AI-Assignment-01

Prepared by: Abad Naseer

20i-1815

Submission Date

31 October, 2024

Report on Search Algorithms in a Dynamic Maze

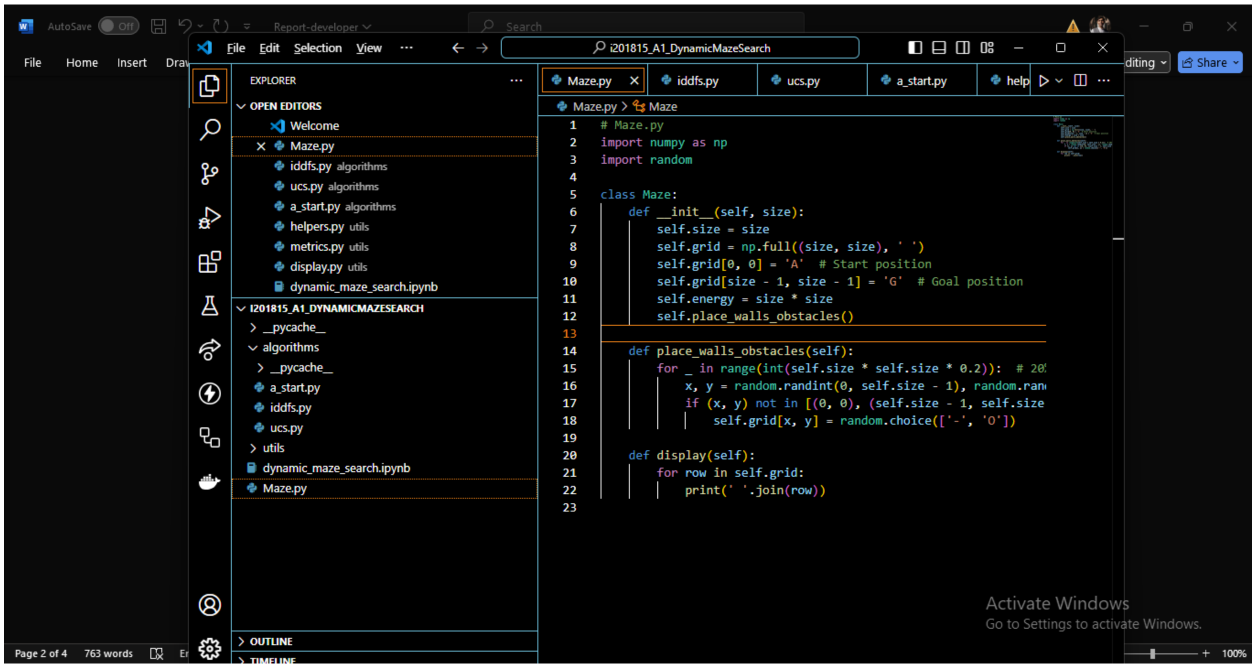
Introduction

This report covers the implementation and comparison of three search algorithms—Iterative Deepening Depth-First Search (IDDFS), Uniform Cost Search (UCS), and A* Search—in navigating a dynamic maze. The maze environment includes fixed walls, obstacles that reduce the agent's energy, and an energy constraint that must be managed as the agent finds the path to the goal.

Environment Setup

The maze grid is defined by the user, with randomly placed obstacles (o) and walls (-). The agent starts in the top-left corner (position A) and aims to reach the goal (position G) in the bottom-right. Each move consumes 1 unit of energy, while crossing obstacles costs an additional 2 energy points.

The grid and energy levels are dynamically set based on the grid size (`n x n`). This generic setup ensures the maze can adapt to different grid sizes while retaining consistent functionality.



