De La Salle University

Term 3 Academic Year 2023-2024

In Partial fulfillment of the course

CSINTSY - S13

MCO1 - State-Based Model Report

Submitted By:

Ariaga, Marian Ricci N.

Guillarte, Dana Louise A.

Llorando, Yesha P.

O’Neil, Samantha Erica R.

So, Chrysille Grace L.

Submitted to:

Dr. Norshuhani Zamin

June 21, 2024

I. Introduction

Optimal pathfinding is a significant topic in Computer Science due to its widespread relevance and practical applications. Numerous algorithms have been developed to determine the optimal path efficiently. In this report, we will explore two categories of search algorithms: blind search and heuristic search. Specifically, we will examine the Uniform-Cost Search (UCS) as an example of a blind search algorithm and the A* Search as an example of a heuristic search algorithm.

Uniform-Cost Search

Uniform-Cost Search (UCS) is designed to find the path from a source node to a goal node with the lowest cumulative cost [1]. This algorithm uses a brute force method by visiting nodes based on their current cost and prioritizing the next path with the smallest overall cost from the source node [2]. This method guarantees the path with the lowest cumulative cost. A step-by-step visual representation of UCS is provided below.

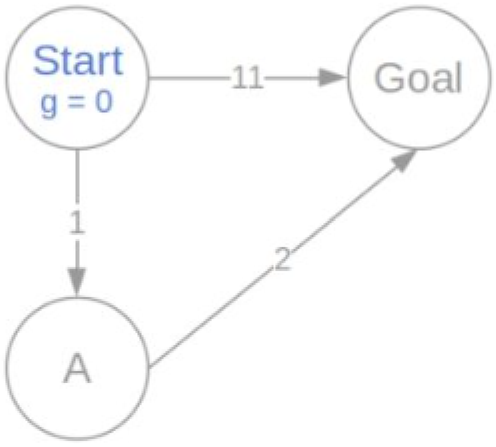


Figure 1.1: Sample Graph for UCS

The graph above illustrates a graph for the UCS procedure. The algorithm begins at the source node, where the cumulative cost (g) is initially 0 because no other nodes have been explored.

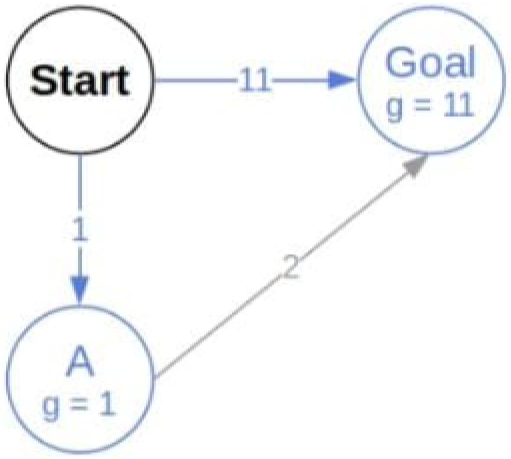


Figure 1.2: Expanding the Start node

Upon expanding the source node, we observe two possible paths: one leading directly to the goal node with a cost of 11, and another to node A with a cost of 1. Despite having a direct path to the goal node, it is not the lowest-cost path. Instead, we choose to move to node A, which has a lower cost of 1. We then update the current node and cumulative cost to 1.

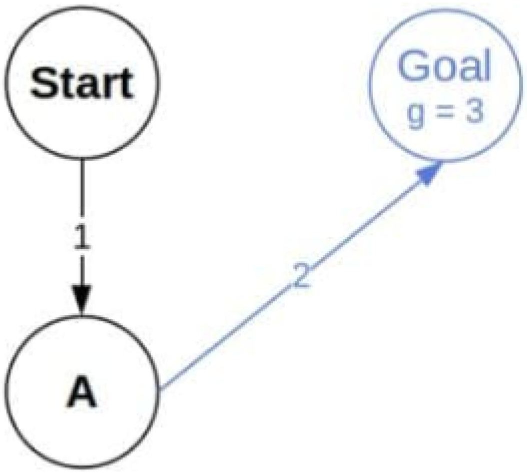


Figure 1.3: Expanding the A node

Expanding node A reveals a single path to the goal node with a cost of 2. Adding this path to the current cumulative cost results in a total cost of 3. Since this path reaches the goal node, it is considered complete. Comparing the available paths from the source to the goal node, we see that the direct path has a cumulative cost of 11, while the newly calculated path through node A costs 3. We select the path with the lesser cost. Thus, the optimal path is Start -> A -> Goal [3].

