Optimizing Shortest Pathfinding for the DLSU Campus Food Map Using A* and Uniform-Cost Search Algorithms

Escano, Ewan Rafael A.

Llanes, Andre Gabriel D.

Royeca, Ethan

Tin, Joaquin Benedict M.

Submitted in partial fulfillment of the requirements in Introduction to Intelligent Systems (CSINTSY)

to Norshuhani Zamin, Ph.D.

Department of Software Technology

November 2024

INTRODUCTION

The rapid expansion of urban spaces and the growing demand for efficient navigation systems have heightened interest in algorithms capable of finding optimal paths in various domains. In educational institutions such as De La Salle University (DLSU), navigating between different locations, including widespread food stalls, can become a daily challenge for students and staff alike. In response, our study aims to develop a pathfinder application designed to identify the shortest path between food stalls on DLSU's campus.

The study compares two commonly utilized pathfinding algorithms—Uniform Cost Search (UCS) and A*—to assess their efficacy in determining the shortest path within a campus environment. UCS is recognized for its comprehensive exploration of paths based on cost, while A* integrates cost and heuristic information, providing distinct methodologies for addressing this challenge. Gaining insight into the disparities between these algorithms is essential for devising an effective navigation system in a dynamic real-world context.

The study aims to evaluate and compare two algorithms, specifically focusing on time complexity, memory usage, and optimality in the context of food stall navigation. The algorithms will be applied to simulate real-time pathfinding within a university campus, where both speed and resource efficiency are crucial. The objectives of the study are the following:

1. Assess and compare the time complexity of both algorithms in real-time food stall navigation scenarios.
2. Analyze the memory usage of each algorithm to determine their resource efficiency when applied to a university campus environment.
3. Evaluate the optimality of each algorithm in finding the shortest or most efficient paths.
4. Provide insights into which algorithm is more suitable for real-time pathfinding in environments where speed and resource constraints are critical.

METHODOLOGY

It was crucial to incorporate various algorithms to automate the process of finding the most efficient route between different food stalls that are easily accessible to DLSU students. This was done to ensure that students could easily navigate and find the best path to their desired food destinations.

Data Implementation

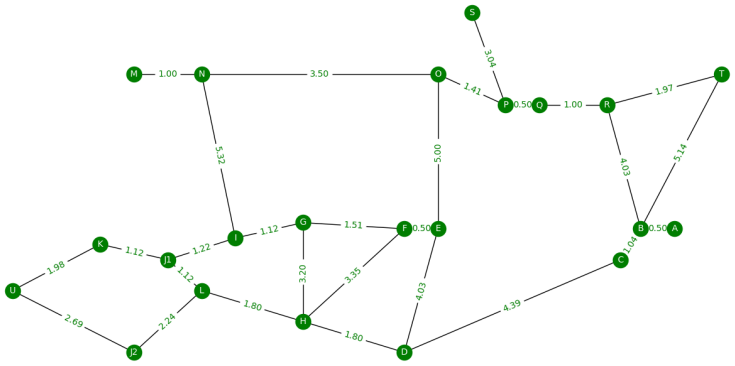

Figure 1: *Mapped DLSU Flood Map*

The mapping was based on the given DLSU flood map in the specifications. The connections (edges) were made in consideration of real life pedestrian lanes and paths.

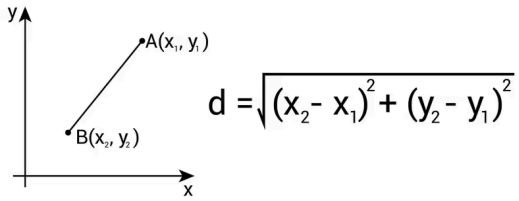

Figure 2: *Euclidean distance formula from HowStuffWorks*

To determine the values of the edges (actual), we made use of the Euclidean distance formula. It takes in inputs of the coordinates of two nodes from the "pos" dictionary. This is used in both UCS and A* search algorithms. Note that the value it produces from the coordinates has a conversion rate of *1 unit = 10 meters*, hence the unit for actual cost is in meters.

