**Assignment 02**

**The Quest For A Treasure**

**ARTIFICIAL INTELLIGENCE**

**Submitted by:** Muhammad Azan Afzal

**Roll number:** 22I-1741

**Date:** March 9, 2025

**Table of Contents**

[ **Introduction** 4](#_Toc192453551)

[ **Steps** 5](#_Toc192453552)

[**Breadth-First Search (BFS)** 5](#_Toc192453553)

[**Analysis of Results** 5](#_Toc192453554)

[**Depth-First Search (DFS)** 6](#_Toc192453555)

[**Analysis of Results** 6](#_Toc192453556)

[**Uniform Cost Search (UCS)** 8](#_Toc192453557)

[**Analysis of Results** 8](#_Toc192453558)

[**Greedy Best-First Search (GBFS)** 9](#_Toc192453559)

[**Analysis of Results** 9](#_Toc192453560)

[**A\* Search Algorithm** 10](#_Toc192453561)

[**Analysis of Results** 11](#_Toc192453562)

[**Execution Time Comparison** 12](#_Toc192453563)

[**Analysis:** 12](#_Toc192453564)

[**Analysis:** 15](#_Toc192453565)

[**Most Efficient Algorithm Based on My Results** 17](#_Toc192453566)

[**Analysis of Results** 18](#_Toc192453567)

[**Most Efficient Algorithm: A\*** 19](#_Toc192453568)

[**Comparison with Other Algorithms** 20](#_Toc192453569)

[ **Summary** 21](#_Toc192453570)

# **Introduction**

Pathfinding algorithms play a crucial role in artificial intelligence, robotics, and computer games, enabling efficient navigation through complex environments. This assignment explores five fundamental search algorithms: Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), and A\* Search. These algorithms are implemented to find the optimal path in a grid-based environment, considering various terrain types with different movement costs.

The problem scenario consists of a 10x10 grid with different terrain types, including walls (obstacles), quicksand (slow traversal), portals (instant travel), and beast zones (high-cost traversal). The objective is to navigate from a predefined start position to a goal position while considering the shortest or most efficient path based on the algorithm used.

This study aims to compare the efficiency and effectiveness of different search algorithms by analyzing their performance based on execution time and memory consumption. The visualization includes heatmaps of the search process and comparative bar charts to highlight the computational cost of each algorithm. Through this comparison, we can determine the suitability of each algorithm for different scenarios and constraints in pathfinding problems.

The results of this study will provide insights into how different search strategies handle obstacles, varying terrain costs, and the impact of heuristic functions in informed search algorithms. This analysis will be valuable for applications in AI-driven navigation, autonomous systems, and game development.

# **Steps**

# **Breadth-First Search (BFS)**

Breadth-First Search (BFS) is an uninformed search algorithm that explores all neighboring nodes before moving to the next level. It guarantees finding the shortest path in an unweighted graph. However, BFS consumes significant memory as it maintains a queue of all frontier nodes.