In real-world applications, UCS is utilized for larger and more complex graphs involving numerous paths. However, this example provides a fundamental understanding of UCS [4].

A* Search

Similar to UCS, A* is another search algorithm designed to find the shortest path between two nodes in a graph. The primary difference lies in the presence of a heuristic function in A* that estimates the cost from any given node n to the destination node. The A* search algorithm uses the following equation:

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

Where:

- $g(n)$ is the actual cost to get from the initial node to node n, calculated as the sum of the costs of the outgoing edges from the initial node to node n.
- $h(n)$ is the heuristic or estimated cost from node n to the destination node.
- $f(n)$ represents the estimated total cost of the cheapest solution through node n.

A* selects the next node to explore based on the lowest value of f(n), thereby preferring nodes with the lowest estimated total cost to reach the goal [5]. The following is an illustrative example of the A* algorithm.

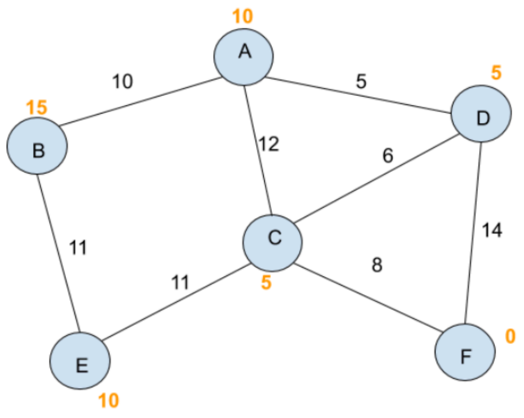


*Figure 1.4: Sample Graph for A**

In this explanation, we will refer to the actual cost as the g-score and the estimated/heuristic total cost of the solution as the f-score. We will find the shortest path between nodes A and F. Initialize the start node (A) with a g-score of 0 and an f-score of 10, which is derived from the heuristic function for node A. In the A* algorithm, the node selected for exploration is the one with the lowest f-score. Initially, since node A is the only explored node, it has the lowest f-score, while all other nodes are considered to have infinite f-scores.

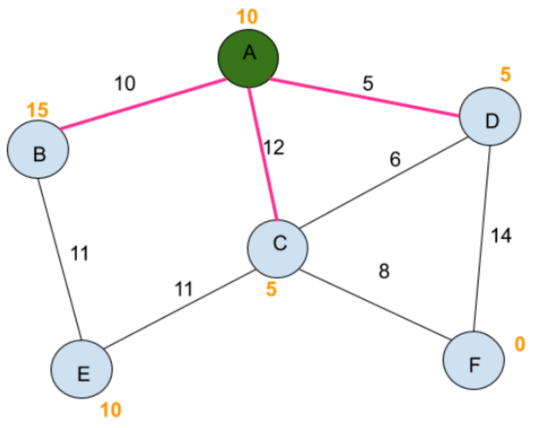


Figure 1.5 Node A Expanded

After visiting node A, we expand and explore its connected branches to find the next node to visit. Using the equation $f(n) = g(n) + h(n)$, we calculate the f-scores: A to B is 25, A to C is 17, and A to D is 10. Since the path from A to D has the lowest f-score, we choose to visit node D.

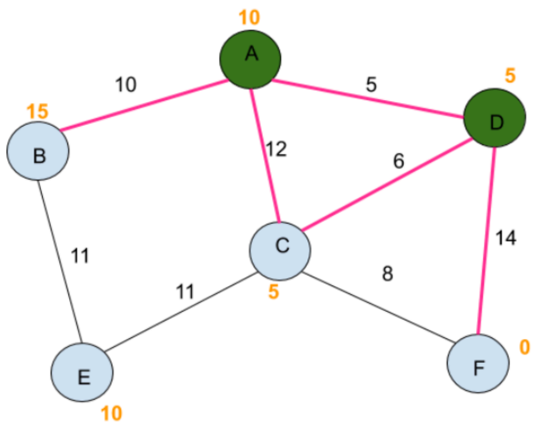


Figure 1.6: D Node Visited

Upon visiting node D, we examine its branches and calculate their f-scores: B has an f-score of 25, C has an f-score of 11, and F has an f-score of 19. We select the path from D to C since 11 is the lowest f-score among the options.

