Presented to the College of Computer Studies
De La Salle University - Manila
Term 3, A.Y. 2024-2025

In partial fulfillment of the course
In Introduction to Intelligent Systems (CSINTSY)

MCO1 - State-Based Models

Group No. 18

Submitted by:
Alamay, Carl Justine S.
Ang, Czarina Damienne N.
Culanag, Saimon Russel W.
Esteban, Janina Angela M.
Marinas, Carl Mervyn G.

Submitted to:
Mr. Nathaniel A. Oco

July 01, 2025

I. Introduction

There are a lot of various eateries around De La Salle University, which makes it challenging for students to decide where to eat. Factors such as distance, price, and food ratings are some of the details that might determine where a student might eat. This project aims to simulate optimal pathfinding to aid students in identifying the best-rated eatery and the shortest route to various eating establishments around DLSU. The group proposes the use of two search algorithms: (1) Uniform Cost Search (UCS) for blind search algorithms, and (2) A* search for heuristic search algorithms. This program uses the food establishment's ratings as its heuristics. The algorithms implemented could also dynamically compute the most efficient path to a selected eatery, considering the changes in the environment, like newly opened or closed eating establishments. This simulation uses a graphical user interface (GUI) to show the DLSU eateries map for visual representation. The program allows users to select an algorithm and displays the optimal path, total cost, and path average rating.

II. Methodology

This project uses state-based models, which use a set of states to represent a problem [1]. Each state captures a specific situation or information about the problem at a specific point in time [1], [2]. A weighted graph is used to illustrate the problem, where each node represents a different eatery, and its weight is the food rating of that eatery. Each edge represents the walkable paths between the eateries or nodes, and each edge cost represents the distance between the eateries or nodes. As mentioned, the algorithms chosen were Uniform Cost Search (UCS) and A* search. This simulation is also implemented using the Java programming language.

Uniform Cost Search is a type of Dijkstra's algorithm that is designed for graphs with different weights. Its main purpose is to find the path from the starting node to the destination node with the minimum total cost [3]. With this, UCS is chosen as the blind search algorithm because this could give the optimal shortest distance between the starting point and the destination. The algorithm works like trying to get somewhere while always taking the cheapest path at each step. It starts from the beginning point and looks at all the possible paths it can take. Then, it chooses the one with the lowest total cost so far without guessing what is ahead. It keeps track of every place it visits and explores the cheapest options until it reaches its destination. If

two paths have the exact cost, it chooses the one that comes first alphabetically (for instance, choosing “A” before “B”). UCS continues this process until it finds its destination or realizes no possible way exists. Figure 2-1 shows how the UCS works.

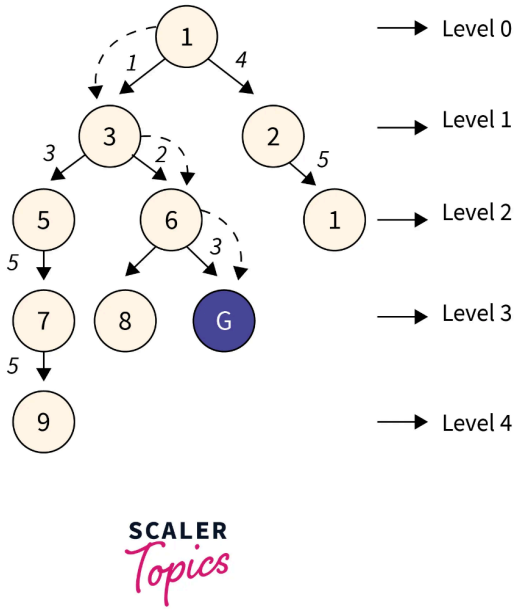


Figure 2-1. Visualization of UCS algorithm [4]

Another algorithm used is the A* search for the heuristic algorithm. The A* search algorithm is similar to UCS; it works the same way if it has a zero heuristic function. Instead of following the shortest path, it also considers which direction looks most promising. At each step, it looks at all the possible places you could go next and chooses the one that seems best based on two things: how far you’ve already traveled to get there, and how far you think you still have to go to reach your goal. It adds those two distances together and always picks the path with the lowest total. By doing this, A* does not waste time exploring paths that look long; it focuses on the smartest route. With this, Equation 2-1 shows how A* avoids extending pathways that are already more expensive [5].

$$f(n) = g(n) + h(n) \text{ where:}$$

$$f(n) = total\ estimated\ cost\ of\ route\ across\ n$$
$$g(n) = the\ cost\ to\ go\ to\ n$$
$$h(n) = estimated\ cost\ from\ n\ to\ goal$$

Equation 2-1. A* Function

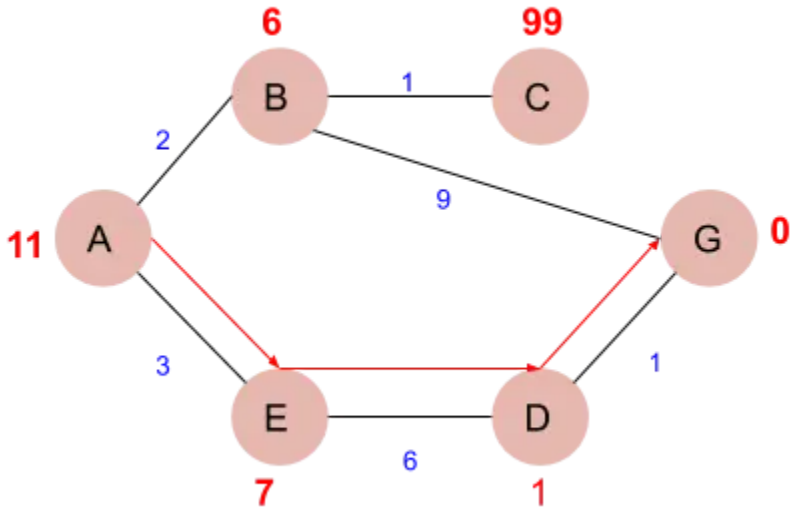


Figure 2-2. Visualization of A* search algorithm [6]

A* search is chosen as the heuristic search for this project because it could properly balance both the distances and food ratings. It could solve the problem by finding a path that is short and an eatery with a higher food rating.

