State-Based Model (MCO1)

In Partial Fulfillment of the Requirements for the Course:

CSINTSY (Introduction to Artificial Intelligence)

Submitted by:

Dwight Daryl J. Astrero

Andrei Zarmin D. De Jesus

Rafael Luis L. Navarro

John Cristian N. Sayat

Matthew Gavin A. Tiopes

Submitted to:

Dr. Norshuhani Zamin

November 04, 2024

I. Introduction

In today's modern computing systems, solving pathfinding algorithms is crucial in several applications, including robotics, network routing, and artificial intelligence [1]. The common problem usually found in a state-based model is which path is optimal among different possibilities from a start point to a goal in a search space. However, pathfinding includes difficulties where an unpredictable environment applies to computing resources, and the requirement to explore all possible paths can be critical. For instance, pathfinding resources like reviews, price searches, or, specifically, navigation applications have become a long-run service over the years as users employ them as a daily guide for transportation [2].

To construct such an application, the group proposes utilizing and scrutinizing a state-based algorithm to calculate the shortest path from source to destination. As a preference, the De La Salle University Food Map contains various points of destination for malls, restaurants, and food stalls. Using two distinct search algorithms, the group suggests using Uniform Cost Search (UCS) as a blind search algorithm and A* as a heuristics search method. The reason behind choosing these methods is that they are often considered optimal and efficient for various cases. This dynamic proposal would be an advantageous approach to seeking the search algorithm's optimality and efficiency, which can be used in many applications. This project seeks to implement a user-centered methodology for optimum pathfinding on the DLSU Food Map, emphasizing efficiency. The report also aims to compare UCS with A* to discover which algorithm produces the shortest and most efficient pathway, offering insights for analogous applications in dynamic contexts.

II. Methodology

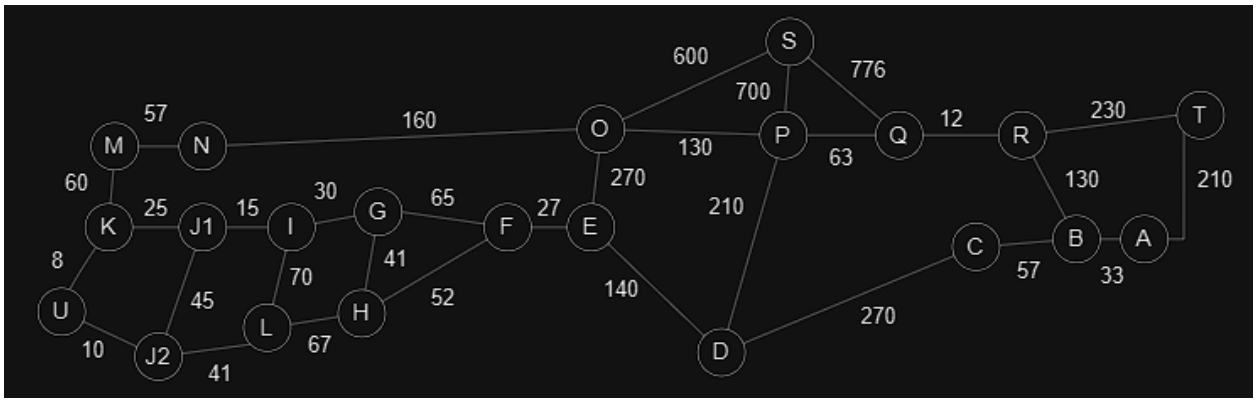

Fig 1. **Undirected Graph of the DLSU Food Map**

The program involves implementing and utilizing the Uniform Cost Search (UCS) and A* search algorithms to locate the optimal path between nodes in a graph. As seen in Fig. 1, it provides an undirected graph of available food destination nodes and various edges connected between nodes, which are computed based on the distance measurement (in meters) inside the map. The graph represents locations, and the proposed algorithms will be used to determine the most efficient path based on given criteria.

