Exploring Search Algorithms in Python

Introduction

This report explores the implementation and evaluation of three fundamental search algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), and A* (A-star). These algorithms are implemented within a modular PathFinding class that encapsulates all core functionalities. Applied to a grid-based maze environment representing a graph with start and goal positions, the task involves finding the shortest path while evaluating each algorithm's performance in terms of runtime, memory usage, and nodes expanded.

Code Implementation Details

This implementation consists of three main Python files that work together to solve the pathfinding problem. The `init.py` file serves as the entry point and orchestrator, where it defines the grid structure (a 2D list representation) and creates instances of the pathfinding algorithms. The core logic is in the `pathfinding.py` file, which contains the `PathFinding` class implementing three different search algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), and A* Search. Each algorithm contains performance metrics tracking execution time, memory usage, and nodes expanded. The `visualization.py` file handles the visual representation of the results, using `matplotlib` to create a color-coded grid where start (green), goal (red), obstacles (black), and the found path (blue) are clearly distinguished. The files interact through method calls: `init.py` creates a `PathFinding` instance with the grid, calls the search methods, and passes their resulting paths to the visualization function. Each search algorithm returns a path as a list of coordinates, which the visualizer then transforms into a colored grid representation. The code includes performance monitoring, real-time node expansion tracking, and the ability to compare different algorithms and heuristics through the `compare_algorithms` function.

The project also contains an independent `test_pathfinding.py` file that serves as a testing and performance analysis framework for the pathfinding implementation. It creates two different test environments: a sparse grid (with few obstacles) and a dense grid (with many obstacles) to evaluate how the algorithms perform under different conditions. The file systematically tests each pathfinding algorithm (BFS, DFS, and A* with different heuristics) on both grids, collecting and comparing metrics like execution time, memory usage, and the number of nodes expanded. The results are presented in a clear, formatted output that makes it easy to compare and analyze the performance characteristics of each algorithm.

Running the Project

1. Clone the project repository
2. Run `pip install -r requirements.txt` to install all necessary dependencies
3. Run `python init.py` will show the pathfinding results using all three algorithms (BFS, DFS, and A*) along with visualizations and performance metrics
4. Run `python test_pathfinding.py` for testing dense and sparse grids

The visualizations will automatically appear in separate windows using matplotlib, showing the paths found by each algorithm with color-coded grids (green for start, red for goal, black for obstacles, and blue for the found path), and performance measurements can be found in the console.

Algorithm Implementations

1. Breadth-First Search (BFS)

Description

BFS explores the grid in a layer-by-layer manner using a queue (FIFO). It guarantees finding the shortest path in an unweighted grid.

Pseudocode

```
function BFS(start, goal, grid):
    queue ← [(start, [start])]
    visited ← set()
    visited.add(start)

    while queue is not empty:
        current, path ← queue.dequeue()

        if current == goal:
            return path

        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.enqueue((neighbor, path + [neighbor]))

    return None
```

Python Code Snippet

```
def bfs(self):
    queue = deque([(self.start, [self.start])])
        visited = {self.start}

        path = None
        while queue:
            current, path = queue.popleft()

            if current == self.goal:
                break

            for neighbor in self.get_neighbors(current):
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor, path + [neighbor]))
        return path
```

The method explores the grid by starting at the start position and systematically exploring all neighboring cells layer by layer. It uses a queue (First-In-First-Out) to keep track of nodes to visit, and a set to remember which nodes it has already seen. For each position, it checks all four adjacent cells (up, down, left, right). When it explores a cell, it also keeps track of the path taken to reach that cell. The search continues until it either finds the goal position or exhausts all possible paths. Because BFS explores all positions at the current "distance" before moving to positions that are further away, it guarantees finding the shortest possible path in an unweighted graph (where each step costs the same). The method returns the complete path once the goal is found.

2. Depth-First Search (DFS)

Description

DFS explores the grid deeply using a stack (LIFO). Unlike BFS, it may not find the shortest path but efficiently finds a valid path if one exists.