The whole graph is created through the Edge, Graph, and Node classes. The Edge class has two attributes, which are the destination node's UID ($destUid$) and the edge cost ($weight$). To create a node, it should have a unique id (uid), which is automatically assigned, a name ($name$), a heuristic value (val), and coordinates ($x,\ y$). It also has multiple constructors so that a node can be created with or without a heuristic. The graph class manages all the nodes and edges through its method, initGraph(), which connects edges and nodes together. It does this by adding the nodes (addNode()), records the UIDs in Maps that records the UID and Node, and Name and UID. It also connects the edges to the nodes and adds these into adjVertices.

```java
public class Edge {
  int destUid;
  int weight;

  public Edge(int destUid, int weight) {
    this.destUid = destUid;
    this.weight = weight;
  }

  public int getdestUid() {
    return this.destUid;
  }

  public int getWeight() {
    return this.weight;
  }


}
```

Figure 2-1. Edge Class

```java
public class Node {
  private int val;
  private static int idCounter = 0;
  private int uid;
  private String name;
  private int x, y; // for heuristic (optional)


  /**
   * Class Constructor
   * @param name    Name of the node
   * @param nodeVal Value of the node (Heuristic)??
   */

  public Node(String name) {
    this.uid = idCounter++;
    this.name = name;
  }

  public Node(String name, int nodeVal){
    this.uid = idCounter++;
    this.val = nodeVal;
    this.name = name;
  }
```

```java
  // Get methods
  // Can implemenet this method by possible having same string
  // or name for a node differing with its uid
  public int getUid() {
    return this.uid;
  }

  public String getName() {
    return this.name;
  }

  public int getVal() {
    return this.val;
  }

  // A STAR
public int getX() {
    return x;
}

public int getY() {
    return y;
}

public void setVal(int val) {
    this.val = val;
}
```

Figure 2-2. Node Class

```
public class Graph {
  public static final String FILENAME = "test.csv";
  ReadFile rf;
  // Map implmentation from https://www.baeldung.com/java-graphs
  private Map<Integer, ArrayList<Edge>> adjVertices;
  public Map<String, Integer> nameToUid;
  Map<Integer, Node> UidToNode;

  /**
   * Initialize Nodes and Edges within Graph constructor.
   */
  public Graph(){
    rf = new ReadFile();
    rf.initRead(FILENAME);
    adjVertices = new HashMap<>();
    nameToUid = new HashMap<>();
    UidToNode = new HashMap<>();
    initGraph();
  };
```

```
private void initGraph() {
  for (List<String> row : rf.records) {
    if (row.size() < 3) continue;

    String fromNodeName = row.get(0);
    String toNodeName = row.get(1);

    int weight;
    try {
      weight = Integer.parseInt(row.get(2));
    } catch (NumberFormatException e) {
      weight = 0;
    }

    int heuristic = 0;
    if (row.size() >= 4) {
      try {
        heuristic = Integer.parseInt(row.get(3));
      } catch (NumberFormatException e) {
        heuristic = 0;
      }
    }

    // Create or update FROM node
    if (!nameToUid.containsKey(fromNodeName)) {
      Node newFromNode = new Node(fromNodeName, heuristic);
      addNode(newFromNode);
    } else {
      int fromUid = nameToUid.get(fromNodeName);
      UidToNode.get(fromUid).setVal(heuristic); // update heuristic if already exists
    }

    // Add undirected edge
    int fromUid = nameToUid.get(fromNodeName);
    int toUid = nameToUid.get(toNodeName);
    addEdgeByUid(fromUid, toUid, weight);
    addEdgeByUid(toUid, fromUid, weight);
  }
}
```

Figure 2-3. Graph Class

Each search algorithm has its own class. For Uniform Cost Search (UCS), UniformCost class implements this blind search algorithm. The important attributes and instances of this class are the graph (g), priority queue ($PQpaths$), visited nodes list ($visited$), starting node ID, and destination node ID ($fromUid,\ toUid$) . Initially, the algorithm starts by clearing the priority queue, creating a new visited nodes list, and adding the $initialPath$ or the root node to the priority queue.

```
// Clear and initialize data structures
PQpaths.clear();
visited = new HashSet<Integer>();

// Start with initial path containing only the source node
Path initialPath = new Path(fromUid);
PQpaths.add(initialPath);
```

Figure 2-4. Initialization of UCS

Next, it runs the main loop and executes the algorithm, where while the priority queue is not empty, it first starts at the starting node and places its UID into *currentNodeUid*. If the current node UID is the same as the destination node UID, it will exit the function as the goal has been reached. However, if the goal is not yet reached, it would add the currentNodeUid to the visited nodes list. Next, it would explore the neighbors of the current node by checking if the neighbor node is not in the visited nodes list, then extend the current path and add it to the priority queue. If no path is found, it will print out that no path has been found from the starting node to the destination node.

