Project Report

Contents

Introduction:

In this project, we aim to implement a naive Dijkstra algorithm in order to find the shortest path from one point on AUC campus to another. This is a straightforward application of the algorithm we have defined in our code; however, the implementation is flexible and can apply to different problems than the one we defined here. Other examples include finding the best path for electric current flow, creating a schedule for different projects in a timeline (by finding the critical projects and building an efficient plan for projects execution order) or a path line from one place to another using public transportation. In short, since our implementation uses a graph class to implement the nodes, problems that can be represented as a simple graph fit into our implementation of Dijkstra. Another notable point regarding our implementation is that it handles different types of graphs with combinations of directionality and weight (undirected weighted, directed weighted, undirected unweighted, directed unweighted). The distances between nodes are represented by an adjacency matrix, suitable for dense networks (Singh & Sharma, 2012). This is suitable for our case since most points in AUC are reachable from another point.

Purpose :

This small project for finding your way through AUC campus can be a fun way to explore campus for new students. The friendly user interface displays the locations names clearly for the user to choose from as well as the stops they will need to go through on their way, in addition to an estimated distance to cover. This application can help freshmen grow familiar with the naming of the buildings and AUC campus layout

Data specification:

In order to model a graph and feed it into our model, we created a text file where each line stores 3 values: the source node number, the destination node number, and the distance between the two nodes. Each location has a number corresponding to it in the text file, and this relationship is defined in our graph class.

At the top of the text file, there are two lines that define the nature of the graph. The first line specifies whether the graph is: Weighted ("W") or Unweighted ("UW") or Directed ("DG") or Undirected ("UG")

For example:
"W UG" means the graph is weighted and undirected.
"UW DG" means the graph is unweighted and directed.

The second line contains a single integer representing the total number of nodes in the graph. This value is used to initialize the adjacency matrix accordingly.

The nodes correspond to important landmarks and buildings on the AUC New Cairo campus, such as the AUC Library, Campus Center, Sports Complex, and residential halls. This mapping is hardcoded in the Graph class and is used during output to provide clear, readable paths for users.

The edge weights (distances) are manually estimated based on walking paths and intuitive proximity across the campus. These are not GPS-verified measurements, but rough values based on our time at AUC. The distances and the paths suggested are true to the data given, but may not be realistically accurate. The goal was to capture a simplified spatial relationship for educational and demonstrative purposes.

Pipeline:

1. Graph Representation

The graph structure is built from the text file discussed in the previous section. This information is stored into an adjacency matrix. If the graph is undirected, each weight is stored twice, in the cell corresponding from point i to j, and once in the cell corresponding from point j to i. If the graph is unweighted, the weight values in the text file are ignored and assume a default value for connections. This adjacency matrix is initialized with a large value representing infinity to indicate no direct connection. When valid edges are parsed, the corresponding matrix cells are updated with the actual weights.

2. Dijkstra’s Algorithm Logic

This is the core of the path-finding system. The algorithm maintains three key vectors:

a. A distance vector that holds the currently known shortest distance from the source node to every other node.
b. A Boolean vector that marks whether the shortest path to a given node has been finalized.
c. A path vector that stores the predecessor of each node on the shortest path.

The algorithm iteratively selects the unknown node with the smallest tentative distance and then updates the distances of its neighbors if a shorter path is found. This process repeats until the shortest distances to all nodes have been calculated or the destination node has been reached.

3. Path Reconstruction

Once the algorithm completes, the shortest path from the source to the destination is reconstructed using the predecessor path vector. This reconstruction is done by tracing backward from the destination to the source. The resulting sequence is then reversed and printed in a readable route format.

If there is no path between the source and destination, the system detects this and clearly communicates that the nodes are unreachable from one another.

4. User-Friendly Output

To improve readability and usability, each node in the graph corresponds to a named point. The system uses a fixed list of AUC building names, and the final output route is printed using these names instead of raw node indices. This provides a familiar and intuitive interface for users navigating the university.

Algorithm details:

In this section, we will give a detailed explanation of the Dijkstra algorithm implemented in our code:

1.Initialization

The adjacency matrix, predecessor matrix, and the current distance matrix (called dist) are initialized. In the adjacency matrix, all distances are initialized to infinity except for the source node where the distance is 0. The priority queue is seeded with the source node and its distance (zero).

2. Main Loop:

While the priority queue is not empty:

a. Extract the node u with the smallest tentative distance from the priority queue.
b. If this node has already been visited (finalized), it is skipped.
c. Mark node u as visited.

3. Relaxation Step:

For every neighbor v of the current node u:

a. Check if there is a direct edge from u to v (i.e., the adjacency matrix entry is not infinity).
b. Calculate a potential new distance to v as dist[u] + weight(u, v).
c. If this new distance is less than the currently recorded dist[v]:
d. Update dist[v] to the new smaller distance.
e. Update the predecessor of v in the path vector to u.
f. Push (dist[v], v) into the priority queue to process v later with the updated distance.