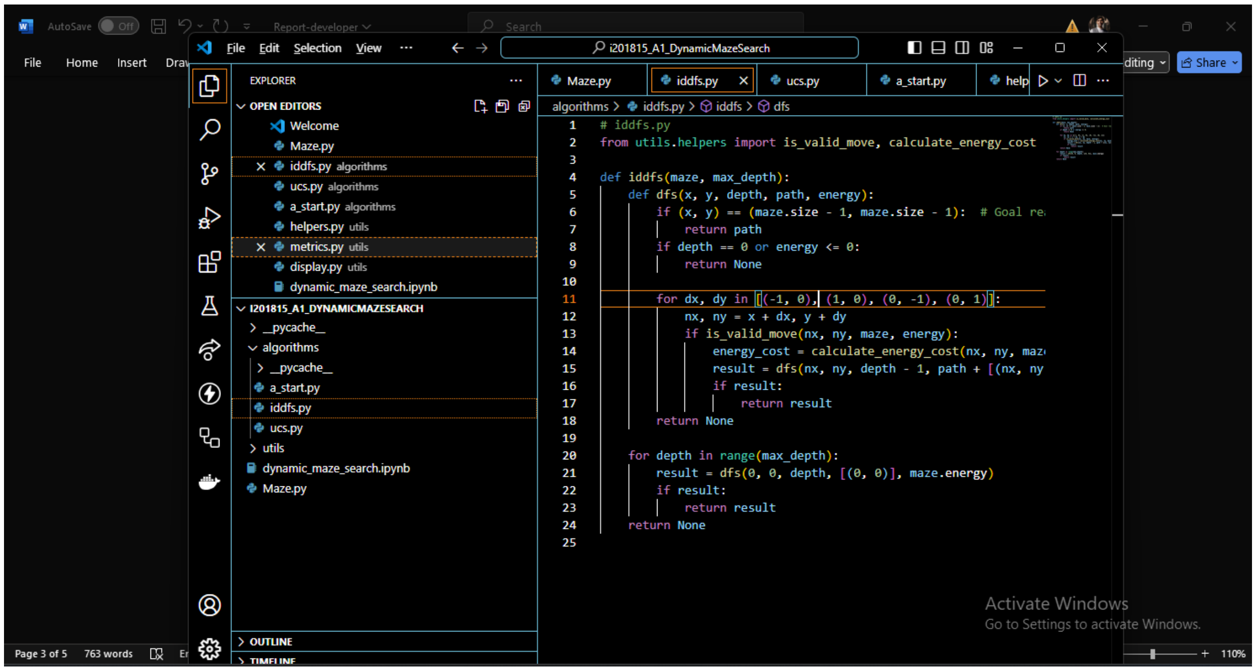
```
# Maze.py
import numpy as np
import random

class Maze:
    def __init__(self, size):
        self.size = size
        self.grid = np.full((size, size), ' ')
        self.grid[0, 0] = 'A'  # Start position
        self.grid[size - 1, size - 1] = 'G'  # Goal position
        self.energy = size * size
        self.place_walls_obstacles()

    def place_walls_obstacles(self):
        for _ in range(int(self.size * self.size * 0.2)):  # 20
            x, y = random.randint(0, self.size - 1), random.ran
            if (x, y) not in [(0, 0), (self.size - 1, self.size
                self.grid[x, y] = random.choice(['-', 'O'])

    def display(self):
        for row in self.grid:
            print(' '.join(row))
```

Part 1: Algorithm Implementation

Iterative Deepening Depth-First Search (IDDFS)

IDDFS is a depth-limited search where we increase the depth limit incrementally until a path is found or the maximum depth is reached. This algorithm works well with minimal memory usage but has a risk of exhausting energy before reaching the goal, especially in mazes with many obstacles.

- **Advantages**: Low memory requirements and effective in small mazes.
- **Disadvantages**: Often explores the same nodes multiple times, leading to high energy consumption in larger grids or complex mazes.