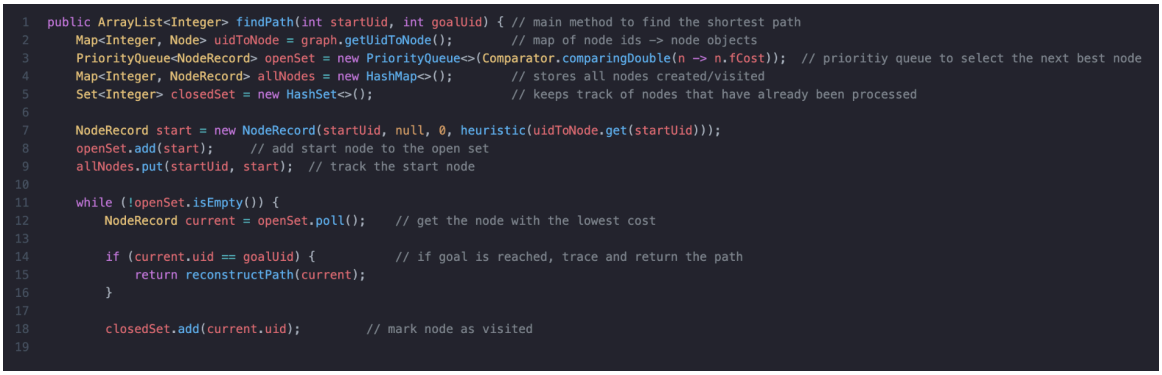
```
while (!PQpaths.isEmpty()) {

  Path currentPath = PQpaths.poll();
  int currentNodeUid = currentPath.getCurrentNodeUid();

  // Check if goal is reached
  if (currentNodeUid == toUid) {
    this.finalPath = currentPath;
    return;
  }

  if (visited.contains(currentNodeUid)) {
    continue;
  }

  visited.add(currentNodeUid);

  // Get edges from current node
  ArrayList<Edge> nodeEdges = g.getNodeEdges(currentNodeUid);
```

```
    // Explore all neighbors
    if (nodeEdges != null) {
      for (Edge edge : nodeEdges) {
        int neighborUid = edge.getdestUid();

        if (!visited.contains(neighborUid)) {
          // Create new path by extending current path
          Path newPath = currentPath.clone().addEdge(edge);
          PQpaths.add(newPath);

        }
      }
    }
  }

  // No path found
  System.out.println("No path found from " + g.getUidToName(fromUid) +
      " to " + g.getUidToName(toUid));
  finalPath = null;
  return;
}
```

Figure 2-5. Main Loop of UCS

```
1  public class AStar {
2      private Graph graph;
3
4      public AStar(Graph g) {
5          this.graph = g;
6      }
7
8      private double heuristic(Node node) {    // node's stored value
9      return node.getVal();
10 }
```

Figure 2-6. AStar Class



```
public ArrayList<Integer> findPath(int startUid, int goalUid) { // main method to find the shortest path
    Map<Integer, Node> uidToNode = graph.getUidToNode();        // map of node ids -> node objects
    PriorityQueue<NodeRecord> openSet = new PriorityQueue<>(Comparator.comparingDouble(n -> n.fCost));  // prioritiy queue to select the next best node
    Map<Integer, NodeRecord> allNodes = new HashMap<>();        // stores all nodes created/visited
    Set<Integer> closedSet = new HashSet<>();                   // keeps track of nodes that have already been processed

    NodeRecord start = new NodeRecord(startUid, null, 0, heuristic(uidToNode.get(startUid)));
    openSet.add(start);     // add start node to the open set
    allNodes.put(startUid, start);  // track the start node

    while (!openSet.isEmpty()) {
        NodeRecord current = openSet.poll();    // get the node with the lowest cost

        if (current.uid == goalUid) {           // if goal is reached, trace and return the path
            return reconstructPath(current);
        }

        closedSet.add(current.uid);         // mark node as visited

```

Figure 2-7. findPath() function

The class implements the A* pathfinding algorithm by using a Graph, Node, and Edge structure. As per the definition above, the algorithm finds the most efficient path between a starting node ($startUid$) and a goal node ($goalUid$) based on both the actual distance traveled so far ($gCost$) and an estimated distance to the goal ($heuristic$). Similar to how Google Maps finds the fastest route to your destination, it combines two things and hence prevents itself from randomly trying paths since there is a calculation of the distance traveled so far and the estimated distance. When these two numbers are added together, it helps the algorithm decide which point to explore next.

The priority queue called the $openSet$ gets set up as a program starts. What it does is that it picks up the next node with the lowest estimated total cost ($fCost = gCost + heuristic$). A map called $allNodes$ is used to keep track of all nodes visited or created, while $closedSet$ stores nodes that have already been processed so they are not accidentally revisited.

```
for (Edge edge : graph.getNodeEdges(current.uid)) {     // this for looking at the neighbors
                int neighborUid = edge.getDest();
                if (closedSet.contains(neighborUid)) continue;      // skip if neighbor already visited

                double tentativeG = current.gCost + edge.getWeight(); // calculates the cost of reaching the neighbor via selected current node

                // this is to keep track of the neighbor records (getter and setter)
                NodeRecord neighbor = allNodes.getOrDefault(neighborUid,
                        new NodeRecord(neighborUid, null, Double.POSITIVE_INFINITY, 0));
                allNodes.putIfAbsent(neighborUid, neighbor);

                if (tentativeG < neighbor.gCost) {  // compares, if new path to the neighbor is cheaper then update
                    neighbor.gCost = tentativeG;
                    neighbor.fCost = tentativeG + heuristic(uidToNode.get(neighborUid));    //
                    neighbor.parent = current;  // makes the parent trace the path later

                    openSet.remove(neighbor);   // for updating priority in the queue
                    openSet.add(neighbor);
                }
            }
        }

        return new ArrayList<>();   // if no path is found
    }
}
```

Figure 2-8. Path finding loop

The start node is initialized with a $gCost$ value of 0 and an $fCost$ based on its set heuristic value. It adds this start node to the open set and begins a loop that runs as long as there are nodes left to explore. For every new place it finds, it checks whether this new path is better (or cheaper) than the one it already knows. If the new path is better (i.e., $gCost$ is lower) than what it is set previously, then the neighbor's record is updated with the new costs, and the current node is set as its parent.

```
74    private double heuristic(Node node) {
        int weight = 2;
        if (node.getUid() == goalUid) {
            return 0;
        } else {
            // modified heuristic calculation to make the difference substantial enough
            // for both UCS and A* pathing in some nodes to
            float currentRating = node.getVal();
            float goalRating = graph.getNodeByUid(goalUid).getVal();

            float ratingFactor = (5 - currentRating) * 20;

            float goalDiff = Math.abs(currentRating - goalRating);

            return Math.pow(ratingFactor + goalDiff, 2) * weight; // added exponential scaling for heuristic
                                                                  // without this the path for AStar and UCS
                                                                  // doesn't change that much
        }
    }
```