Uniform Cost Search (UCS)

The Uniform Cost Search or UCS algorithm implemented in this program is designed to find the shortest path from a start node to a goal node in a graph. The graph shown in Fig. 1 uses this program for both UCS and A* algorithms. The algorithm finds the shortest path by comparing the cumulative path costs of each node and then extending the children of the node with the lowest cumulative path cost. This process repeats until the node with the lowest path cost is a goal node. This algorithm is considered a blind search because no heuristic values are considered in the path cost calculation.

The actual graph, including every node, edge, and cost of each edge, is stored in the "*connectionsMap*" data structure. At the start of the actual traversal, some data structures are initialized. These are a priority queue (*openSet*), a closed set (*closedSet*), a cost map (*gScore*), and a map used to track the best path (*cameFrom*). The priority queue (*openSet*) contains the nodes that can currently be explored and their cumulative path cost, prioritized by the lowest path cost. The closed set (*closedSet*) includes the nodes that have already been fully explored so that the algorithm does not waste time re-evaluating them. The cost map (*gScore*) stores the cumulative path cost of each node from the start node. Finally, the *cameFrom* map stores the best path and is used at the end of the algorithm to restore the shortest path from the start node to the goal node.

The main loop of the program is executed and only stops if either *openSet* is empty (all nodes have been fully explored without finding the goal node) or if a goal node is found. For each iteration, the head of the priority queue *openSet* is dequeued and designated as the current node. It is then checked if it is the goal node. If it is, the total cost of the shortest path is printed, and then the actual path is reconstructed using the *cameFrom* map, which is printed from the start node to the goal node, and the algorithm ends there.

If the current node is not a goal node, then it is added to *closedSet,* and its neighbors/children are explored. These neighbor nodes, edges, and costs are taken from *connectionsMap*. Each neighbor of the current node is checked if it was already explored (if it is in *closedSet*). If it is already explored, we move to the next neighbor node. If it has not been studied yet, the cumulative path cost or *tentativeGScore* of the current neighbor node is

calculated. When this *tentativeGScore* is less than the previously recorded path cost or gScore of the neighbor node in the *gScore* map or if the neighbor node has no recorded gScore yet in the map, the *cameFrom* path is updated to include the neighbor node, the *gScore* map is updated with the neighbor node's *tentativeGScore*, and the neighbor node is added to *openSet* along with its *tentativeGScore.* After all that, the main loop continues until a goal node is found at the head of *openSet* or all the nodes in *openSet* have been explored without finding a goal node.

A* Heuristic

The A* algorithm implemented in the program is designed to find the shortest path from a start node to a destination node within a graph. This graph is represented by nodes connected by edges, with actual travel costs sourced from Google Maps stored in a data structure called "*connectionsMap*." Each edge in this map reflects the travel distance between two nodes, allowing the algorithm to calculate the most efficient route.

In addition to the actual costs, the algorithm incorporates heuristic values stored in the *parentNodeHeuristicMap*, which estimate the number of nodes left to reach the goal. This heuristic function *h(n)* was explicitly chosen to prioritize paths that minimize the number of remaining nodes. In real-world scenarios, such as navigating through areas with multiple eateries close together, the paths leading to these locations can become congested due to increased foot traffic. This congestion can significantly slow down travel times, making it essential to find routes considering distance and the likelihood of encountering delays. The total score for each node, *f(n)*, is calculated as the sum of *g(n),* the cost of reaching the current node, and *h(n)*, the estimated remaining cost to the goal. This combination allows the A* algorithm to efficiently

balance the actual travel costs with the heuristic estimates, guiding it toward the most promising paths in the search for the optimal route.

The algorithm begins by initializing several data structures: a priority queue (*openSet*), a cost map (*gScore*), a closed set (*closedSet*), and a map for tracking the optimal path (*cameFrom*). The priority queue, *openSet*, is used to store nodes along with their calculated *f(n)* values, which are a combination of the actual path cost *g(n)* from the start node and the heuristic *h(n)* from the current node to the goal. The *gScore* map holds the lowest cost from the start node to each node encountered. The *closedSet* keeps track of nodes that have already been fully explored, preventing redundant processing. At the same time, the *cameFrom* map records the parent node for each node, allowing for path reconstruction once the goal is reached.