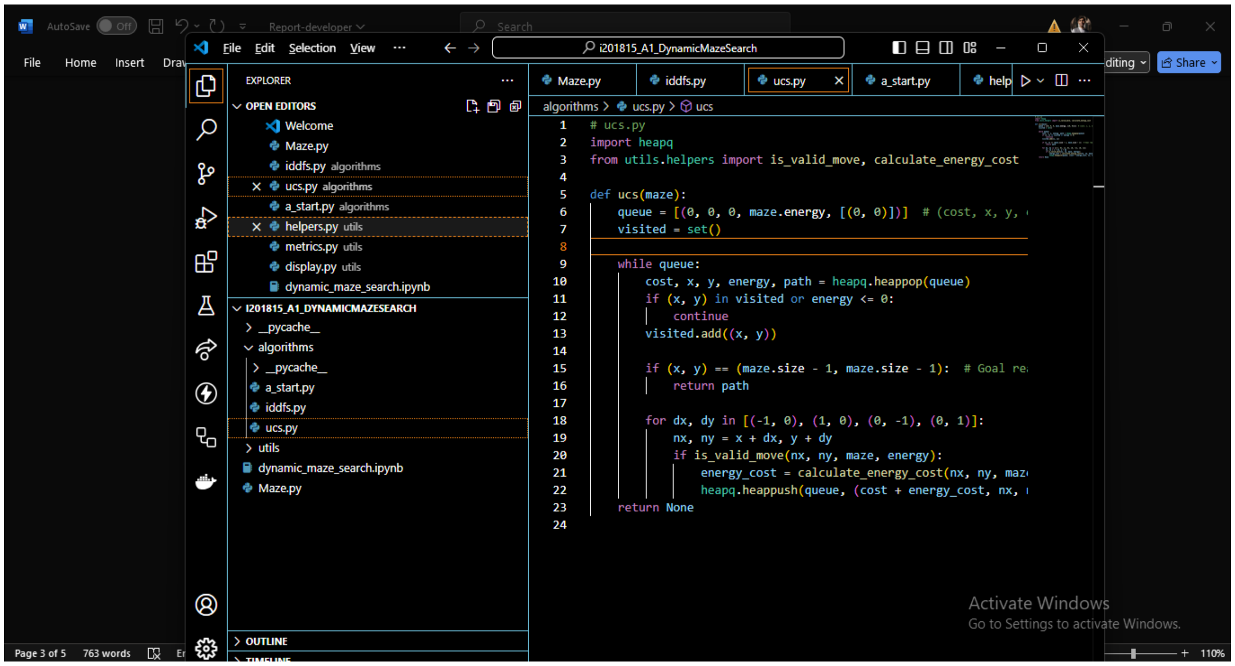


```python
# iddfs.py
from utils.helpers import is_valid_move, calculate_energy_cost

def iddfs(maze, max_depth):
    def dfs(x, y, depth, path, energy):
        if (x, y) == (maze.size - 1, maze.size - 1):  # Goal re
            return path
        if depth == 0 or energy <= 0:
            return None

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if is_valid_move(nx, ny, maze, energy):
                energy_cost = calculate_energy_cost(nx, ny, maz
                result = dfs(nx, ny, depth - 1, path + [(nx, ny
                if result:
                    return result
        return None

    for depth in range(max_depth):
        result = dfs(0, 0, depth, [(0, 0)], maze.energy)
        if result:
            return result
    return None
```

Uniform Cost Search (UCS)

UCS expands nodes based on cumulative cost, ensuring the path with the least energy consumption is found. It is optimal for energy management, as it prioritizes paths with fewer obstacles or shorter distances.

- **Advantages**: Finds the most energy-efficient path due to its cumulative cost-based expansion.
- **Disadvantages**: Can be slow due to exploring all potential paths, leading to increased node exploration and memory usage.

