Artificial Intelligence Assignment-01

Submitted by: Quazi Ghulam Rafi
ID: 293631

Introduction

In this assignment, several game agents for the Maze Runner and Checkers games are implemented using different heuristic search algorithms based on which the agents choose actions depending on the states of various problems. Later, the performances of the algorithms and heuristics used are statistically evaluated and analyzed. In the Maze Runner game, A* and Best-First Search with path memory and 40% randomness algorithms and Euclidean, Manhattan, Chebyshev, Octile distances, and Zero heuristics are used for making move decisions. Contrarily, the Checkers game uses the Minimax algorithm with alpha-beta pruning and piece count, king piece count, and board control heuristics for making move decisions. The rest of the report holds details about the implementations, results, and analyses of each of the games. The report then concludes the findings in the conclusion.

Maze Runner Implementation

The maze runner game is implemented using Python 3.12 version. The code comprises of the following classes for the following reasons.

Heuristics Class:

```python
import math

class Heuristics:
    @staticmethod
    def euclidean_distance(pos1, pos2):
        x1, y1 = pos1
        x2, y2 = pos2
        return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

    @staticmethod
    def manhattan_distance(pos1, pos2):
        x1, y1 = pos1
        x2, y2 = pos2
        return abs(x1 - x2) + abs(y1 - y2)

    @staticmethod
    def chebyshev_distance(pos1, pos2):
        # Calculate the Chebyshev distance heuristic from 'pos' to the goal.
        x1, y1 = pos1
        x2, y2 = pos2
        dx = abs(x1 - x2)
        dy = abs(y1 - y2)
        return max(dx, dy)

    @staticmethod
    def octile_distance(pos1, pos2):
        x1, y1 = pos1
        x2, y2 = pos2
        dx = abs(x1 - x2)
        dy = abs(y1 - y2)
        return dx + dy + (math.sqrt(2) - 2) * min(dx, dy)

    @staticmethod
    def zero_heuristic(pos1, pos2):
        # Zero Heuristic always returns zero, effectively turning A* into Dijkstra's algorithm.
        return 0
```

This class holds the definitions of the heuristics used in the game. The Heuristics class contains the following functions.

- euclidean_distance(pos1, pos2): This function returns the Euclidean distance, which is the straight line between the current and goal positions, as a heuristic for the concerned algorithm. This is a more optimistic estimation of the remaining path length.
- manhattan_distance(pos1, pos2): This function calculates the sum of both the horizontal and vertical distances between the current and the goal positions. This distance calculation assumes movement can only occur along grid lines and thus offer a conservative estimate of the remaining path length.
- chebyshev_distance(pos1, pos2): This heuristic function computes the maximum of the horizontal and vertical distances between the current and the goal positions. As this calculation allows diagonal movement, therefore, it is considered a more liberal estimation of the remaining path length.
- octile_distance(pos1, pos2): This function estimates the remaining path length by considering both horizontal and vertical movements as well as diagonal movements between the current and the goal positions. Therefore, this approach can be considered as a relatively more realistic estimation of the remaining path length.
- zero_heuristic(pos1, pos2): This function implements the zero heuristic or null heuristic, and returns a constant value of 0 for all positions. This approach results in an admissible heuristic but not informative, as it does not provide any estimate of the remaining path length to the goal. In the case of the A* algorithm, this heuristic basically turns the algorithm to Dijkstra algorithm.

Game Class:

```
class Game:
    def __init__(self, maze):
        self.maze = maze

    def get_neighbors(self, pos):
        # Get neighboring positions that are open (0) in the maze.
        neighbors = []
        x, y = pos
        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            new_x, new_y = x + dx, y + dy
            if (
                0 <= new_x < len(self.maze)
                and 0 <= new_y < len(self.maze[0])
                and self.maze[new_x][new_y] == 0
            ):
                neighbors.append((new_x, new_y))
        return neighbors

    def is_goal_state(self, state):
        return state == self.goal
```

This class defines the game environment of the maze and has the following functions.

- __init__(self, maze): This is the constructor of the Game class and has only the maze parameter representing the maze configuration for the game. Essentially, this function sets up the initial state of the game environment.
- get_neighbors(self, pos): Given a position, this function returns the list of neighboring positions that are open to travel, that is, has a 0 value.
- is_goal_state(self, state): This function takes in a position as input and compares it to a predefined goal position to return a judgement whether the position is the final or goal state or not.

Agent Class:

```python
import heapq
import time
import random

class Agent:
    def __init__(self, game, start, goal, heuristic_name):
        self.game = game
        self.start = start
        self.goal = goal
        self.heuristic_name = heuristic_name

    def heuristic(self, pos):
        # Choose and call a heuristic function from the Heuristics class.
        if self.heuristic_name == "zero":
            return Heuristics.zero_heuristic(pos, self.goal)
        elif self.heuristic_name == "euclidean":
            return Heuristics.euclidean_distance(pos, self.goal)
        elif self.heuristic_name == "manhattan":
            return Heuristics.manhattan_distance(pos, self.goal)
        elif self.heuristic_name == "chebyshev":
            return Heuristics.chebyshev_distance(pos, self.goal)
        elif self.heuristic_name == "octile":
            return Heuristics.octile_distance(pos, self.goal)

    def find_path_A_star(self):
        # Implementation of the A* search algorithm starts here.
        start_time = time.time()
        # Priority queue for open nodes.
        open_set = []
        heapq.heappush(open_set, (0, self.start))
        # Dictionary to track where each node came from.
        came_from = {}
        # Dictionary for storing the actual cost from start to each node.
        g_score = {pos: float('inf') for row in self.game.maze for pos in row}
        g_score[self.start] = 0
        # Dictionary for storing the total estimated cost from start to goal through each node.
        f_score = {pos: float('inf') for row in self.game.maze for pos in row}
        # The estimated cost to reach the goal from the start node.
        f_score[self.start] = self.heuristic(self.start)
        # Variable to track the maximum size of the open set during the search.
        max_open_set_size = 1
        # Variable to count the number of nodes explored during the search.
        nodes_explored = 0

        while open_set:
            max_open_set_size = max(max_open_set_size, len(open_set))
            # Get the node with the lowest f-score.
            _, current = heapq.heappop(open_set)
            nodes_explored += 1

            if current == self.goal:
                # Goal reached, calculate execution time and return the path and statistics.
                path = self.reconstruct_path(came_from, current)
                path_length = len(path)
                end_time = time.time()
                execution_time = end_time - start_time
                return path, nodes_explored, max_open_set_size, execution_time, path_length
            for neighbor in self.game.get_neighbors(current):
                # Calculate tentative g-score, check if a better path is found, and update accordingly.
                tentative_g_score = g_score[current] + 1
                if tentative_g_score < g_score.get(neighbor, float('inf')):
                    came_from[neighbor] = current
                    g_score[neighbor] = tentative_g_score
                    f_score[neighbor] = g_score[neighbor] + self.heuristic(neighbor)
                    heapq.heappush(open_set, (f_score[neighbor], neighbor))
        # If the open set is exhausted and no path is found, calculate execution time and return None with statistics.
        end_time = time.time()
        execution_time = end_time - start_time
        return None, nodes_explored, max_open_set_size, execution_time, 0

    # def find_path_best_first_search(self, random_factor=0.05):
    def find_path_best_first_search(self, random_factor=0.4):
        start_time = time.time()
        open_set = []  # Priority queue for open nodes.
        heapq.heappush(open_set, (0, self.start))  # Add the start node to the open set.
        came_from = {}  # Dictionary to track where each node came from.
        max_open_set_size = 1  # Variable to track the maximum size of the open set during the search.
        nodes_explored = 0  # Variable to count the number of nodes explored during the search.
        visited = set()  # Set to track visited nodes.

        while open_set:
            max_open_set_size = max(max_open_set_size, len(open_set))  # Update the maximum open set size.
            # Get the node with the lowest f-score.
            if random.random() < random_factor:
                # Occasionally, choose a random node instead of the one with the lowest f-score.
                _, current = random.choice(open_set)
            else:
                # Get the node with the lowest f-score.
                _, current = heapq.heappop(open_set)
            nodes_explored += 1  # Increment the nodes explored count.
            if current == self.goal:
                # Goal reached, calculate execution time and return the path and statistics.
                path = self.reconstruct_path(came_from, current)
                path_length = len(path)  # Calculate path length
                end_time = time.time()
                execution_time = end_time - start_time
                return path, nodes_explored, max_open_set_size, execution_time, path_length

            visited.add(current)  # Mark the current node as visited.

            for neighbor in self.game.get_neighbors(current):
                if neighbor not in visited:
                    # Calculate the heuristic value for the neighbor.
                    neighbor_heuristic = self.heuristic(neighbor)
                    # Add the neighbor to the open set with the heuristic value as priority.
                    heapq.heappush(open_set, (neighbor_heuristic, neighbor))
                    # Update the came_from dictionary to track the path.
                    came_from[neighbor] = current

        # If the open set is exhausted and no path is found, calculate execution time and return None with statistics.
        end_time = time.time()
        execution_time = end_time - start_time
        return None, nodes_explored, max_open_set_size, execution_time, 0

    def reconstruct_path(self, came_from, current):
        # Initialize the path list with the current (goal) node.
        path = [current]
        # Backtrack from the goal node to the start node using the 'came_from' dictionary.
        while current in came_from:
            current = came_from[current]
            path.append(current)

        # Reverse the path to obtain the correct order from start to goal.
        path.reverse()
        return path
```