Example:

pos = { "A" : (10, 10), $\sqrt{(10 - 10)^2 + (9.5 - 10)^2}$

“B” : (9.5,10), } = 0.5 *(or 5 meters in conversion)*

```
def calculate_heuristics(goal, pos):
    goal_pos = pos[goal]
    heuristics = {node: euclidean_distance(pos[node], goal_pos)for node in
pos}

    for node, heuristic in heuristics.items():
        print(f"Node: {node}, Heuristic: {heuristic:.2f}")

    return heuristics
```

Code used for generating heuristics values for A.* O(n)

To determine the heuristic value of each node, we use the Euclidean distance again but this time, it is used for each node and goal node; hence the goal node will always have a heuristic value of 0. This is an approximation because this type of calculation takes into account a non-existent path from the flood map. It simply calculates the distance between the current node and goal node, hence a shortcut. The unit of the value this function returns will be assumed in meters.

Uniform Cost Search

The search algorithm is a graph traversal designed to determine the least cost path from the start node to the end node. UCS explores nodes based on cumulative path cost, ensuring that it expands the least costly path at each step. This section details the UCS implementation, emphasizing its systematic use of priority queues and cost accumulation.

1. **Initialization**

   The algorithm starts by setting up a priority queue to store tuples containing the cumulative cost, the current node, and the path to reach that node. The starting node is then added to the queue with an initial cost of zero. This ensures the algorithm commences its traversal from the starting point without any prior cost.

2. **Main Loop**

The algorithm operates within a loop until the priority queue is empty. During each iteration, it extracts the node with the lowest cumulative cost from the queue using the based on path cost.

3. Conditional Statement

4. Node Expansion

5. Cost Calculation and Queue Update

A* Search

The A* search algorithm is an informed graph traversal technique that combines path cost with heuristic estimates to find the least-cost path from a start node to a goal node. Unlike Uniform Cost Search, A* utilizes heuristics to prioritize nodes, balancing both the actual path cost from the start node and an estimated cost to reach the goal. This approach allows A* to explore promising paths while minimizing detours efficiently. The heuristic function is crucial; it estimates the remaining cost to the goal, helping A* prioritize nodes that seem closer to the target. This systematic combination of path cost and heuristic values enables A* to achieve optimal, efficient searches when an admissible heuristic is used.

1. **Initialization**
   The algorithm starts by initializing a priority queue to store tuples containing the priority (sum of cumulative cost and heuristic), cumulative cost, current node, and the path to reach that node. The starting node is added to the queue with an initial cost of zero and its heuristic value. This ensures the algorithm begins its traversal from the starting point without any prior cost.

```
Python

priority_queue = [(0 + heuristics[start], 0, start, [start])]
```

2. **Main Loop**

   The algorithm operates within a loop until the priority queue is empty. During each iteration, it extracts the node with the lowest priority (cumulative cost + heuristic) from the queue using *heapq.heappop*.

```
Python
while priority_queue:
    _, cumulative_cost, current_node, path = heapq.heappop(priority_queue)
```

3. **Conditional Statement**
   The algorithm checks if the current node is the goal node. If it is, the path and cumulative cost are returned, indicating that the goal has been reached.

```python
if current_node == goal:
    return path, cumulative_cost
```

4. **Node Expansion**
   If the current node has already been visited, the algorithm skips it. Otherwise, it marks the current node as visited and proceeds to explore its neighbors.

```python
if current_node in self.visited:
    continue
self.visited.add(current_node)
```

Libraries and Tools

- **Tkinter**

  The tkinter library is the standard Graphical User Interface toolkit provided by Python. It allows developers to create simple and efficient desktop applications with graphical interfaces, including windows, buttons, labels, text inputs, and more. Tkinter is built on top of the Tk GUI framework and comes pre-installed with Python, making it accessible without needing to install additional libraries.

