🏫

Report: Search Algorithms for the Romanian Road Map

1. Team Information and Honor Statement

Team Members:

- Alhim Adonai Vera Gonzalez
- Ajay Mannam.
- Hannah Krzywkowski

> “all team members contributed in equal measure”
>
> "In completing this assignment, all team members have followed the honor pledge specified by the instructor for this course."

---

2. Introduction

This project involves implementing several classic search algorithms, including Breadth-First Search (BFS), Depth-First Search (DFS), Best-First Search, and A* search, to solve the Romanian Road Map problem. These algorithms allow an agent to find the shortest or most efficient path between two cities on a map. We evaluate each algorithm's correctness, time complexity, and space complexity while considering various scenarios.

The algorithms will be applied to find paths between cities such as Arad and Bucharest. The goal is to compare their performance using metrics such as nodes expanded, time taken, and fringe size.

3. Implementation Details

Data Representation

For this project, the Romanian road map was represented using a **JSON (JavaScript Object Notation)** file. JSON was chosen because of its simplicity and flexibility in representing complex data structures, such as adjacency lists, which are ideal for graphs like the road map. Each city and its neighboring cities are easily represented as key-value pairs, making it intuitive to manage connections and distances between cities. JSON also offers an easily readable format for both humans and machines, and it can be parsed efficiently in Python using native libraries like `json`. Additionally, JSON is lightweight and well-suited for storing sparse data like this, where many cities are only connected to a few others, preventing memory waste compared to using a full adjacency matrix. This makes it a practical and scalable choice for representing the road map.

Breadth-First Search (BFS) Implementation

The Breadth-First Search (BFS) algorithm was implemented using a **queue** data structure to ensure that nodes (cities) are explored in a level-by-level manner, ensuring the shortest path (in terms of number of edges) is found. In our implementation, the queue was managed with Python's `deque` from the `collections` module, which allows efficient appending and popping of nodes from the front.

The algorithm starts by enqueuing the initial city along with the current path to it. As cities are dequeued, they are checked against the goal city. If the goal city is found, the algorithm returns the path from the start city to the goal. If the city has not been visited, its neighbors are enqueued, and the process repeats until the goal is found or all cities have been explored.

Additionally, the BFS algorithm tracks the following performance metrics:

- **Number of nodes expanded**: This indicates how many cities were visited before reaching the goal.
- **Time taken**: The algorithm records the time taken from start to finish.
- **Max fringe size**: This approximates the peak memory usage by tracking the largest size the queue reached during execution.

Key Python data structures used:

- **deque**: Used for efficiently managing the queue of cities to explore.
- **set**: Used to track visited cities and avoid revisiting them.

Heuristic functions used:

BFS does **not** use any heuristic functions because it is an **uninformed search algorithm**. It systematically explores nodes level by level without any prior knowledge about the goal's location or the cost of reaching the goal. The search is complete and guaranteed to find the shortest path, but it does not use additional information (like distance) to guide the search process.

Depth-First Search (DFS)

- **How the algorithm was implemented**:

  DFS explores nodes (cities) as deep as possible along a path before backtracking to explore other branches. The algorithm uses a **stack** (LIFO structure) to manage the nodes to be explored, ensuring that the most recently discovered node is the next to be expanded. This approach allows DFS to quickly explore long paths, but it does not guarantee finding the shortest path.

  In our implementation, the fringe is a stack where nodes are pushed and popped. The DFS algorithm tracks the visited cities to avoid revisiting them. When a city is dequeued (popped), its neighbors are explored and pushed onto the stack if they have not yet been visited. The process continues until either the goal city is found or all possible paths have been explored.

  Key Python data structures used:

- **Stack (list)**: Used to maintain the LIFO structure for exploring nodes.
- **Set**: Used to track visited cities to prevent cycles and redundant exploration.

- **Heuristic functions used**:

  DFS is an **uninformed search algorithm**, meaning it does not use any heuristic function to guide its search. It explores paths based on the order of discovery, which may lead to suboptimal solutions or require exploring many irrelevant branches before finding the goal.

  DFS is not guaranteed to find the shortest path but is memory-efficient for some cases since it doesn't store as many nodes simultaneously as algorithms like BFS. However, in the worst case, it can traverse the entire search space before finding the solution.