Pseudocode

```
function DFS(start, goal, grid):
    stack ← [(start, [start])]
    visited ← set()

    while stack is not empty:
        current, path ← stack.pop()
```

```
        if current == goal:
            return path

        if current not in visited:
            visited.add(current)
            for neighbor in get_neighbors(current):
                stack.push((neighbor, path + [neighbor]))

    return None
```

Python Code Snippet

```
def dfs(self):
    stack = [(self.start, [self.start])]
        visited = set()

        path = None
        while stack:
            current, path = stack.pop()

            if current == self.goal:
                break

            if current not in visited:
                visited.add(current)
                for neighbor in self.get_neighbors(current):
                    if neighbor not in visited:
                        stack.append((neighbor, path + [neighbor]))
        return path
```

The Depth-First Search (DFS) method explores paths by going as deep as possible before backtracking. It starts at the `self.start` position and uses a stack to keep track of nodes to visit. For each node, it explores one of its unvisited neighbors completely (including that neighbor's neighbors) before moving on to other neighbors. The method maintains a visited set to avoid cycles, and for each node, it stores both the current position and the path taken to reach it. While DFS will successfully find a path to the goal if one exists, it doesn't guarantee finding the shortest path (unlike BFS or A) because it simply takes the first complete path it discovers to the goal. The method returns the complete path once the goal is found.

3. A* Search

Description

A* uses a priority queue to explore the most promising nodes first. It combines the cost of the path so far (g-score) with a heuristic estimate of the remaining cost (h-score).

Pseudocode

```
function A*(start, goal, grid, heuristic):
    pq ← priority_queue()
    pq.insert((0, start, [start]))
    g_score[start] ← 0
    visited ← set()

    while pq is not empty:
        _, current, path ← pq.pop()

        if current == goal:
            return path

        if current not in visited:
            visited.add(current)
            for neighbor in get_neighbors(current):
                tentative_g = g_score[current] + 1
                if neighbor not in g_score or tentative_g <
g_score[neighbor]:
                    g_score[neighbor] = tentative_g
                    f_score = tentative_g + heuristic(neighbor, goal)
                    pq.insert((f_score, neighbor, path + [neighbor]))

    return None
```

Python Code Snippet

```
def astar(self):
   pq = [(0, self.start, [self.start])]
        visited = set()
        g_score = {self.start: 0}

        path = None
        while pq:
            _, current, path = heapq.heappop(pq)

            if current == self.goal:
                break
```

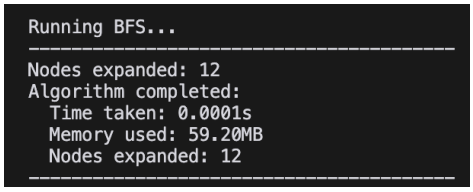
```
            if current not in visited:
                visited.add(current)
                for neighbor in self.get_neighbors(current):
                    tentative_g = g_score[current] + 1

                    if neighbor not in g_score or tentative_g <
g_score[neighbor]:
                        g_score[neighbor] = tentative_g
                        heuristic = getattr(self, 'current_heuristic',
self.manhattan_distance)
                        f_score = tentative_g + heuristic(neighbor)
                        heapq.heappush(pq, (f_score, neighbor, path +
[neighbor]))
        return path
```

The A* algorithm method in this implementation maintains a priority queue `(pq)` of nodes to explore, where each node's priority is determined by the sum of two values: `g_score` (the actual cost from start to current node) and the result of a heuristic function (either Manhattan or Euclidean distance, selected through the `current_heuristic` attribute). The algorithm uses `heapq` to repeatedly select the most promising node (lowest `f_score`), marks it as visited, and processes its unvisited neighbors using `get_neighbors()`. For each neighbor, it calculates a tentative `g_score` and updates the neighbor's score if this new path is better than any previous one. The total estimated cost (`f_score = tentative_g + heuristic(neighbor)`) determines the neighbor's position in the priority queue. The algorithm continues until either the goal is reached or all possible paths are exhausted, tracking execution time and memory usage along the way. The method returns the complete path once the goal is found.