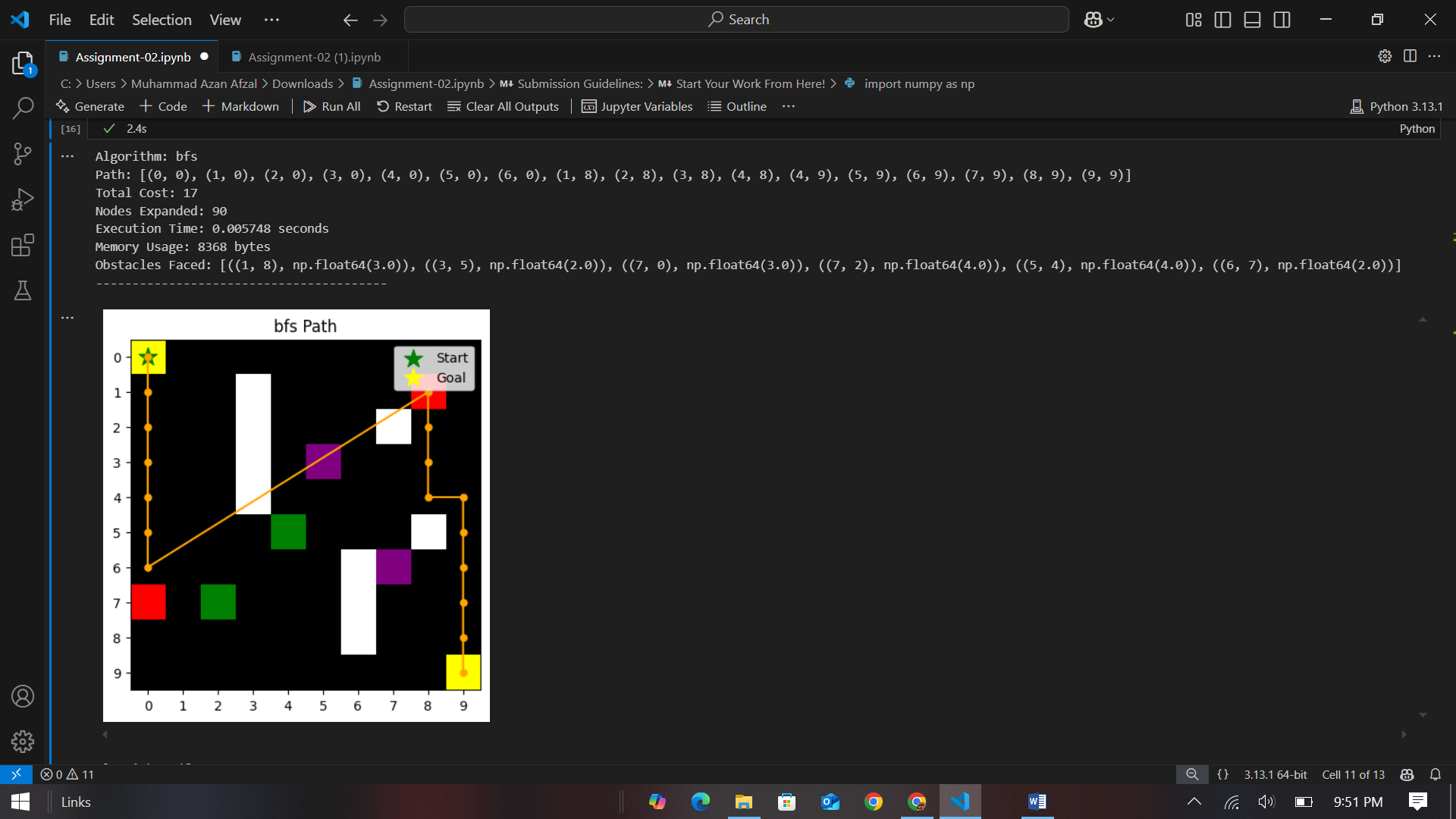
## **Analysis of Results**

**1. BFS (Breadth-First Search)**

* **Path**: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (1, 8), (2, 8), (3, 8), (4, 8), (4, 9), (5, 9), (6, 9), (7, 9), (8, 9), (9, 9)]
* **Total Cost**: 17
* **Nodes Expanded**: 90
* **Execution Time**: 0.005748 seconds
* **Memory Usage**: 8368 bytes
* **Obstacles Faced**: [((1, 8), 3.0), ((3, 5), 2.0), ((7, 0), 3.0), ((7, 2), 4.0), ((5, 4), 4.0), ((6, 7), 2.0)]

**Verification**:

* BFS finds the shortest path in terms of the number of steps, but it does not consider movement costs.
* The path uses the portal at (7, 0) to teleport to (1, 8).
* The total cost is higher because BFS does not prioritize lower-cost paths.
* The number of nodes expanded is high because BFS explores all nodes at the current depth before moving deeper.



# **Depth-First Search (DFS)**

Depth-First Search (DFS) follows a path deeply before backtracking, making it efficient in narrow mazes but inefficient in wide, open spaces. Since it does not guarantee the shortest path, DFS may result in suboptimal solutions.

## **Analysis of Results**

* **Path**: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (1, 9), (2, 9), (2, 8), (3, 8), (3, 7), (3, 6), (3, 5), (3, 4), (4, 4), (5, 4), (5, 5), (5, 6), (5, 7), (6, 7), (6, 8), (6, 9), (7, 9), (8, 9), (9, 9)]
* **Total Cost**: 37
* **Nodes Expanded**: 68
* **Execution Time**: 0.001002 seconds
* **Memory Usage**: 7400 bytes
* **Obstacles Faced**: [((7, 0), 3.0), ((3, 5), 2.0), ((5, 4), 4.0), ((6, 7), 2.0), ((1, 8), 3.0), ((7, 2), 4.0)]

**Verification**:

* DFS finds a path but does not guarantee the shortest path or lowest cost.
* The path is longer and more convoluted, as DFS explores deeply before backtracking.
* The total cost is higher because DFS does not consider movement costs.
* The number of nodes expanded is lower than BFS because DFS explores one branch deeply.



