Implementation and Comparative Analysis of Pathfinding Algorithms

Author:

Tanner Wagner
Jennifer Diaz
Odysseus Valdez
Jubilation Megill

8 December 2024

Contents

1 Introduction

Pathfinding algorithms are a cornerstone of computer science and artificial intelligence, finding applications in a wide range of domains such as robotics, video games, and navigation systems. This project aims to explore the implementation and comparative analysis of five fundamental pathfinding algorithms: **A***, **Breadth-First Search (BFS)**, **Depth-First Search (DFS)**, **Dijkstra's Algorithm**, and **Floyd-Warshall Algorithm**. Each of these algorithms is studied in the context of a grid-based environment with varying levels of complexity, obstacles, and goals.

The objective of this report is to:

- Provide a clear and concise explanation of each algorithm, including its formal definition, theoretical time and space complexity, and practical implementation.
- Demonstrate the performance of these algorithms across multiple test cases, each designed to highlight specific characteristics and challenges inherent to grid-based pathfinding.
- Present a comparative analysis of the algorithms, focusing on their execution time, memory usage, and behavior relative to their theoretical complexities.
- Discuss coding complexity, debugging efforts, and the observed strengths and weaknesses of each algorithm in different scenarios.

The report is organized as follows:

- **Algorithm Descriptions and Implementations:** Detailed descriptions and pseudocode for each of the five algorithms, including explanations of key concepts such as heuristics, priority queues, and adjacency matrices.
- **Performance Analysis:** A discussion of the algorithms' execution times, memory usage, and overall behavior based on the generated outputs and test cases.
- **Comparative Analysis and Insights:** Insights into why certain algorithms outperform others in specific scenarios and how theoretical complexities translate into actual performance.
- **Conclusion:** A summary of the findings, highlighting the key takeaways from the project and the practical implications of the results.
- **Team Member Acknowledgements:** A list of all team members and their individual contributions to the project as a whole.
- **Bibliography:** A list of all sources consulted during the project, including foundational texts and course materials.

Through this project, we aim to deepen our understanding of the underlying mechanics of these algorithms and their practical trade-offs, providing a comprehensive evaluation of their suitability for different types of pathfinding problems. Whether the task involves finding a single shortest path, navigating through multiple goals, or solving all-pairs shortest path problems, this report offers an in-depth exploration of the methods and their applications.

2 Tanner Wagner: A^* Algorithm

2.1 Introduction to A^*

The A algorithm is a widely used pathfinding and graph traversal algorithm that efficiently finds the shortest path between nodes in a weighted graph. It combines the strengths of Dijkstra's algorithm and Greedy Best-First Search by utilizing heuristic functions to guide its search, making it both optimal and efficient in many applications.

2.2 Formal Definition

Let $G = (V, E)$ be a finite graph, where:

- V is the set of vertices (nodes).
- E is the set of edges connecting the vertices.

Each edge $(u, v) \in E$ has an associated non-negative cost $c(u, v)$, representing the cost to move from node u to node v.

2.3 Objective

Given:

- A start node $s \in V$.
- A goal node $g \in V$.
- A heuristic function $h : V \to \mathbb{R}_{\geq 0}$ that estimates the cost from a node to the goal.

Find the path P^* from s to g that minimizes the total cost:

$$P^* = \arg\min_P \left(\sum_{i=0}^{n-1} c(v_i, v_{i+1}) \right)$$

where $P = (v_0, v_1, \ldots, v_n)$ is a path such that $v_0 = s$ and $v_n = g$.

2.4 Algorithm Description

The A algorithm maintains the following for each node v:

- $g(v)$: the cost from the start node s to node v.
- $h(v)$: the heuristic estimate from node v to the goal node g.
- $f(v) = g(v) + h(v)$: the estimated total cost of a solution path passing through v.

2.5 Algorithm Steps

1. **Initialize**
   - Set $g(s) = 0$.
   - Add s to the open set O (priority queue) ordered by f-value.
   - The closed set C is initialized as empty.
2. **Loop**
   (a) While O is not empty:
      i. Remove the node v from O with the lowest $f(v)$.
      ii. If $v = g$, reconstruct and return the path from s to g.
      iii. Add v to C.
      iv. For each neighbor u of v:
         - If $u \in C$, continue.
         - Compute tentative score $g_{\text{tentative}} = g(v) + c(v, u)$.
         - If $u \notin O$ or $g_{\text{tentative}} < g(u)$:
           – Set $g(u) = g_{\text{tentative}}$.
           – Set $f(u) = g(u) + h(u)$.
           – Set the parent of u to v.
           – If $u \notin O$, add u to O.
3. **Failure**
   - If O is empty and the goal has not been reached, return failure.