Figure 2-9. Get Heuristic Value

```java
private ArrayList<Integer> reconstructPath(NodeRecord goal) {
    ArrayList<Integer> path = new ArrayList<>();
    NodeRecord current = goal;
    while (current != null) {
        path.add(0, current.uid);   // insert node at the beginning to build path in correct order
        current = current.parent;
    }
    return path;
}
```

Figure 2-10. reconstructPath() function

The $heuristic()$ function gets the node's value via $getVal()$ and additional equations are added to influence heuristic values for each node. The rating factor is greatly affected by ultimately raising the final heuristic value to the power of 2 and multiplying by a weight of 2. If the node is reached, the $reconstructPath()$ method traces the optimal path by backtracking from the goal to the start using parent pointers and builds a list of node UIDs in the correct order.

```java
private static class NodeRecord {
    int uid;                // id of the node
    NodeRecord parent;      // node we came from.. self-explanatory
    double gCost;           // actual cost from start to this node
    double fCost;           // total cost

    public NodeRecord(int uid, NodeRecord parent, double gCost, double fCost) {
        this.uid = uid;
        this.parent = parent;
        this.gCost = gCost;
        this.fCost = fCost;
    }

    @Override   // this is for .remove() and for storing hashmap
    public boolean equals(Object o) {
        return o instanceof NodeRecord && ((NodeRecord) o).uid == this.uid;
    }

    @Override
    public int hashCode() {
        return Objects.hash(uid);
    }
}
```

Figure 2-11. NodeRecord Class

Lastly, the inner class $NodeRecord$ stores the information, such as how far each point is from the start, what it thinks is the total cost of the goal, and which point it came from. It overrides $equals()$ and $hashCode()$ to make sure that the records behave properly and the priority queue.

The data gathered to be used as the edge of a node was a distance measurement from and to Node A and Node B. Measurements were taken by using the Google Maps measure distance function.

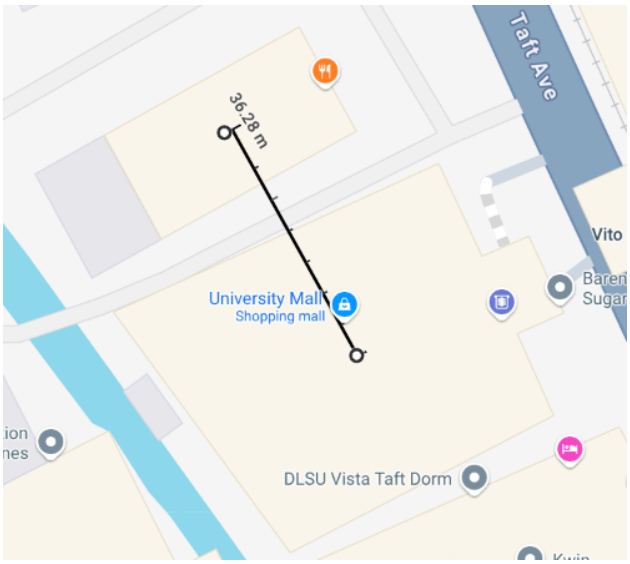


Figure 2-12. Google Maps Distance from U.Mall to McDonald's

The connection between each node was decided most of the time using the line of sight concept; if the source node can see the destination node, then most likely a connection (edge) will be created. If the node is too far away from the rest of the nodes, a connection to the nearest node will be made.

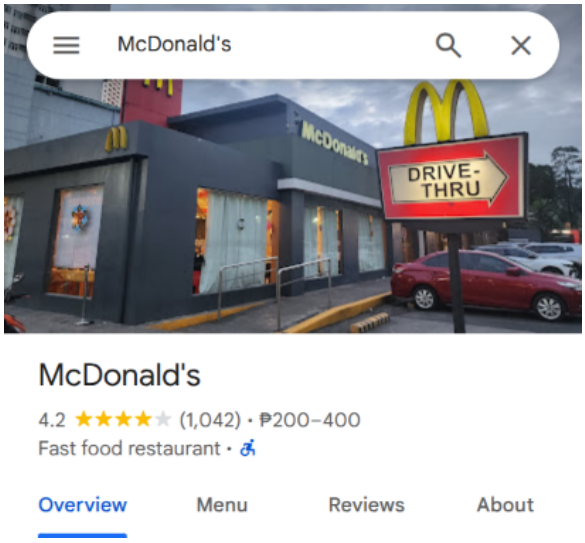


Figure 2-13. Google Maps McDonald's Ratings

Heuristic values or establishment ratings were also gathered through Google Maps by getting its rating posted on it. Data was stored in CSV file, $distances.csv$ and $distances\_h.csv$ where in distances, the edge values and node connections are stored and distances_h the heuristic values, also the establishment's name is stored.

The GUI ties the entire program together, creating a visual platform for users to interact with. It was chosen over a text-based interface because it offers a clearer and more user-friendly way to present and explore the data. The GUI includes a map of the surrounding area of De La Salle University Manila campus, a legend below the map that lists the various eateries around the map, buttons that invoke different functions, and a results tab displaying the results of the search algorithms used.