- **Networkx**

  Networkx is a powerful Python library used for the creation, manipulation, and study of complex graphs and networks. It allows you to represent nodes, edges, and weights, which are essential components for algorithms like A* and Uniform Cost Search (UCS). In our project, we will utilize networkx to model the university campus as a graph, where food stalls represent nodes and the paths between them represent edges. This structure will enable us to efficiently run A* and UCS for shortest pathfinding between food stalls.

- **Numpy**

  Numpy is a fundamental package for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. In the context of our implementation, numpy will be used to store and perform fast mathematical computations on the coordinates of food stalls, enabling efficient calculation of distances, such as the Euclidean distance, which is crucial for the heuristic function in the A* algorithm.

- **Heapq**

  Heapq is a Python library that provides an implementation of the heap queue algorithm, also known as the priority queue. It allows for efficient push and pop elements from a priority queue while maintaining order. This is particularly useful in both A* and UCS, as it is essential to explore nodes in the order of their cumulative cost. During the implementation of the algorithms, the program uses heapq to manage the open list, ensuring that nodes with the lowest cost or estimated total cost (for A*) are processed first.

- **Matplotlib**

  A popular Python plotting library called Matplotlib offers a versatile framework for making a wide range of static, animated, and interactive visualizations. More advanced functions like geographic projections, 3D plotting, and library integration are also supported by Matplotlib. In the code, Matplotlib is used to plot each food stall's nodes, edges, and weights. In addition, it has a feature that encompasses color to signify the traversed nodes to the non-traversed.

- **Time**

  The time module in python provides various functions to work with time-related operations. In this case, we use the .time() function in order to determine the starting and end time of the two algorithms which is just a small bonus feature in our program.

RESULTS AND ANALYSIS

Upon developing the program, it was essential to compare both algorithms to evaluate their overall performance and efficiency, as these factors play a crucial role in achieving the successful pathfinding experience the team aims to deliver to users. In this section, we will highlight the various test cases conducted to determine the effectiveness of each algorithm in specific areas. We will first examine the paths produced by the Uniform Cost Search (UCS) algorithm, followed by an analysis of the A* algorithm. Both algorithms have similar implementations that utilize the Euclidean formula for cost computation; however, the key distinction is that the A* algorithm incorporates heuristic values present in each node, which alters the manner in which nodes are visited. This section of the paper seeks to

demonstrate the optimality and efficiency of both algorithms for pathfinding through diverse test cases. The two test cases under consideration are detailed below:

Test Cases:

1: A to G

2: S to U

3: J1 to O

Uniform Cost Search

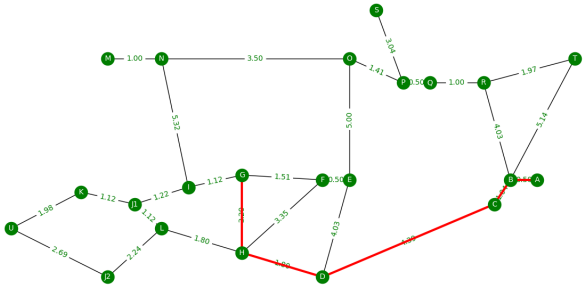

Figure 3: *Test Case 1 Pathfinding A to G*

Cost: 10.93

Shortest path: Yes

A*

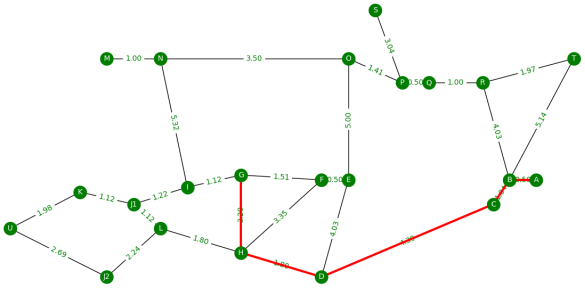

Figure 4: *Test Case 1 Pathfinding A to G*

Cost: 10.93

Shortest path: Yes

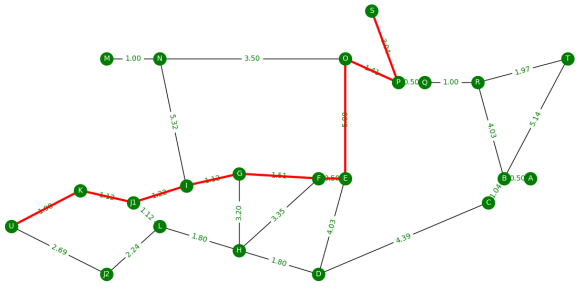

Figure 5: *Test Case 2 Pathfinding S to U*

Cost: 16.91

Shortest path: Yes

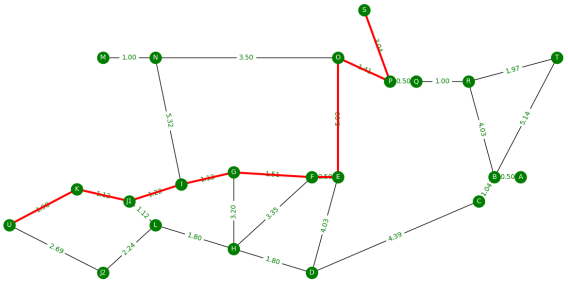

Figure 6: *Test Case 2 Pathfinding S to U*

Cost: 16.91

Shortest path: Yes

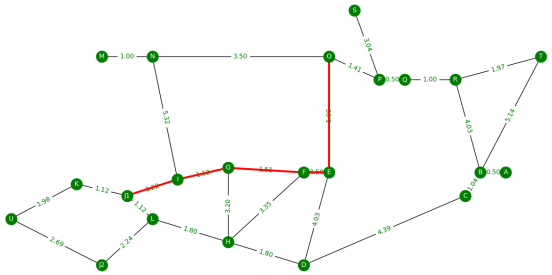

Figure 7: *Test Case 3 Pathfinding J1 to O*

Cost: 9.35

Shortest path: Yes

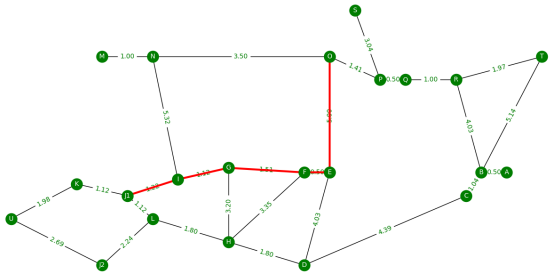

Figure 8: *Test Case 3 Pathfinding J1 to O*

Cost: 9.35

Shortest path: Yes

Let n *be the number of nodes, and* **m** *be the number of edges*

Algorithm	Time complexity	Memory complexity	Optimality
UCS	O((m + n) log n)	O(n)	Optimal
A*	O((m + n) log n)	O(n)	Optimal

Table 2: *Pathfinding using A* Search Algorithm*

Time Complexity

The *uniform cost search algorithm* (UCS) *and A* algorithm* both uses the said time complexity because of how the algorithm was implemented. Both algorithms utilize a priority queue in order to perform the push and pop operations and it was implemented using the heapq library. This operation takes O(log n).

```
heapq.heappop
heapq.heappush
```

The graph traversal for both takes O(n) time

```
for neighbor in self.graph.neighbors(current_node):
```

With this in mind, each time we push or pop a node, we rearrange the priority queue to ensure that the lowest cost path is at the top. Thus, making its time complexity O((n + m) log n). The **only difference** in the two algorithms is that A* uses a heuristic function and takes O(n) to determine the heuristic value of each node..

```
def euclidean_distance(pos1, pos2):
    return np.sqrt((pos2[0] - pos1[0]) ** 2 + (pos2[1] - pos1[1]) ** 2)
```

Code used for generating edge weight for both UCS and A.* O(1)

```
def calculate_heuristics(goal, pos):
    goal_pos = pos[goal]
    heuristics = {node: euclidean_distance(pos[node], goal_pos)for node in
pos}

    for node, heuristic in heuristics.items():
        print(f"Node: {node}, Heuristic: {heuristic:.2f}")

    return heuristics
```

Code used for generating heuristics values for A.* O(n)

Memory complexity

Both UCS and A* algorithms have a memory complexity of O(n). The priority queue may store in the worst case all the nodes in the graph. Furthermore, both algorithms are designed to expand the optimal path, minimizing the expansion of unnecessary nodes.