2.6 Heuristic Function

The heuristic function $h(v)$ must satisfy:

- **Admissibility**: It never overestimates the actual minimal cost from v to g.
- **Consistency (Monotonicity)**: For every edge (v, u), the estimated cost satisfies

$$h(v) \leq c(v, u) + h(u).$$

For grid-based pathfinding without diagonal movement, the **Manhattan distance** is used:

$$h(v) = |x_v - x_g| + |y_v - y_g|$$

where (x_v, y_v) and (x_g, y_g) are the coordinates of node v and the goal node g, respectively.

2.7 Algorithm Implementation Details

In my implementation for a grid-based environment:

- **Grid Representation**:
  - The environment is a 2D grid where each cell is either empty or contains an obstacle.
  - Movements are allowed in four directions: up, down, left, and right.
- **Data Structures**:
  - **Open Set** O: A priority queue implemented as a min-heap, ordered by $f(v)$.
  - **Closed Set** C: A set containing nodes that have already been evaluated.
  - **Parents Map**: A mapping from each node to its parent node, used for path reconstruction.
- **Handling Multiple Items**:
  1. Find the shortest path to the nearest item using A.
  2. Update the start position to the item's location.
  3. Remove the collected item from the list of items.
  4. Repeat until all items are collected.
- **Path Reconstruction**:
  - Once the goal is reached, the path is reconstructed by tracing back from the goal node to the start node using the parents map.

2.8 Time and Space Complexity

The theoretical time complexity of the A algorithm in the worst case is $O(|E| + |V| \log |V|)$, where $|E|$ is the number of edges and $|V|$ is the number of vertices. For a grid of size $N \times N$:

$$|V| = N^2 \quad \text{and} \quad |E| = O(N^2)$$

Thus, the worst-case time complexity becomes:

$$O(N^2 \log N)$$

In practice, the actual performance is often better due to the heuristic guiding the search efficiently towards the goal. The constant factors in the running time are influenced by:

- The efficiency of the priority queue implementation.
- The quality of the heuristic function.
- The branching factor, which is up to 4 in a grid without diagonal movement.

2.9 Coding Complexity and Debugging

Implementing A required careful management of data structures like the open and closed sets and ensuring the heuristic function is admissible and consistent. Debugging efforts focus on verifying that:

- The heuristic function is correctly implemented.
- The priority queue correctly orders nodes by $f(v)$.
- Path reconstruction accurately retrieves the optimal path.

2.10 Discussion

The A algorithm is a powerful tool for finding the shortest path in grid-based environments. By utilizing an admissible and consistent heuristic, such as the Manhattan distance, the algorithm efficiently navigates through the grid, avoiding obstacles and minimizing the total path cost. The implementation challenges are balanced by the algorithm's optimality and practical efficiency.

3 Jennifer Diaz: Depth-First Search (DFS) Algorithm

3.1 Introduction to DFS

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. Unlike A*, which uses heuristics to guide its search, DFS is an uninformed search technique that does not guarantee finding the shortest path in weighted graphs. Instead, DFS systematically explores the search space, potentially visiting all reachable vertices.

When applied to a grid-based pathfinding scenario, DFS can find a path between two points if one exists, but it may not be optimal in terms of path cost or length. Nevertheless, DFS is conceptually simple and easy to implement, making it a good baseline for comparison with more sophisticated algorithms.

3.2 Formal Definition

Consider the same graph setup $G = (V, E)$ and cost function $c(u, v)$ as defined in the A* section. For DFS, we focus on a simple exploration of the graph structure:

- We have a start node $s \in V$ and a goal node $g \in V$.
- The objective is to find *some* path P from s to g if one exists.

Unlike A*, DFS does not maintain a global cost function or heuristic estimate. Its goal is solely to determine reachability and return one valid path if found. The notion of optimality, in terms of path cost or length, is not guaranteed.

