Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

SCHOOL OF COMPUTER SCIENCE & STATISTICS

CS7IS2-202324 ARTIFICIAL INTELLIGENCE

REPORT - ASSIGNMENT 1

PRIYA NAWAL

23330557

MSc. COMPUTER SCIENCE – INTELLIGENT SYSTEMS

Introduction

Based on my comprehension of the task, I have investigated and evaluated the effectiveness of different pathfinding and decision-making algorithms in artificial maze settings. By putting a set of search algorithms—Breadth-First Search (BFS), Depth-First Search (DFS), and A* into practice and contrasting them with two Markov Decision Processes (MDPs) algorithms—Policy Iteration and Value Iteration, my goal is to determine their relative merits and restrictions. A thorough grasp of the algorithms' performance measures, including execution time, memory usage, and path optimality, is provided by this examination, which also emphasises the algorithms' theoretical foundations and practical applications in real-world settings.

Algorithm Descriptions, Implementation and Output Analysis

Maze Generation

- The 'Pyamaze' library, which provides the ability to select the number of rows and columns, is utilised in the maze generator function to create intricately laid out mazes. A goal position is used to start the labyrinth, and depending on the loop percentage parameter, the algorithm may include loops and dead ends, making each maze different and difficult.

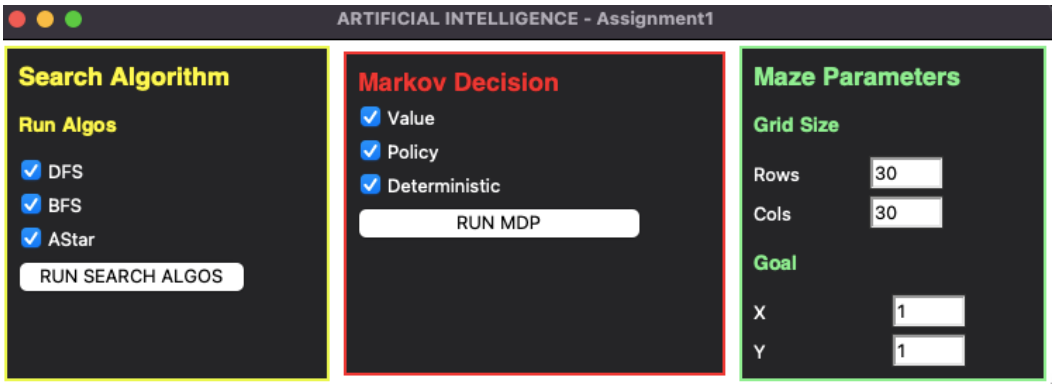

Figure 1: GUI for User Input

- The GUI for the user input is set at default with Grid Size as (30,30).
- The Goal (1,1) states the end point for the maze.
- The Search Algorithms include BFS, DFS, and AStar algorithms and all are set at default to run when clicked on Run Search Algos button.
- The Markov Decision include Value iteration and Policy iteration are set to default when clicked on Run MDP button with Deterministic value set to True.

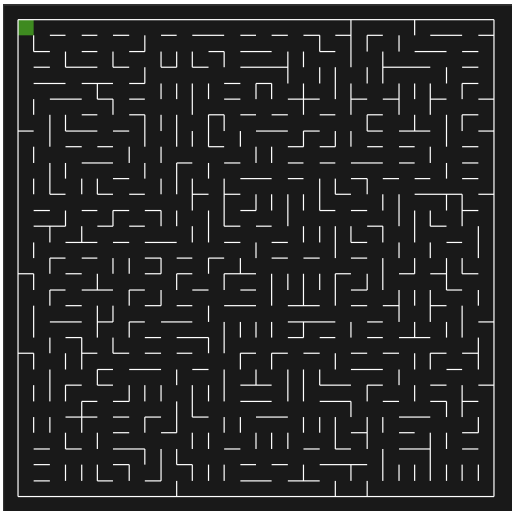

Figure 2: Maze Generated on a grid of 30x30 with goal value as 1x1

Search Algorithms:

- **Breadth-First Search (BFS):**
    - BFS starts at the entry point and works its way through the maze, level by level. The shortest path is determined in an unweighted graph by using a queue to keep track of all potential paths to explore next. Broadening the most superficially unexpanded node is the fundamental idea.
    - **Formula:** BFS does not follow a specific mathematical formula but implements the principle of visiting adjacent unvisited nodes and marking them for future exploration. It follows the First In First Out (FIFO) strategy.
    - **Pseudo code:**

```
BFS(maze, goal):
  Initialize explored = [start], frontier = [start]
  While frontier is not empty:
        currCell = frontier.pop() // For BFS, frontier is treated as a queue
        If currCell == goal, break
        For each direction in 'ESNW':
              If path exists in that direction:
                    child = move(currCell, direction)
                    If child not in explored:
                          Add child to explored
                          Append child to frontier
                          Record path: algoPath[child] = currCell
  Trace back from goal to start using algoPath to find the solution path
```

    - The output maze for BFS displays the path generated in blue. It shows the Final Path taken (59), Searched Path (899), and the Time taken (0.036) on the maze grid of 30x30.

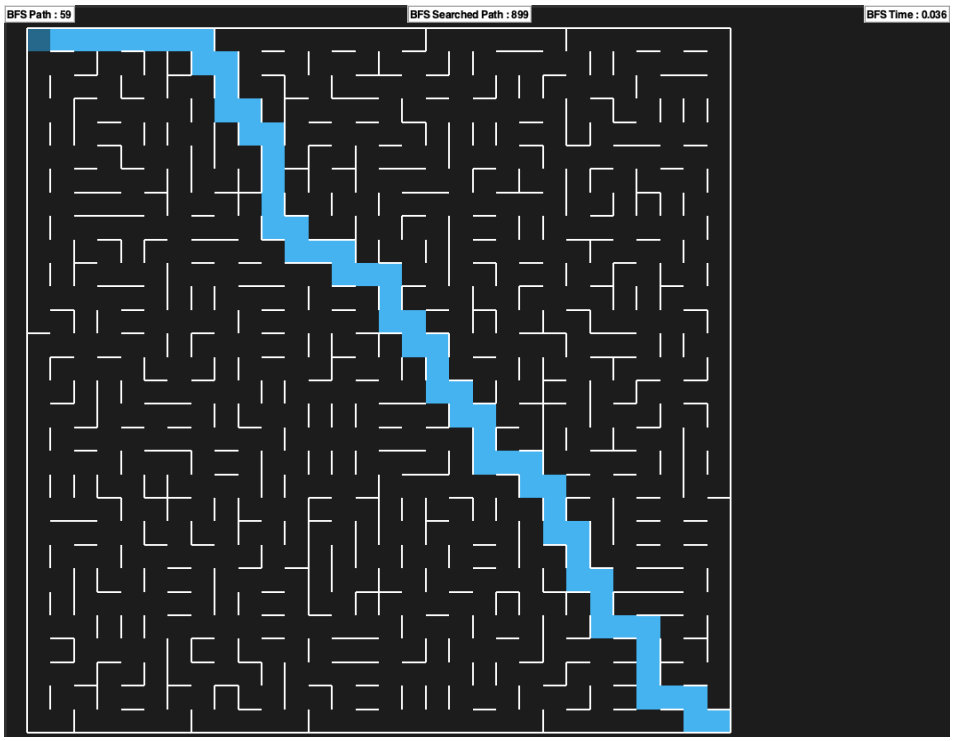

Figure 3: BFS implementation on (30,30)

- **Depth-First Search (DFS):**
    - DFS plunges far into the paths of the maze, discovering as much as it can before turning around. Although it is economical in using memory, it cannot ensure that it will find the

shortest path because it employs a stack (or recursion) to maintain track of the path that is presently being examined.