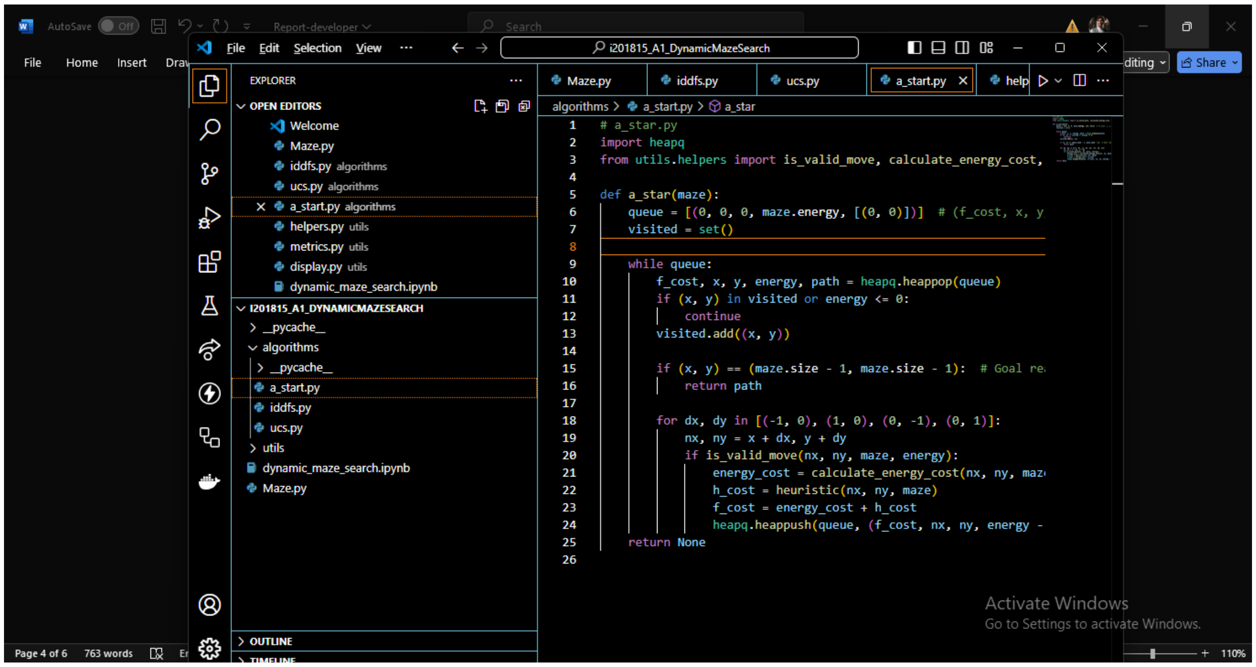


```python
# ucs.py
import heapq
from utils.helpers import is_valid_move, calculate_energy_cost

def ucs(maze):
    queue = [(0, 0, 0, maze.energy, [(0, 0)])]  # (cost, x, y,
    visited = set()

    while queue:
        cost, x, y, energy, path = heapq.heappop(queue)
        if (x, y) in visited or energy <= 0:
            continue
        visited.add((x, y))

        if (x, y) == (maze.size - 1, maze.size - 1):  # Goal re
            return path

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if is_valid_move(nx, ny, maze, energy):
                energy_cost = calculate_energy_cost(nx, ny, maz
                heapq.heappush(queue, (cost + energy_cost, nx,
    return None
```

A* Search

A* combines the actual cost from the start node with a heuristic that estimates the cost to reach the goal. Here, we used the Manhattan distance as the heuristic, which helped guide the agent towards the goal while minimizing energy consumption.

- **Advantages**: Balances between finding an optimal path and reducing search time through the heuristic.
- **Disadvantages**: Dependent on the heuristic's accuracy; may explore more nodes if the heuristic is not ideal.



