored:
            grid[pos[0], pos[1]] = 4
```

A Python Code Snippet to Solve Mazes using DFS Algorithm

2.2.3 A* Algorithm (Manhattan And Euclidean Heuristics)

A* algorithm is an informed search algorithm that uses a heuristic function to guide its search towards the goal node. It combines both the cost of the path and the estimated distance to the goal to decide which node to expand next. We used two different heuristic functions for A* algorithm, Manhattan and Euclidean distance, to evaluate their effectiveness in solving mazes of different sizes and complexities.

```python
# function to calculate the cost of a node using the heuristic function (Euclidean distance)
    def heuristic1(self, pos):
        x, y = pos
        x_goal, y_goal = self.goal
        cost = np.sqrt((x_goal-x)**2 + (y_goal-y)**2)
        return cost
```

A Python Code Snippet For Euclidean Heuristic

```python
# function to calculate the cost of a node using the heuristic function (Manhattan distance)
    def heuristic2(self, pos):
        x, y = pos
        x_goal, y_goal = self.goal
        cost = abs(x_goal-x) + abs(y_goal-y)
        return cost
```

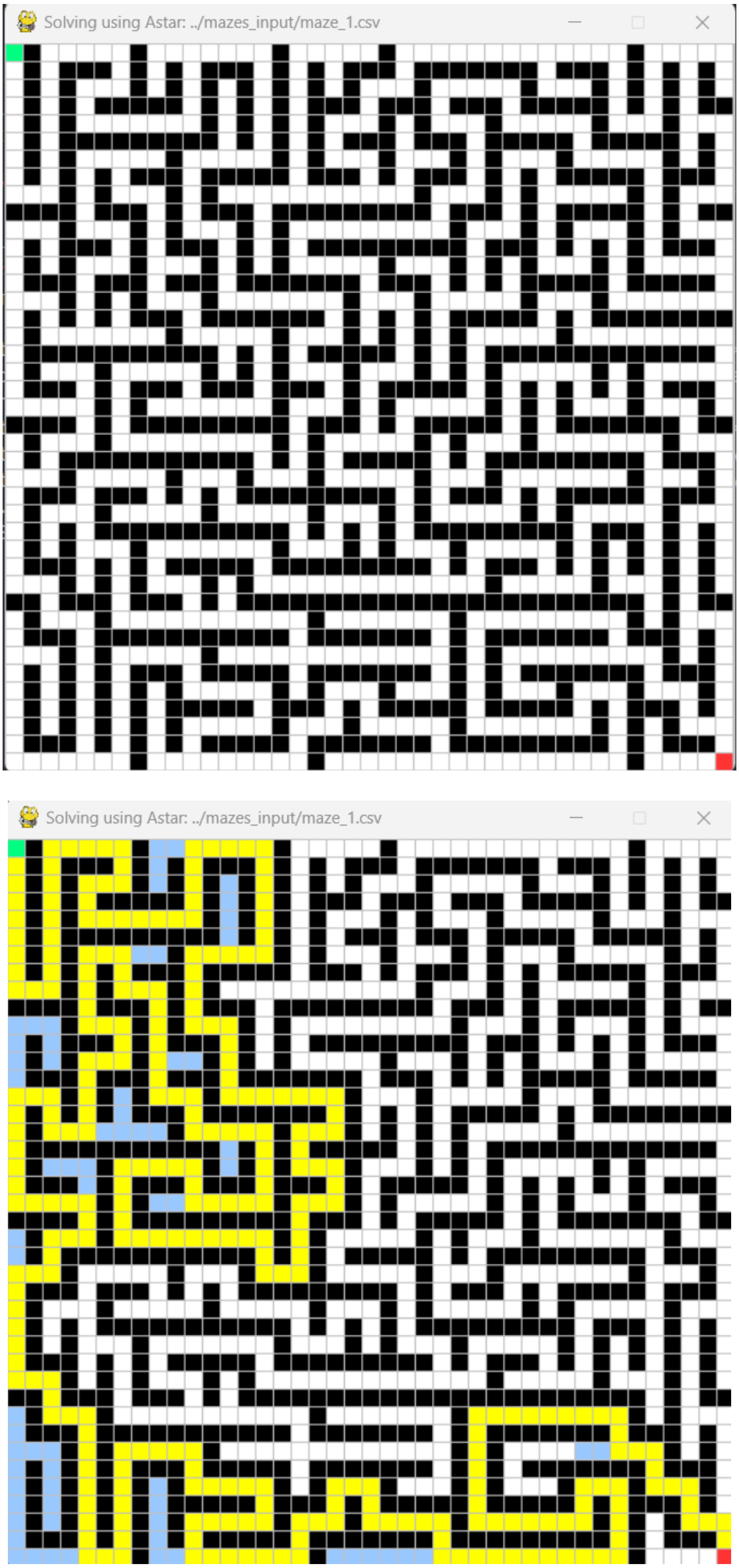
A Python Code Snippet For Manhattan Heuristic

```python
# function to solve the maze using A* algorithm with Euclidean distance
    def solution(self):
        unique = count()
        # calculate the cost of the start node
        h = self.heuristic1(self.start)
        g = 0
        f = h + g
        # create the start node
        start_node = Node(None, g, f, self.start)
        # add the start node to the open list
        self.open_list.put((f, next(unique), start_node))
        while True:
            curr = self.open_list.get()[2] # get the node with the lowest cost
            # if the current node is the goal, then we are done
            if curr.position == self.goal:
                while curr is not None:
                    self.path.append(curr.position)
                    curr = curr.parent
                self.path.reverse()
                return self.path, True
            # generate the children of the current node using the heuristic function
            for child in self.generate_children(curr,1):
                # if the child is not in the closed list, then add it to the open list
                if child not in self.closed_list:
                    self.open_list.put((child.f, next(unique), child))
            # add the current node to the closed list
            self.closed_list.append(curr.position)
        return self.path, False
```

A Python Code Snippet to Solve Mazes using A* Algorithm

2.3 Maze Solving Representation(Pygame GUI)

To enhance the user experience, we used Pygame GUI to represent the maze-solving process. This allowed the user to visualize how the algorithms explored the maze and displayed the path it found. As the algorithm progressed, the Pygame GUI updated the maze's exploration status, displaying which nodes were visited and the path taken. When the algorithm found the end node, the Pygame GUI displayed the path found, highlighting it for the user to see. This makes it easier for the user to understand how the algorithm was solving the maze and allows them to follow the process closely.



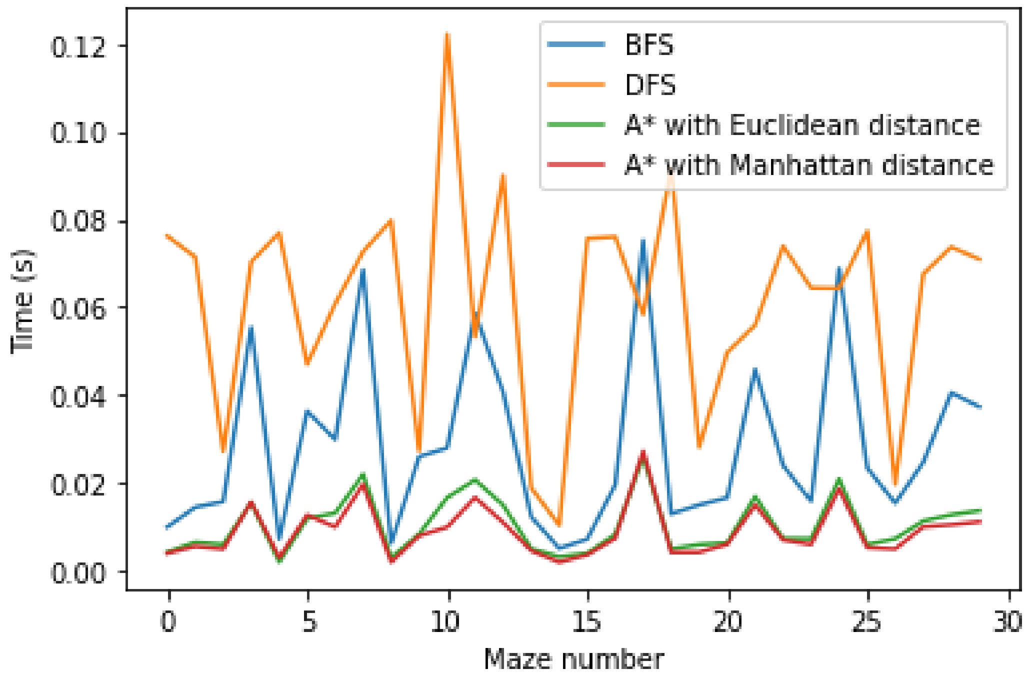
Solved Maze vs Unsolved Maze Representation

Report submitted to -: Dr. Ram Prasad Padhy

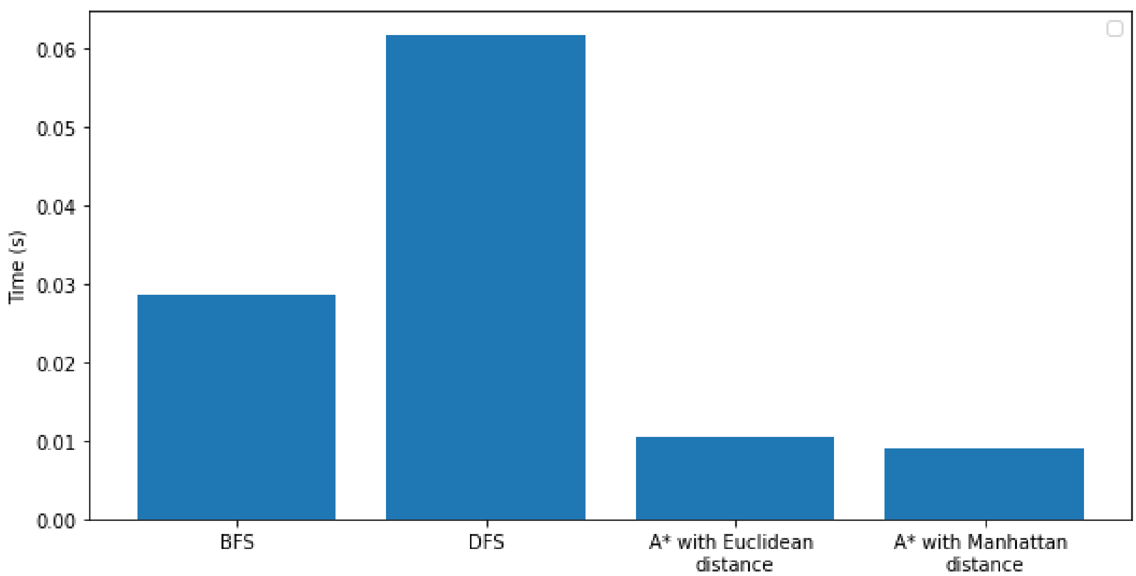
2.4 Performance Comparison

After performing different algorithms on the mazes we have generated in subsection 1, we then calculated and used the times taken by the different algorithms on each maze to perform data analysis and represented the comparative study using different Plots such as a line plot and a bar plot between different Algorithms

Line Plot Comparison of Solving time taken by Each Algorithm