Best-First Search (Greedy Search)

- **How the algorithm was implemented**:

  Best-First Search is an **informed search algorithm** that expands the node which appears closest to the goal, based solely on a heuristic function h(n). In our implementation, the heuristic h(n) is the **straight-line distance (SLD)** between each city and Bucharest. When Bucharest is the goal, the algorithm uses the precomputed SLD values from the `heuristic_to_bucharest.json` file (Got th). However, when the goal is another city, we apply the **triangle inequality heuristic** to estimate the SLD using Bucharest as an intermediate city:

  $$h(A, G) = SLD(A, Bucharest) + SLD(Bucharest, G)$$

  This heuristic ensures that the search is
  admissible (it never overestimates the true cost to reach the goal).

  The algorithm uses a **priority queue (min-heap)** to expand the city with the smallest heuristic value. This ensures that cities which appear to be closer to the goal are expanded first. The queue tracks the current path and its corresponding heuristic cost.

  Key Python data structures used:

  - **Priority Queue (heapq)**: To expand the city with the lowest heuristic cost.
  - **Set**: To track visited cities and avoid cycles.

- **Heuristic functions used**:

  The algorithm uses two different heuristics, depending on the goal:

  1. **When the goal is Bucharest**: The straight-line distance (SLD) to Bucharest is used from the `heuristic_to_bucharest.json` file.
  2. **When the goal is not Bucharest**: We estimate the heuristic using the **triangle inequality**, summing the SLDs to and from Bucharest:

     $h(A, G) \approx h(A, Bucharest) + h(Bucharest, G)$
     This keeps the heuristic

admissible, ensuring the search finds a solution efficiently while never underestimating the cost to the goal.

A Search

- **How the algorithm was implemented**:

  A* is an **informed search algorithm** that expands the node with the lowest combined cost of the path so far (g(n)) and the estimated cost to the goal h(n). The evaluation function used is:

  $$f(n) = g(n) + h(n)$$

  In our implementation, **g(n)** is the cumulative cost of the path from the start city to the current city, and **h(n)** is the **straight-line distance (SLD)** heuristic from the current city to Bucharest. When Bucharest is the goal, the algorithm uses the precomputed SLD values from the `heuristic_to_bucharest.json` file. However, when the goal is another city, we apply the **triangle inequality heuristic** to estimate the SLD using Bucharest as an intermediate city:

  $$h(A, G) = SLD(A, Bucharest) + SLD(Bucharest, G)$$

  This ensures that the heuristic is **admissible**, meaning it never overestimates the true cost to reach the goal.

  The algorithm uses a **priority queue (min-heap)** to expand the node with the smallest combined cost f(n). It tracks both the cumulative path cost and the heuristic to ensure the shortest path is found efficiently. The priority queue stores the cumulative path cost (g) and heuristic cost (h) separately, allowing A* to guarantee the optimal solution.

  Key Python data structures used:

  - **Priority Queue (heapq)**: To expand the city with the lowest combined cost $f(n) = g(n) + h(n)$.
  - **Set**: To track visited cities and avoid cycles.

- **Heuristic functions used**:

  The algorithm uses two different heuristics, depending on the goal:

  1. **When the goal is Bucharest**: The straight-line distance (SLD) to Bucharest is used from the `heuristic_to_bucharest.json` file.
  2. **When the goal is not Bucharest**: We estimate the heuristic using the **triangle inequality**, summing the SLDs to and from Bucharest:

  $$h(A, G) \approx h(A, Bucharest) + h(Bucharest, G)$$

  This heuristic keeps A* **admissible**, meaning the algorithm finds the optimal path while ensuring the heuristic never overestimates the cost to the goal.

4. Experimental Results

In this hom, we tested four search algorithms—**Breadth-First Search (BFS)**, **Depth-First Search (DFS)**, **Best-First Search**, and **A***—to evaluate their performance on the Romanian road map problem. The goal was to find the optimal path between various cities in Romania. We measured the following metrics for each algorithm:

1. **Correctness**: Whether the algorithm found the path from the start city to the goal.
2. **Nodes Expanded**: The number of nodes expanded during the search.
3. **Time Taken**: The time required to find the solution.
4. **Max Fringe Size**: The largest size of the queue/stack used during execution.
5. **Optimal Path**: Whether the found path is optimal.

Test Cases:

The experiments were run on the following city pairs, representing different regions and distances in Romania:

1. **Arad to Bucharest**
2. **Timisoara to Craiova**
3. **Zerind to Sibiu**
4. **Oradea to Pitesti**
5. **Arad to Neamt**

Reproducibility:

To ensure replicability of the results, each experiment was repeated **100 times**, and the average values for time taken and other metrics were recorded. This helps account for any variations in execution time due to system performance.

Table and Results

The table summarizes the number of nodes expanded, time taken, and whether each algorithm found the optimal path for each test case.