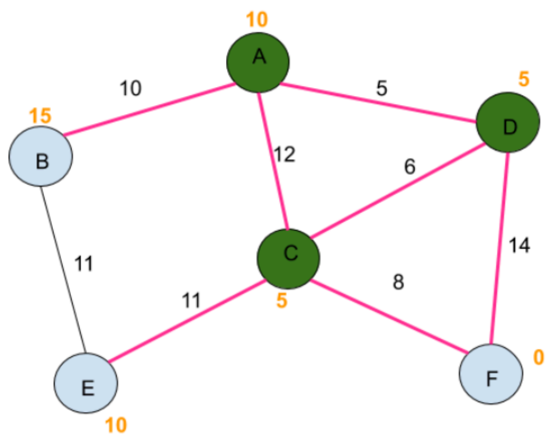


Figure 1.7: C Node Visited

With C as the current node, we explore its possible paths. The f-score from C to E is 32, derived from the sum of C's g-score (11), the weight of the edge C → E (11), and the heuristic value of E (10). The f-score from C to F is 19, calculated from C's g-score (11), the weight of the edge C → F (8), and the heuristic value of F (0). We choose the path from C to F, successfully reaching our destination node. However, we still need to confirm the shortest path. To determine this, we trace back from node F using the parent records. The previous node to F is D, and the previous node to D is A. Thus, the shortest path from A to F is A → D → F [6].

II. Pseudocode

Uniform-Cost Search

The code implementation for the Uniform-Cost Search algorithm is inspired by multiple sources on the internet [2, 3, 4, 7]. Below is the pseudocode for the code implementation.

```
Algorithm UniformCostSearch
    Input: graph (a dictionary representing the graph)
    Variables: frontier (priority queue), cost_so_far (dictionary), came_from (dictionary),
visited_order (list)

    Function Initialize(graph)
        Set self.graph to graph
    End Function
```

Figure 2.1: Uniform Cost Search Algorithm Class and Initialization

The initialization process involves defining a class named UniformCostSearch that accepts a graph as input during its instantiation. This graph is then assigned to an instance variable within the class.

```
 Function search(start, goal)
        Initialize priority queue frontier with (0, start)
        Set cost_so_far[start] to 0
        Set came_from[start] to None
        Initialize visited_order as empty list

        While frontier is not empty
            Pop the node with the lowest cost from frontier into (current_cost,
current_node)
            Append current_node to visited_order

            If current_node is goal
                Return visited_order, current_cost

            For each neighbor, cost in graph[current_node].items()
                Calculate new_cost as current_cost + cost
                If neighbor is not in cost_so_far or new_cost < cost_so_far[neighbor]
                    Set cost_so_far[neighbor] to new_cost
                    Push (new_cost, neighbor) into frontier
                    Set came_from[neighbor] to current_node

        Return visited_order, infinity (if goal is not reachable)
     End Function
```

Figure 2.2: Search Function of Uniform Cost Search

The search function begins by initializing a priority queue, frontier, with the start node and a cost of zero. It sets the cost to reach the start node to zero in the cost_so_far dictionary and notes that there is no preceding node with came_from[start] set to None. Then, an empty list, visited_order, is initialized to keep track of the nodes in the order they are visited.

Next, the function enters a loop that continues as long as the priority queue is not empty. Within the loop, it removes the node with the lowest cost from the frontier and assigns it to current_cost and current_node, then appends current_node to visited_order. If the current_node is the goal, it returns the visited_order and the current_cost.

For each neighbor and its associated cost in the adjacency list of current_node, a new_cost is calculated as the sum of current_cost and the edge cost. Suppose the neighbor has not been visited or the new_cost is less than the previously recorded cost. In that case, the function updates the cost_so_far for that neighbor, pushes the neighbor and its new_cost onto the frontier, and records the current_node as the predecessor of the neighbor it came from. If the goal is not reachable, the function returns visited_order and infinity.

```
     Function update_graph(new_graph)
         Set self.graph to new_graph
    End Function
End Algorithm
```

Figure 2.3: Update Graph Function of Uniform Cost Search

After that, the update_graph method resets the current graph and its values with a new graph.