Bar Plot Comparison of Average time taken by Each Algorithm



3 Conclusion

In Summary of the above report we can see that from the performance comparison subsection the Intelligent heuristic A* Algorithm using both Manhattan And Euclidean distance performances significantly better than brute-force based algorithms such as BFS and DFS displaying how the efficiency and effectiveness of Intelligent AI Search algorithms can help in solving complex maze navigation problems .In addition to this Conclusion we have also built a GUI Based application to display the process of maze solving by a computer using different algorithms

Report submitted to -: Dr. Ram Prasad Padhy

Individual Contributions for the Project

1 CS20B1044 AVINASH R CHANGRANI

- Researched about the Existing AI Paradigms for Maze Solving
- Developed The implementation of the maze solver GUI Using Pygame
- Programmed Different algorithms BFS,DFS and A* Using Manhattan and Euclidean Distance Heuristics to Solve the mazes
- Documented The Text for the Report Using LaTex

2 CS20B1006 SRAVANTH CHOWDARY POTLURI

- Researched The Existing Solutions/Codebases For Maze Solving Methods
- Programmed the code to generate different mazes to solve using the algorithms
- helped in proof reading and debugging of for PyGame GUI and Maze Solving Methods
- Performed Data Analysis for Runtime comparison between different search algorithms
- Documented The Text for the Report Using LaTex

References

1. Artificial Intelligence: A Modern Approach by Peter Norvig and Stuart J. Russel
2. https://en.wikipedia.org/wiki/A*_search_algorithm
3. https://en.wikipedia.org/wiki/Depth-first_search
4. https://en.wikipedia.org/wiki/Breadth-first_search
5. https://github.com/Mostafa-Ebrahim/AI-Maze
6. https://github.com/raulorteg/ai-maze-python
7. https://github.com/Noy-Bo/AI-Maze-Solver

Report submitted to -: Dr. Ram Prasad Padhy