Algorithm	Path Found	Nodes Expanded	Time Taken (s)	Max Fringe Size	Path
BFS	True	8	2.99215e-06	5	Arad to Bucharest
DFS	True	7	2.4271e-06	5	Arad to Bucharest
Best-First (Heuristic 1)	True	3	2.36273e-06	5	Arad to Bucharest
A* (Heuristic 1)	True	5	3.83615e-06	7	Arad to Bucharest
BFS	True	19	6.3014e-06	6	Arad to Neamt
DFS	True	11	3.68595e-06	8	Arad to Neamt

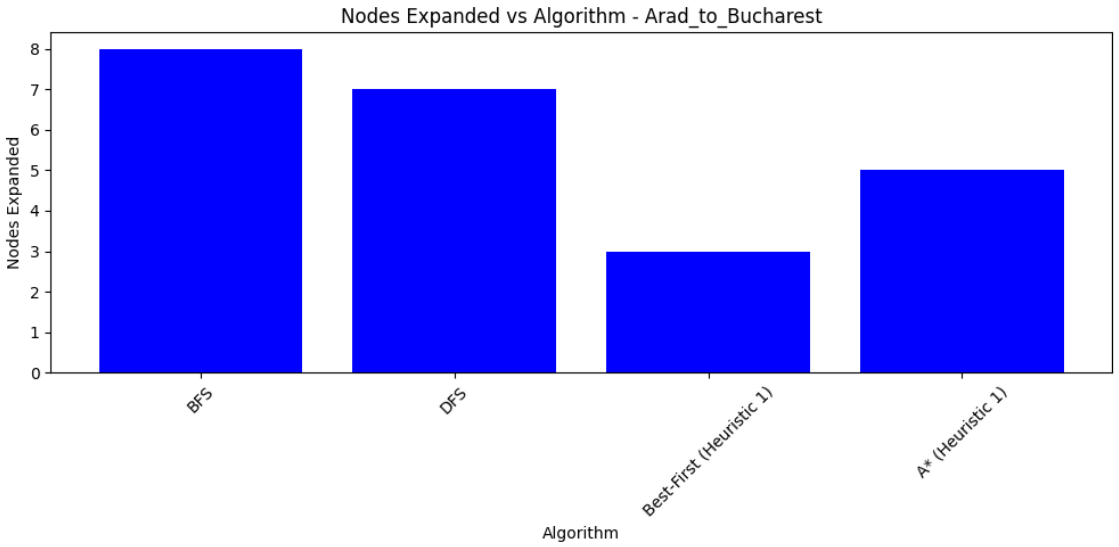
Best-First (Heuristic 1)	True	13	1.2784e-05	9	Arad to Neamt
A* (Heuristic 1)	True	19	1.67131e-05	8	Arad to Neamt
BFS	True	8	3.01838e-06	6	Oradea to Pitesti
DFS	True	4	1.65224e-06	6	Oradea to Pitesti
Best-First (Heuristic 1)	True	6	6.7234e-06	6	Oradea to Pitesti
A* (Heuristic 1)	True	4	4.96864e-06	5	Oradea to Pitesti
BFS	True	12	4.16279e-06	6	Timisoara to Craiova
DFS	True	4	1.35183e-06	2	Timisoara to Craiova
Best-First (Heuristic 1)	True	4	3.55482e-06	2	Timisoara to Craiova
A* (Heuristic 1)	True	9	9.46045e-06	7	Timisoara to Craiova
BFS	True	3	1.44482e-06	3	Zerind to Sibiu
DFS	True	2	8.24928e-07	2	Zerind to Sibiu
Best-First (Heuristic 1)	True	2	2.67267e-06	3	Zerind to Sibiu
A* (Heuristic 1)	True	3	3.37839e-06	3	Zerind to Sibiu