This is the most vital class of the game implementation which performs the pathfinding in a game environment using A* and Best-First Search algorithms. This class has the following functions.

- __init__(self, game, start, goal, heuristic_name): This constructor of the Agent class initializes the agent with the game, start, and goal positions and the selected heuristic name.
- heuristic(self, pos): This calls a specific heuristic function based on the heuristic name provided.
- find_path_A_star(self): This function holds the Implementation of the A* search algorithm to find the optimal path from the start to the goal, considering the selected heuristic function. After completing the search, it returns the path, number of nodes explored, maximum open set size, execution time, and path length.
- find_path_best_first_search(self, random_factor=0.4): This function implements the Best-First Search algorithm with the option to introduce randomness by specifying a random factor. This function also keeps track of the visited nodes, thus allowing the algorithm to have a memory of the explored paths. The randomness and the memory ascertain that the agent does not get stuck in a loop while traversing the maze. After completing the

search, this function returns the path, number of nodes explored, maximum open set size, execution time, and path length.

- reconstruct_path(self, came_from, current): This function reconstructs the path from the goal node back to the start node. The came_from dictionary helps this function to ensure the correct order of positions in the path.

Apart from the above three classes, there is a part of the code that serves as the main function and helps to conduct a comparative analysis of the two pathfinding algorithms and heuristics on the set maze configurations. The code snippet of this section is given below.

```
# Define the maze as a 2D grid (1 represents a wall, 0 represents an open path).
mazes = [[
    [0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
    [0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1],
    [0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1],
    [1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1],
    [0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1],
    [0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1],
    [0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1],
    [0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0],
    [1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
    [1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0],
    [0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
    [0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1],
    [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1],
    [0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0]
],
[
    [0, 0, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 1, 0, 0, 1],
    [1, 1, 1, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 1, 0, 1],
    [1, 1, 1, 1, 1, 0, 0, 0],
    [1, 0, 0, 1, 0, 0, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 1, 0]
]
]

# Define the start and goal positions.
starts = [(0, 0),(0,0)]
goals = [(19, 19),(7,7)]

execution_times_euclidean_A_star = []
execution_times_manhattan_A_star = []
execution_times_chebyshev_A_star = []
```

```
execution_times_octile_A_star = []
execution_times_zero_A_star = []

node_explored_euclidean_A_star = []
node_explored_manhattan_A_star = []
node_explored_chebyshev_A_star = []
node_explored_octile_A_star = []
node_explored_zero_A_star = []

open_sets_euclidean_A_star = []
open_sets_manhattan_A_star = []
open_sets_chebyshev_A_star = []
open_sets_octile_A_star = []
open_sets_zero_A_star = []

path_lengths_euclidean_A_star = []
path_lengths_manhattan_A_star = []
path_lengths_chebyshev_A_star = []
path_lengths_octile_A_star = []
path_lengths_zero_A_star = []

execution_times_euclidean_best_first = []
execution_times_manhattan_best_first = []
execution_times_chebyshev_best_first = []
execution_times_octile_best_first = []
execution_times_zero_best_first = []

node_explored_euclidean_best_first = []
node_explored_manhattan_best_first = []
node_explored_chebyshev_best_first = []
node_explored_octile_best_first = []
node_explored_zero_best_first = []

open_sets_euclidean_best_first = []
open_sets_manhattan_best_first = []
open_sets_chebyshev_best_first = []
open_sets_octile_best_first = []
open_sets_zero_best_first = []

path_lengths_euclidean_best_first = []
path_lengths_manhattan_best_first = []
path_lengths_chebyshev_best_first = []
path_lengths_octile_best_first = []
path_lengths_zero_best_first = []

# Create a game instance.
for maze,start,goal in zip(mazes, starts, goals):
    heuristic_names = ["euclidean", "manhattan", "chebyshev", "octile", "zero"]
     algorithm_names = ["a_star", "best_first"]
```

```python
    for heuristic_name in heuristic_names:
        for algorithm_name in algorithm_names:
            game = Game(maze)
            # Create an agent and find the path.
            agent = Agent(game, start, goal, heuristic_name)
            # path, nodes_explored, max_open_set_size, execution_time, path_length = agent.find_path_A_star()
            path, nodes_explored, max_open_set_size, execution_time, path_length = agent.find_path_best_first_search()

            # Print the path and statistics.
            if heuristic_name == "euclidean" and algorithm_name == "a_star":
                print("A Star Euclidean: ")
                execution_times_euclidean_A_star.append(execution_time)
                node_explored_euclidean_A_star.append(nodes_explored)
                open_sets_euclidean_A_star.append(max_open_set_size)
                path_lengths_euclidean_A_star.append(path_length)

            elif heuristic_name == "euclidean" and algorithm_name == "best_first":
                print("Best First Euclidean: ")
                execution_times_euclidean_best_first.append(execution_time)
                node_explored_euclidean_best_first.append(nodes_explored)
                open_sets_euclidean_best_first.append(max_open_set_size)
                path_lengths_euclidean_best_first.append(path_length)

            elif heuristic_name == "manhattan" and algorithm_name == "a_star":
                print("A Star Manhattan: ")
                execution_times_manhattan_A_star.append(execution_time)
                node_explored_manhattan_A_star.append(nodes_explored)
                open_sets_manhattan_A_star.append(max_open_set_size)
                path_lengths_manhattan_A_star.append(path_length)

            elif heuristic_name == "manhattan" and algorithm_name == "best_first":
                print("Best First Manhattan: ")
                execution_times_manhattan_best_first.append(execution_time)
                node_explored_manhattan_best_first.append(nodes_explored)
                open_sets_manhattan_best_first.append(max_open_set_size)
                path_lengths_manhattan_best_first.append(path_length)

            elif heuristic_name == "chebyshev" and algorithm_name == "a_star":
                print("A Star Chebyshev: ")
                execution_times_chebyshev_A_star.append(execution_time)
                node_explored_chebyshev_A_star.append(nodes_explored)
                open_sets_chebyshev_A_star.append(max_open_set_size)
                path_lengths_chebyshev_A_star.append(path_length)

            elif heuristic_name == "chebyshev" and algorithm_name == "best_first":
                print("Best First Chebyshev: ")
                execution_times_chebyshev_best_first.append(execution_time)
                node_explored_chebyshev_best_first.append(nodes_explored)
                open_sets_chebyshev_best_first.append(max_open_set_size)
                path_lengths_chebyshev_best_first.append(path_length)

            elif heuristic_name == "octile" and algorithm_name == "a_star":
                print("A Star Octile: ")
                execution_times_octile_A_star.append(execution_time)
                node_explored_octile_A_star.append(nodes_explored)
                open_sets_octile_A_star.append(max_open_set_size)
                path_lengths_octile_A_star.append(path_length)

            elif heuristic_name == "octile" and algorithm_name == "best_first":
                print("Best First Octile: ")
                execution_times_octile_best_first.append(execution_time)
                node_explored_octile_best_first.append(nodes_explored)
                open_sets_octile_best_first.append(max_open_set_size)
                path_lengths_octile_best_first.append(path_length)


            elif heuristic_name == "zero" and algorithm_name == "a_star":
                print("A Star Zero: ")
                execution_times_zero_A_star.append(execution_time)
                node_explored_zero_A_star.append(nodes_explored)
                open_sets_zero_A_star.append(max_open_set_size)
                path_lengths_zero_A_star.append(path_length)

            elif heuristic_name == "zero" and algorithm_name == "best_first":
                print("Best First Zero: ")
                execution_times_zero_best_first.append(execution_time)
                node_explored_zero_best_first.append(nodes_explored)
                open_sets_zero_best_first.append(max_open_set_size)
                path_lengths_zero_best_first.append(path_length)


            if path:
                for pos in path:
                    print(pos)
                print(f"Execution Time: {execution_time:.10f} seconds")
                print(f"Nodes Explored: {nodes_explored}")
                print(f"Max Open Set Size: {max_open_set_size}")
                print(f"Path Length: {path_length}")
            else:
                print("No path found.")
```