A* Search

The A* Search algorithm's code implementation is taken from a number of online sources [5, 6, 8, 9]. The pseudocode for our code implementation is shown below.

```
Function aStarSearch(graph, start, goal, heuristic_values, visualize_step)
    Initialize open_nodes as a priority queue
    Push (0, start) to open_nodes

    Initialize actual_costs with start: 0
    Initialize function_values with start: heuristic_values[start] (or 0 if not in
heuristic_values)
    Initialize parent_records as an empty dictionary
    Initialize visited_nodes as an empty set

    Initialize traversed_path as an empty list
    Initialize nodes_expanded to 0
    Initialize max_frontier_size to 0
    Initialize visit_count with start: 1

    While open_nodes is not empty:
        Update max_frontier_size to the maximum of its current value and the size of
open_nodes
        Pop the node with the smallest function value from open_nodes and set it to current
        If current is in visited_nodes:
            Continue to the next iteration of the loop
        Add current to visited_nodes
        Append current to traversed_path
        Increment nodes_expanded by 1
        If current equals goal:
            Initialize path as an empty list
            While current is in parent_records:
                Append current to path
                Set current to parent_records[current]
            Append start to path
            Reverse path
            Set total_cost to actual_costs[goal]
            Return path, traversed_path, total_cost, nodes_expanded, max_frontier_size,
visit_count
        Call visualize_step(current, visited=True)
        For each neighbor_node, cost in graph[current]:
            Set accumulative_cost to actual_costs[current] + cost
            If neighbor_node is not in actual_costs or accumulative_cost is less than
actual_costs[neighbor_node]:
                If neighbor_node is not in visited_nodes:
                    Set parent_records[neighbor_node] to current
                    Set actual_costs[neighbor_node] to accumulative_cost
                    Set function_values[neighbor_node] to accumulative_cost +
heuristic_values[neighbor_node] (or 0 if not in heuristic_values)
                    Push (function_values[neighbor_node], neighbor_node) to open_nodes

                    Update visit_count for neighbor_node, incrementing it by 1 if it exists,
otherwise set to 1

                    Call visualize_step(current, neighbor_node)

        Sleep for 0.5 seconds

    Return None, traversed_path, infinity, nodes_expanded, max_frontier_size, visit_count
```

Figure 2.3: A Search Function*

The code initializes a priority queue, open_nodes, with the start node set to a priority of zero, using Python's heapq to retrieve nodes with the lowest function cost efficiently. It maintains dictionaries for actual costs (actual_costs) and A* function values (function_values), a dictionary for parent nodes (parent_records), and a set for visited nodes (visited_nodes).

During the search, while open_nodes is not empty, the algorithm extracts the node with the lowest function value. If this node is the goal, it reconstructs and returns the path. For each neighbor, it calculates a new path cost and updates records if the new path is better. If no path is found by the end, it returns None and infinity.

The setup involves considering each node with unknown travel costs (g-score) and estimated total costs (f-score). Nodes are managed in "unvisited" and "visited" lists, with the start node's g-score initialized to zero and its f-score based on a heuristic function.

In the exploration loop, the algorithm picks the node with the lowest f-score from the "unvisited" list, examines its neighbors, updates costs and records for better paths, and moves fully explored nodes to the "visited" list (visit_count). Simultaneously, a traversed_path list is kept for a clean record of all nodes visited in order. The variable nodes_expanded is also incremented. If the goal node is found, the path is traced back using "previous node" pointers. If no path is found and "unvisited" is empty, it indicates no possible path exists between the start and goal.

At the end of the function, variables path, traversed_path, total_cost, nodes_expanded, max_frontier_size, visit_count will be returned.

III. Results and Analysis

In this section of the report, the performance comparison will be explored in terms of time complexity, memory usage and complexity, efficiency and optimality, and the researchers' recommendations moving forward.

A. Program Outputs

To gain insight into the differences between the outputs of UCS and A Star Search, a similar test case was used to run both algorithms. The results of having Dallas as start node and Chicago as goal node can be seen below.