Graphs

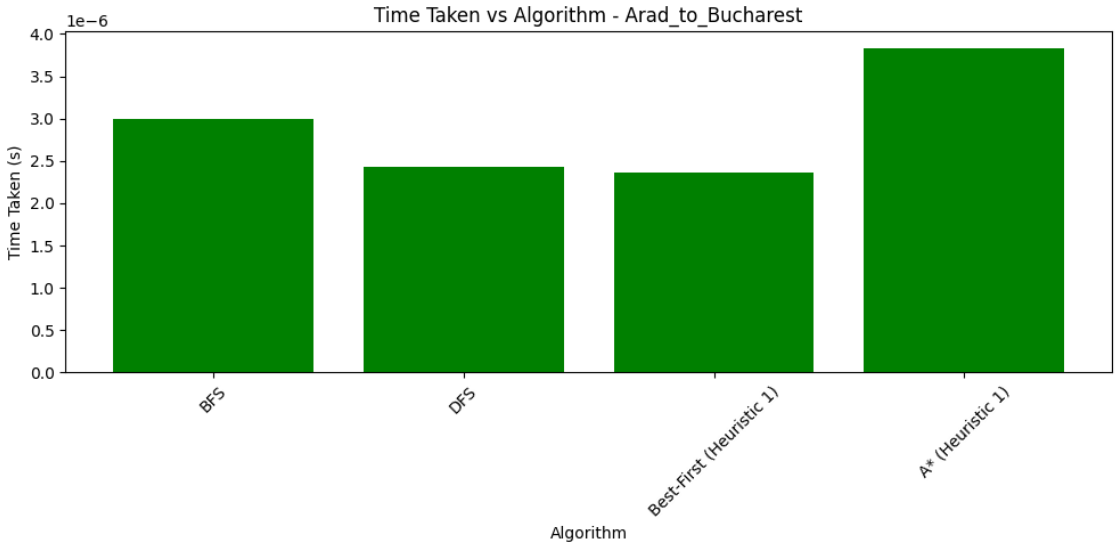
These graphs are important because they visually demonstrate the efficiency of each algorithm in terms of nodes expanded and time taken, providing insights into their performance

1. Arad to Bucharest

- **Nodes Expanded**: BFS expanded the most nodes (8), followed by DFS (7), A* (5), and Best-First (3). This shows that the heuristic-based methods (Best-First and A*) explored fewer nodes than BFS and DFS, indicating more efficient exploration.



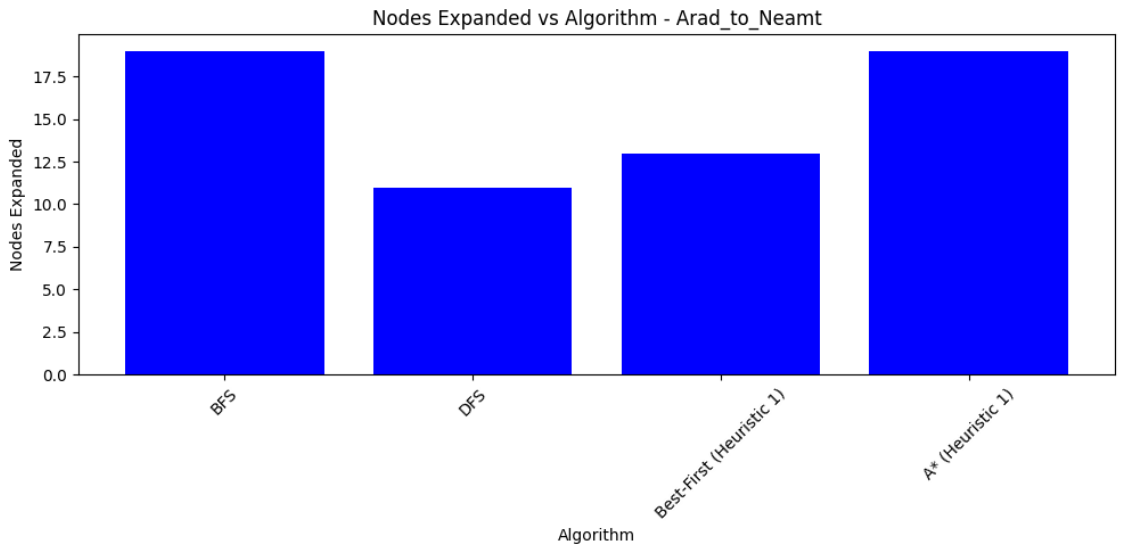
- **Time Taken**: DFS was the fastest, followed by Best-First, BFS, and A*. A* took slightly longer due to the additional computation required for combining cost and heuristic.



