AI Coursework – 2403206c

Introduction

The aim of this exercise is to evaluate my implementation of a discrete-space search algorithm in solving teleportable mazes, where teleport links connect points in the maze. The algorithm will attempt to find the shortest path from a starting position to a goal position while dynamically discovering and utilizing teleports. I decided to implement Breadth-First Search (BFS).

This report discusses the implementation of BFS algorithm and critically evaluates its performance. My approach was based on Russell & Norvig's *Artificial Intelligence: A Modern Approach (AIMA)*[1], which provided a theoretical basis for BFS.

Task 1: Implementation and Explanation of Searching Algorithms

Maze Implementation

The maze for this problem is generated using the gen_maze_with_teleports function, which creates a perfect maze using Prim's algorithm. This algorithm generates a single-solution maze by iteratively carving paths between cells while maintaining a connected structure.

Two parameters define the maze:

- ***n*:** The size of the maze (*n* x *n*)
- ***t*:** The number of teleportation links, which randomly connect two open cells.

```python
import random
from mazelib.solve.BacktrackingSolver import BacktrackingSolver

def gen_maze_with_teleports(n, t):
    """
    Generate a random maze with teleportation links

    n: size of the maze (n x n)
    t: number of teleportaion links

    return: maze object, dictionary of teleportation links
    """

    # create the maze
    m = Maze()
    m.generator = Prims(int(n/2), int(n/2))
    m.generate()
    m.generate_entrances(True, True)

    # add teleportation links
    cells = [(x, y) for x in range(len(m.grid)) for y in range(len(m.grid[0])) if m.grid[x][y] == 0]
    teleport_links = {}

    while len(teleport_links) < t:
        source = random.choice(cells)
        destination = random.choice(cells)

        if source != destination and source not in teleport_links and destination not in teleport_links.values():
            teleport_links[source] = destination

    return m, teleport_links
```

Figure 1: Implementation of the maze generation function including teleports.

The above figure demonstrates how teleports are incorporated. Open cells are identified, and teleports are created by randomly selecting a pair of source and destination cells. Each teleport link is represented in a dictionary where the key is the source cell, and the value is the destination cell.

```
1  N = 10
2  T = 2
3  m, teleport_links = gen_maze_with_teleports(N, T)
4
5  problem = MazeProblem(m.start, m.end, m, teleport_links)
6  print(problem.initial, problem.goal)
7  print(m)
8  print(teleport_links)
```

✓ 0.0s

```
(0, 1) (10, 5)
#S#########
#   # #   #
### # # ###
#         #
# ### # ###
#   # #   #
# ##### ###
# #       #
####### # #
#       # #
#####E#####
{(3, 2): (7, 5), (3, 3): (9, 6)}
```

Figure 2: Example output of generating a maze with teleport links.

MazeProblem Class

The **MazeProblem** class models the maze as a state-space search problem. Key components of this class include:

1. **State Representation**:
   - The maze is a 2D grid where 0 represents walls and 1 represents open paths. States are defined as the coordinates of a cell (x, y).

2. **Actions**:
   - The agent can move in four cardinal directions: up, down, left, or right.
   - Movement is only allowed if:
     - The target cell is an open path (0).
     - The target cell is not a wall (1).
     - The movement does not exceed the maze boundaries.

        - If the movement action leads closer to the goal, it is evaluated as part of the frontier during the search process.
    - The agent can take the teleport action if:
        - The current cell is a teleport source.
        - The corresponding teleport destination has been discovered.
        - Teleportation allows the agent to jump between distant points in the maze, potentially having significant impact on the exploration efficiency and path costs.
        - Initially, all teleport destinations are unknown to the agent.
        - When the agent reaches a teleport source, the **discover_teleport** method dynamically reveals the corresponding destination and adds it to **self.discovered_teleports**.
        - This mechanism ensures that teleports are only used after they are encountered, aligning with the problem constraints that simulate limited agent knowledge.
    - If the movement action leads closer to the goal, it is evaluated as part of the frontier during the search process.

3. **Path Cost**:
    - The cost of each action is uniform (incremented by 1 for every move or teleport).

4. **Goal Test**:
    - The agent reaches the goal when the current state matches the goal state.

The agent's actions directly influence the exploration efficiency and results:

1. **Movement Actions**:
    - These actions allow the agent to explore adjacent cells in the maze.
    - When movement actions are prioritized (e.g., in BFS), they ensure systematic exploration, guaranteeing that all reachable cells are evaluated.

Observation:

1. Movement actions should dominate the exploration in areas without teleport sources.
2. BFS should evaluate movement actions in breadth-first order, minimizing redundant exploration.

2. **Teleport Actions**:
    - Teleport actions introduce shortcuts, significantly reducing the effective search space.
    - The dynamic discovery mechanism ensures that teleports are only used after their destinations are revealed, simulating real-world constraints where the agent lacks prior knowledge.