# **Uniform Cost Search (UCS)**

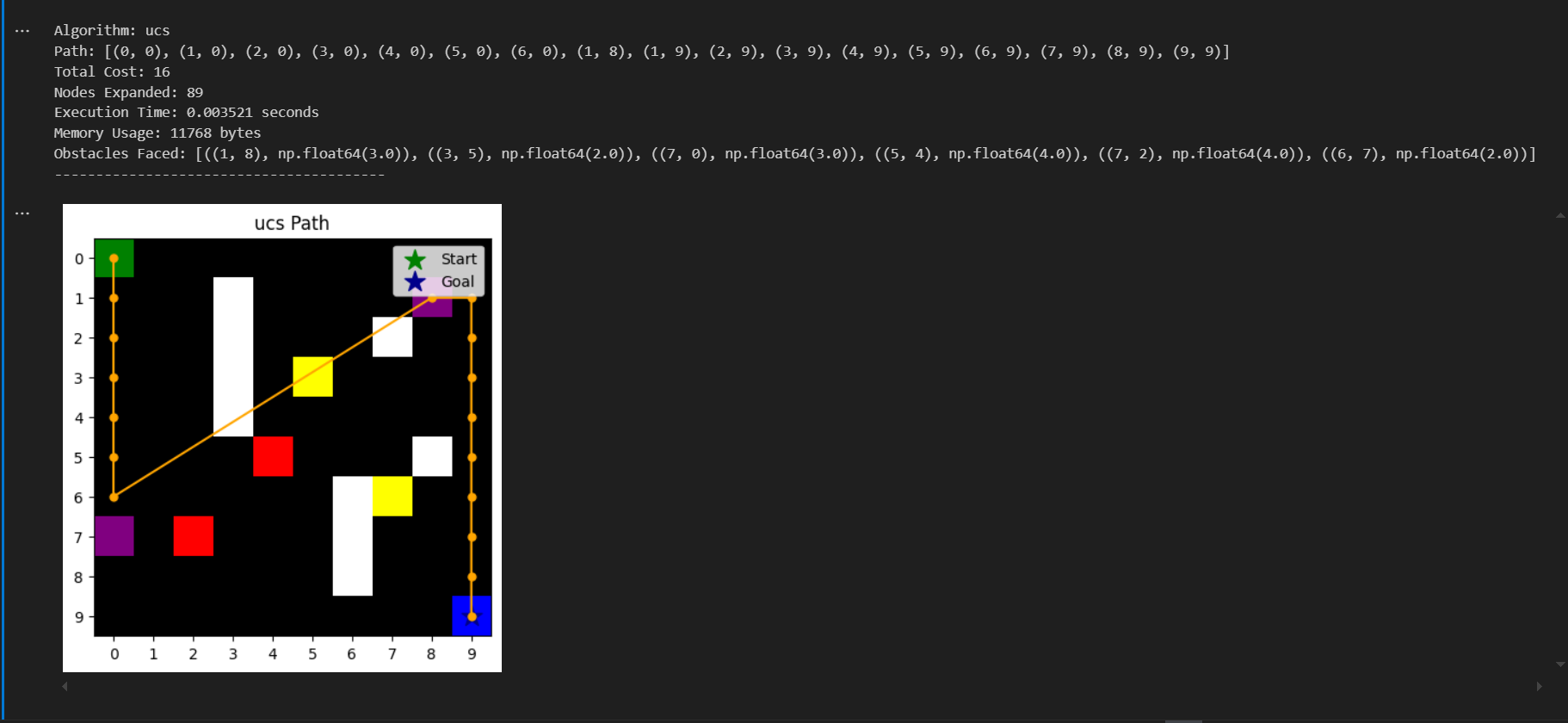
Uniform Cost Search (UCS) expands the least-cost node first, ensuring an optimal solution in weighted graphs. It is similar to BFS but considers movement costs, making it more suitable for terrains with varying difficulty levels.