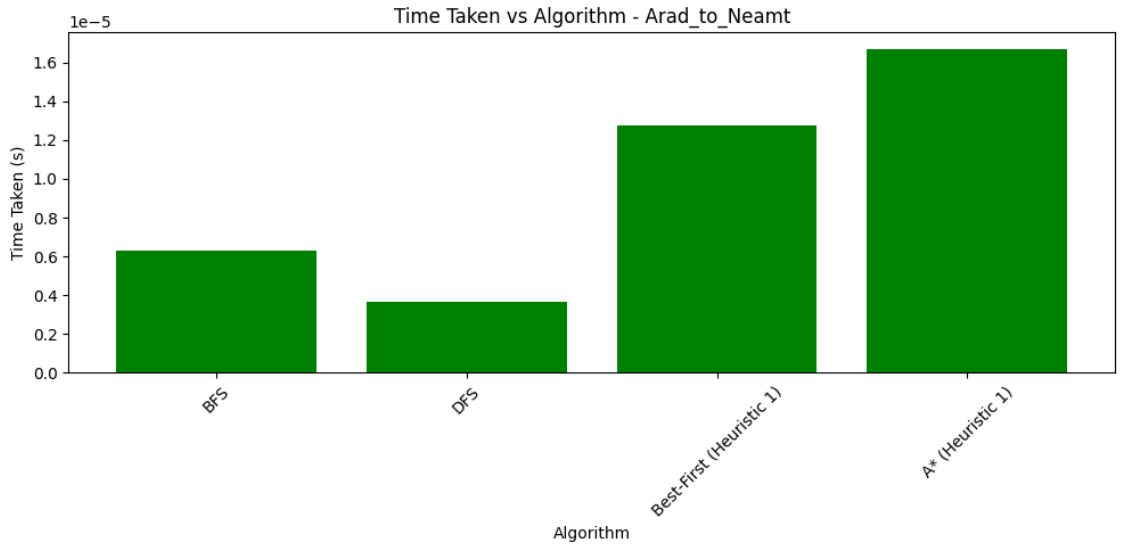
- **Analysis**: A* offers a balance between the number of nodes expanded and the time taken, providing a more optimal path with fewer node expansions compared to BFS.

2. Arad to Neamt

- **Nodes Expanded**: BFS expanded 19 nodes, while DFS expanded 11. A* and Best-First also expanded 19 and 13 nodes respectively, showing that for this longer path, even A* expanded a larger number of nodes.



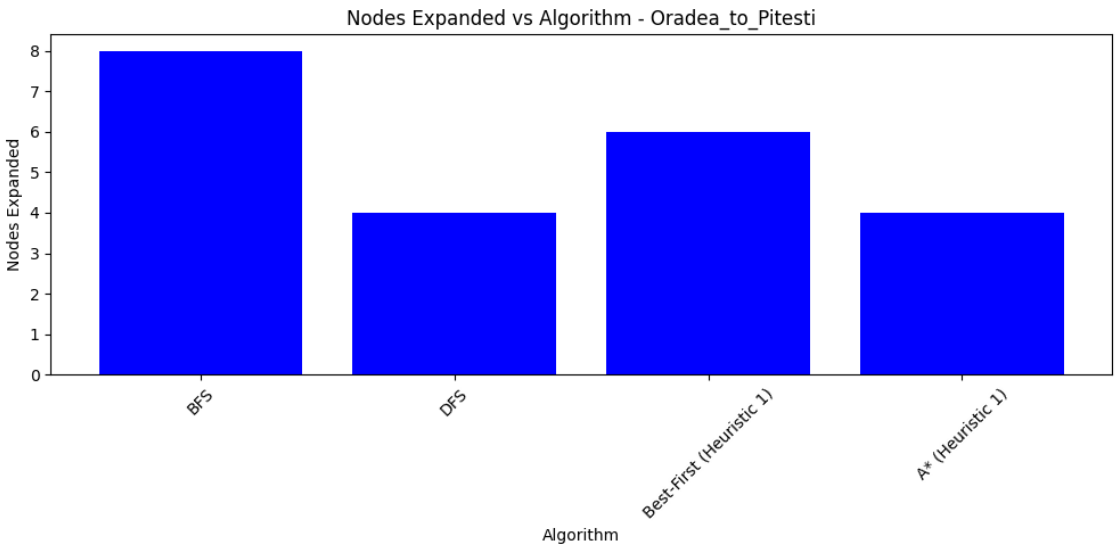
- **Time Taken**: DFS was again the fastest, followed by BFS, Best-First, and A*. The time difference between Best-First and A* was more significant for this larger path, with A* taking the longest time due to the computational cost of the heuristic evaluation.