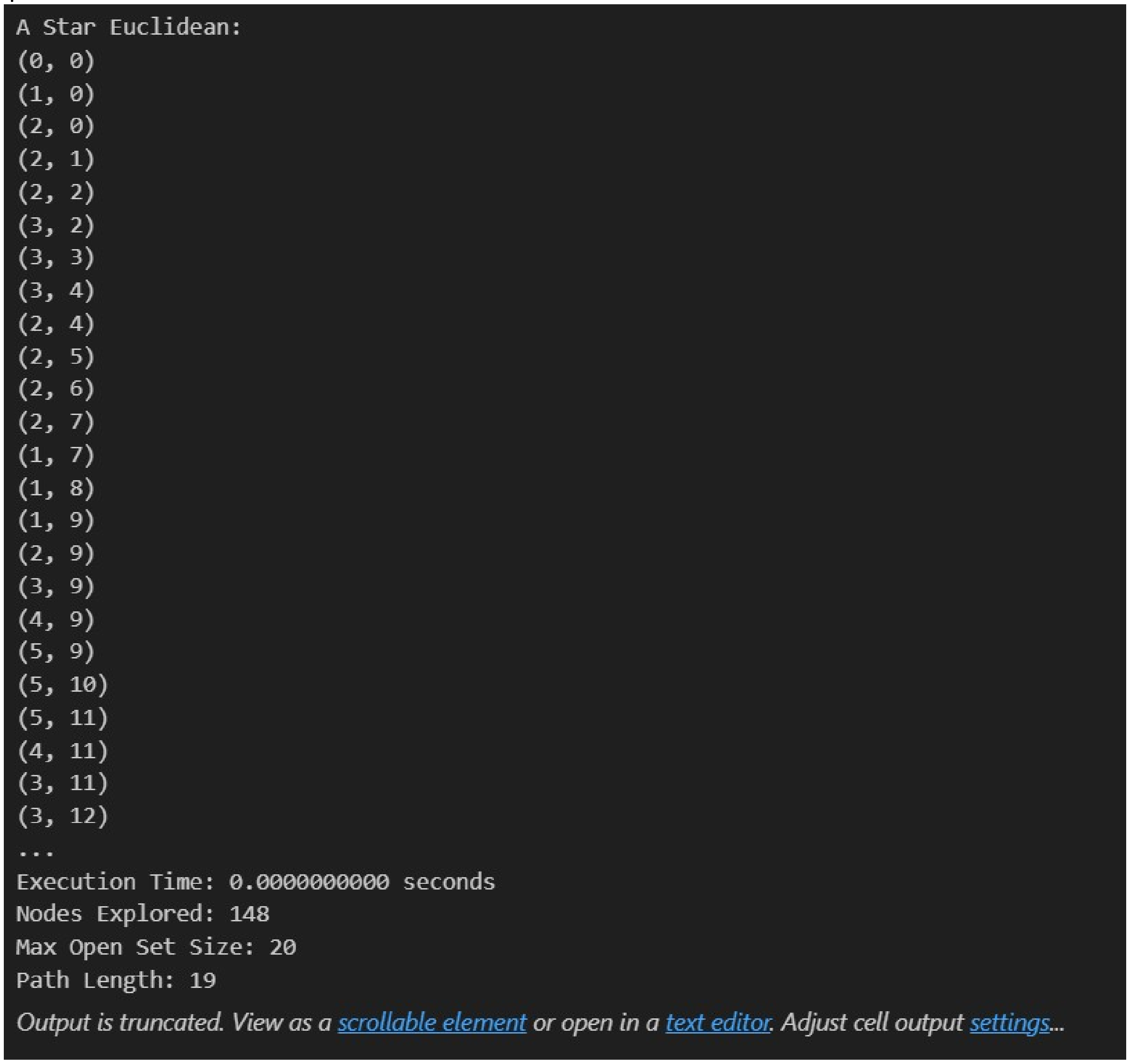
This part defines the maze layouts in the mazes list in which every 1 is an obstacle, and 0 is an open or empty path. It also defines the starting and goal positions for the earlier defined mazes. It further applies A* search and Best-First Search algorithms to find the paths utilizing the Euclidean distance, Manhattan distance, Chebyshev distance, Octile distance, and zero heuristics. This code section also runs several game instances in various combinations of maze layouts, algorithms, and heuristics. It collects the execution times, nodes explored, maximum open set sizes, and path lengths for statistical performance analysis.

Experimentation Setup for Maze Runner

To evaluate the A* and Best First Search algorithms and the various associated heuristics, two mazes were declared in the mazes list, and their start and goal positions were defined in the starts and goals lists. In a loop, various combinations of algorithms and heuristics were run for both the mazes. The associated execution time, nodes explored, maximum open set size, and path lengths data were collected for statistical evaluation of the performance of the algorithms and heuristics.