3.3 Algorithm Description

DFS can be implemented using a stack, either explicitly (with a stack data structure) or implicitly (with recursion). In this implementation, a stack is used explicitly:

1. Initialize a stack with the start node s and an empty path.
2. Keep a set of visited nodes to avoid revisiting the same state repeatedly.
3. Pop the top element $(current, path)$ from the stack.
4. If $current = g$, return the path found.
5. Otherwise, expand the neighbors of *current* that are not visited and push them onto the stack with the updated path.
6. Repeat until the stack is empty or the goal is found.

If the stack empties before reaching g, no path exists.

3.4 Grid Environment and Multiple Items

In this project, DFS is applied to a 2D grid environment:

- Each cell can be free or contain an obstacle.
- Movements are allowed up, down, left, and right.

To handle multiple items, the algorithm:

1. Starts at the initial position and attempts to find a path to the first item in the list.
2. If a path is found, the agent moves along this path, collects the item, updates its current position to the item's location, and then attempts to find a path to the next item.
3. If at any point no path to the next item exists, the algorithm reports failure.
4. After all items are collected, the algorithm optionally attempts to return to the start position.

This sequential approach ensures that DFS tries to solve a series of point-to-point pathfinding problems.

3.5 Time and Space Complexity

For a grid of size $N \times N$:

- **Time Complexity**: In the worst case, DFS may visit every cell. The complexity is $O(N^2)$ since each cell is visited at most once (due to the visited set). Without weights or heuristics, DFS doesn't prune paths based on cost, so it could explore large portions of the grid.
- **Space Complexity**: DFS stores only the stack and visited set. In the worst case, the visited set may contain all cells, giving a complexity of $O(N^2)$.

3.6 Coding Complexity and Debugging

The DFS implementation is straightforward:

- No heuristic function is needed.
- The primary challenge lies in correctly maintaining a visited set and ensuring the stack operations are implemented correctly.
- Debugging is simpler relative to algorithms like A* or Dijkstra's algorithm since there are no priority queues or advanced cost computations.

3.7 Discussion

DFS provides a baseline for performance comparison. While it can find a path, it may not find the shortest one. In environments where memory is sufficient and optimality is not a priority, DFS can be a simple and fast solution to implement. However, when path cost or distance is critical, DFS is often outperformed by algorithms like A*, which use heuristics to guide the search towards the goal more efficiently.

4 Jennifer Diaz: Breadth-First Search (BFS) Algorithm

4.1 Introduction to BFS

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that explores all neighbors of a node before moving on to the next level of nodes. In an unweighted graph (or grid), BFS is guaranteed to find the shortest path in terms of the number of edges (or moves) from the start to the goal, making it optimal for such scenarios.

When applied to a grid-based environment, BFS can efficiently find the shortest path between two points, assuming uniform step costs. However, like DFS, BFS is uninformed—no heuristic is used. This makes BFS simpler than A*, but less directed. Nonetheless, BFS is often considered a baseline for comparison due to its simplicity and optimality (in unweighted settings).

4.2 Formal Definition

As with A* and DFS, consider a grid-based graph $G = (V, E)$ where:

- Each vertex $v \in V$ corresponds to a cell in the grid.
- Each edge $(u, v) \in E$ represents a valid move from cell u to cell v.

In BFS, the objective, given a start node $s \in V$ and a goal node $g \in V$, is to find the shortest path P from s to g in terms of the number of steps. Since the grid is unweighted and moves have uniform cost, BFS will find a path P^* that minimizes:

$$|P^*| = \arg\min_{P \subseteq V} \left(\sum_{i=0}^{n-1} 1 \right)$$

Here the cost is essentially the number of moves, each counting as one.

4.3 Algorithm Description

BFS typically uses a queue structure to manage frontier nodes:

1. Initialize a queue with the start node s, along with a path containing only s.
2. Maintain a visited set to avoid revisiting nodes.
3. Dequeue the front element $(current, path)$ from the queue.
4. If $current = g$, return the path.
5. Otherwise, enqueue all unvisited neighbors of $current$ with their respective paths.
6. Repeat until the queue is empty or the goal is found.

If the queue empties before finding the goal, no path exists.

4.4 Grid Environment and Multiple Items

In this project, BFS is applied to the same 2D grid environment used by A* and DFS:

- Each cell may be free or blocked by an obstacle.
- Movements are allowed up, down, left, and right.

For multiple items:

1. Start from the initial position, run BFS to find the shortest path to the first item.
2. Move along that path, collect the item, and set the current position to the item's location.
3. Proceed to the next item in the list, using BFS again from the new start position.
4. If any item is unreachable, report failure.
5. After collecting all items, the algorithm may attempt to return to the start position using BFS once more.