The algorithm's main loop continues until *openSet* is empty or the goal node is reached. At each iteration, the node with the lowest *f(n)* value is retrieved from *openSet* and set as the *current* node. The algorithm checks if this node is the goal; if so, it terminates the search, calculates the total path cost from *gScore*, and uses the *cameFrom* map to reconstruct the entire path by tracing from the goal node back to the start. This path is then printed, showing the nodes traversed and the corresponding node names.

The algorithm explores all its neighbors for each node, as retrieved from the *connectionsMap*. For each neighbor, it checks if the neighbor has been thoroughly explored by referencing the *closedSet*. If the neighbor has not been visited, the algorithm calculates a tentative score, *tentativeGScore*, by adding the current node's *gScore* and the actual cost of

reaching the neighbor. Suppose this tentative score is lower than any previously recorded score for the neighbor in *gScore. In that case*, the algorithm updates *cameFrom* to record the *current* node as the optimal parent for the neighbor and updates *gScore* with the lower tentative score. The new *f(n)* value for the neighbor, combining *tentativeGScore* and the heuristic value, is added to *openSet*, ensuring that nodes with the lowest *f(n)* values are prioritized in the next iteration. The neighbor and its *f(n)* score are printed for verification.

Once the goal node is reached, the algorithm reconstructs the path by following the *cameFrom* map from the goal node back to the start node. This path is stored in reverse order to facilitate efficient backtracking. The path is then reversed and printed to display the sequence of nodes from start to goal.

III. Results and Analysis

```
!! DLSU FOOD NODES !!
University Mall(A = 0)
McDonald's(B = 1)
Perico's(C = 2)
Bloemen Hall(D = 3)
W.H Taft Residence(E = 4)
EGI TAFT(F = 5)
Castro Street(G = 6)
Agno Food Court(H = 7)
One Archers'(I = 8)
Andrew's La Casita(J1 = 9)
Enrique's La Casita(J2 = 10)
Green Mall(K = 11)
Green Court(L = 12)
Sherwood(M = 13)
Jollibee(N = 14)
Dagonoy St.(O = 15)
Burgundy(P = 16)
Estrada St.(Q = 17)
D' Student's Place(R = 18)
Leon Guinto St.(S = 19)
P.Ocampo St.(T = 20)
Fidel A. Reyes St.(U = 21)
```

***Fig. 2*. Variation of Food Nodes of DLSU and its int values**

The program overview lets users indicate multiple options on the following nodes, representing food locations along DLSU. The program reads a file called "*finaldata.txt*" containing the data about the nodes and their corresponding connections. The heuristic value of each node is automatically set to 0 and will only be updated after the user has entered a valid starting and end node. The program provides options to view node details and heuristic values from initial to goal nodes and test two proposed algorithms such as UCS and A* Search.

To analyze the difference between algorithms, the group tested two destinations to compare heuristic values and the total cost of the path. Destination A starts at node A or the University Mall, and a goal path to node O or Dagonoy Street. Destination B starts at node U or Fidel A. Reyes Street and goes to node I or One Archers' Place.

Heuristic Values

```
**********************************************
<NODE, HEURISTIC VALUE>
Node: (0 = A), H(n): 5.0
Node: (1 = B), H(n): 4.0
Node: (2 = C), H(n): 3.0
Node: (3 = D), H(n): 2.0
Node: (4 = E), H(n): 1.0
Node: (5 = F), H(n): 2.0
Node: (6 = G), H(n): 3.0
Node: (7 = H), H(n): 3.0
Node: (8 = I), H(n): 4.0
Node: (9 = J1), H(n): 4.0
Node: (10 = J2), H(n): 5.0
Node: (11 = K), H(n): 3.0
Node: (12 = L), H(n): 4.0
Node: (13 = M), H(n): 2.0
Node: (14 = N), H(n): 1.0
Node: (15 = O), H(n): 0.0
Node: (16 = P), H(n): 1.0
Node: (17 = Q), H(n): 2.0
Node: (18 = R), H(n): 3.0
Node: (19 = S), H(n): 1.0
Node: (20 = T), H(n): 4.0
Node: (21 = U), H(n): 4.0
```

```
**********************************************
<NODE, HEURISTIC VALUE>
Node: (0 = A), H(n): 7.0
Node: (1 = B), H(n): 6.0
Node: (2 = C), H(n): 5.0
Node: (3 = D), H(n): 4.0
Node: (4 = E), H(n): 3.0
Node: (5 = F), H(n): 2.0
Node: (6 = G), H(n): 1.0
Node: (7 = H), H(n): 2.0
Node: (8 = I), H(n): 0.0
Node: (9 = J1), H(n): 1.0
Node: (10 = J2), H(n): 2.0
Node: (11 = K), H(n): 2.0
Node: (12 = L), H(n): 1.0
Node: (13 = M), H(n): 3.0
Node: (14 = N), H(n): 4.0
Node: (15 = O), H(n): 4.0
Node: (16 = P), H(n): 5.0
Node: (17 = Q), H(n): 6.0
Node: (18 = R), H(n): 7.0
Node: (19 = S), H(n): 5.0
Node: (20 = T), H(n): 8.0
Node: (21 = U), H(n): 3.0
```