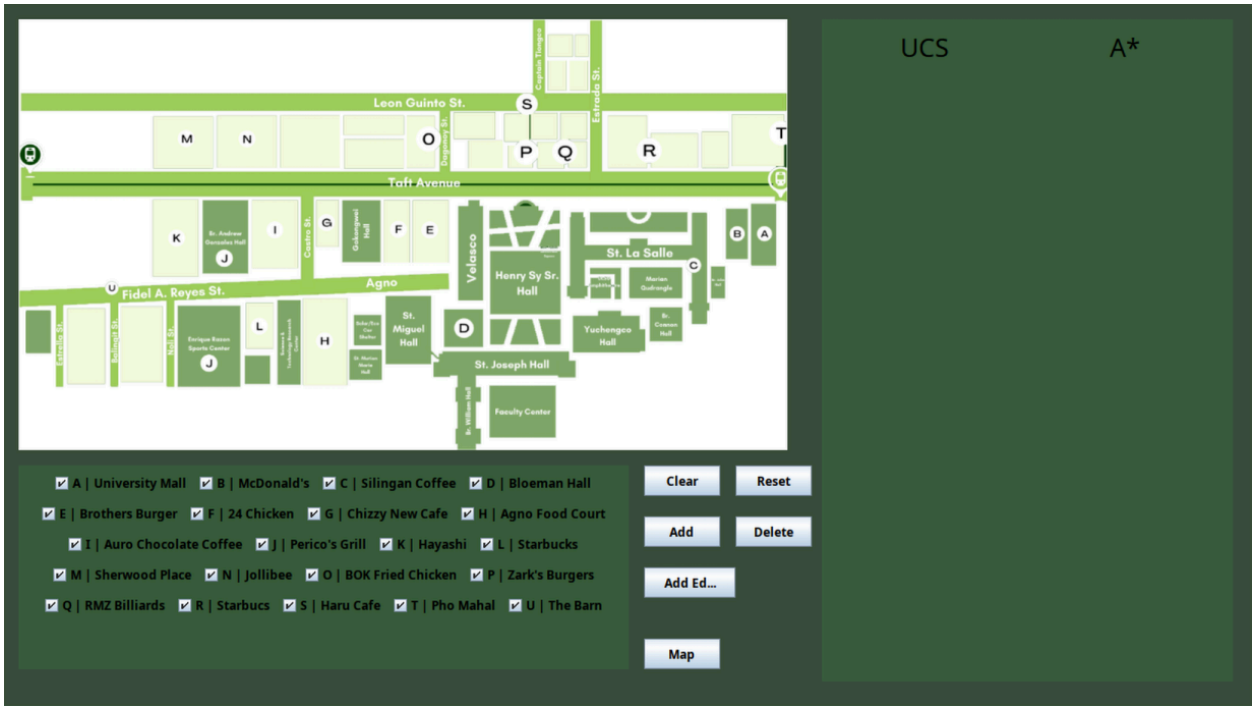


Figure 2-14. App GUI

Proper error handling was applied to the different functions, preventing invalid inputs and guiding users to enter appropriate values when prompted. For example, The add function requests for a name and heuristic value. Only positive integers are allowed to be entered for the heuristic value field, any other number or string inputs will be prompted to input the correct values.

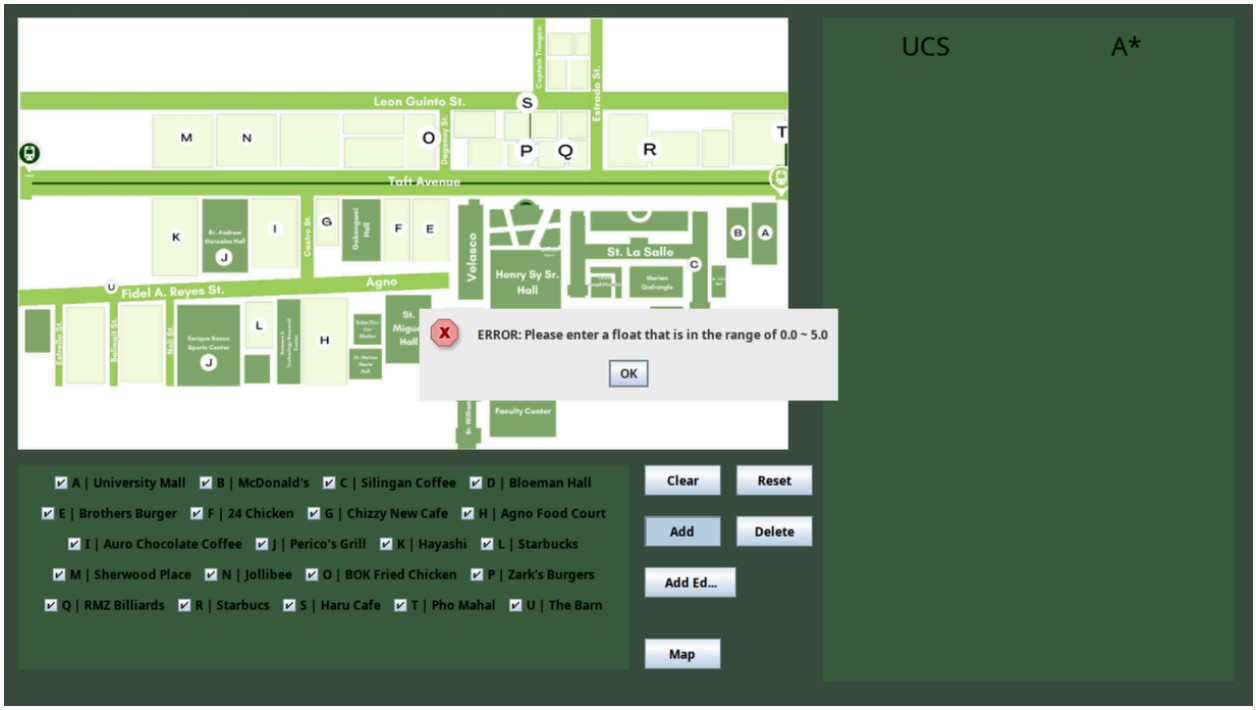


Figure 2-15. App GUI error handling example

To run the algorithms, the user must click on the “Map” button. They are then prompted to input the start and goal nodes. After inputting the details, the program will run through the algorithms and output the results on the right tab.