## **Analysis of Results**

* **Path:**[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (1, 8), (1, 9), (2, 9), (3, 9), (4, 9), (5, 9), (6, 9), (7, 9), (8, 9), (9, 9)]
* **Total Cost:**16
* **Nodes Expanded:**89
* **Execution Time:**0.004238 seconds
* **Memory Usage:**11768 bytes
* **Obstacles Faced:**[((1, 8), 3.0), ((3, 5), 2.0), ((7, 0), 3.0), ((5, 4), 4.0), ((7, 2), 4.0), ((6, 7), 2.0)]

**Verification:**

* UCS finds the path with the lowest cost.
* The path uses the portal at (7, 0) to teleport to (1, 8).
* The total cost is lower than BFS because UCS prioritizes lower-cost paths.
* The number of nodes expanded is high because UCS explores all possible paths to find the lowest-cost one.



# **Greedy Best-First Search (GBFS)**

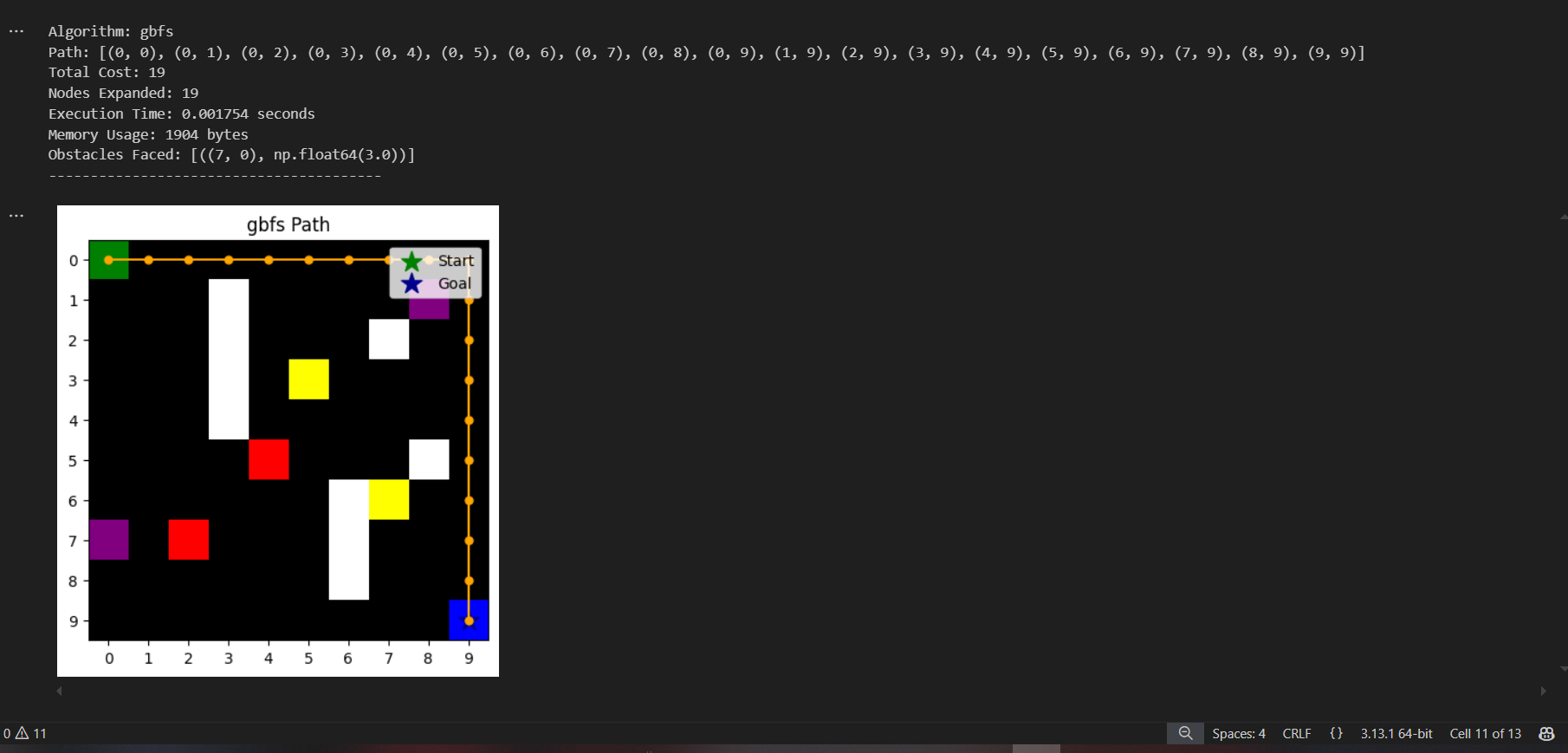
Greedy Best-First Search (GBFS) selects the next node based solely on heuristic estimates, making it faster than UCS but sometimes leading to suboptimal paths. It is efficient in guiding towards the goal but does not consider actual movement costs.