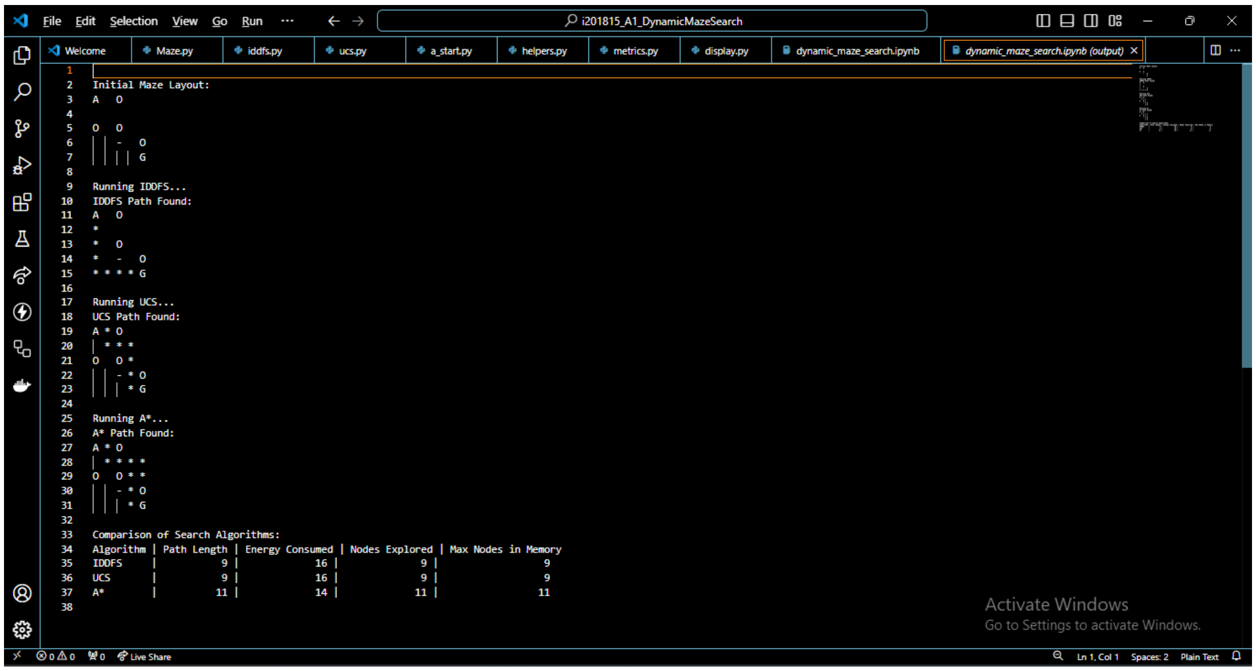
```python
# a_star.py
import heapq
from utils.helpers import is_valid_move, calculate_energy_cost,

def a_star(maze):
    queue = [(0, 0, 0, maze.energy, [(0, 0)])]  # (f_cost, x, y
    visited = set()

    while queue:
        f_cost, x, y, energy, path = heapq.heappop(queue)
        if (x, y) in visited or energy <= 0:
            continue
        visited.add((x, y))

        if (x, y) == (maze.size - 1, maze.size - 1):  # Goal re
            return path

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if is_valid_move(nx, ny, maze, energy):
                energy_cost = calculate_energy_cost(nx, ny, maz
                h_cost = heuristic(nx, ny, maze)
                f_cost = energy_cost + h_cost
                heapq.heappush(queue, (f_cost, nx, ny, energy -
    return None

```

Part 2: Implementation Overview

The main code was structured as follows:

- **Maze Class**: Set up and manages the maze, including displaying the grid layout.
- **Search Algorithms**: Functions for IDDFS, UCS, and A* to find paths and calculate energy consumption.
- **Utilities**: Display and metrics utilities for visualizing paths and evaluating each algorithm's performance.