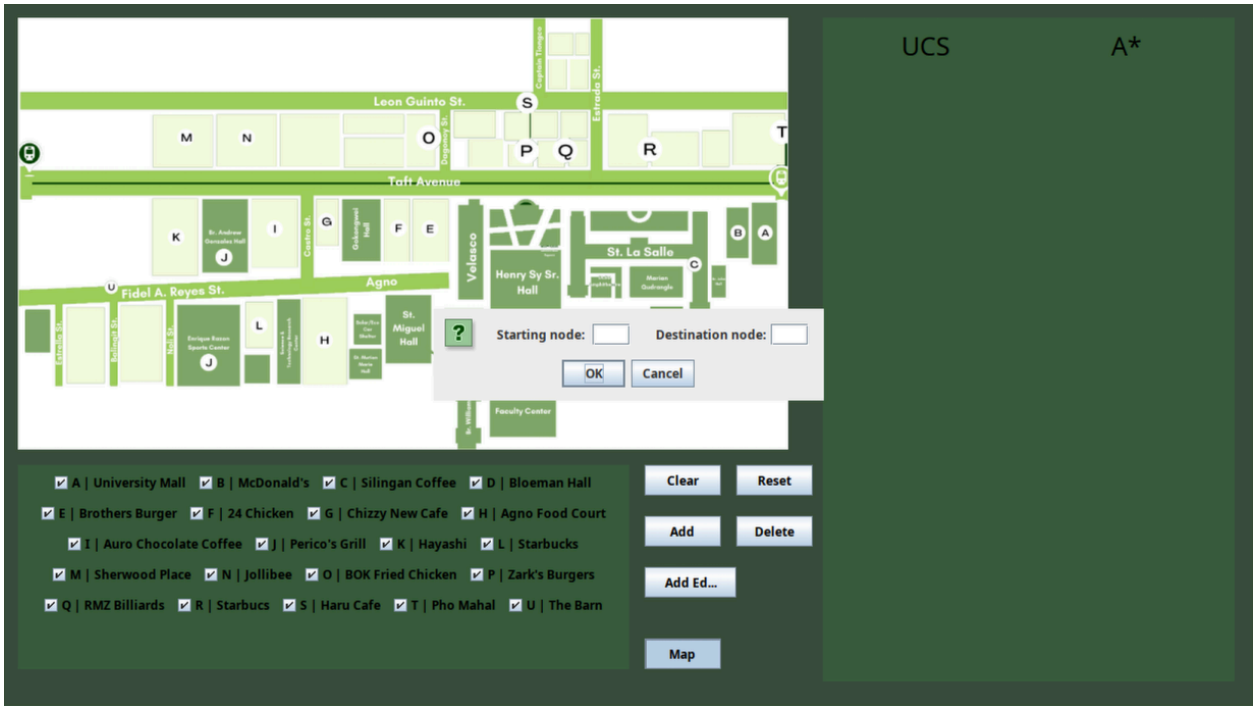


Figure 2-16. Map function inputs

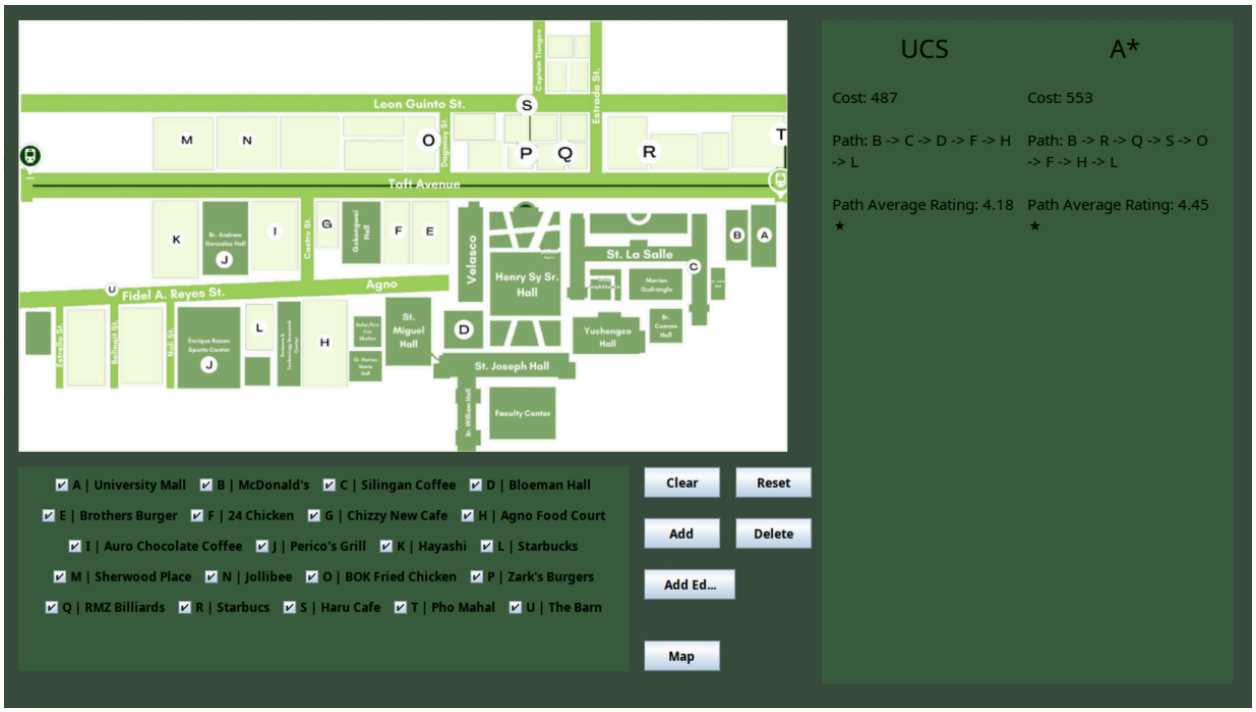


Figure 2-17. Screenshot of map function from Node B to L

III. Results and Analysis

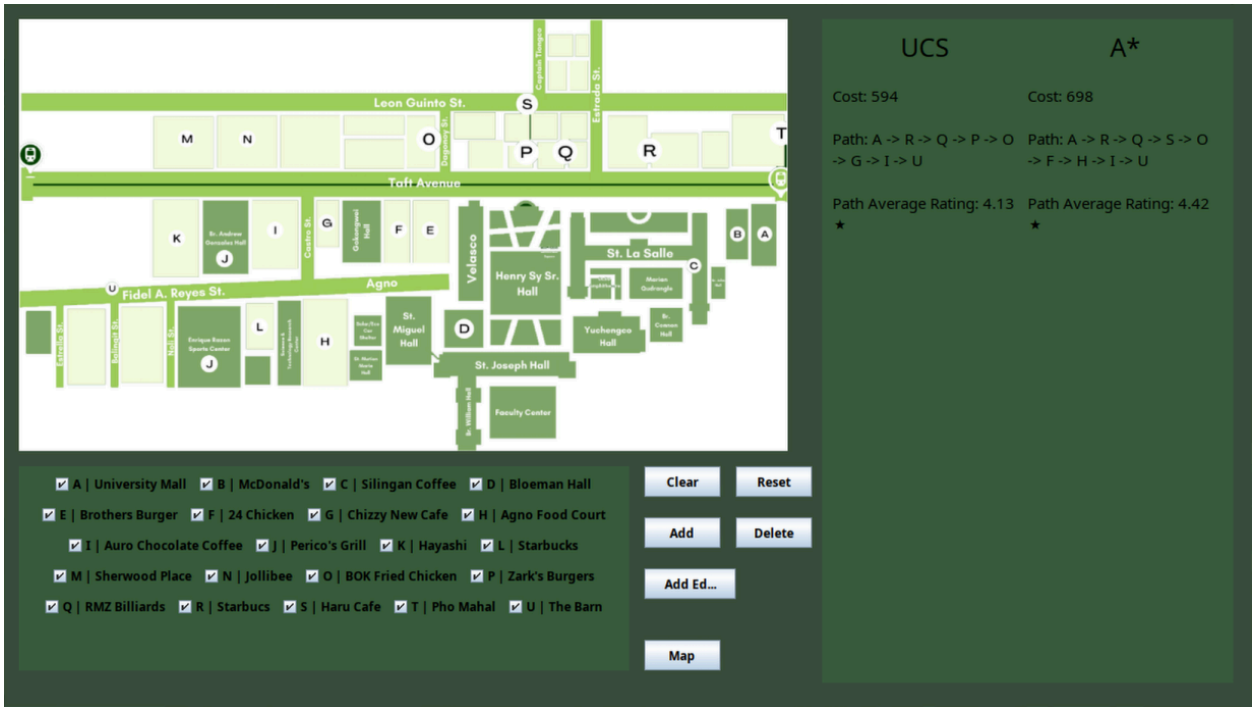


Figure 3-1. Screenshot of map function from Node A to U

From Figure 3-1, a difference between the A* algorithm and UCS can be observed, as A* has a greater average rating for the path that it chose compared to UCS, but in exchange for a higher cost. Heuristic value plays an important role in determining if there will be a difference between UCS and A*. With the initial implementation of just assigning the establishment's rating on Google Maps as their heuristics, all of the combinations of node pathing for both UCS and A* were not the same. This implies that admissibility in heuristics is an important aspect