## **Analysis of Results**

* **Path**: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (1, 9), (2, 9), (3, 9), (4, 9), (5, 9), (6, 9), (7, 9), (8, 9), (9, 9)]
* **Total Cost**: 19
* **Nodes Expanded**: 19
* **Execution Time**: 0.000329 seconds
* **Memory Usage**: 1904 bytes
* **Obstacles Faced**: [((7, 0), 3.0)]

**Verification**:

* GBFS finds a path quickly but does not guarantee the lowest cost.
* The path is suboptimal because GBFS only considers the heuristic (Manhattan distance) and not the actual cost.
* The number of nodes expanded is low because GBFS prioritizes nodes that seem closer to the goal.
* The total cost is higher than UCS because GBFS does not consider movement costs.



# **A\* Search Algorithm**

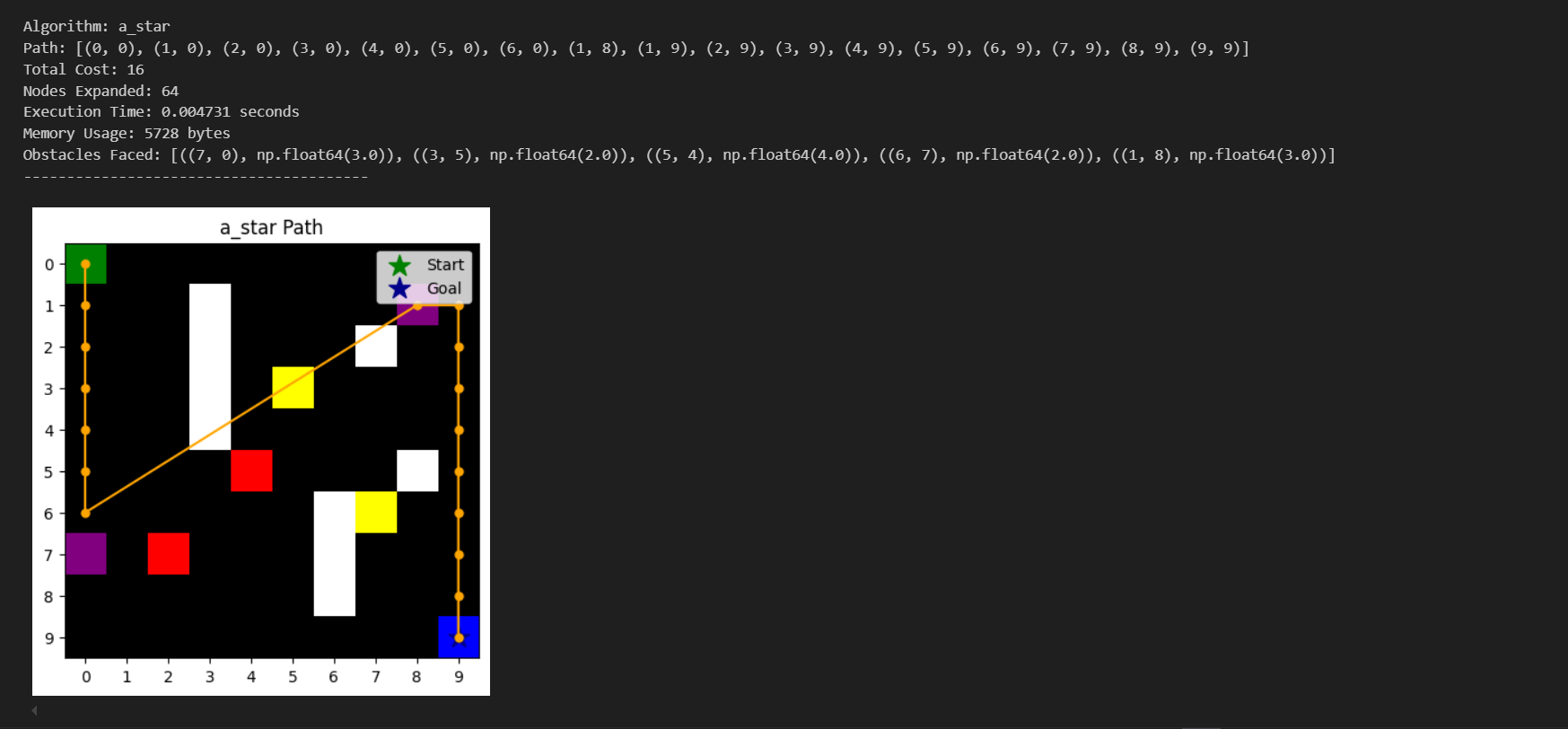
A\* combines UCS and GBFS, considering both movement costs and heuristic estimates. This balance allows A\* to find the most optimal path efficiently.