Optimality

Both UCS and A* algorithms gave the best possible paths. So in terms of optimality we can already say A* is optimal as it is admissible wherein $h(n) \leq h^*(n)$. Likewise, we can say UCS is optimal if all edge weights are non-negative values. Furthermore, a bonus feature showing the execution time using the time library import in python was used to show the algorithm's execution time. 0.0000 values mean that the execution time is smaller than the allotted digits.

REFERENCES

[2:51] Bro Code, "Python Time module" *YouTube*, Jan 26, 2021. [Online]. Available: https://www.youtube.com/watch?v=Qj3GlL5ckQA

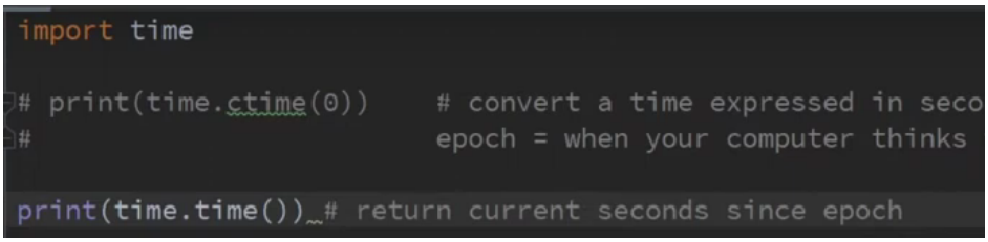

Figure 9: *Sample code for using .time() function*

[Parts 1-5] Koolac , "How to create an Undirected Graph using Python | Networkx Tutorial - Part 01 " *YouTube*, Sep. 18, 2021. [Online]. Available: https://youtu.be/rldKl1CNx-A

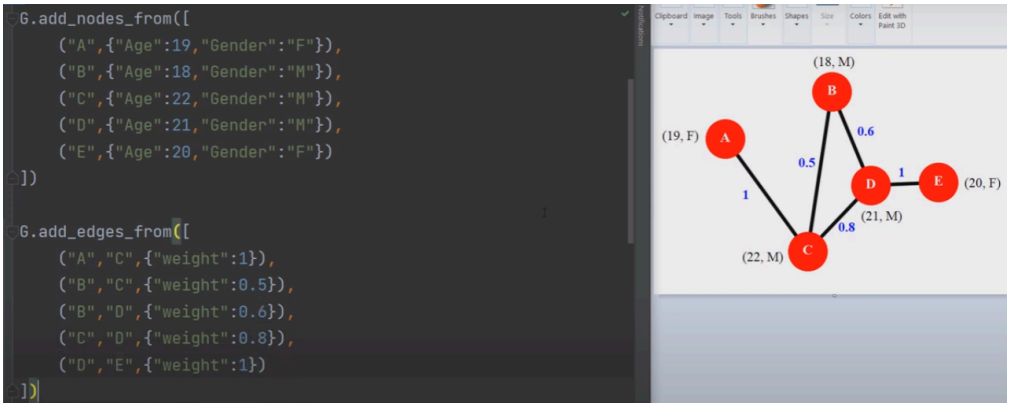

Figure 10: *Sample code for node and edge attributes*

CONTRIBUTION OF EACH MEMBER

Name	Contribution
Escano, Ewan Rafael	- Written Introduction and Methodology - Implemented the GUI

	- Contributed to the implementation of the A* Algorithm
Llanes, Andre Gabriel D.	- UCS algorithm implementation - Results and Analysis (test cases, complexities, and optimality) - Written Methodology on UCS and A* - Deliverables
Royeca, Ethan	- Contributed to the implementation of distance calculation between nodes and the A* Algorithm
Tin, Joaquin Benedict M.	- Written Methodology and Results and Analysis