- **Formula:** DFS exploration can be represented as recursively exploring each branch before backtracking, $DFS(v) = v + DFS(children(v))$ where v is a vertex and $children(v)$ are the unvisited adjacent vertices. It follows the Last In First Out (LIFO) strategy.
- **Pseudo code:**

```
DFS(maze, goal):
    Initialize explored, frontier with start node
    Initialize algoPath
    While frontier is not empty:
        currCell = Pop last cell from frontier
        If currCell is goal:
            break
        For each direction in NODES:
            If path exists in direction from currCell:
                child = Move in direction from currCell
                If child not in explored:
                    Add child to explored and frontier
                    algoPath[child] = currCell
    Return Path from algoPath
```

- The output maze for DFS displays the path generated in green. It shows the Final Path taken (73), Searched Path (151), and the Time taken (0.0331) on the maze grid of 30x30.

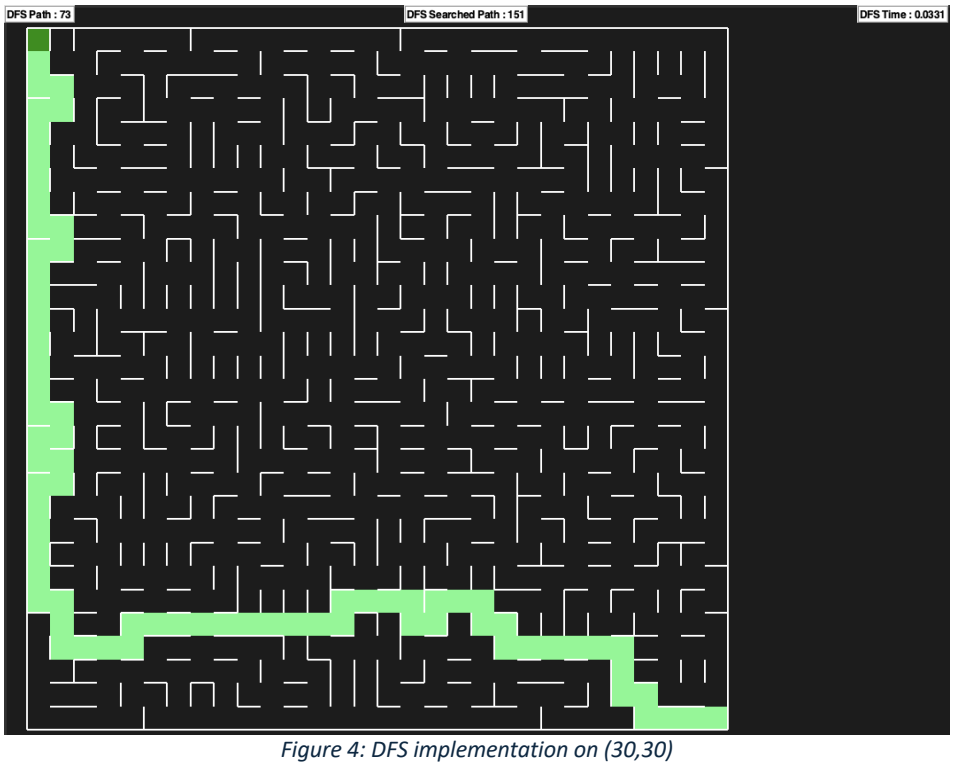

Figure 4: DFS implementation on (30,30)

- **A* Search:**
  - To determine the most efficient route, A* Search makes use of both the real cost $g(n)$ from the start node to the current node and a heuristic estimate $h(n)$ of the cost from the current node to the objective. By combining the expected cost to attain the goal and the cost to reach the node, it ranks the nodes with the lowest $f(n) = g(n) + h(n)$.
  - **Formula:** $f(n) = g(n) + h(n)$, where $g(n)$ is the cost from the start node to n, and $h(n)$ is the heuristic estimate of the cost from n to the goal.

- **Pseudo code:**

```
A*(maze,goal):
    Initialize g_score[start] = 0, f_score[start] = heuristic(start)
    Add start to open set with f_score[start]
        while open set is not empty:
        current = node in open set with the lowest f_score
        if current is goal:
                return reconstruct_path(cameFrom, current)
        remove current from open set
        for each neighbor of current:
                temp_g_score = g_score[current] + dist_between(current, neighbor)
        if temp_g_score < g_score[neighbor]:
                cameFrom[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = g_score[neighbor] + heuristic(neighbor)
                if neighbor not in open set:
                        add neighbor to open set
    return failure
```

- The output maze for A* displays the path generated in red. It shows the Final Path taken (59), Searched Path (797), and the Time taken (0.03) on the maze grid of 30x30.

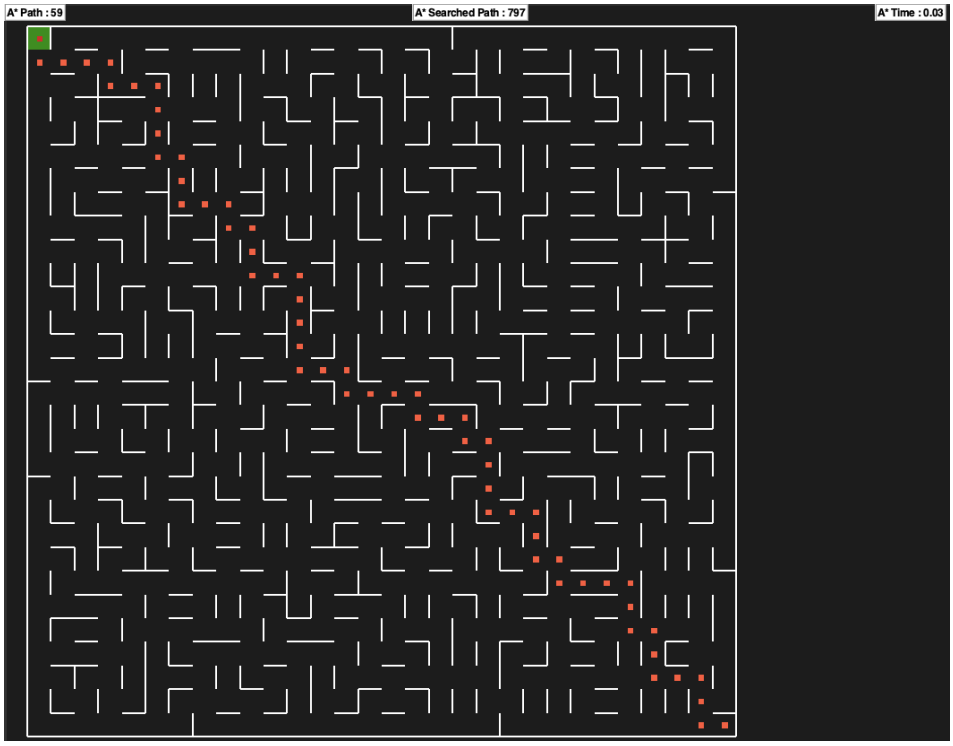

Figure 5: A implementation on (30,30)*

Heuristic Comparison

A heuristic evaluation has been carried out in order to use the best possible heuristic for the A* search algorithm. The performance of Manhattan distance was compared with Euclidean distance using two parameters that are search path length and forward path length. Search Path is a list of all cells that the algorithm has explored to reach the final goal. The forward path is a final list of all cells that an agent can use to travel from start cell to goal cell. The choice of heuristic significantly impacts A*'s performance. A well-chosen heuristic that underestimates the distance to the goal can lead to faster and more optimal pathfinding.