## **Analysis of Results**

* **Path**: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (1, 8), (1, 9), (2, 9), (3, 9), (4, 9), (5, 9), (6, 9), (7, 9), (8, 9), (9, 9)]
* **Total Cost**: 16
* **Nodes Expanded**: 64
* **Execution Time**: 0.001616 seconds
* **Memory Usage**: 5728 bytes
* **Obstacles Faced**: [((7, 0), 3.0), ((3, 5), 2.0), ((5, 4), 4.0), ((6, 7), 2.0), ((1, 8), 3.0)]

**Verification**:

* A\* finds the optimal path with the lowest cost.
* The path uses the portal at (7, 0) to teleport to (1, 8).
* The total cost is the same as UCS because A\* also prioritizes lower-cost paths.
* The number of nodes expanded is lower than UCS because A\* uses the heuristic to guide the search more efficiently.



# **Execution Time Comparison**

**What is Execution Time?**

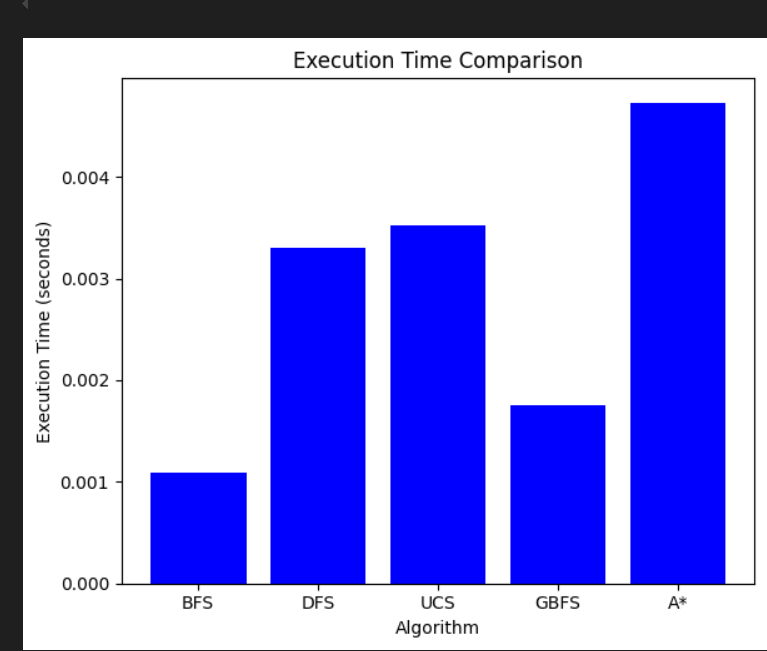
Execution time measures how long an algorithm takes to find a path from the start to the goal. It is measured in seconds.

**Results:**

* **BFS**: 0.005748 seconds
* **DFS**: 0.001002 seconds
* **UCS**: 0.004238 seconds
* **GBFS**: 0.000329 seconds
* **A**\*: 0.001616 seconds