Results and Analyses of the Results of Maze Runner

Sample output of the main function:



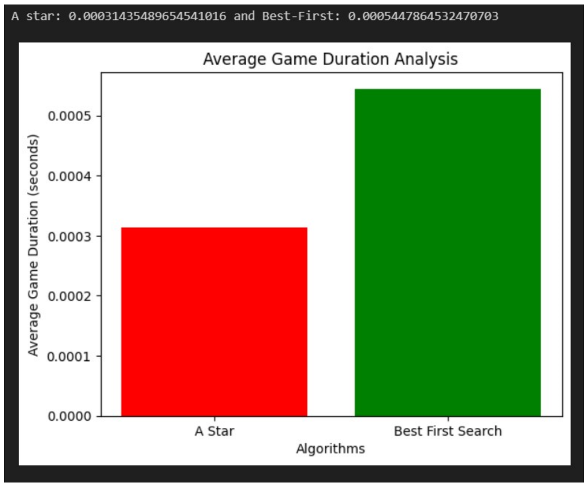
```
A Star Euclidean:
(0, 0)
(1, 0)
(2, 0)
(2, 1)
(2, 2)
(3, 2)
(3, 3)
(3, 4)
(2, 4)
(2, 5)
(2, 6)
(2, 7)
(1, 7)
(1, 8)
(1, 9)
(2, 9)
(3, 9)
(4, 9)
(5, 9)
(5, 10)
(5, 11)
(4, 11)
(3, 11)
(3, 12)
...
Execution Time: 0.0000000000 seconds
Nodes Explored: 148
Max Open Set Size: 20
Path Length: 19
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Fig. 01: Truncated output of the Maze Game

Statistical Results and Analyses:

Average Game Duration A VS Best First Search:*

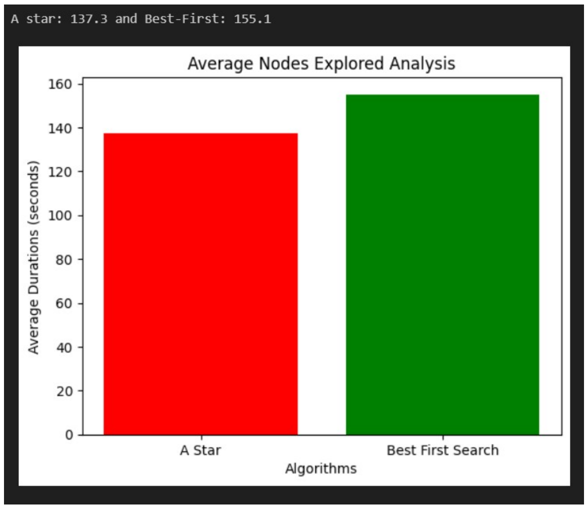


```
A star: 0.00031435489654541016 and Best-First: 0.0005447864532470703
```

Fig. 02: Comparison of A* and Best First Search in terms of average game duration

Both the A* and Best First Search with memory and randomness are time efficient in solving the input mazes. However, the prior algorithm takes less time to solve each maze. Fig. 02 shows the average time required to solve the two input mazes is 0.00314 seconds for the A*, whereas it is 0.00544 seconds for the Best First Search algorithm. However, during the experimentations, it was observed that without randomness and memory, Best First Search cannot solve the puzzle and gets stuck in a loop, thus running out of time.

Average Nodes Explored A VS Best First Search:*



```
A star: 137.3 and Best-First: 155.1
```

Fig. 03: Comparison of A* and Best First Search in terms of average nodes explored

To find the paths of both the mazes, A* needed to explore 137.3 nodes on average, whereas Best First Search was required to explore 155.1 nodes. A higher node exploration means the algorithm had to explore more potential solutions to reach the desired goal state, indicating a more extensive search effort. This directly has implications for execution time and thus conforms to this as A* had less execution time, according to Fig. 02.

Average Maximum Open Sets A VS Best First Search:*

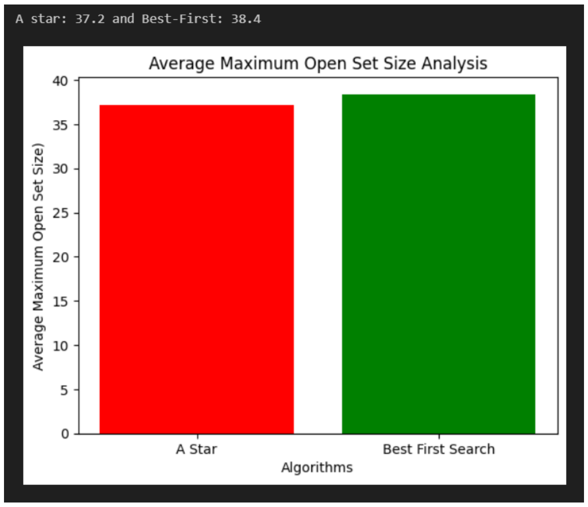

```
A star: 37.2 and Best-First: 38.4
```

Fig. 04: Comparison of A* and Best First Search in terms of maximum open sets

Maximum open sets represent the largest number of nodes that were present at once in the open set. Open set means the nodes that have been discovered but not explored yet. Fig. 04 depicts both the A* and Best First Search algorithms had almost similar maximum open set number, Best First Search had 1.2 higher value than A*. Higher maximum open sets refer to higher memory usage. However, this defers to the general case, where A* has a space complexity of O(b^m) and Best First Search has a polynomial space complexity. So theoretically, Best First Search should require less memory, however, this defer can be caused due to the introduction of randomness and memory.

Average Path Lengths A VS Best First Search:*

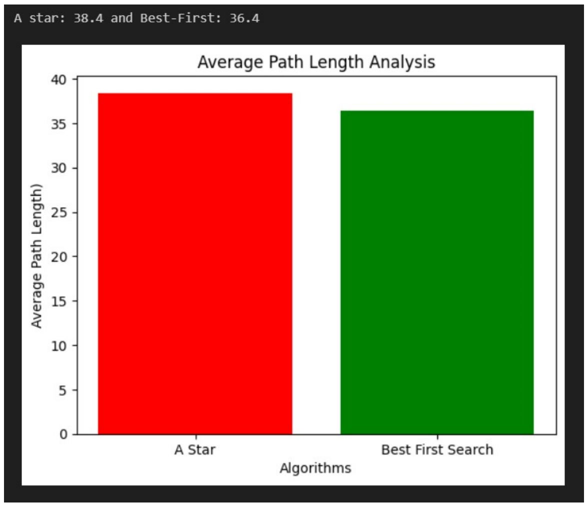