Fig 3. **Heuristic Values of Destination A and B**

Based on the given output (Fig. 3), Destinations A (A => O) and B (U=>I) have distinct values to get into the goal nodes. The heuristic values decrease as nodes get closer to the destination. Values with 0.0 indicate it is the destination node (Node O & I). The output highlights how the heuristic values change based on the given destination node, reflecting the estimated cost to reach the destination from each node.

Cost of Path

```
(Main Node) -> [CostPath, Neighbour Node]
(A = 0) -> [33, B] [210, T]
(B = 1) -> [57, C] [130, R] [33, A]
(C = 2) -> [270, D] [57, B]
(D = 3) -> [140, E] [210, P] [270, C]
(E = 4) -> [27, F] [270, O] [140, D]
(F = 5) -> [65, G] [52, H] [27, E]
(G = 6) -> [30, I] [41, H] [65, F]
(H = 7) -> [67, L] [41, G] [52, F]
(I = 8) -> [15, J1] [70, L] [30, G]
(J1 = 9) -> [25, K] [45, J2] [15, I]
(J2 = 10) -> [10, U] [45, J1] [41, L]
(K = 11) -> [60, M] [8, U] [25, J1]
(L = 12) -> [41, J2] [70, I] [67, H]
(M = 13) -> [57, N] [60, K]
(N = 14) -> [57, M] [160, O]
(O = 15) -> [160, N] [270, E] [600, S] [130, P]
(P = 16) -> [130, O] [210, D] [700, S] [63, Q]
(Q = 17) -> [63, P] [776, S] [12, R]
(R = 18) -> [12, Q] [130, B] [230, T]
(S = 19) -> [600, O] [700, P] [776, Q]
(T = 20) -> [230, R] [210, A]
(U = 21) -> [8, K]
```

***Fig 4*. Cost Path of Nodes (Edges)**

As seen in Fig. 3, the undirected graph represents a node and its edges. The program allows users to select an option to determine the costs associated with each path. Each line in the output represents a node, its connections to other nodes, and the path cost between them. The output format can be represented as "*(Main Node = int) → [Cost Path, Neighbour Node]*" to simplify the readability of each cost. The graph's connectivity is well-connected and ensures that several nodes have multiple paths compared to others, indicating potential hubs. Each cost varies significantly, ranging from 8 as the lowest to 776 as the highest; higher costs may indicate longer distances or more difficult paths. The given output in Fig. 4 provides detailed information for

understanding the graph's structure. It provides the necessary information for the following algorithms, like UCS and A*, to locate the most applicable path between nodes.