A wide range of loop percentages has been used to evaluate the heuristics for a 30 x 30 sized maze. LoopPercent, when set to highest value 100, means the maze generation algorithm will maximize the number of multiple paths to the goal. A value of 0 means that only 1 path to the goal is available.

Markov Decision Process:

- **Value Iteration:**
    - In order to reach the ideal values, this strategy iteratively adjusts the value of each state to represent the maximum predicted benefit of acting from that state. The Bellman optimality equation forms its foundation.
    - **Formula:** $V(s) = maxa\ \Sigma P(s'|s,a)[\big(R(s,a) + \gamma V(s')\big)]$, where $V(s)$ is the value of state s, $P(s'|s,a)$ is the transition probability, $R(s,a)$ is the reward, and γ is the discount factor.
    - **Pseudo code:**

```
Initialize values of all states to 0
Repeat until convergence:
  delta = 0
  For each state in the maze:
    if state is the goal:
      continue
    utilityMax = -infinity
    For each action from the current state:
      Sum = 0
      For each possible next state:
        Sum += Probability(next state | current state, action) *
             (Reward(current state, action, next state) +
              Discount Factor * Value(next state))
      if Sum > utilityMax:
        utilityMax = Sum
    delta = max(delta, |utilityMax - Value(current state)|)
    Value(current state) = utilityMax
  if delta < a small threshold:
    break
```

    - The output maze for value iteration displays the path generated in red with Deterministic value set as True and False both. It shows the Final Path taken (59) for both the options but the time taken for Deterministic is 0.7429 whereas, for Non-Deterministic is 0.3459 on the maze grid of 30x30.

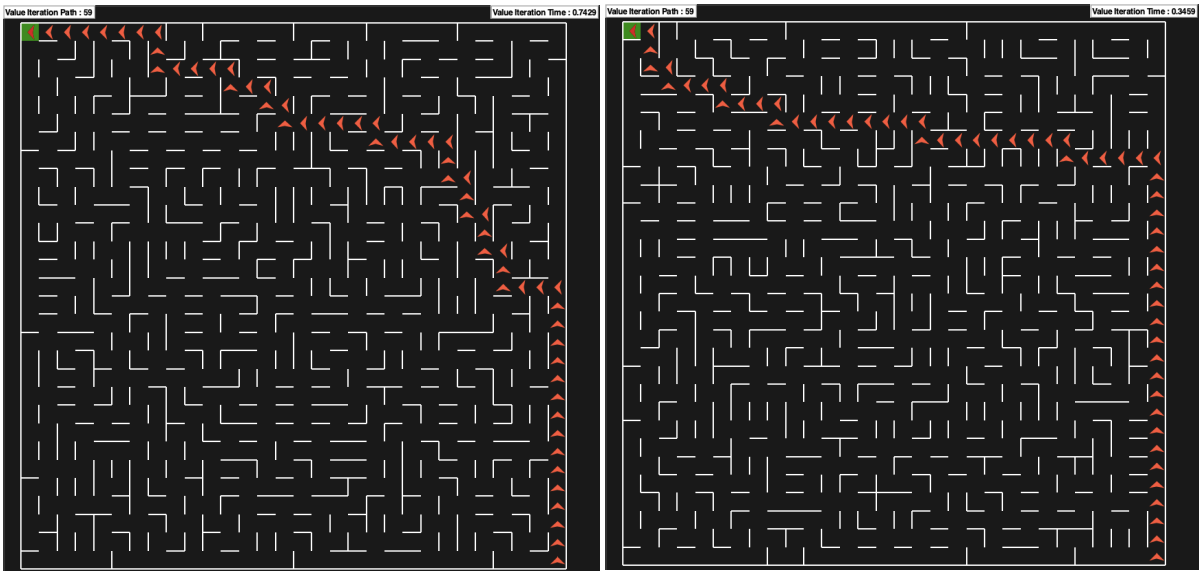

Figure 6: Value iteration on (30,30) with Deterministic & Non-Deterministic values

- **Policy Iteration:**
    - The two stages of policy iteration are policy evaluation, in which the utility of adhering to the existing policy is determined, and policy improvement, in which the policy is modified to take into account the activities that result in the maximum utility.
    - Policy Evaluation Formula: $U(s) = R(s) + \gamma \Sigma P(s'|s, \Pi(s))U(s')$, where $U(s)$ is the utility of state s, $\Pi(s)$ is the policy, and $R(s)$ is the immediate reward.
    - Policy Improvement: Updates the policy by choosing the action that maximizes the expected utility for each state.
    - **Pseudo code:**

```
Initialize policy randomly
Repeat:
  Policy Evaluation:
    Repeat until convergence:
      For each state s in States:
        utility[s] = reward[s] + γ * Σ P(s'|s, policy[s]) * utility[s']
  Policy Improvement:
    policy_stable = True
    For each state s in States:
      chosen_action = policy[s]
      policy[s] = argmax_a Σ P(s'|s,a) * (reward[s,a,s'] + γ * utility[s'])
      If chosen_action != policy[s]:
        policy_stable = False
If policy_stable, exit loop
```

    - The output maze for policy iteration displays the path generated in blue with Deterministic value set as True and False both. It shows the Final Path taken (59) for both the options but the time taken for Deterministic is 0.0319 whereas, for Non-Deterministic is 0.0312 on the maze grid of 30x30.

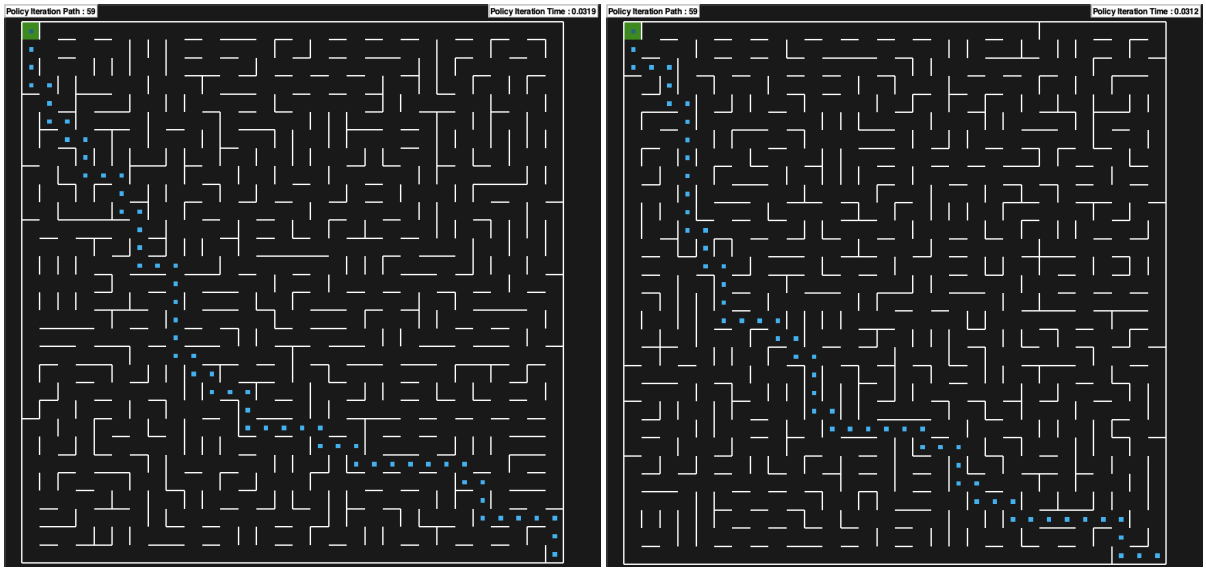

Figure 7: Policy iteration on (30,30) with Deterministic & Non-Deterministic values