Observation:

- Well-placed teleports near the goal may drastically reduce the number of steps required.
- Poorly placed teleports or those discovered late may have minimal or no impact on the results.

```python
from search import Problem

class MazeProblem(Problem):
    def __init__(self, initial, goal, maze, teleports=None):
        """
        initial: The starting position (x, y)
        goal: The goal position (x, y)
        maze: A 2D list where 0 represents a wall and 1 represents an open path
        teleports: Dictionary of teleportation links {source: destination}
        """
        self.initial = initial
        self.goal = goal
        self.maze = maze.grid
        self.teleports = teleports or {}  # All teleports (unknown to agent initially)
        self.discovered_teleports = {}   # Tracks only teleports discovered during traversal

    def actions(self, state):
        """Return the possible actions from a given state, including teleportation."""
        actions = []
        x, y = state

        # Standard movement actions
        if x > 0 and (self.maze[x - 1][y] == 0 or (x - 1, y) == self.goal):  # Up
            actions.append('up')
        if x < len(self.maze) - 1 and (self.maze[x + 1][y] == 0 or (x + 1, y) == self.goal):  # Down
            actions.append('down')
        if y > 0 and (self.maze[x][y - 1] == 0 or (x, y - 1) == self.goal):  # Left
            actions.append('left')
        if y < len(self.maze[0]) - 1 and (self.maze[x][y + 1] == 0 or (x, y + 1) == self.goal):  # Right
            actions.append('right')

        # Teleport action (only include discovered teleports)
        if state in self.discovered_teleports:
            actions.append('teleport')

        return actions

    def result(self, state, action):
        """Return the resulting state after taking an action."""
        x, y = state
        if action == 'up':
            return (x - 1, y)
        elif action == 'down':
            return (x + 1, y)
        elif action == 'left':
            return (x, y - 1)
        elif action == 'right':
            return (x, y + 1)
        elif action == 'teleport' and state in self.discovered_teleports:
            # Jump to the teleport destination
            return self.discovered_teleports[state]

    def discover_teleport(self, state):
        """
        Discover the teleport destination for the current state if it is a teleport input.
        """
        if state in self.teleports and state not in self.discovered_teleports:
            self.discovered_teleports[state] = self.teleports[state]

    def is_goal(self, state):
        """Return True if the state is the goal state."""
        return state == self.goal

    def path_cost(self, c, state1, action, state2):
        """Calculate the path cost."""
        return c + 1
```

*Figure 3: Implementation of the **MazeProblem** class.*

Search Algorithm Implementation

BFS is a graph-based search algorithm that systematically explores all nodes at a current depth level before moving to the next depth level. It uses a queue data structure (FIFO) to maintain the frontier of nodes yet to be explored, ensuring that nodes are expanded in increasing order of their depth.

According to Russell and Norvig [1], BFS guarantees completeness (it will always find a solution if one exists) and optimality (it finds the shortest path in terms of the number of steps, assuming uniform costs). This makes BFS particularly suitable for unweighted mazes, such as the teleportable maze in this problem.

```python
def breadth_first_search(problem, visualize=True):
    "Search for goal; paths with least number of steps first."
    if problem.is_goal(problem.initial):
        print(problem.initial)
        return Node(problem.initial)

    frontier = FrontierQ(Node(problem.initial), LIFO=False)
    explored = set()

    iterations = 0

    while frontier:
        node = frontier.pop()
        explored.add(node.state)
        problem.discover_teleport(node.state)
        if problem.is_goal(node.state):
            if visualize:
                animate_search(problem, visited=list(explored), frontier=[], solution_path=state_sequence(node.path()))
            return node, explored

        for action in problem.actions(node.state):
            child = node.child_node(problem, action)
            if child.state not in explored and child.state not in frontier:
                frontier.add(child)
                if visualize and iterations % 3 == 0:
                    # Update visualization at this step
                    animate_search(problem, visited=list(explored), frontier=[n.state for n in frontier.values()])
        iterations += 1

    return None
```

*Figure 4: Implementation of the **breadth_first_search** function.*

The above figure shows my implementation of the BFS algorithm, the core components are as follows:

1. **Frontier Initialization:**
    - The frontier is initialized as a FIFO queue using the FrontierQ class from
    - It starts with the initial state encapsulated in a Node object, representing the root of the search tree.
    - Using LIFO=False ensures that the queue operated in FIFO mode, which is essential for BFS to process the nodes in breadth-first order.

2. **Explored Set:**
    - An explored set is used to track visited states to prevent revisiting nodes and redundant exploration.

3. **Dynamic Teleport Discovery:**
    - The **problem.discover_teleport(node.state)** function is called whenever a node is expanded. This ensures that teleport destinations are revealed dynamically when their sources are encountered.

4. **Node Expansion:**
    - The solver processes each node by:
        - Removing it from the frontier.
        - Checking if it is the goal.
        - Expanding on it by evaluating all possible actions (up, down, left, right, teleport)
    - For each valid action, a child node is created and added to the frontier if it has not already been explored or added to the frontier.

5. **Visualisation:**
    - The **animate_search** function is called periodically to visually display the agent navigating the maze.

```python
from collections import OrderedDict

class FrontierQ(OrderedDict):
    "A Frontier that supports FIFO or LIFO Queue ordering."

    def __init__(self, initial, LIFO=False):
        """Initialize Frontier with an initial Node.
        If LIFO is True, pop from the end first; otherwise from front first."""
        super(FrontierQ, self).__init__()
        self.LIFO = LIFO
        self.add(initial)

    def add(self, node):
        "Add a node to the frontier."
        self[node.state] = node

    def pop(self):
        "Remove and return the next Node in the frontier."
        (state, node) = self.popitem(self.LIFO)
        return node

    def replace(self, node):
        "Make this node replace the nold node with the same state."
        del self[node.state]
        self.add(node)
```