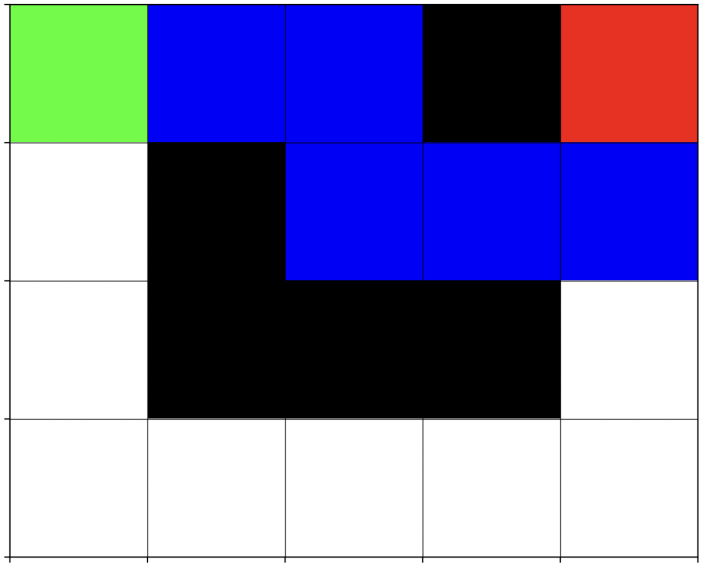
Performance Observations

Init.py Performance Metrics

Performance was measured in terms of runtime, memory usage, and nodes expanded. Sparse and dense grids were used for analysis. A sample `init.py` run output is shown below.

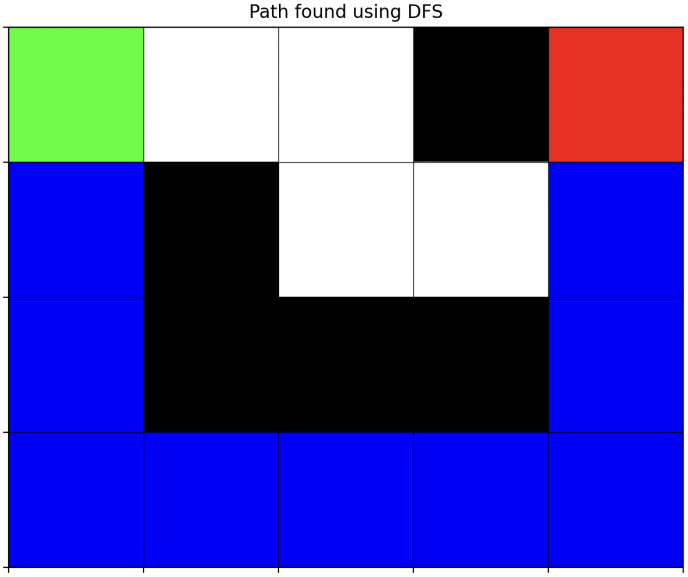


Path found using BFS

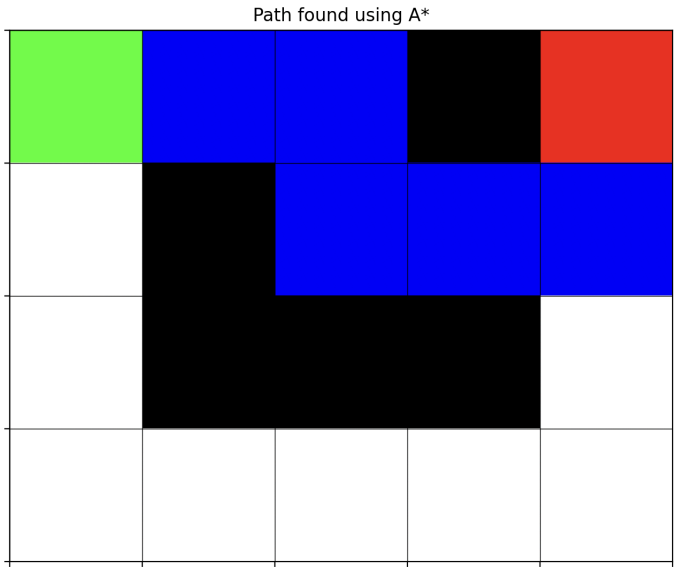


```
Running DFS...
----------------------------------------
Nodes expanded: 10
Algorithm completed:
  Time taken: 0.0001s
  Memory used: 221.20MB
  Nodes expanded: 10
----------------------------------------
```