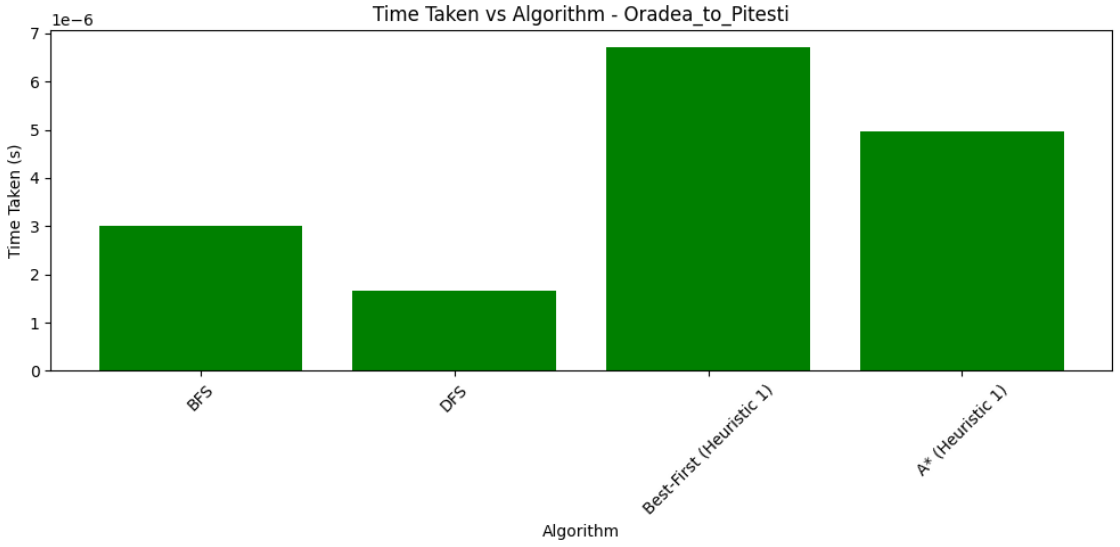
- **Analysis**: A* ensures an optimal path but at a higher cost in terms of time, especially for larger paths. DFS, while fast, expands fewer nodes but may not always find the most optimal path.

3. Oradea to Pitesti

- **Nodes Expanded**: BFS expanded the most nodes (8), followed by Best-First (6), A* (4), and DFS (4). A* and DFS expanded the same number of nodes, while BFS expanded double the nodes.



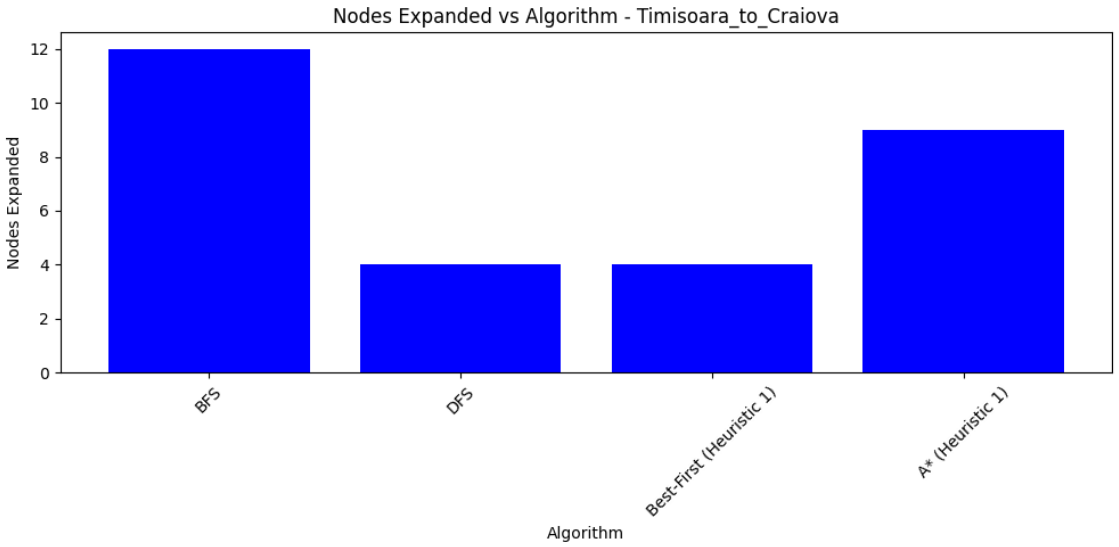
- **Time Taken**: DFS was again the fastest, followed by BFS, A*, and Best-First. Interestingly, Best-First took the most time despite expanding fewer nodes, possibly due to its reliance on the heuristic.