0.0s

*Figure 5: Implementation of the **FrontierQ** class.*

```python
class Node(object):
    """A node in a search tree. A search tree is spanning tree over states.
    A Node contains a state, the previous node in the tree, the action that
    takes us from the previous state to this state, and the path cost to get to
    this state. If a state is arrived at by two paths, then there are two nodes
    with the same state."""

    def __init__(self, state, parent=None, action=None, path_cost=1):
        "Create a search tree Node, derived from a previous Node by an action."
        self.state     = state
        self.parent    = parent
        self.action    = action
        self.path_cost = path_cost

    def __repr__(self): return "<Node {}: {}>".format(self.state, self.path_cost)

    def __lt__(self, other): return self.path_cost < other.path_cost

    def child_node(self, problem, action):
        "The Node you get by taking an action from this Node."
        next_state = problem.result(self.state, action)
        return Node(next_state, self, action,
                    problem.path_cost(self.path_cost, self.state, action, next_state))

    def expand(self, problem):
        """List the nodes reachable in one step from this node."""
        return [self.child_node(problem, action)
                for action in problem.actions(self.state)]

    def solution(self):
        """Return the sequence of actions to go from the root to this node."""
        return [node.action for node in self.path()[1:]]

    def path(self):
        """Return a list of nodes forming the path from the root to this node."""
        node, path_back = self, []
        while node:
            path_back.append(node)
            node = node.parent
        return list(reversed(path_back))
```

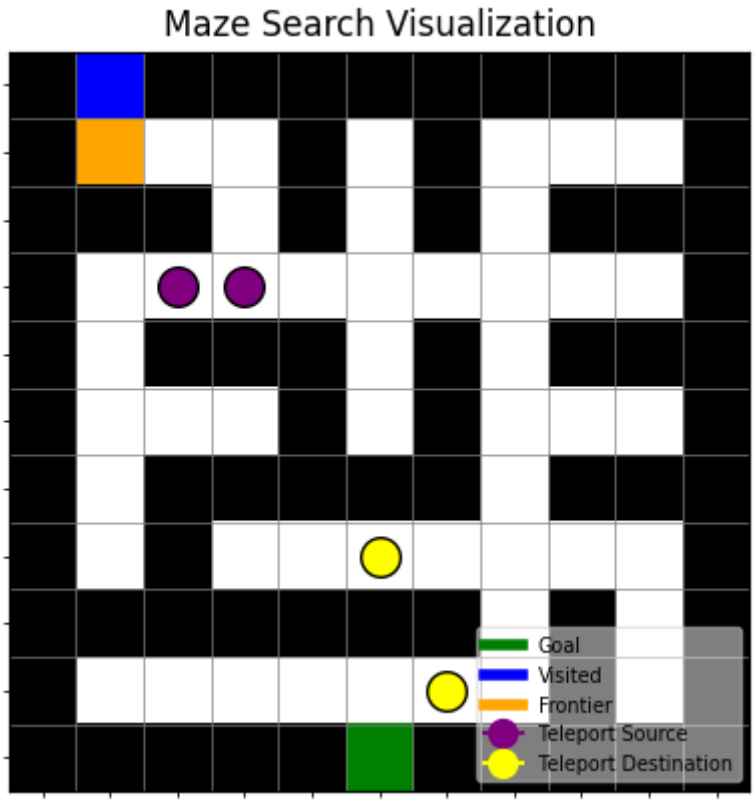
*Figure 6: Implementation of the **Node** class.*