Performance Comparison & Evaluation of Results

An evaluation between all the Search and MDP algorithms has been carried out across a range of three metrics. The metrics used to evaluate the performance of each algorithm are:

1. **Final Path:** It refers to the number of cells from start to end. This includes the start and goal cell. It is an effective measure of how efficient an algorithm is in finding the path to a goal.
2. **Searched Path:** It states the number of nodes taken to find the most optimal path.
3. **Time taken:** To execute a function time taken is calculated using the timeit module in python. It is an effective measure of how quickly a function executes within the given constraints.
4. **Memory Usage:** Another metric that can be used is to compare the memory used by the two algorithms, if looppercent has been set to 0, which means that only one path exists between start node and end node. This allows for an effective evaluation of the memory used in reaching the same path. Memory Usage is calculated using the tracemalloc module in python.

Maze size ranges between 10 x 10 and 90 x 90. Larger sizes have not been taken due to long run times. Mazes smaller than 10 x 10 seemed trivial and hence not used for evaluation. A 20 x 20 maze is used to visually compare the path taken by two algorithms.

Please note that mazes generated are random so if the code is run again, the maze generated will be different.

Comparing Search Algorithms (BFS vs DFS vs A*)

- **BFS vs DFS vs A***
    - The green path displays the results for DFS algorithm, blue for BFS algorithm, and red for A* algorithm.
    - The results display that the final path taken by DFS (green squares) is the largest and also takes more time.
    - BFS and A* algorithms takes the same final path but time taken by A* is slightly less as compared to BFS.
    - Hence, A* algorithm is the most efficient in terms of path optimality and the number of searched nodes, followed by BFS, with DFS being the least efficient due to its exhaustive nature and non-optimality in pathfinding.

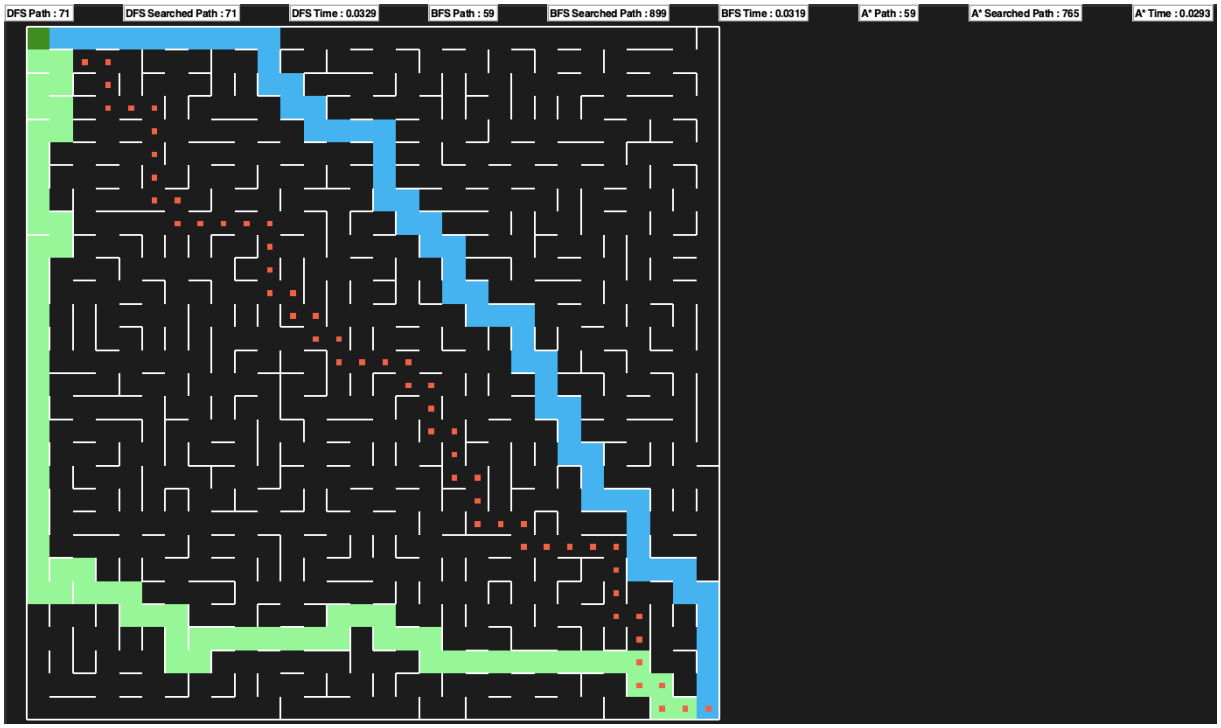

Figure 8: Search Algorithms running together on the maze grid of 30x30

- **DFS vs BFS**
    - DFS and BFS are two uniformed search algorithms that differ in the way they reach the goal. DFS may not give us the shortest path to the goal, whereas BFS guarantees the shortest path (given that the cost to the path is same for all edges).
    - The output displays that the path taken by BFS is shorter when compared with DFS and the time taken by DFS is slightly more as compared to BFS.
    - The increasing time complexity of DFS with the maze size makes it less suitable for mazes of larger sizes. On the other hand, BFS algorithm is not affected by the size of the maze.