This sequential approach ensures BFS solves multiple subproblems of point-to-point shortest path queries in an unweighted grid.

4.5 Time and Space Complexity

For a grid of size $N \times N$:

- **Time Complexity**: BFS may in the worst case visit every cell. Each cell is enqueued and dequeued at most once. Hence, the time complexity is $O(N^2)$.
- **Space Complexity**: BFS stores the queue and visited set. In the worst case, both can grow to include all cells, yielding $O(N^2)$ space complexity.

Since BFS explores nodes level by level, it is guaranteed to find the shortest path (in terms of moves) in an unweighted scenario.

4.6 Coding Complexity and Debugging

BFS is relatively easy to implement:

- No heuristic or priority queue is needed.
- The primary considerations are maintaining the queue for the frontier nodes and the visited set.
- Debugging typically involves ensuring that the queue operations and visited checks function correctly.

4.7 Discussion

BFS serves as a strong baseline algorithm. While it does not leverage heuristics to reduce search time, it provides guaranteed shortest paths in an unweighted grid. This makes BFS useful as a benchmark for evaluating the performance of more complex algorithms like A*. By comparing BFS results against those of A* or other heuristic-driven methods, we can measure the efficiency and benefits that heuristics provide.

5 Jubilation Megill: Floyd–Warshall Algorithm

5.1 Introduction to the Floyd–Warshall Algorithm

The Floyd–Warshall algorithm is a classic dynamic programming approach that finds shortest paths between all pairs of vertices in a weighted graph. Unlike single-source shortest path algorithms (e.g., BFS, DFS, A*, or Dijkstra's), Floyd–Warshall simultaneously computes the shortest distance from every node to every other node. This global perspective is especially useful when planning routes that must pass through multiple intermediate points.

In the context of a grid-based environment, Floyd–Warshall:

- Converts the grid into a graph where each free cell is a vertex.
- Computes the shortest path cost between all pairs of free cells.
- Once all-pairs shortest paths are known, it applies a greedy strategy to find an approximate route passing through all required items, starting and potentially returning to the initial position.

This approach is different from other algorithms discussed (like A*, BFS, or DFS), which focus on finding a single path between specified start and goal nodes. Floyd–Warshall, by contrast, provides a complete shortest-path picture of the environment.

5.2 Formal Definition

Given a set of vertices V representing free cells and edges E representing movements between adjacent cells, the Floyd–Warshall algorithm computes a distance matrix D such that:

$$D[i, j] = \text{the shortest path cost from } V_i \text{ to } V_j.$$

If the grid has $N \times N$ free cells, we have:

$$|V| = N^2 \quad \text{(in the worst case, if no obstacles are present).}$$

The Floyd–Warshall algorithm initializes the D matrix using direct edge costs and then iteratively updates it, considering each vertex as an intermediate node:

$$D[i, j] = \min(D[i, j], D[i, k] + D[k, j]) \quad \text{for all } i, j, k.$$

After completing these updates, $D[i, j]$ holds the shortest path cost from V_i to V_j if one exists, or ∞ otherwise.

5.3 Algorithm Description

The Floyd–Warshall steps are:

1. Initialize the distance matrix D so that:
   - $D[i, i] = 0$ for all i.
   - $D[i, j] = 1$ if an edge (i.e., adjacency) exists between V_i and V_j.
   - $D[i, j] = \infty$ if there is no direct edge between V_i and V_j.
2. For each vertex k:

   $$\forall i, j : D[i, j] = \min(D[i, j], D[i, k] + D[k, j])$$

   This step ensures that every possible intermediate vertex is considered.

5.4 Planning a Route Through All Items

Once the all-pairs shortest paths are computed, the algorithm uses a greedy approach to:

1. Start from the initial position.
2. At each step, choose the nearest unvisited item according to the precomputed shortest paths in D.
3. Move to that item, mark it collected, and repeat until all items are obtained.

This approach resembles a heuristic solution to a Traveling Salesman Problem (TSP)-like scenario on the grid. While this method does not guarantee the optimal order of item collection, it leverages the global distance information to find a reasonably good route.

5.5 Time and Space Complexity

The Floyd–Warshall algorithm runs in:

$$O(|V|^3)$$

For a grid of size $N \times N$:

$$|V| = N^2 \implies O((N^2)^3) = O(N^6)$$

This is significantly more expensive than single-source algorithms and quickly becomes infeasible for large grids.