Uniform Cost Search

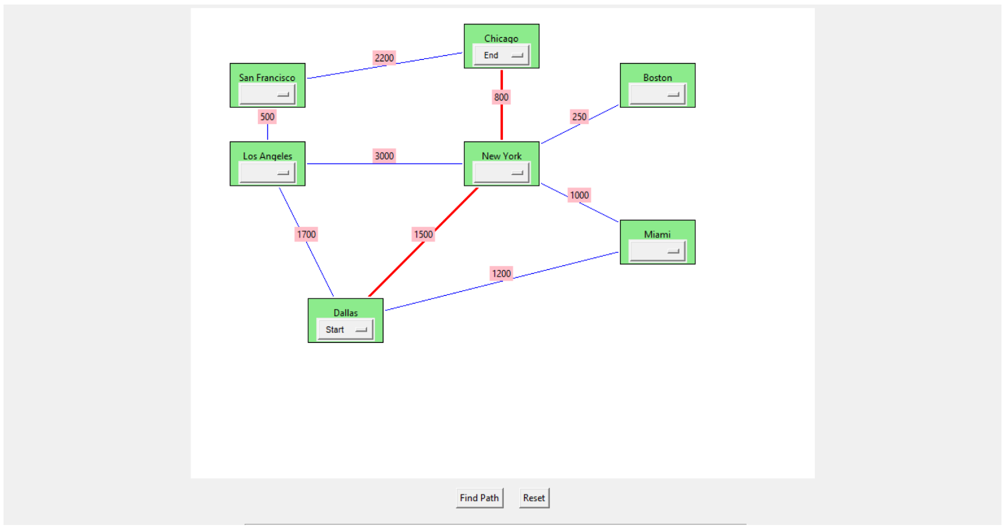


Figure 3.1: Uniform Cost Search Algorithm Used on Dallas to Chicago Path

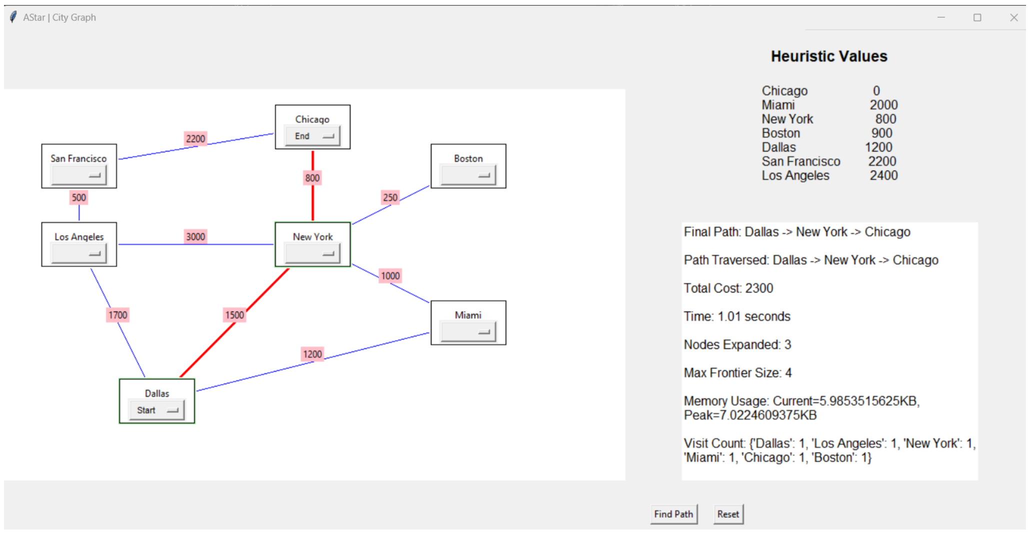
```
UCS Dallas to Chicago Output:

Final Path: Dallas -> New York -> Chicago
Path Traversed: Dallas -> Miami -> New York -> Los Angeles ->
Boston -> San Francisco -> Chicago
```

```
Total Cost: 2300
Time: 6.530240058898926 seconds
Nodes Expanded: 7
Max Frontier Size: 3
Memory Usage: Current=6.0478515625KB, Peak=6.7412109375KB
Visit Count: {'Dallas': 1, 'Los Angeles': 1, 'Miami': 1, 'New
York': 1, 'Chicago': 1, 'Boston': 1, 'San Francisco': 1}
```