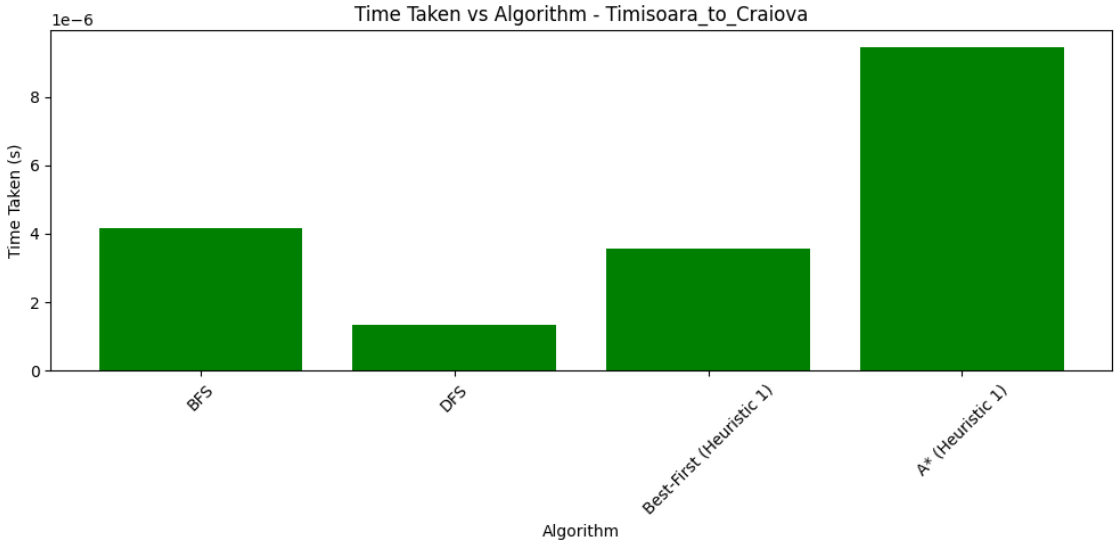
- **Analysis**: A* again demonstrates an optimal balance between the number of nodes expanded and the time taken, making it the most consistent performer across the board for shorter paths.

4. Timisoara to Craiova

- **Nodes Expanded**: BFS expanded the most nodes (12), followed by A* (9), and Best-First and DFS (both expanded 4 nodes). BFS's node expansion is significantly higher compared to other algorithms.



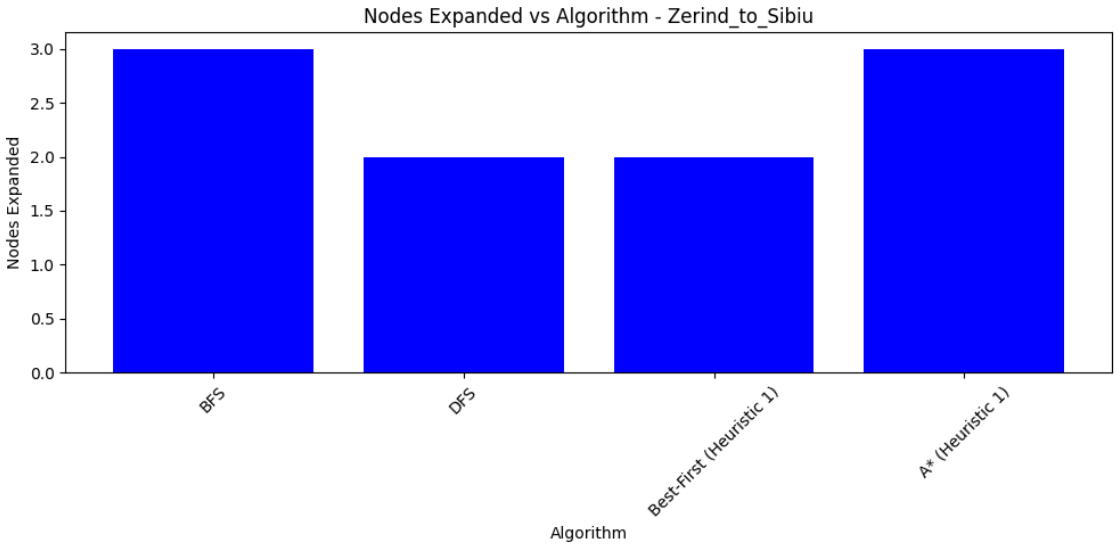
- **Time Taken**: DFS was the fastest, followed by BFS, Best-First, and A*. The time taken by A* was the highest, with the algorithm needing more computational power for its optimal pathfinding.