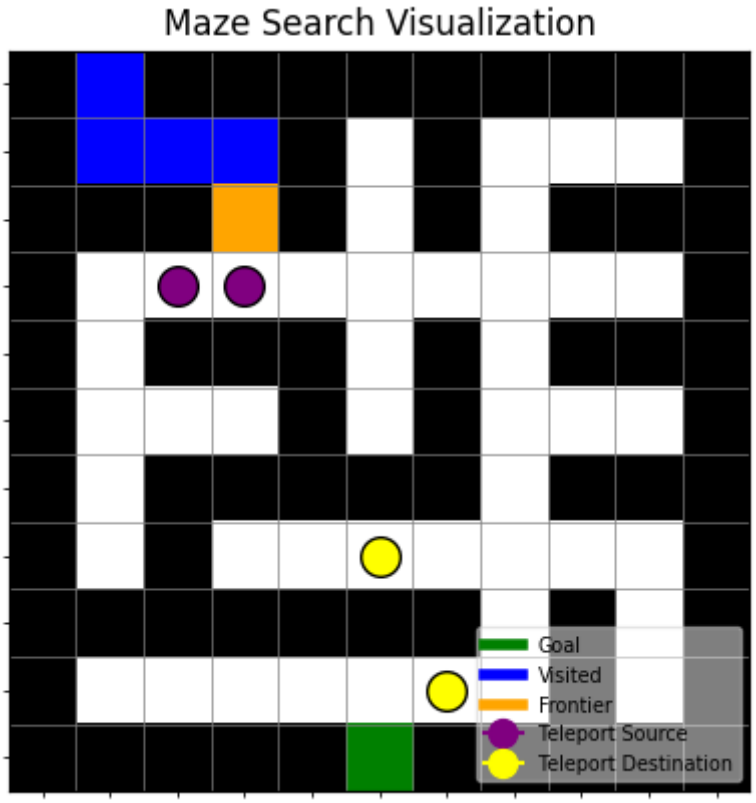
Solving a Maze Using BFS

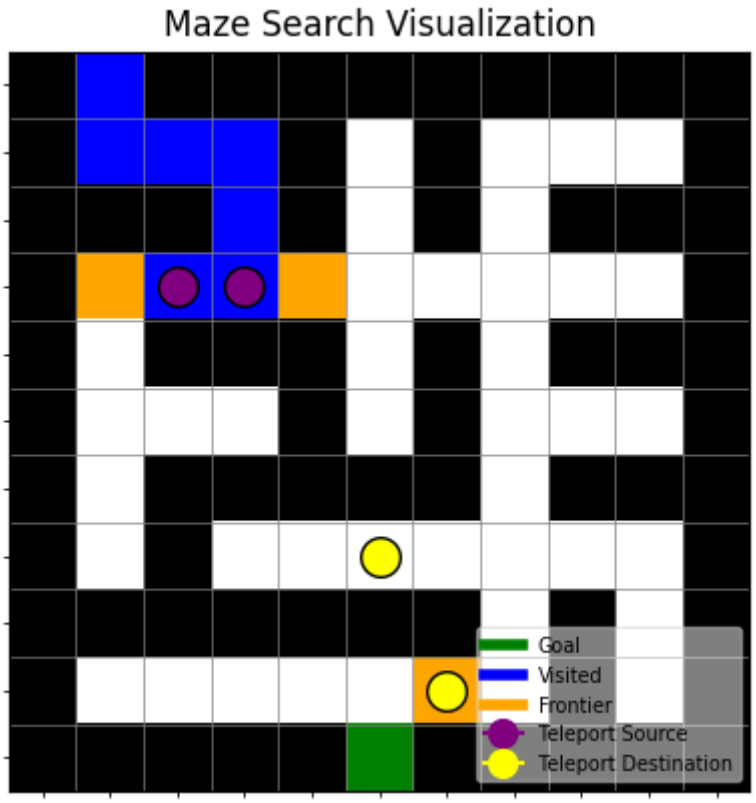
The below figures evidence my agents process in navigating the generated maze, note that the progress is displayed every 3 iterations in order to save space.

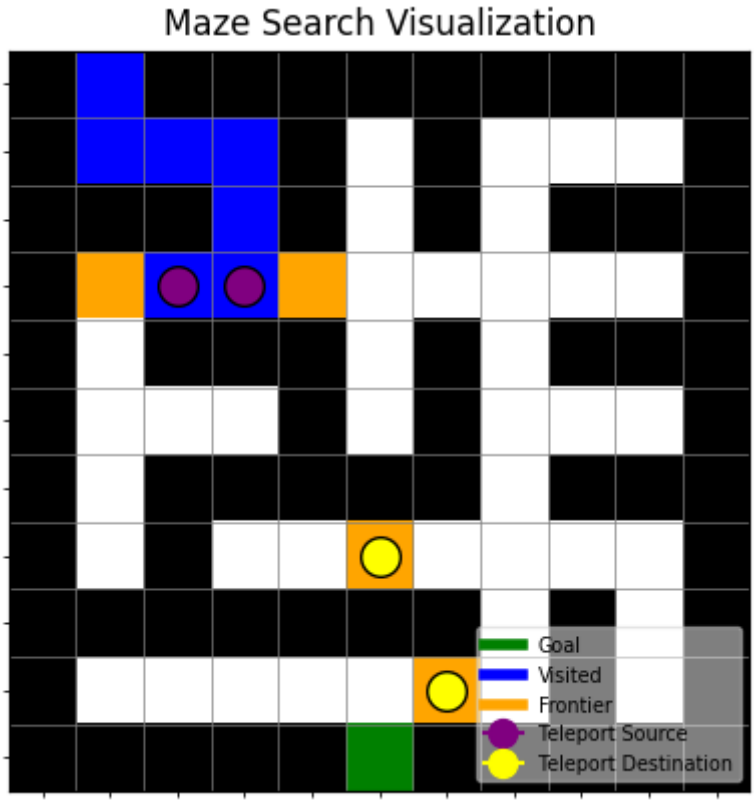
```
1  bfs, bfs_explored = breadth_first_search(problem, visualize=True)
✓ 6.8s
```

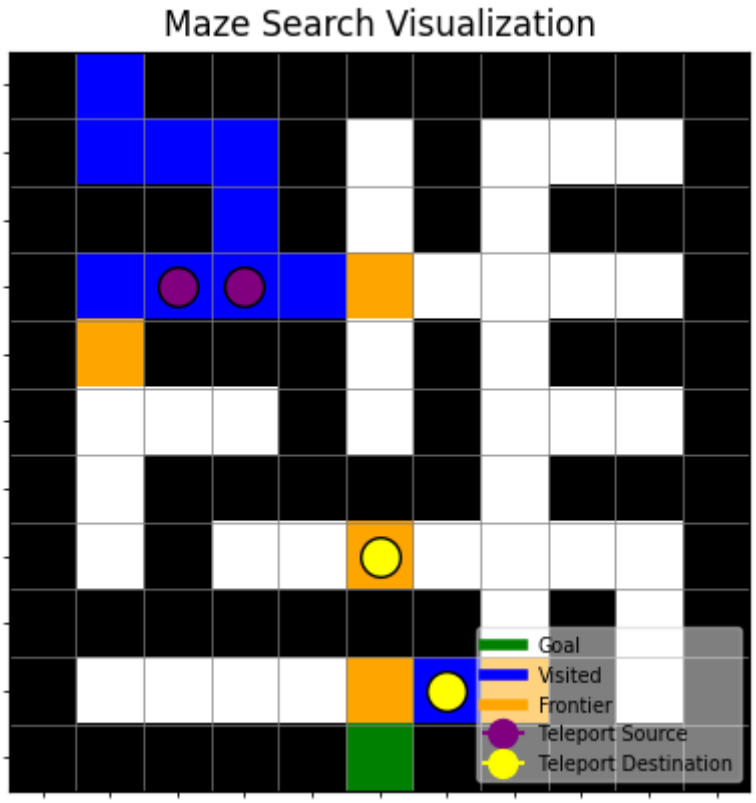
Figure 7: Calling the BFS algorithm on the MazeProblem.