```
A star: 38.4 and Best-First: 36.4
```

Fig. 05: Comparison of A* and Best First Search in terms of path length

According to Fig. 05, A* has a slightly higher path length than Best First Search. At first glance, it might seem counterintuitive as A* is an optimal algorithm, meaning it guarantees the shortest path. However, in practice, the optimality of the algorithm does not guarantee that it will always produce shorter path lengths than other algorithms, and its ability to find the shortest path heavily depends on the use of admissible heuristics. The actual path length also depends on the problem geometry and the order in which nodes are explored.

Comparison of the Mean Nodes Explored for A and Best First Search:*

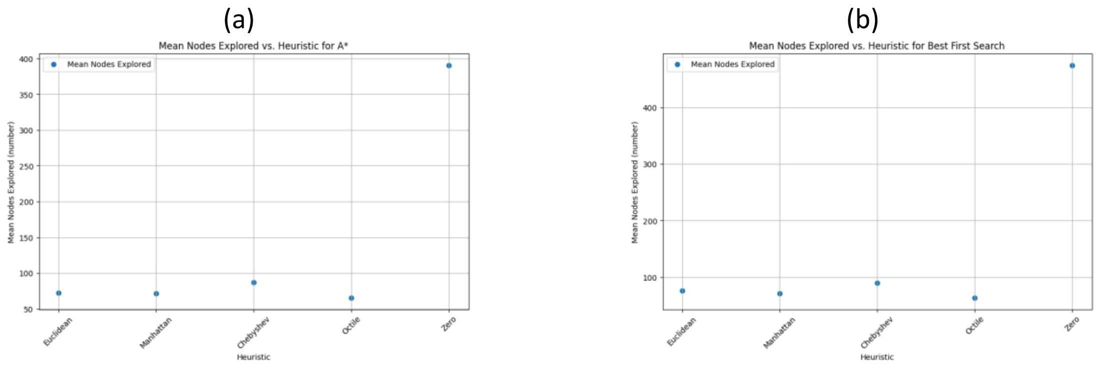


Fig. 06: (a) Comparison of mean nodes explored for A* using different heuristics and (b) Comparison of mean nodes explored for Best First Search using different heuristics

Overall, for both algorithms, the Octile distance heuristic required the least number of nodes to be explored, while the zero heuristic required the highest number of nodes to be explored. Since the number of nodes explored has a relation to time efficiency, it can also be derived that the Octile distance heuristic is the most time efficient while the zero heuristic is the least.

Comparison of the Mean Nodes Explored for A and Best First Search:*

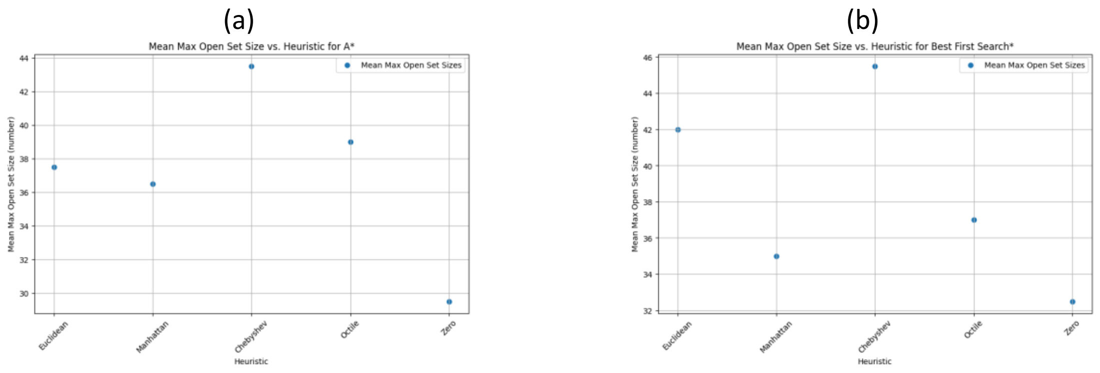


Fig. 07: (a) Comparison of Maximum Open Sets explored for A* using different heuristics and (b) Comparison of Maximum Open Sets explored for Best First Search using different heuristics

For both A* and Best First Search, the Chebyshev distance heuristic had the highest maximum open sets. In contrast, the zero heuristic had the least, suggesting the prior heuristic is the least efficient in terms of memory usage while the other is the most.

Comparison of the Mean Path Lengths for A and Best First Search:*

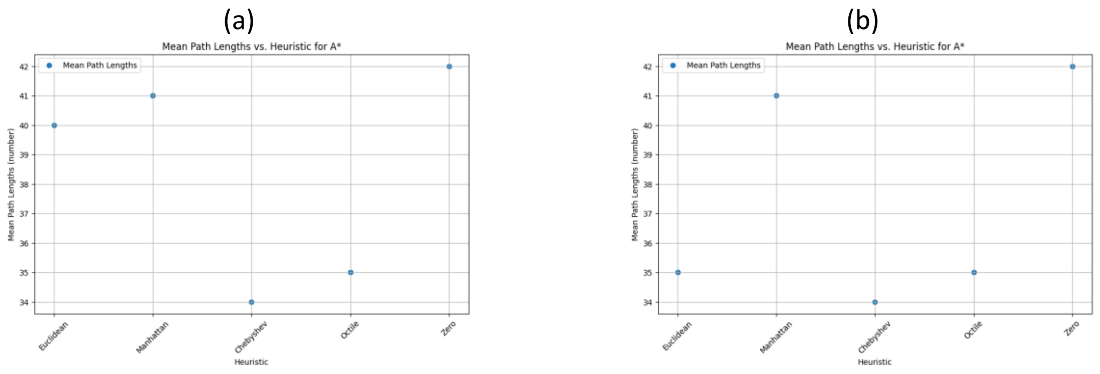


Fig. 08: (a) Comparison of Path Lengths explored for A* using different heuristics and (b) Comparison of Path Lengths explored for Best First Search using different heuristics

For both A* and Best First Search, the Chebyshev distance heuristic had the least path length, while the zero heuristic had the highest path length. Having a lower path length means higher optimality. Therefore, the Chebyshev distance heuristic gives the most optimal path while the zero heuristic gives the worst.

Checkers Game Implementation

The Checkers game is implemented using a Python3 library called imparaai-checkers and Python 3.12 version. The documentation of the Python3 library is found in the link https://pypi.org/project/imparaai-checkers/, and apart from the library classes, the code comprises the following classes for the following reasons.

Heuristics Class:

```
class Heuristics:
    @staticmethod
    def get_player_pieces(game, player):
        player_pieces = []
        for piece in game.board.pieces:
            if piece.player == player and not piece.captured:
                player_pieces.append(piece)
        return player_pieces

    @staticmethod
    def get_opponent_pieces(game, player):
        opponent_pieces = []
        for piece in game.board.pieces:
            if piece.player != player and not piece.captured:
                opponent_pieces.append(piece)
        return opponent_pieces

    #heuristics list starts here
    #increase the number of pieces of the player as much as possible, maximize the difference between the player and
opponent
    @staticmethod
    def simple_piece_count(game, player):
        player_pieces = Heuristics.get_player_pieces(game, player)
        opponent_pieces = Heuristics.get_opponent_pieces(game, player)

        return len(player_pieces) - len(opponent_pieces)

    #try to increase the number of king pieces the player has
    @staticmethod
    def king_count(game, player):
        player_pieces = Heuristics.get_player_pieces(game, player)
        return sum(1 for piece in player_pieces if piece.king)