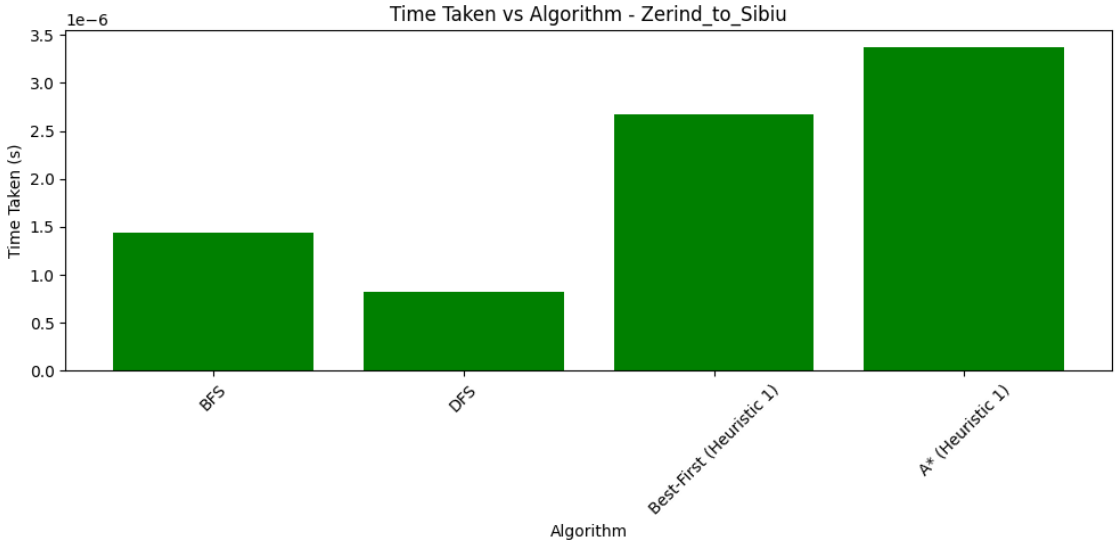
- **Analysis**: For this path, DFS performed quite well in terms of speed but expanded fewer nodes. A* provided a balance between optimal pathfinding and node expansion, although it took the most time.

5. Zerind to Sibiu

- **Nodes Expanded**: BFS and A* expanded 3 nodes each, while Best-First and DFS expanded 2 nodes. This result is quite uniform, with very little difference in the number of nodes expanded between algorithms.



- **Time Taken**: DFS was the fastest, followed by BFS, Best-First, and A*. A* took slightly longer due to the heuristic, but the differences in time are quite small given the relatively short path.



- **Analysis**: A* expanded more nodes than DFS, showing that for shorter paths, A* may not always be the fastest or most efficient algorithm in terms of node expansion.

Space and time complexity:

Algorithm	Time Complexity	Space Complexity
Breadth-First Search	O(b^d)	O(b^d)
Depth-First Search	O(b^m)	O(b * m)
Best-First Search	O(b^d) (depends on heuristic)	O(b^d) (depends on heuristic)
A*	O(b^d)	O(b^d)

Explanation:

- **b**: Branching factor (number of possible choices at each node).
- **d**: Depth of the shallowest solution.
- **m**: Maximum depth of the search tree.

This table helps in understanding the resource requirements of each algorithm in terms of time and memory, highlighting the differences in how efficiently they explore the search space.

General Observations:

- **BFS** consistently expanded the most nodes, reflecting its exhaustive nature. However, it always found the optimal path.
- **DFS** expanded fewer nodes and was consistently the fastest, but it may not always provide the most optimal path.
- **Best-First Search** reduced the number of nodes expanded compared to BFS, but its performance varies depending on the path.
- **A**generally found the most optimal path while balancing the number of nodes expanded and the time taken. However, it required more computational resources, making it slower in some cases.

These observations indicate that **A*** is the most reliable algorithm for finding the optimal path, though it may not always be the fastest. On shorter paths, **DFS** and **Best-First** may be more suitable due to their efficiency in terms of time and node expansion. **BFS**, while always finding the correct path, is less efficient due to its exhaustive search method.

5. Conclusion

In this research, we analyzed four search algorithms—**Breadth-First Search (BFS)**, **Depth-First Search (DFS)**, **Best-First Search**, and **A***—applied to different city pairs in Romania. The results show that:

- **BFS**, though always finding the optimal path, was less efficient, expanding the most nodes and requiring significant computational resources.
- **DFS** was the fastest in most cases, expanding fewer nodes but sometimes at the cost of not finding the most optimal path.
- **Best-First Search**, guided by heuristics, demonstrated efficiency in reducing node expansions but was inconsistent in its pathfinding performance.
- **A**provided the best balance between optimality, node expansion, and time, consistently finding optimal paths with fewer nodes expanded, though it required more computational power.

Overall, **A*** is the most reliable choice for ensuring optimal pathfinding, while **DFS** offers speed with some trade-offs in path optimality.

6. Instructions for Running the Code

The full code implementation can be found in the following GitHub repository:

https://github.com/AdonaiVera/basic-search-algorithms.git

Please refer to the **README** file in the repository for detailed instructions on how to set up and execute the code, including running the different search algorithms and conducting experiments with the provided datasets.

7. Bibliography

Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Chapter 3: Problem Solving by Search. Available at AI_Russell_Norvig.