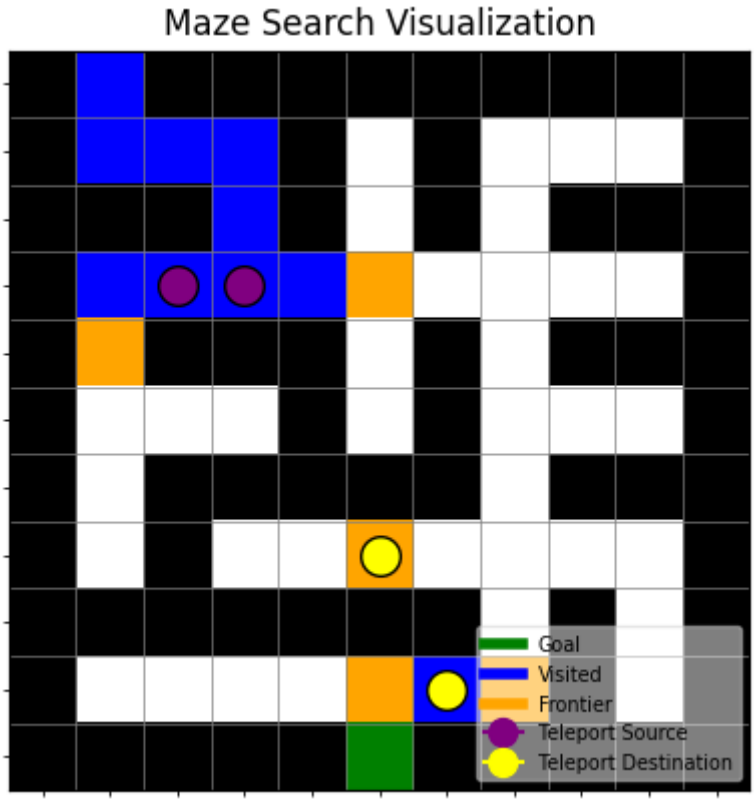

Maze Search Visualization
Goal
Visited
Frontier
Teleport Source
Teleport Destination

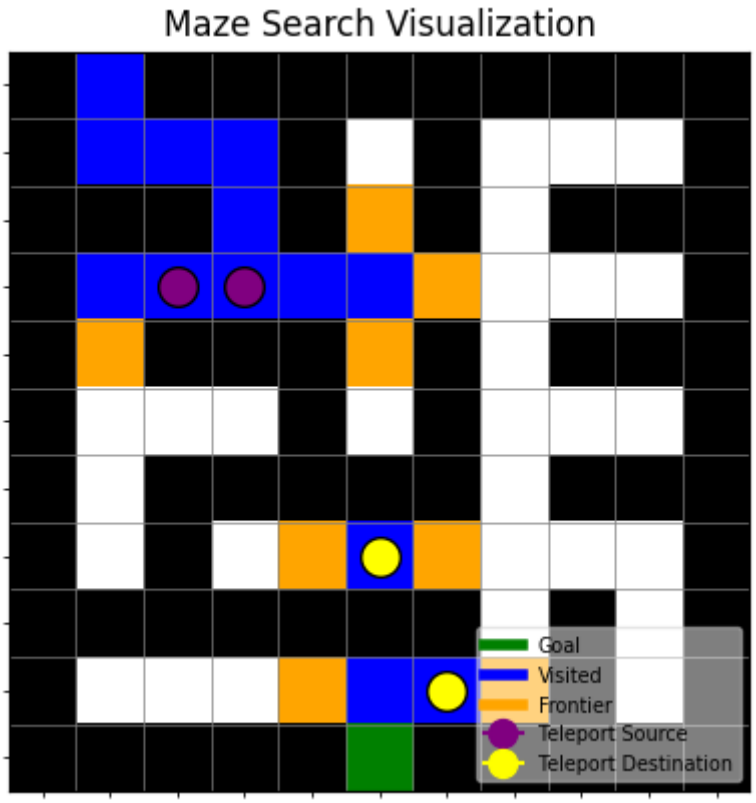

Maze Search Visualization
Goal
Visited
Frontier
Teleport Source
Teleport Destination

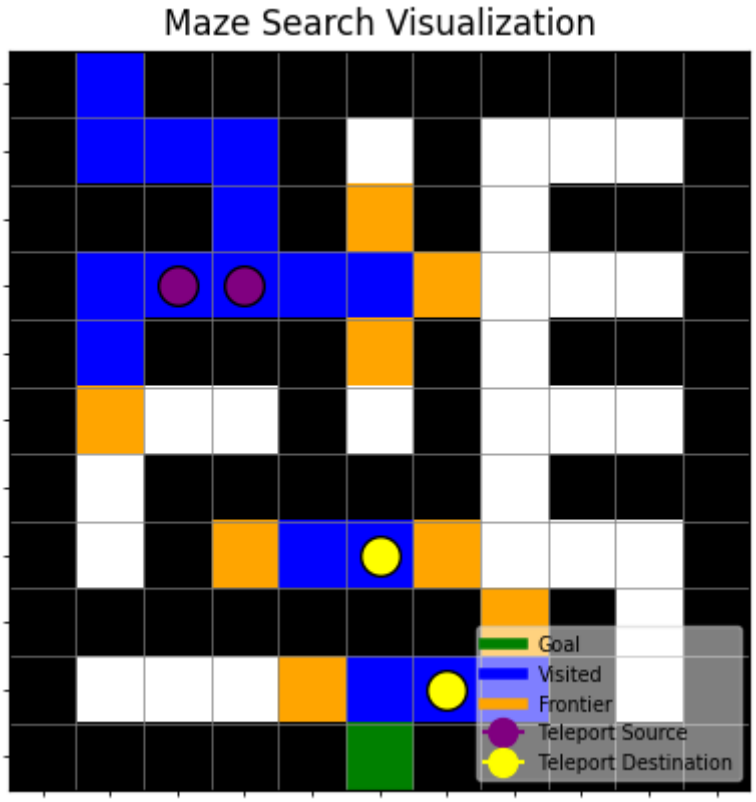

Maze Search Visualization
Goal
Visited
Frontier
Teleport Source
Teleport Destination