of A*, with the first heuristic implementation being admissible for all nodes; thus, there were differences between A* and UCS pathfinding results.

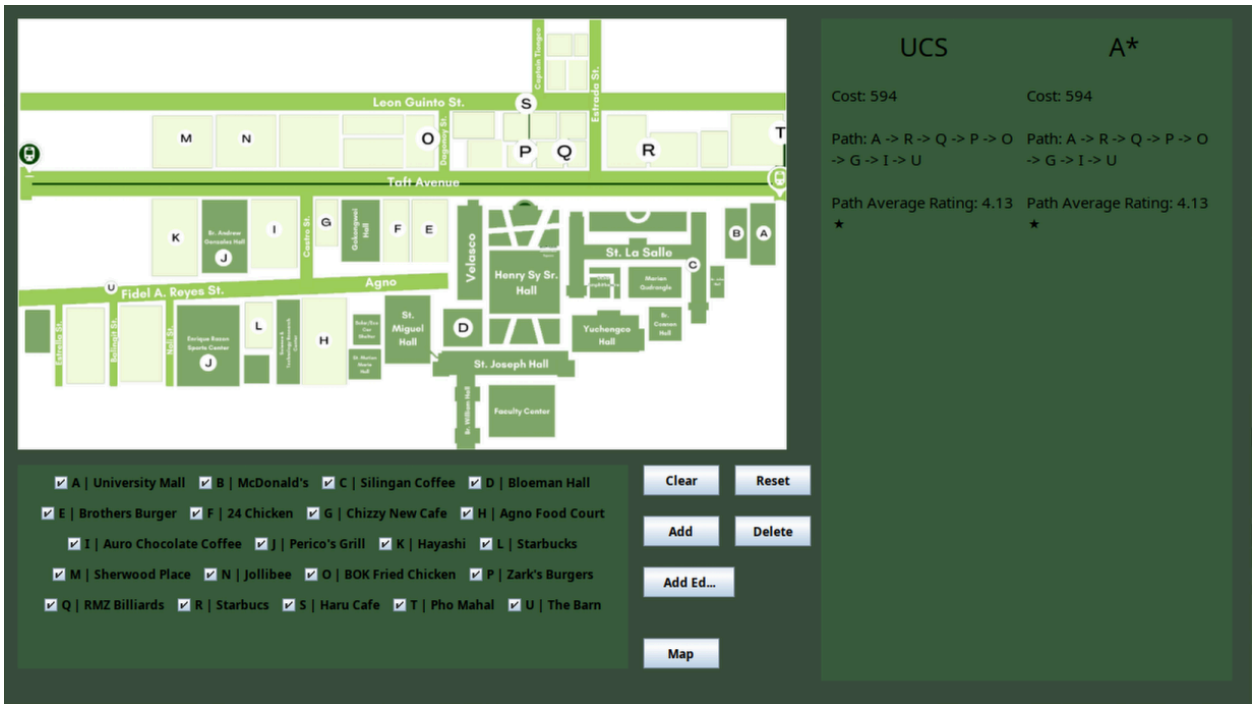


Figure 3-2. Screenshot of map function from A to U with non-admissible for each node (A*)

If all the heuristic values for each node are lower than the cost, A* seems to follow the same path as UCS as seen in Figure 3-2, where the heuristic value for each node is mapped to the establishment's rating. Where the heuristic value only ranges from 0.0 to 5.0, having a function that overestimates (heuristic < cost) always it defeats the purpose of A* while being inefficient compared to UCS by adding a process of checking each edge and node's heuristic value. Thus contributing to a higher execution time and higher memory usage for storing the heuristic value of the nodes.