Figure 3.2: Uniform Cost Search Algorithm Output for Dallas to Chicago Path

A* Search



Heuristic Values

Chicago	0
Miami	2000
New York	800
Boston	900
Dallas	1200
San Francisco	2200
Los Angeles	2400

```
Final Path: Dallas -> New York -> Chicago

Path Traversed: Dallas -> New York -> Chicago

Total Cost: 2300

Time: 1.01 seconds

Nodes Expanded: 3

Max Frontier Size: 4

Memory Usage: Current=5.9853515625KB,
Peak=7.0224609375KB

Visit Count: {'Dallas': 1, 'Los Angeles': 1, 'New York': 1,
'Miami': 1, 'Chicago': 1, 'Boston': 1}
```

Figure 4.1: A Search Algorithm Used on Dallas to Chicago Path*

```
A* Dallas to Chicago Output:

Final Path: Dallas -> New York -> Chicago
Path Traversed: Dallas -> New York -> Chicago
Total Cost: 2300
Time: 1.01 seconds
Nodes Expanded: 3
Max Frontier Size: 4
Memory Usage: Current=5.9853515625KB, Peak=7.0224609375KB
Visit Count: {'Dallas': 1, 'Los Angeles': 1, 'New York': 1,
'Miami': 1, 'Chicago': 1, 'Boston': 1}
```

Figure 4.2: A Search Algorithm Output for Dallas to Chicago Path*

B. Time Complexity

UCS Time Complexity

The time complexity of the Uniform Cost Search (UCS) algorithm is expressed as $O(b^d)$ where:

b: The branching factor, representing the maximum number of successors for any node.

d: The number of nodes in the resulting path computed as C/ε where:

C: The cost of the optimal solution.

ε: The minimum cost between any two states.

The Uniform Cost Search algorithm prioritizes nodes with the lowest accumulated cost for exploration. Its time complexity is exponentially dependent on the ratio of C to ϵ. When ϵ is relatively large compared to C, the time complexity remains manageable. Conversely, if ϵ is much smaller than C, the time complexity increases significantly and may become impractical [4].

A* Search Time Complexity

The worst time complexity of the A* Search algorithm is expressed as $O(b^d)$, where:

b: The branching factor, representing the maximum number of successors for any node.

d: the number of nodes in the resulting path.

In a given graph, it might be necessary to traverse all edges to reach the destination node from the source node. Therefore, the worst-case time complexity of the A* algorithm can be $O(b^d)$ or O(E), where E is the number of edges in the graph [9]. In certain cases where heuristics allow the algorithm to make more optimal decisions, the time complexity can drop due to fewer node expansions.

Comparison

While the worst-case time complexity for both algorithms is similar, a noticeable difference in running time is observed in the output for the test case above (Dallas to Chicago). We can see a clearer representation of the running times below.

	UCS	**A***
Running Time	6.5302 seconds	1.0100 seconds
Nodes Expanded	7	3

Based on the results, A* search executed significantly faster than UCS. The A* search used heuristic information to prioritize promising paths, leading to fewer node expansions and a shorter running time.

C. Memory Complexity

In terms of memory complexity, UCS requires memory to store the explored frontier, which tends to grow considerably larger in big search spaces. As previously mentioned, A* also utilizes a frontier however it is considerably smaller because it focuses on paths that are closer to the goal depending on their heuristic value. Overall, UCS tends to be more memory-hungry due to its exhaustive exploration strategy. A* Search usually has a smaller memory footprint due to its directed search based on heuristic values. However, it is important to note that the difference in memory usage between these two algorithms may be negligible in small search spaces and extremely complex heuristic functions.

Comparison

	UCS	**A***
Current Memory Usage	6.0478515625KB	5.9853515625KB
Peak Memory Usage	6.7412109375KB	7.0224609375KB

The results demonstrate the negligible difference in memory usage between UCS and A* Search, as evidenced by the peak consumption difference of only about 0.3 KB. A* tends to be faster, especially when the heuristic values are close to accurate, as seen in the table above. However, the UCS's current usage and peak usage are more similar to each other, which may indicate that the algorithm is more consistent and reliable.