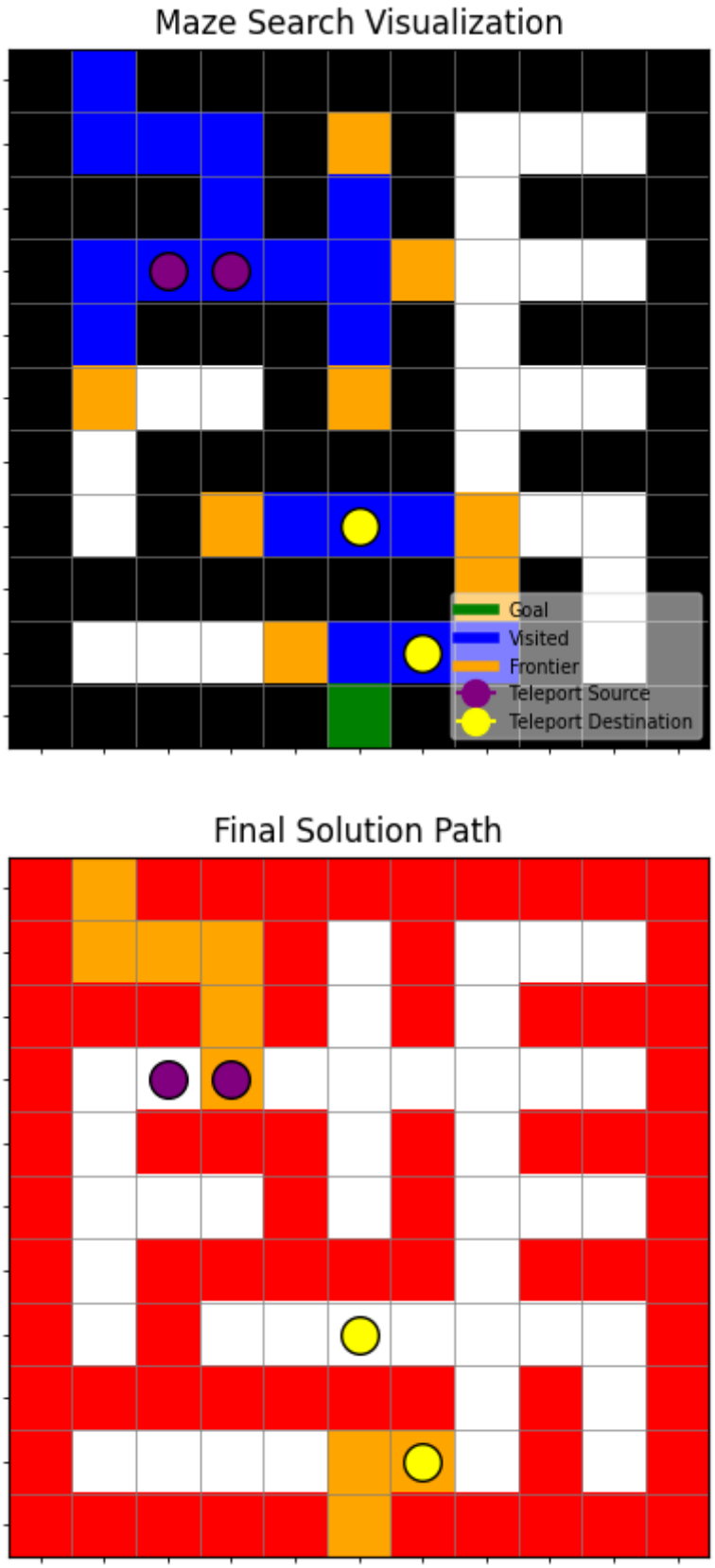

Maze Search Visualization
Goal
Visited
Frontier
Teleport Source
Teleport Destination


Maze Search Visualization
Goal
Visited
Frontier
Teleport Source
Teleport Destination


Maze Search Visualization
Goal
Visited
Frontier
Teleport Source
Teleport Destination


Maze Search Visualization
Goal
Visited
Frontier
Teleport Source
Teleport Destination


Maze Search Visualization
Goal
Visited
Frontier
Teleport Source
Teleport Destination



Figures 8: Output from calling the BFS search function, each image represents the agents progress after 3 iterations, the final image shows the optimal path found.

The series of images in **Figure 8** illustrates the agent's systematic exploration of the maze using Breadth-First Search (BFS). These images capture the state of the search at intervals of three iterations, highlighting the progress made by the agent in finding the optimal path to the goal.

Observations and Rationalization

Observation: The agent expands nodes layer by layer, creating concentric layers of explored cells around the starting position. This is evident in the images, where the frontier (orange) progresses outward uniformly.

Rationalization: BFS's use of a **FIFO queue** ensures that all nodes at the current depth are expanded before moving to the next depth level. This property guarantees that the shortest path to the goal is found, as the algorithm prioritizes breadth over depth.

Observation: In the images, teleports are only utilized after their sources are encountered. Once a teleport source is reached, its corresponding destination becomes part of the explored frontier, allowing the agent to take a shortcut.