Memory usage is also substantial:

$$O(|V|^2) = O(N^4)$$

space is required to store the distance and predecessor matrices.

5.6 Coding Complexity and Debugging

Implementing Floyd–Warshall correctly involves:

- Proper initialization of the distance matrix D and predecessor matrix P.
- Carefully handling ∞ values and edge cases.
- After computing all pairs shortest paths, implementing a greedy strategy to find a route through all items adds complexity.

Debugging this code can be more challenging due to the multiple nested loops and the complexity of verifying all-pairs shortest paths. Adding or removing a single edge or changing the grid structure can affect many computed distances.

5.7 Discussion

Floyd–Warshall stands apart from the other algorithms discussed. Instead of focusing on a single start-goal pair at a time, it calculates the shortest paths for all pairs. This makes it well-suited for situations where multiple queries or multiple intermediate targets exist. However, the high computational cost and memory usage limit its scalability on large grids.

When combined with a heuristic route-planning step (like a greedy approach), Floyd–Warshall can provide a decent solution for collecting items scattered throughout the grid, though it may be overkill or too resource-intensive for simpler pathfinding needs.

6 Odysseus Valdez: Dijkstra's Algorithm

6.1 Introduction to Dijkstra's Algorithm

Dijkstra's algorithm is a classic algorithm for finding the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights. Unlike A*, it does not rely on heuristics but systematically explores nodes in increasing order of distance from the source. In the context of grid-based environments, Dijkstra's algorithm guarantees finding the optimal path to a goal node while handling uniform or varying step costs.

6.2 Formal Definition

Let $G = (V, E)$ be a graph, where:

- V is the set of vertices (nodes), and E is the set of edges.
- Each edge $(u, v) \in E$ has a non-negative weight $w(u, v)$, representing the cost to travel from node u to node v.

Given:

- A source node $s \in V$.
- A set of goal nodes $G = \{g_1, g_2, \ldots, g_k\} \subseteq V$.

The objective is to compute the shortest path P^* from s to each $g \in G$ such that:

$$P^* = \arg\min_P \left(\sum_{(u,v) \in P} w(u, v) \right)$$

where P is a path in G.

6.3 Algorithm Description

Dijkstra's algorithm operates by iteratively expanding the shortest known path to unexplored nodes. Key components include:

- A priority queue to manage frontier nodes, ordered by their current shortest path estimate.
- A distance map $d[v]$ storing the shortest known distance from s to v.
- A parent map $p[v]$ to reconstruct paths once the goal is reached.

Steps of the algorithm:

1. **Initialization:**
   - Set $d[s] = 0$ and $d[v] = \infty$ for all $v \in V \setminus \{s\}$.

- Add $(0, s)$ to the priority queue.

2. **Main Loop:**
   - While the priority queue is not empty:
     (a) Extract the node u with the smallest $d[u]$.
     (b) For each neighbor v of u:
       – Compute $d_{\text{tentative}} = d[u] + w(u, v)$.
       – If $d_{\text{tentative}} < d[v]$, update $d[v]$, set $p[v] = u$, and add v to the queue.
3. **Path Reconstruction:**
   - Starting from each goal node $g \in G$, trace back using $p[v]$ to construct the path from s.

6.4 Handling Multiple Items

In this implementation, the algorithm handles multiple items by:

1. Running Dijkstra's algorithm from the current position to find the nearest item.
2. Moving along the shortest path to collect the item.
3. Updating the current position and repeating until all items are collected.

6.5 Time and Space Complexity

For a grid of size $N \times N$:

- **Time Complexity:** Each node and edge is processed once in the priority queue. Using a min-heap, the complexity is:

$$O(|E| + |V| \log |V|) = O(N^2 \log N) \quad \text{for a grid.}$$

- **Space Complexity:** The algorithm stores distances, parents, and the priority queue. Thus, the complexity is:

$$O(N^2)$$

6.6 Coding Complexity and Debugging

Implementing Dijkstra's algorithm required:

- Efficiently managing the priority queue with a min-heap.
- Properly updating distance and parent maps to avoid redundant calculations.
- Correctly handling multiple goals by iteratively updating the source and recomputing paths.

Debugging challenges included:

- Verifying correctness of the priority queue operations.
- Ensuring no items were left unvisited due to early termination or incorrect updates.
- Handling edge cases such as unreachable nodes.