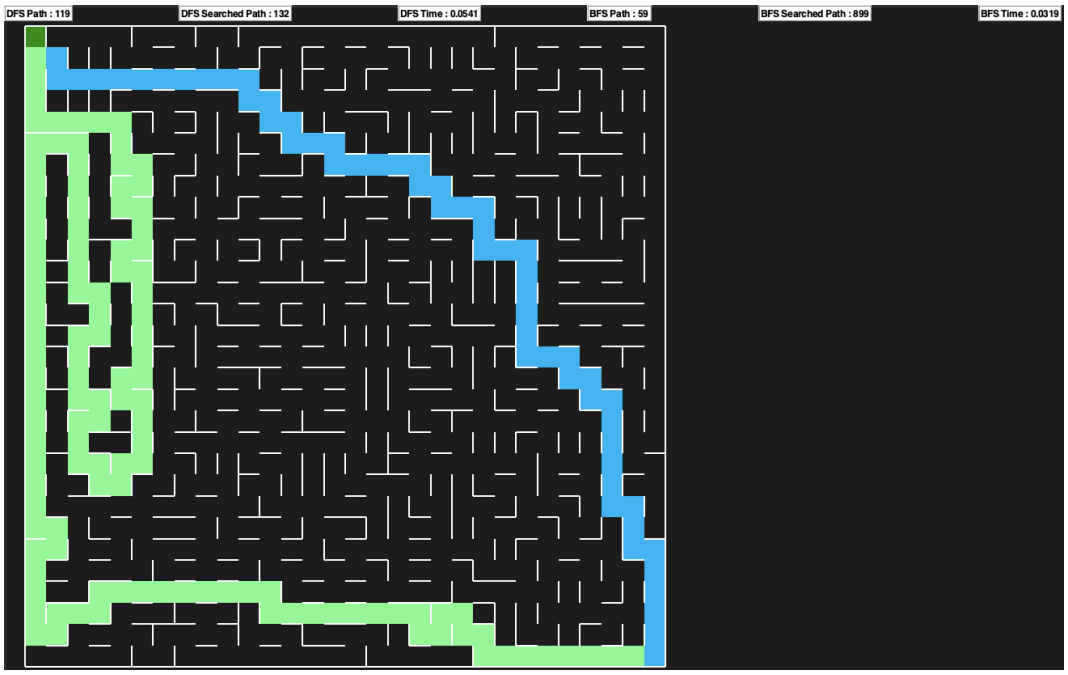

Figure 9: DFS vs BFS Algorithms running together on the maze grid of 30x30

- **DFS vs A***
    - Unlike DFS and BFS, A* is an informed search algorithm which finds the shortest path to the goal based on the estimated cost to the goal from a node (heuristic), and the cost to the node from a starting node.
    - The output displays that the path taken by A* is shorter as compared to DFS. And the time taken to execute DFS grows exponentially as the maze size increases, and is higher than that of A*.
    - Since A* always finds the shortest path, the path length remains fairly consistent and does not show any anomalies. On the other hand, the path length for DFS increases drastically with increasing maze sizes, due to the complexity of the maze and the unguided nature of the algorithm.

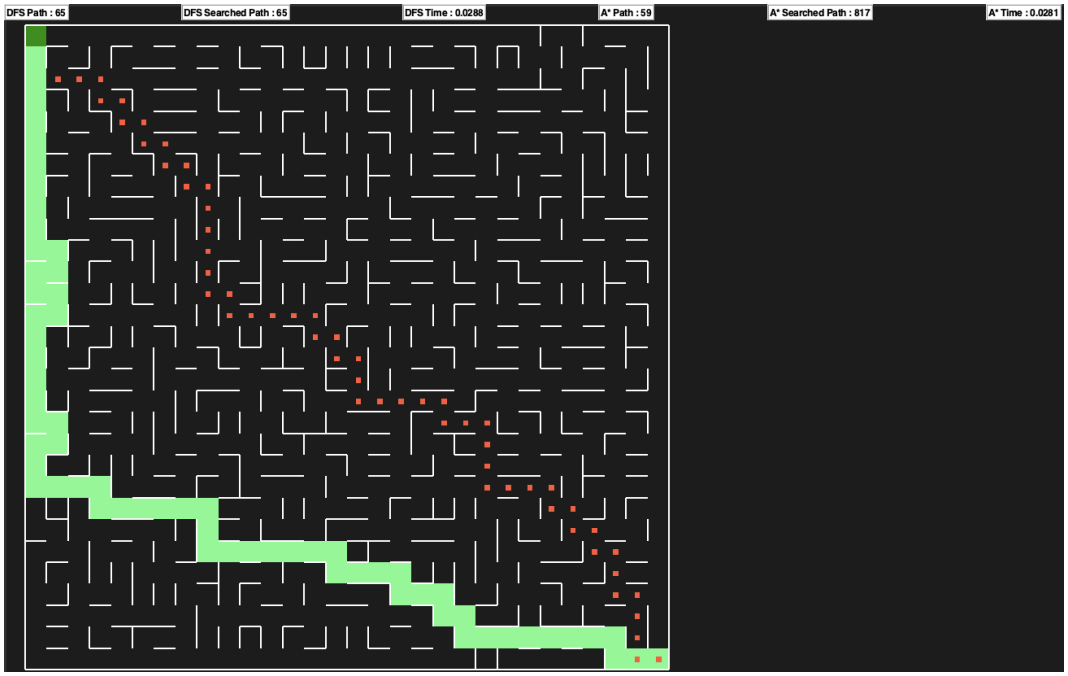

Figure 10: DFS vs A Algorithms running together on the maze grid of 30x30*

- **BFS vs A***
    - Both BFS and A* are designed to find the shortest path to the goal state.
    - The output displays that the path taken by both BFS and A* have same length of 59. However, both algorithms take a different path to the goal. As well as we can see that the time taken by both the algorithms is almost same. So the performance comparison differs with different maze sizes but we can ensure that both the algorithms work equally efficient.

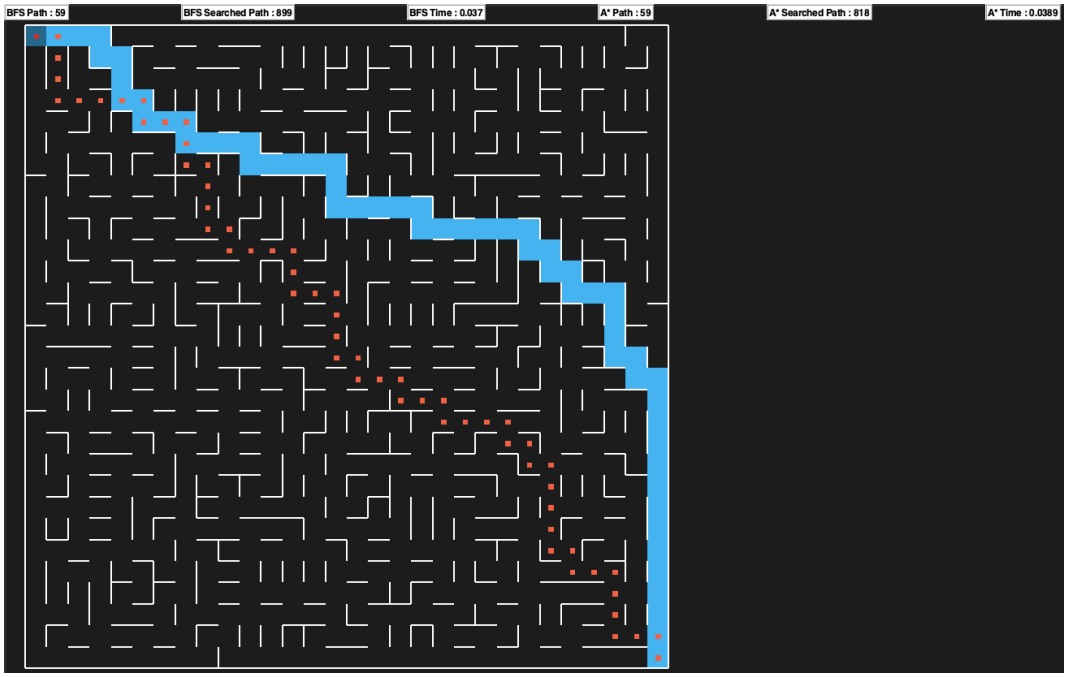

Figure 11: BFS vs A Algorithms running together on the maze grid of 30x30*

- **Value iteration vs Policy iteration**
    - The blue path displays the results for Value iteration, and red for Policy iteration.
    - From the results, we cannot make much difference between the number of steps taken by value iteration and policy iteration as both roughly take the similar number of steps to converge to the target or goal node.
    - Policy Iteration shows a slight edge over Value Iteration in terms of computation time, which could be crucial in larger state spaces or real-time applications. This is mainly because policy iteration has only a finite option of policies in a finite-state MDP, and they eventually converge in a finite number of steps.