Rationalization: The implementation ensures that teleport destinations are dynamically discovered via the **discover_teleport** method. This simulates real-world constraints, where the agent cannot pre-emptively use teleports it has not encountered. Teleports significantly reduce exploration overhead when they provide shortcuts close to the goal.

Observation: Early in the search process, the agent predominantly uses movement actions (up, down, left, right) to explore adjacent cells. Teleports are incorporated later as they are discovered.

Rationalization: Movement actions dominate the initial exploration because teleports are not yet part of the agent's action set. This aligns with BFS's systematic approach, where all immediate neighbours are explored before considering distant nodes or shortcuts.

Observation: The final image in **Figure 8** shows the shortest path from the start to the goal, highlighted in yellow. This path leverages teleports, where beneficial, minimizing the number of steps.

Rationalization: BFS guarantees optimality in unweighted graphs, such as this maze. By systematically exploring all paths at each depth level, it ensures the shortest path is found, even in the presence of teleports.

Observation: The blue cells (explored nodes) steadily increase, while the orange cells (frontier) mark the boundary of the current exploration depth. The images show how the frontier narrows as the agent approaches the goal.

Rationalization: BFS's level-order traversal ensures a gradual expansion of the explored set. The frontier narrows when the goal lies in a localized region or when teleports provide direct access to deeper parts of the maze.

Task 2: Effect of the maze size on the number of steps to goal

In this task, I analysed how increasing maze sizes impact the performance of my Breadth-First Search (BFS) implementation in finding an optimal path. Specifically, the study examines the relationship between maze size (n x n) and the number of steps required for BFS to reach the goal state. The experiment maintains a fixed number of teleportation links (t=2) to evaluate how the algorithm scales as the complexity of the maze increases.

```python
"""Task 2"""

N_list = [10, 15, 20, 25, 30]
T_2 = 2

steps = []

for n in N_list:
    m_n, tp = gen_maze_with_teleports(n, T_2)
    problem = MazeProblem(m_n.start, m_n.end, m_n, teleport_links)
    _, explored = breadth_first_search(problem, visualize=False)
    steps.append(get_len_explored(explored))

plt.figure(figsize=(10, 6))
plt.plot(N_list, steps, marker='o')
plt.title("Effect of Maze Size on Number of Steps to Goal with Two Teleports")
plt.xticks(N_list)
plt.xlabel("Maze Size (n x n)")
plt.ylabel("Number of Steps to Goal")
plt.grid()
plt.show()
```

Figure 9: Implementation for task 2.

The above script iterates through a range of maze sizes (**N_list**) while keeping the number of teleports fixed (T=2). For each maze size, a new maze is generated, and the BFS algorithm is executed to determine the number of explored states. The results are shown in the plot below, with maze size on the x-axis and the number of steps to the goal on the y-axis.

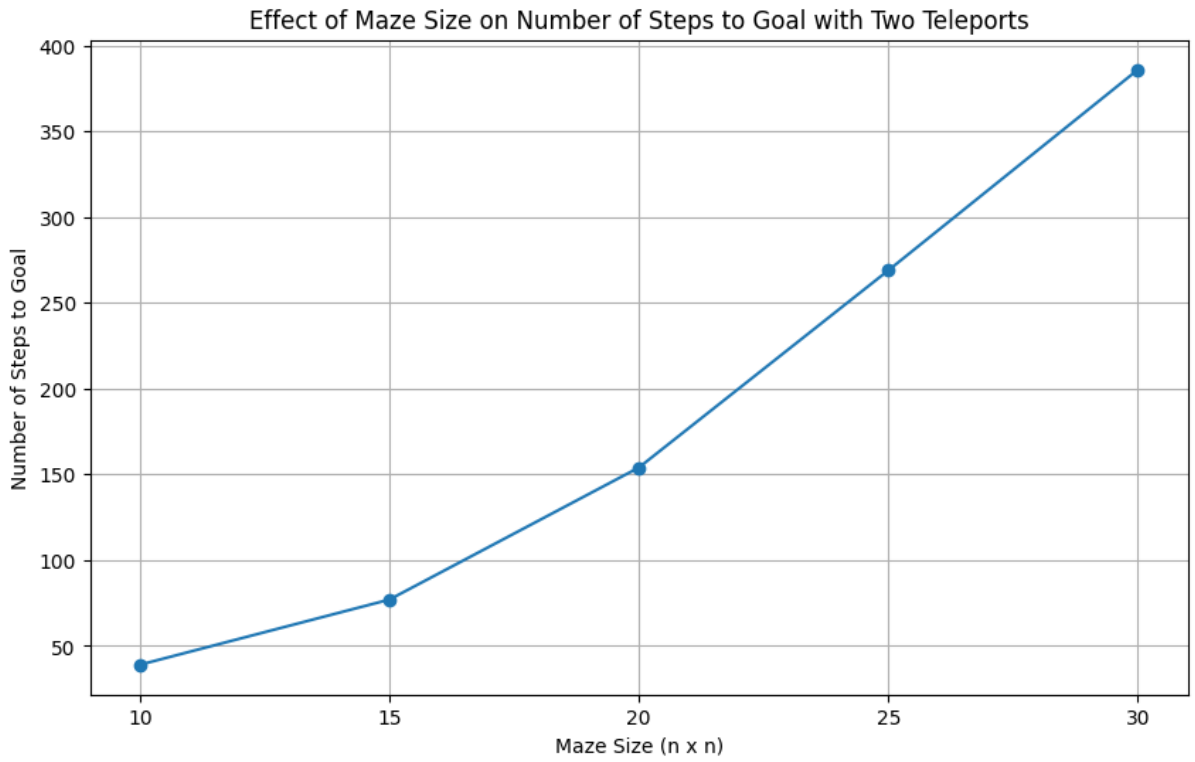


Figure 10: Visualisation of the effect of maze size on the number of steps required to reach the goal.