Uniform Search Algorithm

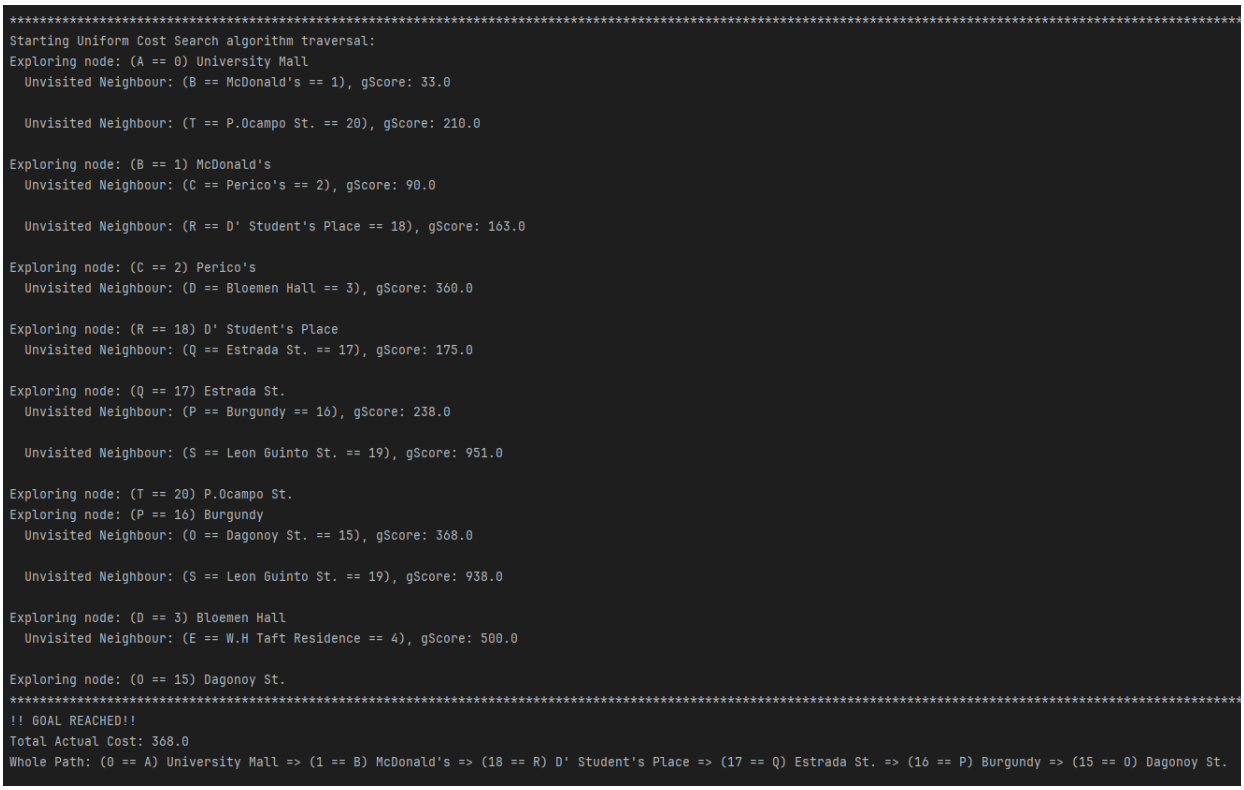

***Fig. 5*. Uniform Cost Search of Destination A**

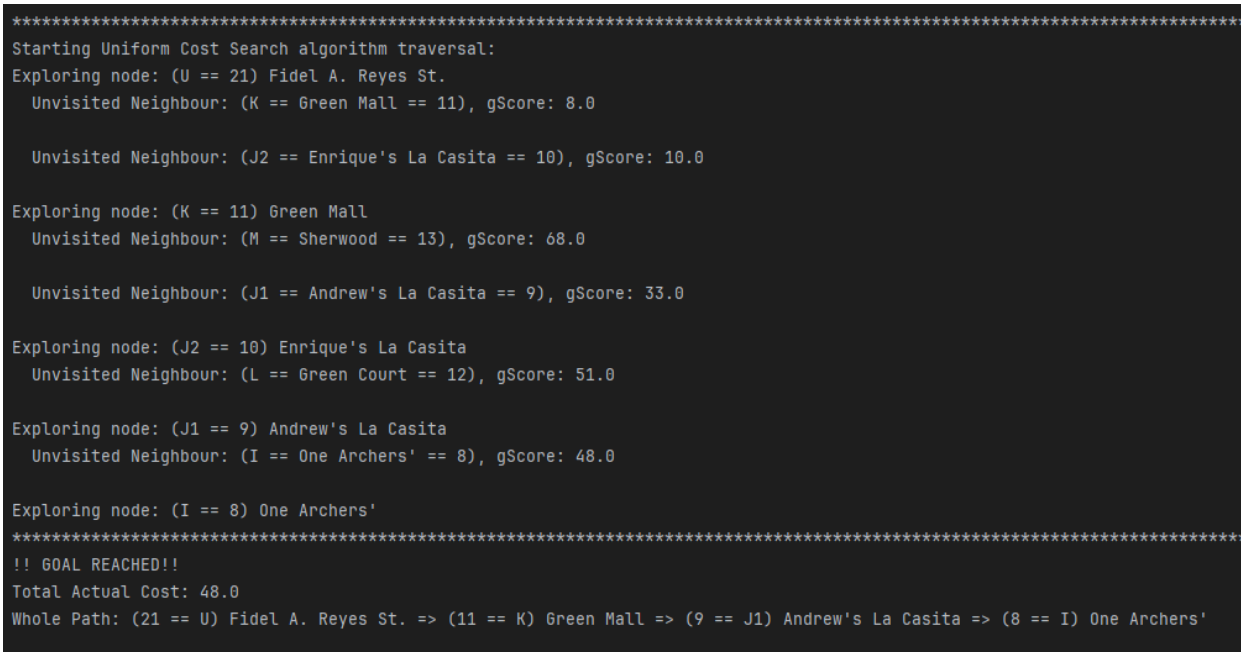

Fig. 6. **Uniform Cost Search of Destination B**

The traversals of the Uniform Cost Search (UCS) algorithm for two destinations, A to O and U to I, contrast a significant path in the output. Destination A's path consists of 6 nodes: A, B, R, Q, P, and O, with a total cost of 368.0. On the other hand, destination B has a path of U, K, J1, and J with a total cost of 48.0. In terms of efficiency, the path from destination B is shorter. It has a significantly lower cost than destination A, which is considered more efficient, and it explored only fewer nodes, resulting in a lower total cost. As a form of analysis, destination A involves more nodes and has a higher total cost. The algorithm had to find a more optimal path to explore more nodes. In other words, destination A is more complex and potentially longer despite the path length. The comparison of the two destinations highlights the significant difference in terms of size, cost, and number of neighbor nodes explored. This concludes with an insight into the complexity and efficiency of the graph and its path.