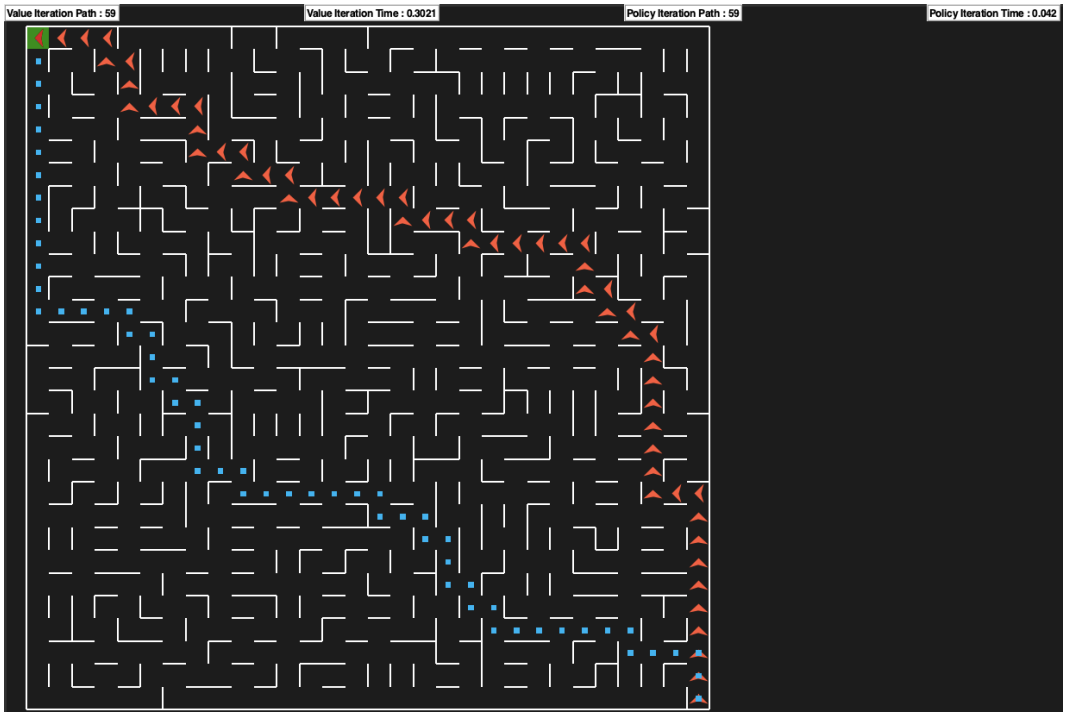

Figure12: MDP Algorithms running together on the maze grid of 30x30

- **Search Algorithms vs MDP Algorithms**

    - We can see that DFS performs the worst for all the three metrics that we have considered for this assignment.
    - BFS takes lesser number of steps to converge to the goal, but the time taken to complete the process is relatively higher when compared to A* and the MDP algorithms.
    - The MDP algorithms have comparable performance for the maze solver problem, and both have superior performance when compared to BFS and DFS for all the metrics.
    - In this problem of solving the maze and finding the optimal path, A* search algorithm performs the best in terms of the steps as well as time taken.

- **Comparison of Search & MDP Algorithm Performance on different Maze sizes**

Algorithm Comparison

Maze Size	Algorithm	Time Taken (s)	Steps Taken
5x15	DFS	8.82e-06	19
5x15	BFS	1.69e-05	35
5x15	A*	2.10e-05	41
5x15	ValueIteration	3.10e-05	62
5x15	PolicyIteration	5.01e-05	119
20x10	DFS	1.10e-05	74
20x10	BFS	1.38e-05	199
20x10	A*	1.48e-05	374
20x10	ValueIteration	6.18e-05	899
20x10	PolicyIteration	5.10e-05	2499
15x25	DFS	9.06e-06	56
15x25	BFS	1.41e-05	155
15x25	A*	2.81e-05	322
15x25	ValueIteration	2.29e-05	767
15x25	PolicyIteration	5.20e-05	2310
30x30	DFS	3.60e-05	18
30x30	BFS	4.77e-05	28
30x30	A*	7.61e-05	38
30x30	ValueIteration	1.09e-04	58
30x30	PolicyIteration	2.27e-04	98
50x50	DFS	1.19e-05	18
50x50	BFS	2.48e-05	28
50x50	A*	5.67e-05	38
50x50	ValueIteration	4.20e-05	58
50x50	PolicyIteration	1.56e-04	98

Figure 13:Comparison of Algorithms with different Maze sizes with Target value as (1,1)

    - **Algorithmic Performance:**
        - Depth-First Search (DFS): This algorithm doesn't always finish the maze faster than the others, but it usually requires fewer steps to finish. This implies that DFS might occasionally be more time-efficient but also more step-efficient.
        - Breadth-First Search (BFS): BFS performs inconsistently in terms of steps taken and time required. It appears to work well on smaller mazes, but as the size of the maze increases, its effectiveness decreases.
        - Reinforcement learning algorithms such as Value Iteration and Policy Iteration typically require longer execution periods but differ in the number of steps. They appear to have more steps on larger mazes than DFS and BFS, which could point to a more deliberate exploration approach.
    - **Maze Size Impact:**
        - The number of steps and time required for each algorithm rises as the maze size increases. But the rate of rise differs depending on the algorithm. For example, DFS increases time and steps more gradually than BFS, which looks to expand more fast as the maze size increases.
    - **Efficiency Analysis:**
        - Time Efficiency: DFS appears to be time-efficient in smaller mazes, but as the labyrinth size increases, its time efficiency falls. However, the time taken by the value

iteration and policy iteration algorithms increases steadily as the maze size increases, indicating that they might not scale as well as DFS in bigger mazes.
        - Step Efficiency: Compared to the reinforcement learning algorithms, DFS and BFS are typically more step-efficient, suggesting that they might be taking shorter routes to the objective.
    - **Scalability:**
        - BFS does not appear to scale as well as DFS when it comes to scalability to larger mazes. Even though they are not the fastest, the reinforcement learning algorithms do not exhibit an exponential increase in steps or time, which may indicate improved scalability with even larger mazes than those put to the test.
    - **Algorithm Sustainability:**
        - The size of the maze and the intended ratio of time to step efficiency can be used to determine whether an algorithm is suitable. Better are DFS and BFS for smaller mazes. Policy iteration or value iteration may be more appropriate for larger mazes when the quality of the solution (i.e., fewer steps) is more essential than the time taken to locate the solution.
    - **Possibility of Optimising:**
        - The algorithms' time efficiency may be increased by refining their implementation, particularly the reinforcement learning ones. It might take less time if there was parallel processing or more effective data structures.
    - **Uniformity Across Experiments:**
        - It would be crucial to see whether there is a substantial variance in the outcomes or if they remain constant throughout several trials. The algorithms' resilience to varying initial conditions would be suggested by consistency.

- **Comparison of Time Taken by Search & MDP algorithms for different Maze sizes**

    - **Scaling with Maze Size:** All algorithms show an increase in time taken as the size of the maze increases, which is expected since larger mazes generally require more computation.
    - **Value Iteration Performance:** The Value Iteration algorithm appears to have a relatively flat performance across smaller maze sizes but shows a significant increase in time for the largest maze size (50x50). This suggests that Value Iteration's performance is affected by the state space size, with larger mazes leading to exponentially higher computation times.
    - **Policy Iteration Consistency:** Policy Iteration shows a steady increase in computation time with maze size. The curve is smoother than that of Value Iteration, indicating a more consistent performance across the tested range of maze sizes.
    - **Comparison of Search Algorithms:** Among the search algorithms, A* generally performs better than BFS and DFS for smaller mazes but starts to converge with BFS in larger mazes. This might suggest that the heuristic used in A* is effective in smaller mazes but less so in larger mazes where the number of nodes increases significantly.