Insights from the Results

The results demonstrate that BFS remains effective at finding optimal paths in larger mazes but at the cost of increased steps. The linear growth reflects BFS's inherent scalability in handling larger search spaces, albeit with a proportional increase in exploration overhead.

While teleports have the potential to reduce steps, their fixed number (t=2) limits their impact as the maze size increases. In larger mazes, additional teleports or more strategic placement would likely yield a more noticeable reduction in steps.

The linear increase in steps highlights BFS's potential inefficiency in very large mazes due to its exhaustive exploration. Memory usage would also grow significantly as maze sizes increases, potentially becoming a bottleneck for n > 30.

Task 3: Effect of Teleportation Links on Steps to Goal

This task investigates whether increasing the number of teleportation links (**t**) in the maze reduces the number of steps required to find the goal using BFS. By fixing the maze size to 30x30 and varying **t** from 0 to 10, the experiment evaluates how teleports influence the agent's efficiency in navigating the maze.

```python
"""Task 3"""
n = 30
T_List = [n for n in range(0, 11)]

steps = []

for t in T_List:
    m_n, tp = gen_maze_with_teleports(n, t)
    problem = MazeProblem(m_n.start, m_n.end, m_n, teleport_links)
    _, explored = breadth_first_search(problem, visualize=False)
    steps.append(get_len_explored(explored))

plt.figure(figsize=(10, 6))
plt.plot(T_List, steps, marker='o')
plt.title("Effect of N. Teleports on Number of Steps to Goal in 30x30 Maze")
plt.xticks(T_List)
plt.xlabel("Number of Teleports")
plt.ylabel("Number of Steps to Goal")
plt.grid()
plt.show()
```

Figure 11: Implementation for task 3.

The above script follows a similar structure to the implementation used in task 2. For each value of **t**, a maze is generated with the corresponding number of teleports, and the BFS algorithm is executed to calculate the number of explored nodes required to reach the goal. The results are plotted, with the x-axis representing the number of teleports and the y-axis representing the steps taken to reach the goal.

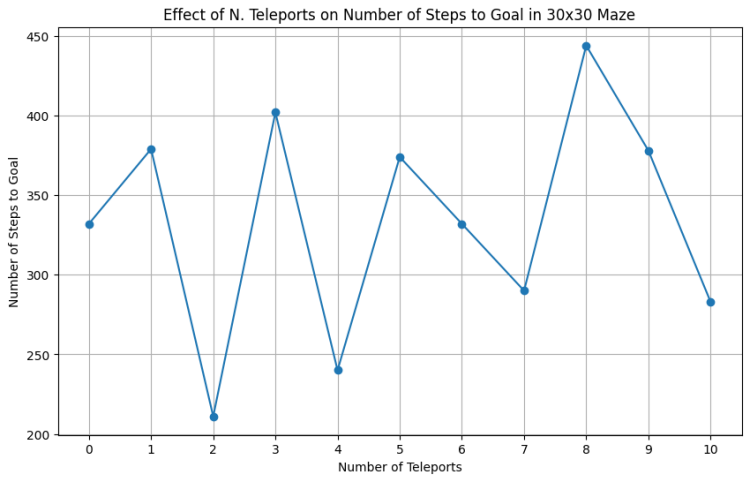


Figure 12: Visualisation of the effect of maze size on the number of steps required to reach the goal.

Insights from the Results

- At certain points (e.g. t=2), the number of steps drops significantly. This is likely because the teleports provide efficient shortcuts. When teleports are positioned close to the start or goal, they reduce the physical distance the agent must traverse, thereby reducing the number of steps. By bypassing large portions of the maze, teleports can effectively reduce the search space needed for BFS. This will be particularly impactful when the teleport leads directly to regions near the goal.

- At some values of **t** (e.g. t=8), the number of steps increases despite more teleports being available. Poorly placed teleports that lead to distant or irrelevant areas of the maze can misdirect BFS. This increases the number of explored nodes as BFS systematically evaluates these unhelpful branches. Furthermore, adding more teleports increases the number of potential paths BFS needs to evaluate. Without a heuristic to prioritise teleports leading closer to the goal, BFS treats all teleports equally, leading to unnecessary exploration.

- The graph exhibits significant variability, with sharp peaks and troughs as t increases. The randomness in teleport generation means some configurations are naturally more effective that others. For example:

    - Troughs: Occur when teleports are well-positioned to create direct shortcuts to the goal.

    - Peaks: Occur when teleports are either redundant or poorly placed, increasing exploration overhead.

  Additionally, the inherent structure of the maze influences how effective a teleport configuration can be. For instance, a teleport that bypasses a highly convoluted region has a greater impact than one in sparsely connected areas.

- Beyond a certain point (t > 4), adding more teleports does not consistently reduce steps and can even lead to an increase. As the number of teleports increases, many teleports may overlap in functionality or connected areas that are already easily reachable through standard movement. This redundancy limits their ability to significantly reduce the search space. BFS also does not inherently prioritise paths that are more likely to lead to the goal. As a result, additional teleports may introduce more paths to evaluate, increasing the number of steps before reaching the goal.

- BFS's completeness guarantees that all paths are explored in breadth-first order, ensuring the shortest path is found but also meaning that any additional complexity, such as poorly placed teleports, increases the number of explored nodes.

References:

[1] Artificial Intelligence: A Modern Approach, Global Edition. Peter Norvig, Stuart Russell