    #try to have as much as dominance possible in the centre board
    @staticmethod
    def evaluate_board_control(game, player):
        #Higher score for central squares, Medium score for middle squares, lower score for back rows
        board_control_scores = {
            (1, 2, 3, 4, 5, 6, 7, 8): 5,
            (9, 10, 11, 12, 21, 22, 23, 24): 3,
            (13, 14, 15, 16, 17, 18, 19, 20): 1,
        }

        total_score = 0
        # Use a set to collect unique positions
        total_positions = set()

        pieces = Heuristics.get_player_pieces(game, player)

        for piece in pieces:
            for positions, score in board_control_scores.items():
                if piece.position in positions:
                    total_score += score
                    # Add the positions in the group to the set
                    total_positions.update(positions)
                    # Assign the highest available score
                    break

        return total_score
    #heuristics list ends here
```

This class contains a collection of static methods for implementing several heuristics for the Checkers game. The heuristics help to evaluate various game situations and provide a score to the agents, thus allowing them to make informed move decisions. Here are the various functions that are related to this class.

- get_player_pieces(game, player): This function returns the number of pieces that belong to the player and are not captured. This particular function is not a heuristic function but rather a function that the simple_piece_count function calls.
- get_opponent_pieces(game, player): This function returns the number of pieces that belong to the opponent and are not captured. This function is also similar to the get_player_pieces function in terms of use.
- simple_piece_count(game, player): This heuristic function aims to maximize the difference between the number of pieces of the player and the opponent. The higher the positive difference between the two numbers is, the better it is for the player.
- king_count_piece(game, player): This heuristic function encourages the player to have as many king pieces as possible. To elaborate, it tries to increase the number of pieces that reach the opposite end of the board and returns the count of the king pieces the player possesses.
- evaluate_board_control: This heuristic function tries to establish the player’s dominance in the center of the board. It assigns different scores to different positions of the game board and calculates the player’s total score based on the positions of the player pieces. Since the central positions hold higher scores, the player is encouraged to have more pieces in the center of the board.

MinimaxAgent Class:

```
import copy

class MinimaxAgent:
    def __init__(self, player):
        self.player = player

    def get_best_move(self, game, heuristic):
        _, best_move = self.minimax(game, 4, True, float('-inf'), float('inf'), heuristic)
        return best_move

    def minimax(self, game, depth, maximizing_player, alpha, beta, heuristic):
        #selection of the heuristics for position evaluation when either the game is over or at the leaf nodes
        if depth == 0 or game.is_over():
            if heuristic == "piece_count":
                return Heuristics.simple_piece_count(game, self.player), None
            elif heuristic == "king_count":
                return Heuristics.king_count(game, self.player), None
            else:
                return Heuristics.evaluate_board_control(game, self.player), None

        if maximizing_player:
            # Initialize the maximum evaluation.
            max_eval = float('-inf')
            # Initialize the best move.
            best_move = None

            # Loop through possible moves in the game.
            for move in game.get_possible_moves():
                # Create a copy of the game for simulation.
                next_game = copy.deepcopy(game)
                # Apply the current move to the copy of the game.
                next_game.move(move)
                eval, _ = self.minimax(next_game, depth - 1, False, alpha, beta, heuristic)

                # If the evaluation is better than the current maximum: update the maximum evaluationand best move
                if eval > max_eval:
                    max_eval = eval
                    best_move = move

                # Update the alpha value.
                alpha = max(alpha, max_eval)
                # Prune the search if the beta value is less than or equal to alpha.
                if beta <= alpha:
                    break

            return max_eval, best_move
        else:
            # Initialization of the minimizing player
            min_eval = float('inf')
            best_move = None

            # Loop through possible moves in the game.
            for move in game.get_possible_moves():
                # Create a copy of the game for simulation.
                next_game = copy.deepcopy(game)
                next_game.move(move)
                eval, _ = self.minimax(next_game, depth - 1, True, alpha, beta, heuristic)

                # If the evaluation is worse than the current minimum: update the minimum evaluation and best move
                if eval < min_eval:
                    min_eval = eval
                    best_move = move

                # Update the alpha value.
                beta = min(beta, min_eval)
                # Prune the search if the beta value is less than or equal to alpha.
                if beta <= alpha:
                    break

            return min_eval, best_move
```

This class implements an agent that uses minimax algorithm to make decisions in the two player Checkers game. The algorithm determines the best move for the player by considering both the player's action and opponent's responses. The functions that are related to this class are mentioned below.

- __init__(self, player): This constructor initializes the class with the player it represents.
- get_best_move(self, game, heuristic): This function tries to determine the best move for a given player in the current game state. It takes the game and a heuristic as input and initializes the minimax search algorithm with a specified depth and finally returns the best move when it finds it.
- minimax (self, game, depth, maximizing_player, alpha, beta, heuristic): This function is the core of the minimax algorithm implementation It utilizes minimax algorithm to evaluate the possible moves and their consequences. This function takes the present game state, depth of the search, a boolean flag to indicate whether the player is a maximizing player and alpha-beta pruning values. While traversing the game tree, when the agent reaches a terminal state, that is, if the depth is 0 (leaf nodes) or the game is over, the function calculates a heuristic value for the present game state, where the heuristic selection depends on the selected heuristic type. If the player is a maximizing player, it initializes the maximum evaluation as negative infinity and iterates through possible moves, updating the maximum evaluation and best move as it explores. It also performs alpha-beta pruning to eliminate branches of the game tree known to be suboptimal. If the beta value becomes less than or equal to alpha, it breaks out of the loop. If the player is a minimizing player, it initializes the minimum evaluation as positive infinity and follows a similar process to select the best move for the opponent. It also performs alpha-beta pruning. The method returns the maximum or minimum evaluation based on whether the player is a maximizing or minimizing player and the corresponding best move.

Apart from these classes the code involves a script section where it performs a series of game simulations to evaluate and compare the three heuristics, namely, piece count, king count and board control.

```
import random
import time

def get_duration(start_time):
    # Record the end time
    end_time = time.time()

    # Calculate and print the execution time
    execution_time = end_time - start_time
    return execution_time

piece_count_wins = []
```

```python
piece_count_losses = []
piece_count_draws = []
piece_count_durations = []
piece_count_agent_moves = []

king_count_wins = []
king_count_losses = []
king_count_draws = []
king_count_durations = []
king_count_agent_moves = []


board_control_wins = []
board_control_losses = []
board_control_draws = []
board_control_durations = []
board_control_agent_moves = []

#perform 10 game iterations where in each iteration the agent uses one of the heuristics
#Therefore in each iterations 3 simulations of the game takes place, one for each heuristics
for i in range(10):
    heuristics_arr = ['piece_count', 'king_count', 'board_control']

    for heuristic in heuristics_arr:

        print(f"Comparing {heuristic} for maximizing agents for iteration-{i}")
        start_time = time.time()

        game = Game()
        # Set the current player (1 or 2)
        current_player = game.whose_turn()

        game.consecutive_noncapture_move_limit = 20

        agent = MinimaxAgent(current_player)
        opponent_moves_count = 0
        agent_moves_count = 0

        while not game.is_over():
            if current_player == agent.player:
                best_move = agent.get_best_move(game, heuristic)
                game.move(best_move)
                agent_moves_count += 1
                print(f"Your agent's move: {best_move}")
            else:
                #the opponent plays randomly for baseline comparison
                opponent_moves = game.get_possible_moves()
                opponent_move = random.choice(opponent_moves)
                game.move(opponent_move)
                opponent_moves_count += 1
                print(f"Opponent's move: {opponent_move}")