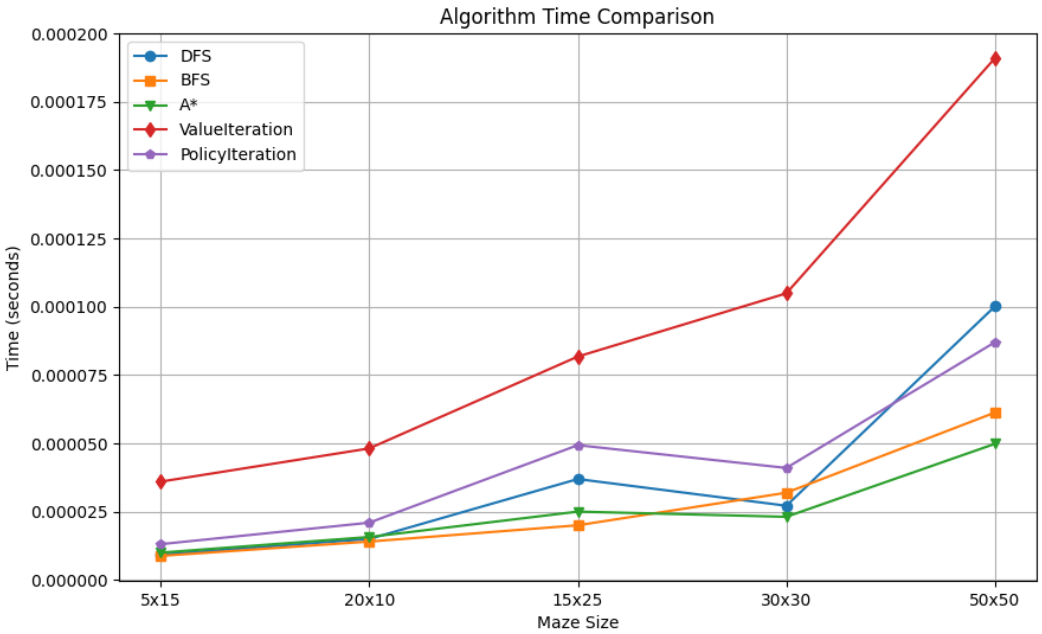

Figure 14: Comparison of Time Taken by Search and MDP Algorithms for different Maze size

- **Comparison of Steps Taken by Search & MDP algorithms for different Maze sizes**
    - **Value Iteration and Policy Iteration:** Both MDP algorithms have identical step counts across all maze sizes, which are significantly lower than those of the Search algorithms. This indicates that the MDP algorithms are more efficient in finding shorter paths under the given reward structure.
    - **Search Algorithm Comparison:** BFS and A* have similar step counts, with A* slightly lower in larger mazes. This indicates that A*'s heuristic is somewhat effective in guiding the search towards the goal more directly than BFS.
    - **Efficiency vs. Optimality:** The difference in step counts for Search algorithms versus MDP algorithms highlights the trade-off between efficiency (fewer steps) and optimality (shortest path). MDP algorithms are finding more optimal paths due to the global knowledge they accumulate during the value/policy computation process.
    - **Optimality of MDP Paths:** The flatness of the MDP algorithms' lines suggests that the increase in maze size does not significantly impact the optimality of the paths they find, which is a result of the algorithms optimizing the policy globally across the entire state space.

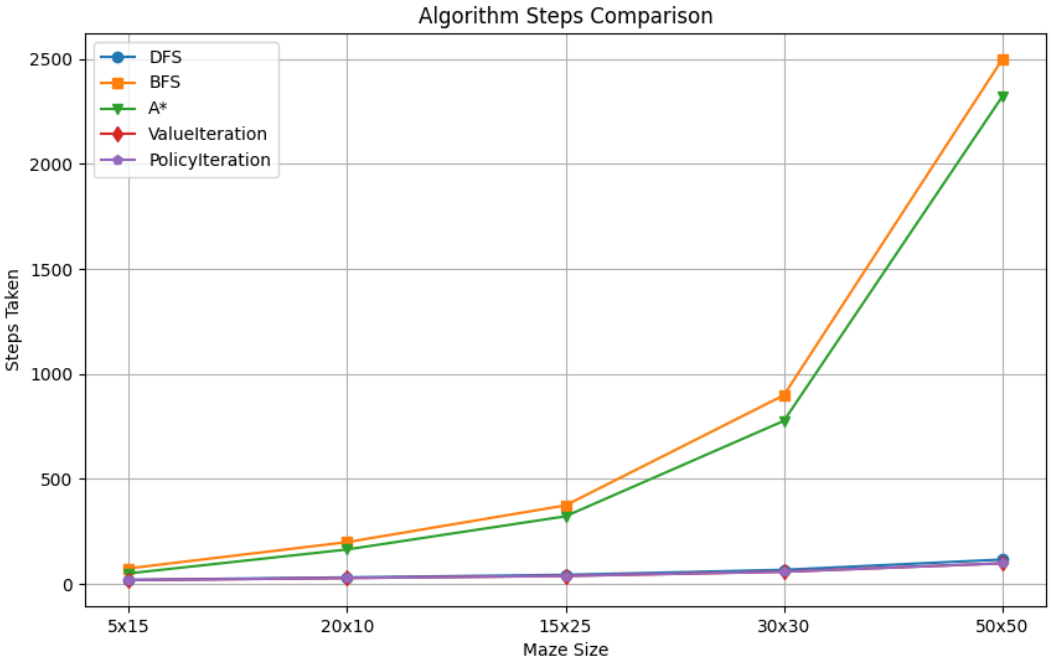

Figure 15: Comparison of Steps Taken by Search and MDP Algorithms for different Maze sizes

Conclusion

In conclusion, interesting trends regarding the computational behaviour of search and MDP algorithms in maze-solving scenarios are shown by comparing their analyses. Though they often operate more quickly, search algorithms don't always produce the best routes, especially when the maze's complexity rises. However, MDP algorithms, which in this study are represented by Value Iteration and Policy Iteration, show a surprising capacity to identify paths that are both shorter and more optimal, even though they frequently do so at the expense of longer computation times, especially in bigger mazes. As such, the exact needs of the task at hand greatly influence an algorithm's suitability. MDP algorithms are clearly the better option if optimality is of the utmost importance and computational resources are plentiful.

The trends in performance highlight how crucial algorithm selection is, and how important it is as problem complexity rises. While selecting an algorithm may not have much of an effect on the result for straightforward mazes, it may be essential to performance for complex and expansive mazes. This work emphasises how important it is to choose algorithms strategically, taking into account the unique characteristics and needs of the issue area. It also provides opportunities for further research to optimise these algorithms for a better trade-off between path optimality and speed, guaranteeing their flexibility and effectiveness in a range of real-world scenarios where maze-solving paradigms are pertinent.

References

[1] MAN1986, (2021) pyamaze [Source Code] https://github.com/MAN1986/pyamaze

[2] Paul E. Black, "Manhattan distance", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 11 February 2019. Available from: https://www.nist.gov/dads/HTML/manhattanDistance.html

[3] Paul E. Black, "Euclidean distance", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 17 December 2004. Available from: https://www.nist.gov/dads/HTML/euclidndstnc.html

[4] Russell, S.J. and Norvig, P. (2022) Artificial Intelligence: A modern approach. Harlow: Pearson Education Limited.