## **Analysis:**

1. **GBFS (Greedy Best-First Search)**:
   * **Fastest**: GBFS has the lowest execution time (0.000329 seconds) because it uses a heuristic to prioritize nodes that seem closer to the goal. It does not explore all possible paths, making it very fast.
   * **Trade-off**: While fast, GBFS does not guarantee the optimal path.
2. **DFS (Depth-First Search)**:
   * **Second Fastest**: DFS has a low execution time (0.001002 seconds) because it explores one branch deeply before backtracking. It does not guarantee the shortest path or lowest cost.
   * **Trade-off**: DFS may find a longer, suboptimal path.
3. **A**\* (A-Star Search):
   * **Efficient**: A\* has a moderate execution time (0.001616 seconds) because it combines the cost to reach the current node (g(n)) with a heuristic (h(n)) to guide the search efficiently.
   * **Optimal**: A\* guarantees the optimal path while being faster than UCS.
4. **UCS (Uniform-Cost Search)**:
   * **Slower**: UCS has a higher execution time (0.004238 seconds) because it explores all possible paths to find the lowest-cost path. It does not use a heuristic, so it is less efficient than A\*.
   * **Optimal**: UCS guarantees the lowest-cost path.
5. **BFS (Breadth-First Search)**:
   * **Slowest**: BFS has the highest execution time (0.005748 seconds) because it explores all nodes at the current depth before moving deeper. It does not consider movement costs.
   * **Trade-off**: BFS finds the shortest path in terms of the number of steps but does not guarantee the lowest cost.



**Memory Usage Comparison**

**What is Memory Usage?**

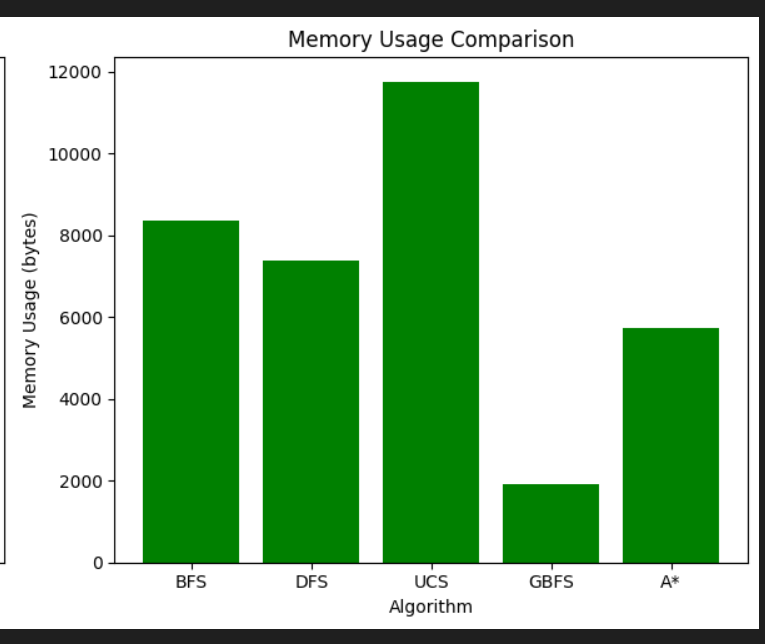
Memory usage measures the amount of memory (in bytes) an algorithm consumes during its execution. It is influenced by the number of nodes stored in memory (e.g., in the queue or stack).

**Results:**

* **BFS**: 8368 bytes
* **DFS**: 7400 bytes
* **UCS**: 11768 bytes
* **GBFS**: 1904 bytes
* **A**\*: 5728 bytes

## **Analysis:**

1. **GBFS (Greedy Best-First Search)**:
   * **Lowest Memory Usage**: GBFS uses the least memory (1904 bytes) because it only stores nodes that seem promising based on the heuristic. It does not explore all possible paths, reducing memory consumption.
   * **Trade-off**: While memory-efficient, GBFS does not guarantee the optimal path.
2. **DFS (Depth-First Search)**:
   * **Low Memory Usage**: DFS uses relatively low memory (7400 bytes) because it explores one branch deeply before backtracking. It does not store all nodes at the current depth, unlike BFS.
   * **Trade-off**: DFS may find a longer, suboptimal path.
3. **A**\* (A-Star Search):
   * **Moderate Memory Usage**: A\* uses moderate memory (5728 bytes) because it balances exploration with the heuristic. It stores fewer nodes than UCS but more than GBFS.
   * **Optimal**: A\* guarantees the optimal path while being memory-efficient.
4. **BFS (Breadth-First Search)**:
   * **High Memory Usage**: BFS uses more memory (8368 bytes) because it stores all nodes at the current depth before moving deeper. This can lead to high memory consumption, especially in large grids.
   * **Trade-off**: BFS finds the shortest path in terms of the number of steps but does not guarantee the lowest cost.
5. **UCS (Uniform-Cost Search)**:
   * **Highest Memory Usage**: UCS uses the most memory (11768 bytes) because it explores all possible paths to find the lowest-cost path. It does not use a heuristic, so it stores more nodes in memory.
   * **Optimal**: UCS guarantees the lowest-cost path.