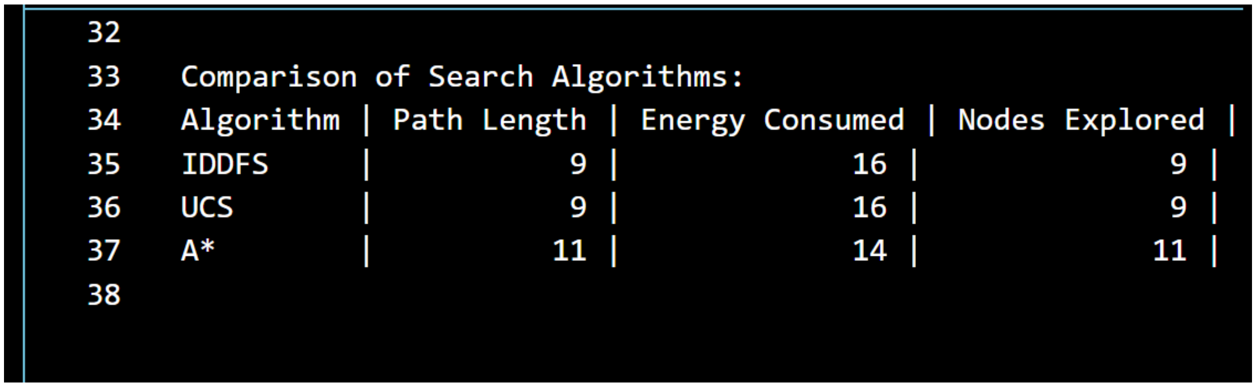
Sample output for each algorithm was visualized using `display_path`, where paths were shown using asterisks (*) in the maze. The metrics were calculated to assess efficiency in terms of path length, energy, nodes explored, and memory used.



```

Initial Maze Layout:
A   O

O   O
| |  -   O
| | | |  G

Running IDDFS...
IDDFS Path Found:
A   O
*
*   O
*   -   O
* * * * G

Running UCS...
UCS Path Found:
A * O
| * * *
O   O *
| |  - * O
| | | * G

Running A*...
A* Path Found:
A * O
| * * * *
O   O * *
| |  - * O
| | | * G

Comparison of Search Algorithms:
Algorithm | Path Length | Energy Consumed | Nodes Explored | Max Nodes in Memory
IDDFS     |           9 |              16 |              9 |                   9
UCS       |           9 |              16 |              9 |                   9
A*        |          11 |              14 |             11 |                  11

```

Part 3: Evaluation and Comparison

The following table summarizes the comparative results:



```

Comparison of Search Algorithms:
Algorithm | Path Length | Energy Consumed | Nodes Explored |
IDDFS     |           9 |              16 |              9 |
UCS       |           9 |              16 |              9 |
A*        |          11 |              14 |             11 |

```

Observations

- **IDDFS** explored more nodes in general due to its iterative nature but had lower memory usage.
- **UCS** was effective in finding the path with the least energy consumption, though it required more nodes in memory due to tracking cumulative costs.
- **A*** demonstrated a balance between energy efficiency and pathfinding speed, benefiting from the heuristic to reduce node exploration.

Part 4: Edge Cases

The program handled several edge cases:

1. **No Solution**: In cases where obstacles or walls entirely block the goal, each algorithm returned an appropriate message indicating no path found.
2. **Low Energy**: If the agent's initial energy was insufficient to reach the goal, the algorithms terminated early without wasting additional resources.

Conclusion

This assignment demonstrated the unique advantages and limitations of IDDFS, UCS, and A* in a dynamic maze environment. The choice of algorithm can vary based on the maze structure and energy constraints. UCS and A* were preferable for energy-efficient pathfinding, while IDDFS worked well with simpler, smaller mazes.

Screenshots and Code Explanation

Screenshots of the output were generated for each algorithm run to visually represent the paths. Below are explanations of core code sections:

- **Maze Class**: Manages grid layout, initializes walls and obstacles, and provides methods for displaying the maze.
- **Search Functions**: Implemented logic for each search algorithm, handling energy constraints and movement directions.
- **Metrics Calculation**: Calculates and displays metrics for each algorithm, aiding in quantitative comparison.

This assignment underscored the trade-offs between memory usage, search time, and energy management in pathfinding algorithms.