6.7 Discussion

Dijkstra's algorithm is a powerful and optimal method for shortest-path computation in non-negative weighted graphs. Its deterministic nature and lack of reliance on heuristics make it a reliable baseline for comparison. However, its performance in dense grids, where many nodes are explored, can lag behind heuristic-driven algorithms like A*. The iterative approach for handling multiple items, while effective, may introduce overhead compared to more global solutions such as Floyd–Warshall.

7 Performance Analysis and Discussions

This section presents a comparative analysis of the implemented pathfinding algorithms based on the generated outputs, performance metrics, and visualized theoretical complexities.

7.1 (a) Comparison of Asymptotic Running Times and Actual Performance

The following bar chart illustrates the average execution time for all algorithms across the test cases:

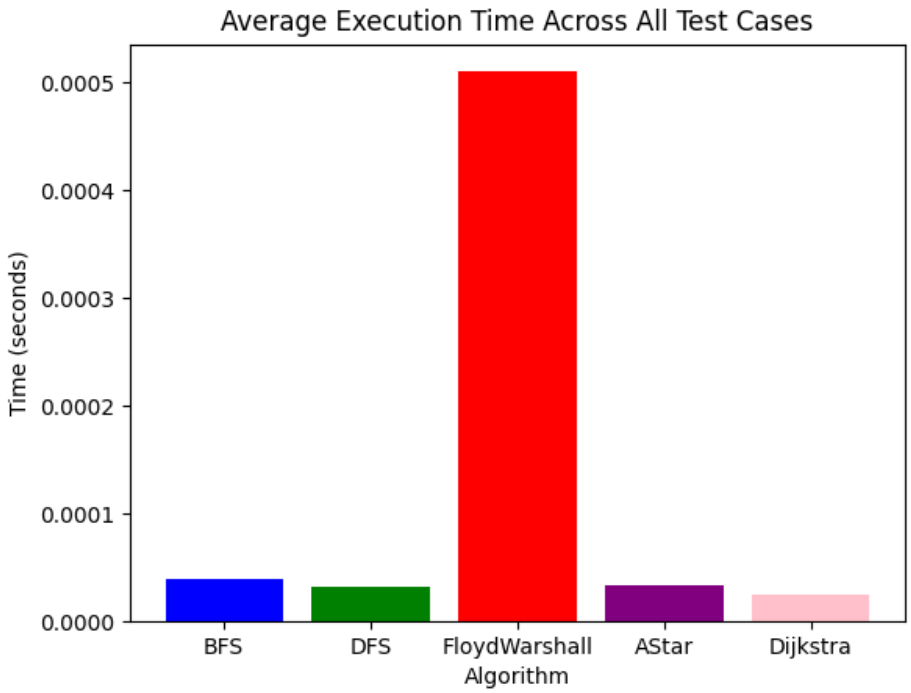

Figure 1: Average Execution Time Across All Test Cases

From Figure 1, we observe:

- Floyd-Warshall is significantly slower than the other algorithms due to its cubic time complexity $O(|V|^3)$, especially evident with larger grids.
- A^*, DFS, and Dijkstra exhibit the fastest execution times, followed closely by BFS. The simplicity of BFS and the efficient use of priority queues in Dijkstra allow for optimal performance in these scenarios.
- DFS is slightly slower than A*, likely due to its exhaustive exploration of paths without pruning based on a heuristic or cost function.
- A* remains efficient but does not outperform Dijkstra in these cases, potentially due to the simplicity of the grid structure and heuristic usage.

The following plot compares the theoretical time complexities:

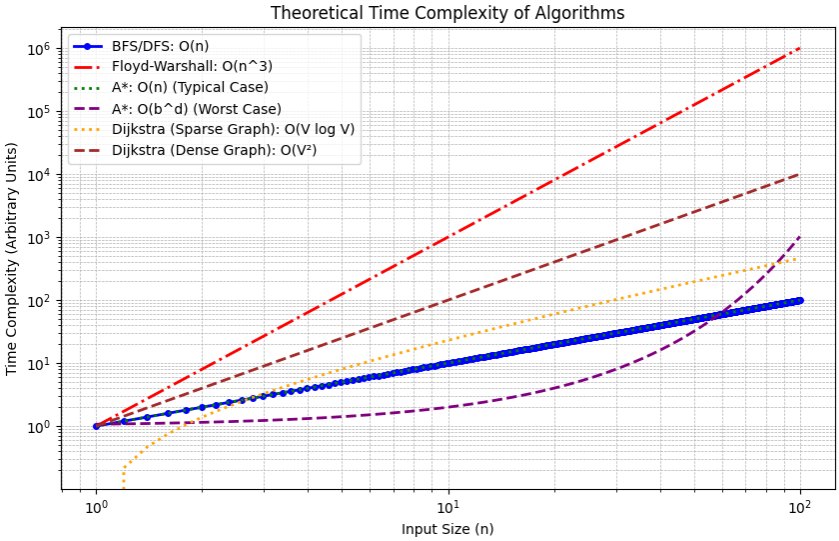

Figure 2: Theoretical Time Complexity of Algorithms

The Figure 2 aligns well with the observed execution times, emphasizing the dominant cubic growth of Floyd-Warshall and the $O(|V|^2)$ complexity of Dijkstra on dense graphs.

7.2 (b) Coding Complexity and Debugging Effort

The coding complexity and debugging effort vary significantly:

- Floyd-Warshall: Debugging was challenging due to the nested loops and managing the distance matrices, but the algorithm itself is straightforward to implement.
- BFS and DFS: These were the easiest to code and debug due to their simple structures.
- A^* and Dijkstra: Both required intricate data structure management (priority queues). A* added complexity with the heuristic function.

7.3 (c) Fastest and Slowest Algorithms

Based on the analysis:

- **Fastest:** A^*, DFS, and Dijkstra. The combination of simplicity (BFS) and optimal use of priority queues (Dijkstra) allows for excellent performance.
- **Slowest:** Floyd-Warshall. Its cubic complexity and all-pairs shortest path computation make it unsuitable for larger grids.

7.4 (d) Memory Usage and Maximum Grid Size

The following bar chart illustrates the average number of nodes explored across all test cases, serving as a proxy for memory usage:

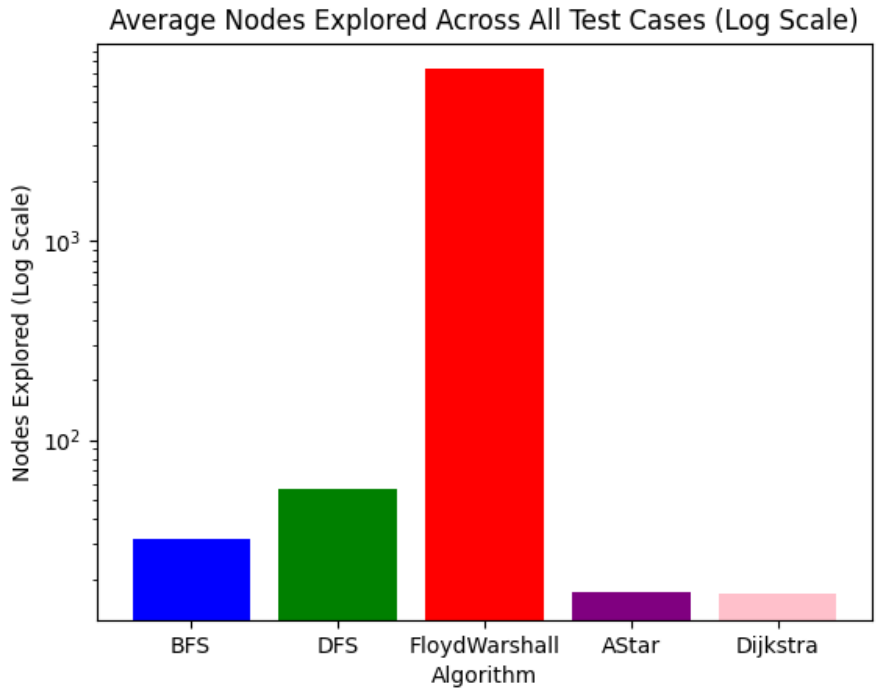

Figure 3: Average Nodes Explored Across All Test Cases (Log Scale)

The Figure 3 highlights:

- Floyd-Warshall: The most memory-intensive, with thousands of nodes explored due to its global computation.
- BFS and DFS: Moderate memory usage due to their straightforward exploration strategies. Though still not as good as A^* and Dijkstra.
- A* and Dijkstra: not nearly as much memory usage, attributed to the priority queue and heuristic/map management.