[5] MDP_VI_PI_Q-learning_AIMA.ipynb by Tirthajyoti Sarkar: https://github.com/tirthajyoti/RL_basics

[6] Markov Decision Process by Joseph Su. Department of Computer Science, Georgia Institute of Technology: https://jsu800.github.io/docs/ml_mdp.pdf

[7] MazeMDP by Sally Gao. GitHub Repository: https://github.com/sally-gao/mazemdp

Appendix 1 (DFS)

```
# ----------------DFS----------------------
class DFS(SearchAlgo):
    def __init__(self, m, goal):
        super().__init__(m, goal, algo='dfs')

    def execute_search(self):
        self.set_params()
        while self.get_stopping_condition():
            self.currCell = self.get_current_cell()
            if self.currCell == self.goal:
                break
            for d in self.nodes:
                if self.m.maze_map[self.currCell][d]:
                    child = self.move(direction=d)
                    if child in self.explored:
                        continue
                    self.explored.append(child)
                    self.frontier.append(child)
                    self.algoPath[child] = self.currCell
        path, steps, time_taken = self.get_forward_path()
        return path, steps, time_taken
```

Appendix 2 (BFS)

```
# ----------------BFS----------------------
class BFS(SearchAlgo):
    def __init__(self, m, goal):
        super().__init__(m, goal, algo='bfs')

    def execute_search(self):
        self.set_params()
        while self.get_stopping_condition():
            self.currCell = self.get_current_cell()
            if self.currCell == self.goal:
                break
            for d in self.nodes:
                if self.m.maze_map[self.currCell][d]:
                    child = self.move(direction=d)
                    if child in self.explored:
                        continue
                    self.explored.append(child)
                    self.frontier.append(child)
                    self.algoPath[child] = self.currCell
        path, steps, time_taken = self.get_forward_path()
        return path, steps, time_taken
```

Appendix 3 (A*)

```
# ----------------A*-----------------------
class AStar(SearchAlgo):
    def __init__(self, m, goal):
        super().__init__(m, goal, algo='a*')
    def execute_search(self):
        self.set_params()
        while self.get_stopping_condition():
            self.currCell, self.currCost = self.get_current_cell()
            if self.currCell == self.goal:
                break
            for d in self.nodes:
                if self.m.maze_map[self.currCell][d]:
                    child = self.move(direction=d)
                    temp_g_score = self.g_score[self.currCell] + 1
                    temp_f_score = temp_g_score + calculateEstimatedDistance(child)
                    if temp_f_score < self.f_score[child]:
                        self.g_score[child] = temp_g_score
                        self.f_score[child] = temp_f_score
                        self.store.put((self.f_score[child], child))
                        self.algoPath[child] = self.currCell
        path, steps, time_taken = self.get_forward_path()
        return path, steps, time_taken
```

Appendix 4 (Value Iteration)

```
# ----------------Value Iteration-----------------------
class ValueIteration(MarkovDecisionProcess):
    def __init__(self, m=None, goal=None, isDeterministic=True):
        super().__init__(m, goal, isDeterministic)
        self._reward = -4  # LIVING REWARD
        self._max_error = 10 ** (-3)
        self.target = [self.goal]
        self.values = {state: 0 for state in self.actions.keys()}
        self.values[self.target[0]] = 1
        self.algoPath = {}
        self.explored = []
        self.mainTime = 0
    def get_maxDelta(self, delta, utilityMax, state):
        return max(delta, abs(utilityMax - self.values[state]))
    def calculate_valueIteration(self):
        start = time.time()
        while True:
            delta = 0
            for state in self.actions.keys():
                if state == self.target[0]:
                    continue
```

```
            utilityMax = float("-inf")
            for action, prob in self.actions[state].items():
                for direction in action:
                    if self.m.maze_map[state][direction]:
                        childCell = self.move(state, direction)
                reward = self._reward
                if childCell == self.target[0]:
                    reward = 10000
                utility = 0
                utility += super().calculate_ValueIterationUtility(prob, reward, childCell, self.values)
                if utility > utilityMax:
                    utilityMax = utility
            delta = self.get_maxDelta(delta, utilityMax, state)
            self.values[state] = utilityMax
        if delta < self._max_error:
            break
    end = time.time()
    self.mainTime = end-start
def create_searchPath(self, currNode):
    start = time.time()
    node = currNode
    steps = 0  # Initialize step counter
    while True:
        if node == self.target[0]:
            break
        selectedNode = None
        selectedNodeVal = None
        for direction in 'ENWS':
            if self.m.maze_map[node][direction]:
                traverseDirection = self.move(node, direction)
                # If the traverseDirection is directly the target, select it immediately
                if traverseDirection == self.target[0]:
                    selectedNode = traverseDirection
                    break
                # For Value Iteration, use the utility values to select the next node
                if selectedNodeVal is None or self.values[traverseDirection] > selectedNodeVal:
                    selectedNode = traverseDirection
                    selectedNodeVal = self.values[selectedNode]
        if selectedNode:
            steps += 1  # Increment step counter only if a new node is selected
            self.explored.append(node)  # Mark the current node as explored
            self.algoPath[node] = selectedNode  # Record the path
            node = selectedNode  # Move to the next node
    end = time.time()
    self.mainTime = end - start
    return self.algoPath, self.mainTime, steps
```

Appendix 5 (Policy Iteration)

```
# ----------------Policy Iteration----------------------
class PolicyIteration(MarkovDecisionProcess):
    def __init__(self, m=None, goal=None, isDeterministic=True):
        super().__init__(m, goal, isDeterministic)
        self.target = [self.goal]
        self.values = {state: 0 for state in self.actions.keys()}
        self.values[self.target[0]] = pow(10, 7)
        self.policyValues = {s: random.choice('N') for s in self.actions.keys()}
        self._reward = {state: -40 for state in self.actions.keys()}  # LIVING REWARD
        self._reward[self.target[0]] = pow(10, 8)
        self.algoPath = {}
        self.mainTime = 0
    def calculate_policyIteration(self):
        start = time.time()
        policyTrigger = True
        while policyTrigger:
            policyTrigger = False
            valueTrigger = True
            while valueTrigger:
                valueTrigger = False
                for state in self.actions.keys():
                    if state == self.target[0]:
                        continue
                    utilityMax = float('-infinity')
                    actionMax = None
                    for action, prob in self.actions[state].items():
                        for direction in action:
                            if self.m.maze_map[state][direction]:
                                childNode = self.move(state, direction)
                        utility = super().calculate_PolicyIterationUtility(prob, self._reward, state, childNode,
                                                                           self.values)
                        if utility > utilityMax:
                            utilityMax = utility
                            actionMax = action
                    self.policyValues[state] = actionMax
                    self.values[state] = utilityMax
                    if self.policyValues[state] != actionMax:
                        policyTrigger = True
                        self.policyValues[state] = actionMax
        end = time.time()
        self.mainTime = end-start
    def create_searchPath(self, currNode):
        start = time.time()
        node = currNode
        steps = 0  # Initialize step counter
```

```
    while node != self.target[0]:
        # Determine the next node based on the current policy
        nextNode = self.move(node, self.policyValues[node])
        if nextNode != node:  # Check to ensure we're not stuck in the same node
            steps += 1  # Increment step counter when moving to a new node
        self.algoPath[node] = nextNode
        node = nextNode  # Move to the next node
        if node == self.target[0]:  # Break the loop if the target is reached
            break
    end = time.time()
    self.mainTime = end - start
    return self.algoPath, self.mainTime, steps
```

The implementation of the Assignment-1 is uploaded on GitHub: https://github.com/priyaaa705/Maze-Generation