            # Switch the current player for the next turn
            if not game.move_limit_reached():
                current_player = game.whose_turn()
            else:
                break

        print(f"Total moves by your agent: {agent_moves_count}")
        print(f"Total moves by the opponent: {opponent_moves_count}")

        winner = game.get_winner()

        if winner == 1:
            print("The agent won!")
            if heuristic == "piece_count":
                piece_count_wins.append(1)
                piece_count_losses.append(0)
                piece_count_draws.append(0)
                piece_count_durations.append(get_duration(start_time))
                piece_count_agent_moves.append(agent_moves_count)

            elif heuristic == "king_count":
                king_count_wins.append(1)
                king_count_losses.append(0)
                king_count_draws.append(0)
                king_count_durations.append(get_duration(start_time))
                king_count_agent_moves.append(agent_moves_count)

            else:
                board_control_wins.append(1)
                board_control_losses.append(0)
                board_control_draws.append(0)
                board_control_durations.append(get_duration(start_time))
                board_control_agent_moves.append(agent_moves_count)

        elif winner == 2:
            print("The opponent won!")
            if heuristic == "piece_count":
                piece_count_wins.append(0)
                piece_count_losses.append(1)
                piece_count_draws.append(0)
                piece_count_durations
                piece_count_durations.append(get_duration(start_time))
                piece_count_agent_moves.append(agent_moves_count)


            elif heuristic == "king_count":
```

```python
                king_count_wins.append(0)
                king_count_losses.append(1)
                king_count_draws.append(0)
                king_count_durations.append(get_duration(start_time))
                king_count_agent_moves.append(agent_moves_count)

            else:
                board_control_wins.append(0)
                board_control_losses.append(1)
                board_control_draws.append(0)
                board_control_durations.append(get_duration(start_time))
                board_control_agent_moves.append(agent_moves_count)

        else:
            print("It's a draw!")
            if heuristic == "piece_count":
                piece_count_wins.append(0)
                piece_count_losses.append(0)
                piece_count_draws.append(1)
                piece_count_durations.append(get_duration(start_time))
                piece_count_agent_moves.append(agent_moves_count)

            elif heuristic == "king_count":
                king_count_wins.append(0)
                king_count_losses.append(0)
                king_count_draws.append(1)
                king_count_durations.append(get_duration(start_time))
                king_count_agent_moves.append(agent_moves_count)

            else:
                board_control_wins.append(0)
                board_control_losses.append(0)
                board_control_draws.append(1)
                board_control_durations.append(get_duration(start_time))
                board_control_agent_moves.append(agent_moves_count)

print(f"piece_count_wins {piece_count_wins}")
print(f"piece_count_losses {piece_count_losses}")
print(f"piece_count_draws {piece_count_draws}")
print(f"piece_count_durations {piece_count_durations}")
print(f"piece_count_moves {piece_count_agent_moves}")


print(f"king_count_wins {king_count_wins}")
print(f"king_count_losses {king_count_losses}")
print(f"king_count_draws {king_count_draws}")
print(f"king_count_durations {king_count_durations}")
print(f"king_count_moves {king_count_agent_moves}")
print(f"board_control_wins {board_control_wins}")
print(f"board_control_losses {board_control_losses}")
print(f"board_control_draws {board_control_draws}")
print(f"board_control_durations {board_control_durations}")
print(f"board_control_moves {board_control_agent_moves}")
print(f"king_count_moves {king_count_agent_moves}")

print(f"board_control_wins {board_control_wins}")
print(f"board_control_losses {board_control_losses}")
print(f"board_control_draws {board_control_draws}")
print(f"board_control_durations {board_control_durations}")
print(f"board_control_moves {board_control_agent_moves}")
```

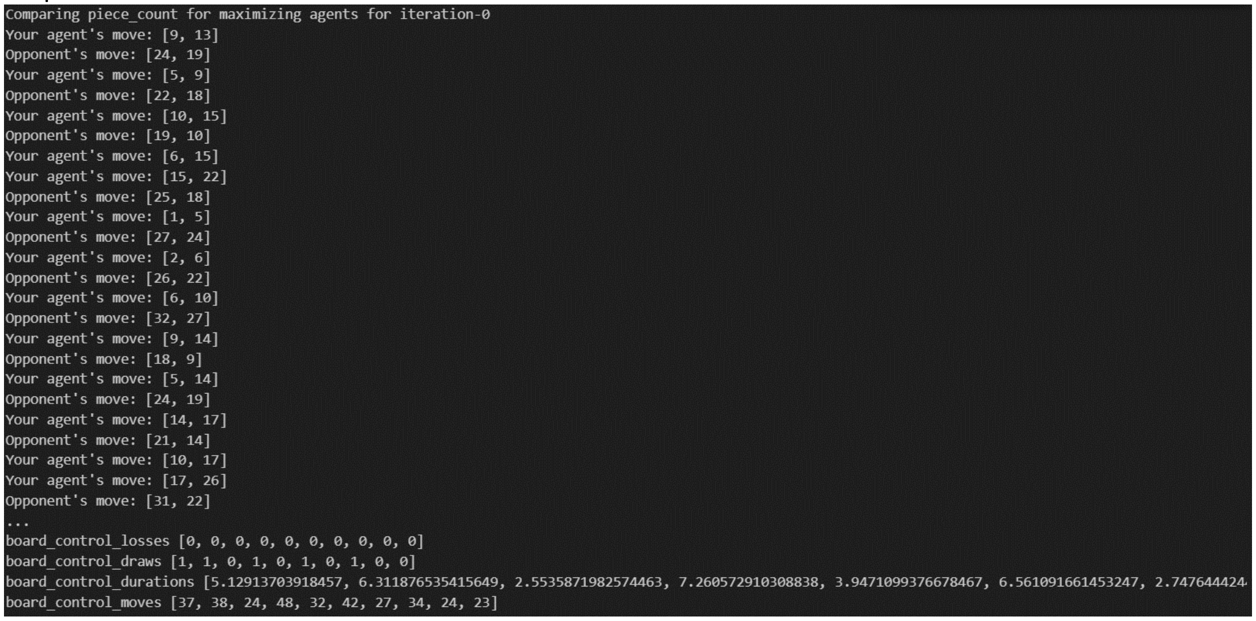
The code runs 10 iterations, and in each iteration, the game outcome is analyzed with the agent using each of the three heuristics. For each simulation, the game outcome, along with move count and game duration, are recorded and stored in separate lists for further statistical analysis.

Experimentation Setup of the Checkers game

To evaluate the Minimax algorithm's performance with various associated heuristics, game simulations were made in 10 iterations where the agent used each of the three heuristics. Therefore, each iteration had three game simulations, and the ten iterations had 30 simulations. The simulation results were stored in lists and later analyzed for statistical analyses.

Results and Analyses of the Results of the Checkers game

Sample output of the main function:



```
Comparing piece_count for maximizing agents for iteration-0
Your agent's move: [9, 13]
Opponent's move: [24, 19]
Your agent's move: [5, 9]
Opponent's move: [22, 18]
Your agent's move: [10, 15]
Opponent's move: [19, 10]
Your agent's move: [6, 15]
Your agent's move: [15, 22]
Opponent's move: [25, 18]
Your agent's move: [1, 5]
Opponent's move: [27, 24]
Your agent's move: [2, 6]
Opponent's move: [26, 22]
Your agent's move: [6, 10]
Opponent's move: [32, 27]
Your agent's move: [9, 14]
Opponent's move: [18, 9]
Your agent's move: [5, 14]
Opponent's move: [24, 19]
Your agent's move: [14, 17]
Opponent's move: [21, 14]
Your agent's move: [10, 17]
Your agent's move: [17, 26]
Opponent's move: [31, 22]
...
board_control_losses [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
board_control_draws [1, 1, 0, 1, 0, 1, 0, 1, 0, 0]
board_control_durations [5.12913703918457, 6.311876535415649, 2.5535871982574463, 7.260572910308838, 3.9471099376678467, 6.561091661453247, 2.747644424
board_control_moves [37, 38, 24, 48, 32, 42, 27, 34, 24, 23]
```

Fig. 09: Truncated output of the Checkers game implementation

Statistical Results and Analyses:

Game Outcome Analyses of Piece Count VS King Count VS Board Control Heuristics:

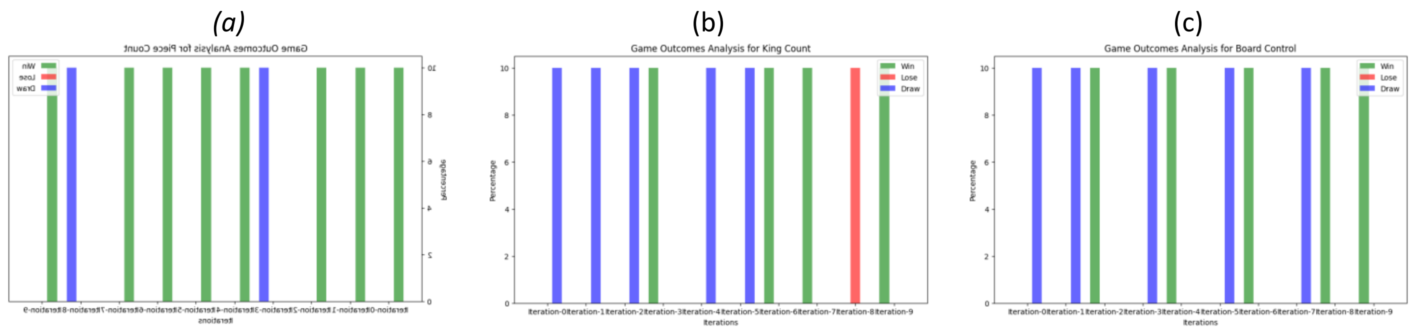


Fig. 10: Game outcome comparison for (a) piece count, (b) king count and (c) board control heuristics

In Fig. 10, each bar represents a 10% winning, losing, or drawing rate depending on the bar colors: green, red, and blue, respectively. Fig. 10 (a) shows that among the ten iterations using piece count as a heuristic for the minimax agent, the agent won eight games and drew two. Therefore, the winning percentage for this heuristic was 80%, the highest among the three heuristics. Both Fig. 10 (b) and (c), representing the game outcomes for king count and board control heuristics, respectively, had 50% draws. However, the winning rate for board control is 10% more than the king count, which had lost a game out of the ten iterations and thus had the lowest winning rate.

Average game durations comparison Piece Count VS King Count VS Board Control:

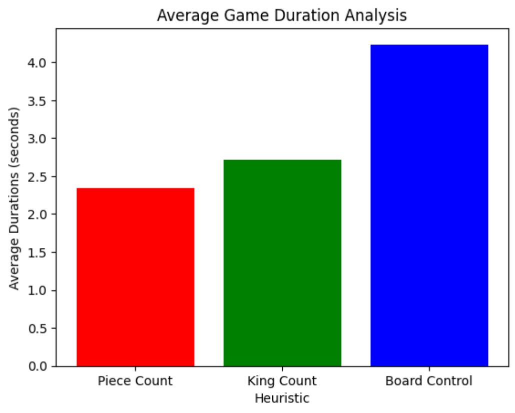


Fig. 11: Comparison of the average game durations of the games for piece count, king count and board control heuristics

Fig. 11 depicts the average time required to complete the ten iterations of games for each of the heuristics, namely, piece count (red), king count (green), and board control (blue). Among the three, the piece count heuristic takes, on average, less than 2.5 seconds to complete the iterations, which is the lowest among the three heuristics. On the other hand, board control takes the highest time (more than 4.0 seconds) to complete the iterations. King count takes more than 2.5 seconds for the game iterations. Therefore, the piece count heuristic is the most time-efficient, while the board control heuristic is the least.

Average agent moves comparison Piece Count VS King Count VS Board Control:

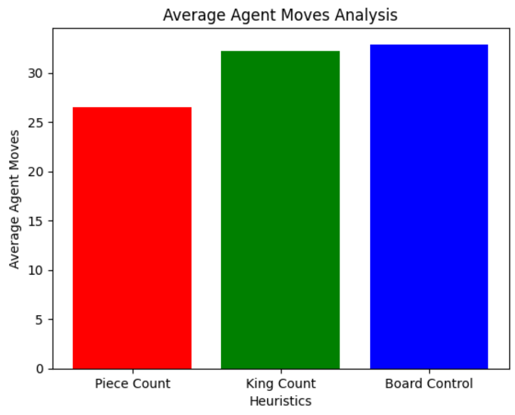


Fig. 12: Comparison of the average agent moves of the games for piece count, king count and board control heuristics

Fig. 12 shows that among piece count, king count, and board control heuristics, piece count takes the least number of moves to complete the games (approximately 26). However, the other two heuristics take almost similar agent moves, which are around 34 and 35, respectively. In games similar to Checkers, a higher number of agent moves signifies how proactive the agent is, that is, if the agent is seeking opportunities to advance, capture opponent pieces, and control the board. However, whether a higher number of moves is advantageous or not depends on the specific game, strategy, and the quality of the moves made by the agent, the evaluation of which can be doubted as when the agent is using piece count heuristic, it is least active but most effective in terms of winning percentage and time efficiency. However, despite comparable activeness, king count is superior to board control in terms of time efficiency but needs to improve in winning percentage.

Conclusion

Overall, the A* algorithm is more effective in solving mazes than the Best-First Search algorithm. It requires less execution time than the other algorithm and thus more time efficient. Furthermore, the algorithm is complete and optimal, unlike the Best First Search algorithm. The effect of optimality and completeness has been reflected during our experimentations as Best First Search alone could not solve the maze and required memory and randomness to get out of loops. There remains a direct proportional relationship between the number of nodes explored and execution times, and agreeing to this relationship, the A* algorithm is superior in terms of time complexity. A similar proportional relationship exists between maximum open set size and memory or space usage. In general, A* uses less memory compared to Best First Search algorithms. However, the experimentation results depict A* having fewer maximum open sets, thus memory usage, compared to the implemented Best First Search algorithms. This deferring from the general could be caused by adding additional memory of visited nodes and randomness in the Best First Search algorithm. The path lengths of A* are slightly longer than the Best First Search. In blatant eyes, it may seem counterintuitive to the optimality of the A* algorithm. However, this optimality does not necessarily mean producing a shorter path length than other algorithms. In terms of heuristics, Octile distance is the most time efficient and requires the least nodes to explore, while zero heuristics is the least efficient and requires the most significant number of nodes to explore to complete the mazes. Zero heuristic also performed the worst in terms of space complexity and had the highest maximum open set size. However, Chebyshev had the lowest maximum open set size and required the most minor memory usage. Chebyshev also had the least path length, while zero heuristic again performed worst in this regard. In conclusion, apart from the zero heuristic, the other four heuristics performed competitively.

For the Checkers game, the minimax algorithm with alpha-beta pruning and Piece Count heuristic had the highest winning percentage, while the king count heuristic had the lowest winning percentage. The Piece Count heuristic is also the most time-efficient as it has the lowest average game duration. However, the Board Control heuristic had the highest average game duration and, thus, the least time efficient. Regarding agent moves, the Piece Count heuristic was the lowest, depicting that using this heuristic, the agent is the least proactive. However, the other two heuristics display similar levels of activeness, but Board Control was slightly more active than King Count. Evaluating all the performance metrics for the Checkers game, it is safe to say that among the three heuristics, Piece Count gives the best result.