Figure 3-3 illustrates how Uniform Cost Search and A* Search differs in evaluating nodes over three iterations. A* uses both actual cost and heuristic estimates ($h(n)$) to guide the search more efficiently toward the goal, while UCS expands nodes based solely on actual path cost. Red nodes represent the next node to be evaluated.

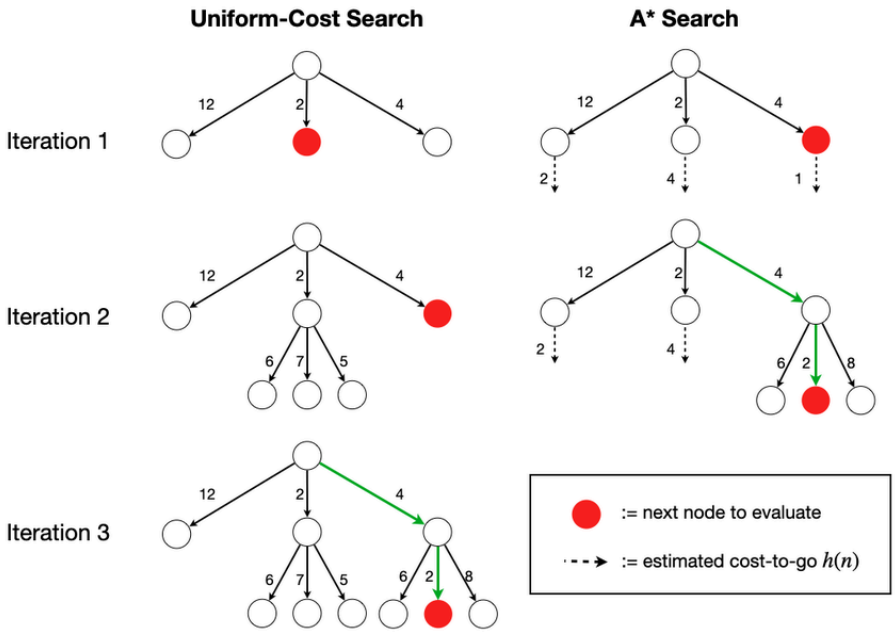


Figure 3-3. Visual Comparison Between Uniform Cost Search and A* Search Over Iterations

While it visually demonstrates the difference in node evaluation between Uniform Cost Search (UCS) and A*. UCS expands nodes solely based on the actual cost from the start, while A* considers both actual cost and an estimated cost to the goal, allowing it to prioritize more goal-directed paths. This behavior explains why A* tends to reach the goal faster and with fewer unnecessary explorations. Table 3-1 summarizes these differences by comparing key aspects of UCS and A*, such as algorithm type, cost consideration, efficiency, and suitability for different use cases.

Aspect	Uniform Cost Search (UCS)	A*
Type of Algorithm	Uniformed	Informed
Cost Considered	Actual cost $g(n)$ only	Actual cost $g(n)$ + estimated cost $h(n) = f(n) = g + h$
Goal Direction	Explores all possible paths equally	Directed towards a goal using a heuristic
Efficiency	Slower, as it explores unnecessary paths	Usually faster due to guided search
Optimality	Yes	Yes (if heuristic is admissible)
Heuristic Needed?	No	Yes

Use Case	When no heuristic is available or when accuracy is prioritized	When a heuristic is available (i.e., maps, games)

Table 3-1. Comparison Between Uniform Cost Search (UCS) and A* Algorithm

To further synthesize the table above: A* offers practical advantages of UCS by combining actual path costs with informed estimates. While UCS guarantees an optimal path via exploring all possibilities, it will be slow and inefficient in larger and complex graphs. A* balances accuracy with efficiency by using heuristics to guide the search toward the goal more directly with fewer node expansions. However, it is possible that A* search could be slower if the heuristic used is not informative, as the heuristics should be related to the edge cost. A*'s flexibility in heuristic design makes it advantageous for real-world applications like navigation systems, which aligns with the goals of this MCO1. In summary, A* is generally more efficient than UCS; it reaches the goal faster without sacrificing correctness.

IV. References

[1] A. Amidi and S. Amidi. "States-based models with search optimization and MDP." Stanford. Accessed: Jun. 27, 2025. [Online]. Available: https://stanford.edu/~shervine/teaching/cs-221/cheatsheet-states-models.

[2] C. Agili. "Understanding State-based Models in Artificial Intelligence." Medium. Accessed: Jun. 27, 2025. [Online]. Available: https://medium.com/@agili.chidinma/understanding-state-based-models-in-artificial-intelligence-0dcfaa76245f.

[3] "Uniform Cost Search (UCS) in AI." GeeksforGeeks. Accessed: Jun. 28, 2025. [Online]. Available: https://www.geeksforgeeks.org/artificial-intelligence/uniform-cost-search-ucs-in-ai/

[4] N. Thakkar, "Uniform-Cost Search Algorithm." Scaler. Accessed: Jun. 28, 2025. [Online]. Available: https://www.scaler.com/topics/uniform-cost-search/.

[5] Board Infinity, "A* Search Algorithm | Board Infinity," Board Infinity. Accessed: Jun. 27, 2025. [Online]. Available: https://www.boardinfinity.com/blog/a-search-algorithm/.

[6] Great Learning Editorial Team, "What is A* Search Algorithm." Great Learning. Accessed: Jun. 28, 2025. [Online]. Available: https://www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence/.

V. Contributions of Each Member

Members	Contribution
Alamay, Carl Justine S.	Helped make the UI and helped write the report
Ang, Czarina Damienne N.	Helped implement the graph class, UCS function, and helped writing the report
Culanag, Saimon Russel W.	Created Graph class, implemented UCS, merged GUI and Graph implementation, and helped with docs in regards to data gathering
Esteban, Janina Angela M.	Created the A* algorithm for the project, helped with menu functions, and helped with the report.
Marinas, Carl Mervyn G.	Created the GUI for the project with proper error handling. Helped with the report.