A* Algorithm

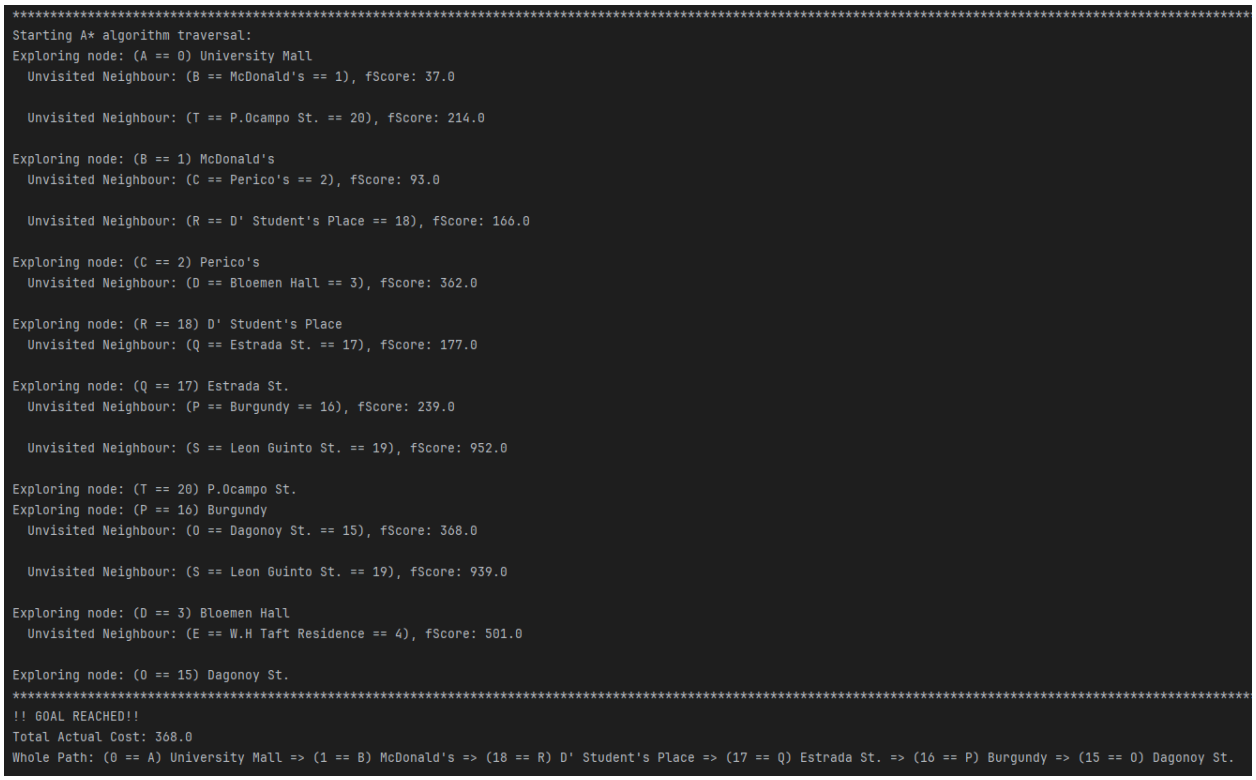

Fig. 7. **A* Search for Destination A**

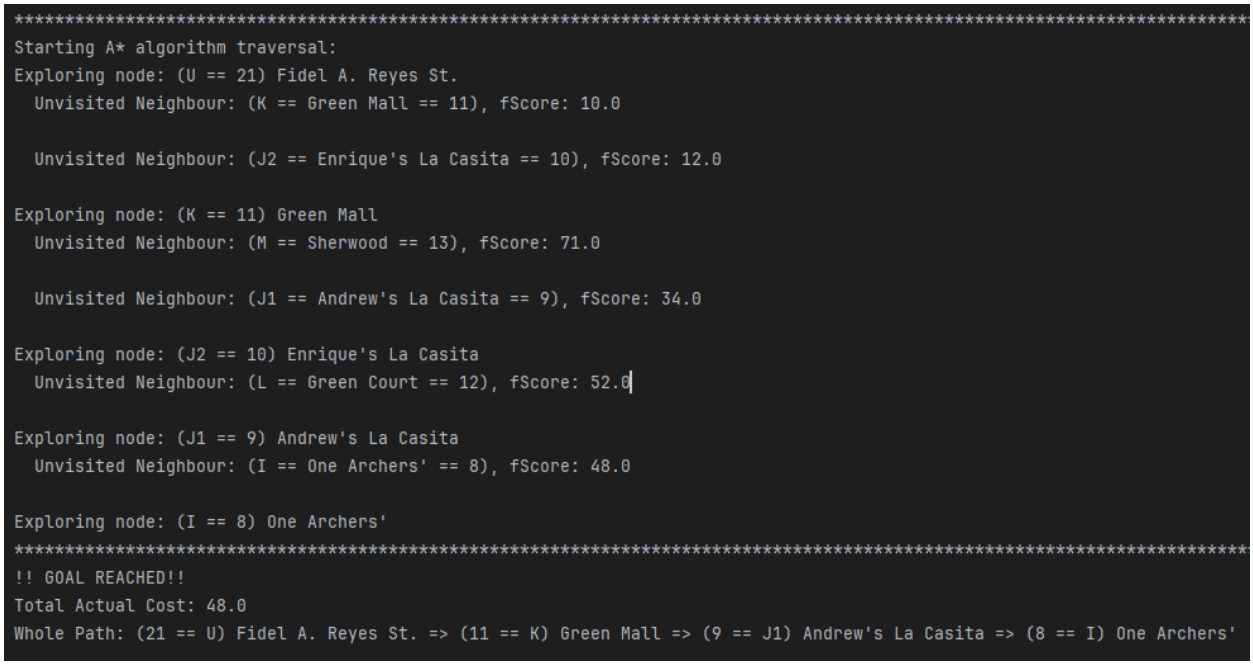

Fig. 8. **A* Search for Destination B**

A* search algorithm for two destinations was also used to provide the traversals of their paths to the goal nodes. Destination A (Fig. 7) consists of a path of 6 nodes (A, B, R, Q, P, O) with a total distance cost of 368.0, while destination B (Fig. 8) has a path length of 4 nodes (U, K, J1, L) with a total distance of 48.0. Compared to UCS, the comparison is similar to the actual cost of traversals. The only difference is that the A* algorithm effectively uses heuristics to guide the search, resulting in optimal paths for the goal nodes. In summary, the analysis of A* provides an alternative insight into the complexity of the paths between different nodes in the graph.

Comparison of UCS and A*

	Destination A	Destination B
UCS	368.0	48.0
A*	368.0	48.0

Table 1. **Cost Path of Destination A & B**

	Destination A	Destination B
UCS	A → B → R → Q → P → O	U → K → J1 → I
A*	A → B → R → Q → P → O	U → K → J1 → I

Table 2. **Path Traversals of Destination A & B**

	Destination A	Destination B
UCS	A, B, C, R, Q, T, P, D, O	U, K, J2, J1, I
A*	A, B, C, R, Q, T, P, D, O	U, K, J2, J1, I

Table 3. **Nodes Visited in Destination A & B**

The results of the destination in terms of the algorithm comparison are identical. This concludes that both algorithms are effective in searching for an optimal path. Both UCS and A*

went to the same set of nodes and their traversals, suggesting that the heuristic used in A* did not significantly reduce the number of nodes explored compared to UCS in other specified cases.

UCS can guarantee the finding of an optimal solution and does not require a heuristic value to traverse through nodes. On the contrary, it can be inefficient in terms of complexity, specifically to a more significant number of nodes with high branch factors [1]. While it does not use heuristic values to guide its pathfinding, it could lead to exploring irrelevant visited nodes. A* Search is