D. Efficiency and Optimality

When using the UCS algorithm, it is guaranteed that you will find the optimal solution since it explores every possible path. In contrast, A* search relies on estimations, which means there is a tendency for the shortest path found to be inaccurate. A* can potentially find a suboptimal path that is faster but not necessarily the shortest. In terms of efficiency, UCS is very meticulous with its checking, which tends to make it slower than the heuristic algorithm. A* prioritizes exploring paths that seem closer, making it generally faster in most cases.

In the tests conducted based on the given problem, both UCS and A* found the optimal path from Dallas to Chicago with a total cost of 2300. However, A* significantly outperformed UCS in terms of efficiency. UCS took 6.53 seconds, expanded 7 nodes, and traversed a longer path, while A* completed the search in just 1.01 seconds and expanded only 3 nodes with a more direct path. Despite UCS having a slightly lower peak memory usage (6.741 KB compared to A*'s 7.022 KB), A* was faster and used less current memory (5.985 KB vs. UCS's 6.048 KB). Although A* would generally have a lower frontier size than UCS, it had a larger max frontier size than UCS (4 compared to UCS's 3), indicating the given heuristic values have a significant effect on the nodes the algorithm taps. From these results, A* is the more efficient algorithm in this test case.

E. Recommendations

Strengths and Weaknesses Of Uniform Cost Search

The main strength of Uniform Cost Search is that it is complete and you are guaranteed a shortest path regardless of the time it takes or the memory it consumes. It has an advantage against A* especially in cases where the heuristic is unreliable. However, it can be inefficient in large search spaces since it exhausts all options and explores every path. A good recommendation for UCS would be to implement some adaptations that could potentially reduce overall exploration costs such as iterative deepening search.

Strengths and Weaknesses of A Search*

The A* search algorithm is strong and efficient, especially when the heuristic is accurate. One of its main strengths is its speed and optimality even in large search spaces. To improve the A* search algorithm, designing better heuristics is crucial. Programmers can also implement dynamical adjustment of heuristic values during a search to improve the algorithm's behavior significantly.

IV. References

[1] Javatpoint, “Uninformed Search Algorithms - Javatpoint,” *www.javatpoint.com*, 2011. https://www.javatpoint.com/ai-uninformed-search-algorithms

[2] N. Thakkar, “Uniform-Cost Search Algorithm,” *Scaler Topics*, Sep. 22, 2023. https://www.scaler.com/topics/uniform-cost-search/

[3] M. Simic, “Uniform-Cost Search vs. Best-First Search,” *Baeldung*, Oct. 15, 2021. https://www.baeldung.com/cs/uniform-cost-search-vs-best-first-search

[4] GeeksforGeeks, “Uniform-Cost Search (Dijkstra for large Graphs),” *GeeksforGeeks*, Mar. 25, 2019. https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

[5] Javatpoint, “Informed Search Algorithms in AI - Javatpoint,” *www.javatpoint.com*, 2011. https://www.javatpoint.com/ai-informed-search-algorithms

[6] Ada Computer Science, “A* search algorithm,” *Ada Computer Science*. https://adacomputerscience.org/concepts/path_a_star?examBoard=all&stage=all

[7] S. Sryheni, “Obtaining the Path in the Uniform Cost Search Algorithm,” *Baeldung*, Mar. 18, 2024. https://www.baeldung.com/cs/find-path-uniform-cost-search

[8] A. S. Ravikiran, “A* Algorithm Concepts and Implementation,” Simplilearn, Feb. 16, 2024. https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm

[9] R. Belwariar, “A* Search Algorithm,” *GeeksforGeeks*, Sep. 07, 2018. https://www.geeksforgeeks.org/a-search-algorithm

V. Contributions of Each Member

Name	Contribution
Ariaga, Marian Ricci N.	● Wrote the introduction and pseudocode portions of the report
Guillarte, Dana Louise A.	● Coded the Uniform Cost Search program and algorithm (backend) ● Assisted in writing pseudocode report
Llorando, Yesha P.	● Coded the final running app and graphical user interface for both algorithms
O’Neil, Samantha Erica R.	● Wrote the results and analysis portion of the report
So, Chrysille Grace L.	● Coded the A* Search program and algorithm (backend) ● Assisted in pseudocode reports, analysis, and proofreading.