Path found using DFS



```
Running ASTAR...
----------------------------------------
Nodes expanded: 7
Algorithm completed:
  Time taken: 0.0001s
  Memory used: 280.86MB
  Nodes expanded: 7
----------------------------------------
```




```
Metrics Summary:

Algorithm: BFS
  Execution Time: 0.000054 seconds
  Memory Usage: 59.20 MB
  Nodes Expanded: 12
  Path: [(0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (1, 4), (0, 4)]

Algorithm: DFS
  Execution Time: 0.000075 seconds
  Memory Usage: 221.20 MB
  Nodes Expanded: 10
  Path: [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3), (3, 4), (2, 4), (1, 4), (0, 4)]

Algorithm: A*
  Execution Time: 0.000075 seconds
  Memory Usage: 280.86 MB
  Nodes Expanded: 7
  Path: [(0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (1, 4), (0, 4)]
```

Conclusion

This 2D maze is a uniform-cost grid where each step has a weight of 1. In this environment, **BFS** guarantees the shortest path by exploring nodes layer-by-layer with minimal memory usage `(59.20 MB)` but expands more nodes `(12)` than **A***. **DFS** consumes significantly higher memory `(221.20 MB)`, expands fewer nodes `(10)`, but does not guarantee the shortest path due to its depth-first approach. **A*** uses heuristics to find the optimal path while expanding the fewest nodes `(7)`, but it requires the highest memory usage `(280.86 MB)` because it manages a priority queue, heuristic values, and g-scores. In uniform-cost grids, **BFS** and **A*** often yield similar results since edge weights are constant and heuristics offer limited additional advantage. However, **A*** shows its true power in weighted scenarios, where varying edge costs make **BFS** unsuitable for guaranteeing optimal paths. By combining edge costs and heuristic estimates, **A*** can intelligently prioritize nodes closer to the goal, significantly reducing unnecessary exploration and ensuring optimal paths with fewer node expansions, provided the heuristic is well-suited to the grid structure.

Test_pathfinding.py Performance Metrics

Sparse Grid Performance

Algorithm	Time (s)	Memory (MB)	Nodes Expanded
BFS	0.0002	22.73	96
DFS	0.0004	22.91	94
A* (Manhattan)	0.0006	22.94	96
A* (Euclidean)	0.0028	23.03	96

Dense Grid Performance

Algorithm	Time (s)	Memory (MB)	Nodes Expanded
BFS	0.0000	23.03	3
DFS	0.0000	23.06	3
A* (Manhattan)	0.0001	23.06	3
A* (Euclidean)	0.0002	23.06	3

Efficiency Comparison (Dense vs Sparse)

Algorithm	Dense/Sparse Node Expansion Ratio
BFS	0.03
DFS	0.03
A* (Manhattan)	0.03
A* (Euclidean)	0.03

Key Observations

In the sparse grid, BFS and A* expanded the same number of nodes (96), while DFS expanded slightly fewer (94). A* (Euclidean) took the longest time but consumed comparable memory to other algorithms. In the dense grid, all algorithms expanded only 3 nodes due to restricted paths, leading to minimal performance differences. Overall, BFS and A* are equally efficient in sparse grids, but A* demonstrates its strength in leveraging heuristics for weighted or complex scenarios. Dense grids inherently simplify the search process for all algorithms.

Conclusions

Based on the implemented pathfinding algorithms (BFS, DFS, and A*), each excels in specific scenarios. BFS is ideal for finding the shortest path in uniform-cost grids and performs well in sparse grids with few obstacles, though it can be memory-intensive in large grids. DFS, while using less memory, does not guarantee the shortest path, making it suitable for quickly finding any valid path. A*, with appropriate heuristics (e.g., Manhattan or Euclidean), combines efficiency and optimality, making it the best choice for most practical scenarios. It excels in dense grids with many obstacles, where heuristic-driven exploration reduces unnecessary node expansions and ensures the shortest path. Overall, the best algorithm depends on the environment, with BFS being reliable for unweighted grids and A* standing out in weighted or complex scenarios.