known to combine the advantages of UCS and Greedy Best-First Search by using heuristics to provide a better optimal path. It is actually more efficient than UCS in other cases as long the heuristic is admissible. Although the efficiency of A* significantly depends on the value of the heuristic function, it may be possible to explore unnecessary nodes if the heuristic is not informative enough to provide.

In conclusion, both the UCS and A* algorithms found optimal paths for the given destination with identical results. However, due to its use of heuristics, A* can potentially be more efficient than UCS in general, specifically for larger and more complex graphs. As a result, based on food ratings, the choice between algorithms is not different. Still, it can improve depending on the specific problem and the available heuristic values the graph can contain.

IV. References

[1] A. Felner, "Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm," *Proceedings of the International Symposium on Combinatorial Search*, vol. 2, no. 1, pp. 47–51, 2021. doi: 10.1609/socs.v2i1.18191 (accessed Nov. 3, 2024).

[2] F. R. S. D. Scientist, "Pathfinding Algorithms- Top 5 most powerful," *Graphable*. [Online]. https://www.graphable.ai/blog/pathfinding-algorithms/ (accessed Nov. 1, 2024).

[3] S. Singh, "The algorithms behind the working of Google Maps," *Medium*. [Online]. https://medium.com/@sachin28/the-algorithms-behind-the-working-of-google-maps-73c379bcc9b9 (accessed Nov. 3, 2024).

V. Contributions of Each Member

Name	Detailed Contribution/s
Dwight Daryl J. Astrero	● A* Algorithm Class ● Report (A* Methodology)
Andrei Zarmin D. De Jesus	● FileReader Class ● DataRecord Class ● Checker Class ● Data information text file ● A* Algorithm Class ● Uniform Cost Search Algorithm Class ● Translation of DLSU Food Map to a Graph with the cost of a path between nodes
Rafael Luis L. Navarro	● Uniform Cost Search Algorithm Class
John Cristian N. Sayat	● Report (Introduction, Results & Analysis)
Matthew Gavin A. Tiopes	● Uniform Cost Search Algorithm Class ● Report (UCS Methodology)