For the current grid size of 5×5, all algorithms operated without memory issues. However, for larger grids:

- BFS, DFS, A*, and Dijkstra are expected to scale better due to their more localized computation.
- Floyd-Warshall would likely encounter memory constraints first due to its $O(N^2)$ memory requirement for the distance matrix.

7.5 Conclusion

The analysis of the pathfinding algorithms reveals some intriguing discrepancies between their theoretical time complexities and actual performance on the 5×5 grid test cases. While the theoretical results align with expectations for larger inputs, the small grid size and specific characteristics of the test cases explain some of the observed deviations:

- **Grid Size and Input Characteristics:** The 5×5 grid is relatively small, minimizing the impact of asymptotic complexities. Algorithms like A* and Dijkstra, which excel on larger inputs due to their efficient handling of complex structures, do not have as much room to showcase their advantages over simpler algorithms like BFS and DFS. In these small grids, the overhead associated with managing priority queues and heuristic functions can outweigh the benefits.

- **Heuristic Limitations in A*:** A* heavily relies on the heuristic function to guide its search efficiently. In our test cases, the heuristic (Manhattan distance) might not always provide significant pruning, especially in grids with multiple optimal paths or closely spaced items. As a result, A* sometimes explored a comparable or greater number of nodes than simpler algorithms.

- **BFS and DFS Simplicity:** The relative simplicity of BFS and DFS becomes advantageous in these small grids. These algorithms avoid the overhead of maintaining additional data structures, allowing them to achieve faster execution times in cases where all items or goals are in accessible regions with straightforward paths.

- **Floyd-Warshall's Computational Intensity:** Floyd-Warshall's all-pairs shortest path computation inherently results in higher execution times and memory usage, as expected from its $O(|V|^3)$ complexity. However, this algorithm's global perspective can be advantageous in scenarios requiring precomputed shortest paths for multiple queries.

- **Dijkstra's Consistent Efficiency:** Dijkstra's strong performance is a testament to its effective use of priority queues to ensure optimal pathfinding. Its behavior closely mirrors that of A* in these grids, but without the added complexity of heuristic calculations, it occasionally outperformed A*.

- **Simpler Algorithms Surpassing Advanced Ones:** Simpler algorithms like BFS and DFS outperforming advanced ones such as A* in these tests underscore the importance of matching algorithm choice to problem size and complexity. The additional features of advanced algorithms are designed to handle large-scale or highly complex scenarios, which are not present in small grid tests.

In summary, while theoretical complexities provide valuable insights into the behavior of algorithms for large inputs, actual performance can vary significantly based on input size, grid structure, and algorithmic overhead. For small grids, simplicity often prevails, but as grid sizes and complexity grow, the advantages of advanced algorithms like A* and Floyd-Warshall become more apparent.

8 Team Member Acknowledgements

8.1 Jennifer Diaz

Implemented the breadth first search (BFS) and depth first search (DFS) algorithms as well as helped with the full implementation found in Main.py and the report. Additionally, implemented extra credit for BFS and DFS.

8.2 Jubiliation Megill

Implemented the algorithm of our choice for this project which ended up being the Floyd-Warshall algorithm. Additionally, helped with the full implementation in Main.py as well as his section of the report. Additionally, implemented extra credit for Floyd-Warshall.

8.3 Odysseus Valdez

Implemented Dijkstra's algorithm as well as helped with the full implementation in Main.py and the report. Additionally, implemented extra credit for Dijkstra's algorithm.

8.4 Tanner Wagner

Implemented A^* algorithm, helped with full implmentation in Main.py, drafted report, helped with full implementation of comparative analysis in Main.py using numpy as well as implemented extra credit for A^* algorithm.

9 Bibliography

1. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100–107.

2. Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall.

3. Dechter, R., & Pearl, J. (1985). *Generalized Best-First Search Strategies and the Optimality of A**. Journal of the ACM, 32(3), 505–536.

4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

5. Dijkstra, E. W. (1959). *A Note on Two Problems in Connexion with Graphs*. Numerische Mathematik, 1, 269–271.

6. Floyd, R. W. (1962). *Algorithm 97: Shortest Path*. Communications of the ACM, 5(6), 345.

7. Warshall, S. (1962). *A Theorem on Boolean Matrices*. Journal of the ACM, 9(1), 11–12.

8. Tarjan, R. E. (1972). *Depth-First Search and Linear Graph Algorithms*. SIAM Journal on Computing, 1(2), 146–160.

9. Course Lecture Notes on Pathfinding Algorithms, CS 361, Fall 2024.