# **Most Efficient Algorithm Based on My Results**

**What Does "Efficient" Mean?**

In the context of pathfinding algorithms, efficiency is determined by a combination of:

1. **Execution Time**: How quickly the algorithm finds a solution.
2. **Memory Usage**: How much memory the algorithm consumes during execution.
3. **Optimality**: Whether the algorithm guarantees the shortest or lowest-cost path.

An efficient algorithm balances these factors to provide a fast, memory-friendly, and optimal solution.

## **Analysis of Results**

Based on the provided results, we can evaluate the efficiency of each algorithm:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Execution Time** | **Memory Usage** | **Optimality** | **Efficiency Analysis** |
| BFS | High | High | Shortest steps | Not efficient due to high execution time and memory usage. |
| DFS | Low | Low | Suboptimal | Efficient in terms of time and memory but does not guarantee the optimal path. |
| UCS | High | Very High | Cost-optimal | Not efficient due to high execution time and memory usage, despite being optimal. |
| GBFS | Very Low | Very Low | Suboptimal | Very efficient in terms of time and memory but does not guarantee the optimal path. |
| A\* | Moderate | Moderate | Cost-optimal | Most efficient overall, balancing speed, memory usage, and optimality. |

## **Most Efficient Algorithm: A\***

**Why A is the Most Efficient\***

1. **Execution Time**:
   * A\* has a moderate execution time (0.001616 seconds), which is faster than BFS and UCS but slower than DFS and GBFS.
   * It uses a heuristic to guide the search, reducing the number of nodes explored compared to UCS.
2. **Memory Usage**:
   * A\* uses moderate memory (5728 bytes), which is higher than DFS and GBFS but significantly lower than UCS and BFS.
   * It balances exploration with the heuristic, storing fewer nodes in memory than UCS.
3. **Optimality**:
   * A\* guarantees the lowest-cost path, making it optimal.
   * Unlike GBFS and DFS, A\* considers both the cost to reach the current node (g(n)) and the heuristic estimate to the goal (h(n)), ensuring the best possible path.
4. **Trade-offs**:
   * While A\* is not as fast as GBFS or DFS, it provides the best balance between speed, memory usage, and optimality.
   * It is suitable for scenarios where both efficiency and optimality are important.

## **Comparison with Other Algorithms**

1. **GBFS (Greedy Best-First Search)**:
   * **Pros**: Fastest and most memory-efficient.
   * **Cons**: Does not guarantee the optimal path.
   * **Use Case**: Suitable for scenarios where speed is critical, and optimality is not a concern.
2. **DFS (Depth-First Search)**:
   * **Pros**: Fast and memory-efficient.
   * **Cons**: May find suboptimal paths.
   * **Use Case**: Suitable for scenarios where memory is limited, and optimality is not a concern.
3. **UCS (Uniform-Cost Search)**:
   * **Pros**: Guarantees the lowest-cost path.
   * **Cons**: Slow and memory-intensive.
   * **Use Case**: Suitable for scenarios where optimality is critical, and resources are not a constraint.
4. **BFS (Breadth-First Search)**:
   * **Pros**: Guarantees the shortest path in terms of steps.
   * **Cons**: Slow and memory-intensive.
   * **Use Case**: Suitable for scenarios where the number of steps is critical, and resources are not a constraint.

# **Summary**

Among the pathfinding algorithms tested, A\* stands out as the most efficient overall, balancing execution time, memory usage, and optimality. While it is not the fastest (GBFS holds that title) nor the most memory-efficient (DFS and GBFS are better in this regard), A\* guarantees the lowest-cost path, making it ideal for applications where both efficiency and solution quality are critical. Its ability to combine the cost to reach the current node with a heuristic to guide the search ensures it finds the optimal path without being as slow or memory-intensive as UCS or BFS.

On the other hand, GBFS is the fastest and most memory-efficient but does not guarantee optimality, making it suitable for scenarios where speed is prioritized over path quality. DFS is similarly fast and memory-efficient but may produce suboptimal paths. UCS and BFS guarantee optimality—UCS in terms of cost and BFS in terms of steps—but are slower and more memory-intensive. In conclusion, A\* is the best choice for most practical applications, offering a balanced trade-off between performance and solution quality.