4. Termination:

The loop continues until all reachable nodes have been processed and the shortest distances from the source to all other nodes are finalized.

Output:

The dist vector and the path vector are the two main outputs from our code implementation that are used to reconstruct the shortest path:

a. The dist vector contains the shortest distance from the source to every node.

b. The path vector can be used to reconstruct the shortest path from the source to the destination by backtracking from the destination node using the predecessors until the source is reached.

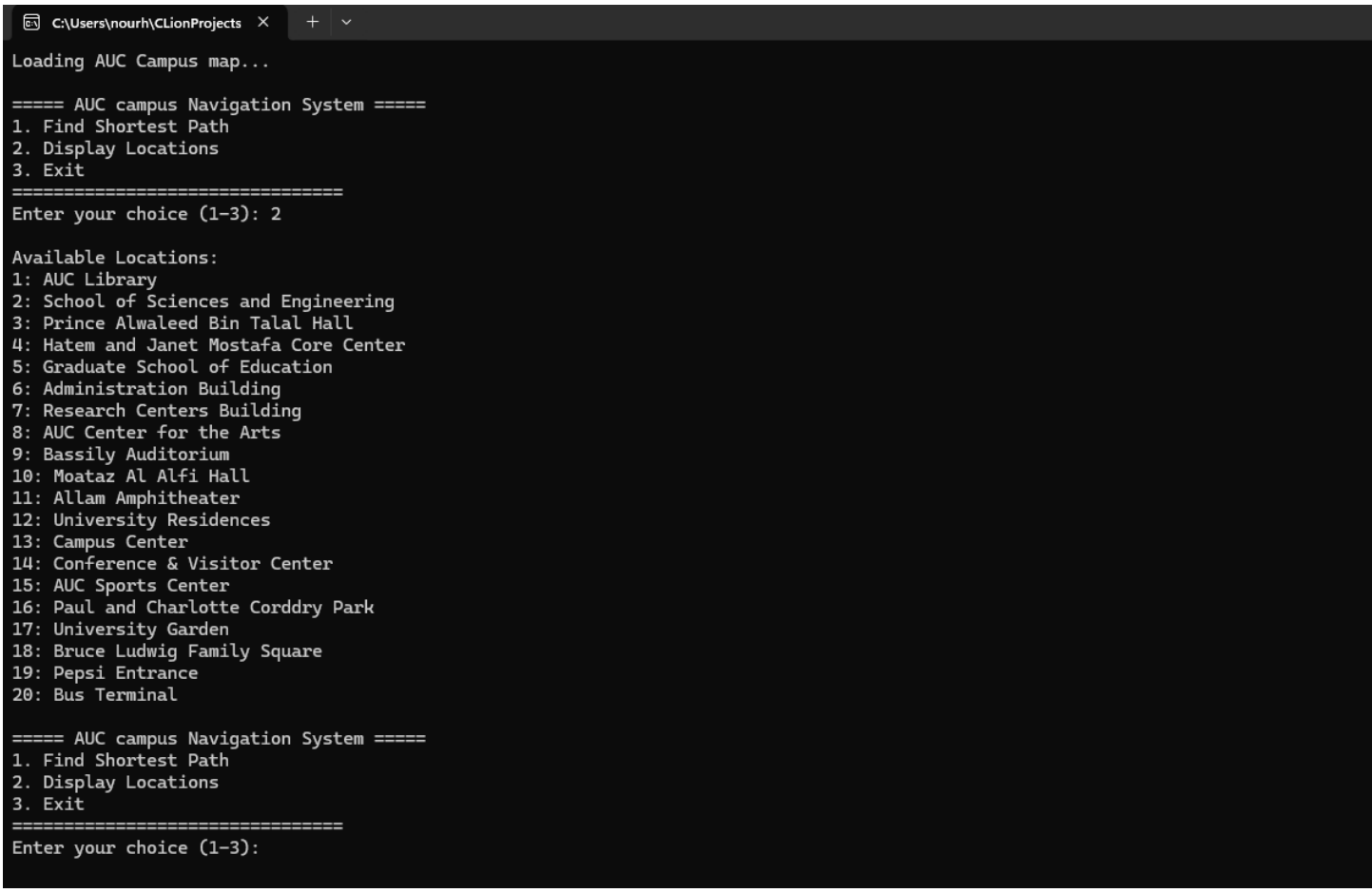
If the distance to the destination is still infinity, it means no path exists.

The output of the code is the number of meters it takes from one place to the other, in addition to the building on the optimal path to the destination. In addition, we display the time it took for the code to run. This piece of information is not of much use to the user, but we printed it to get a sense of how long the code takes on average to run across our different run tests.

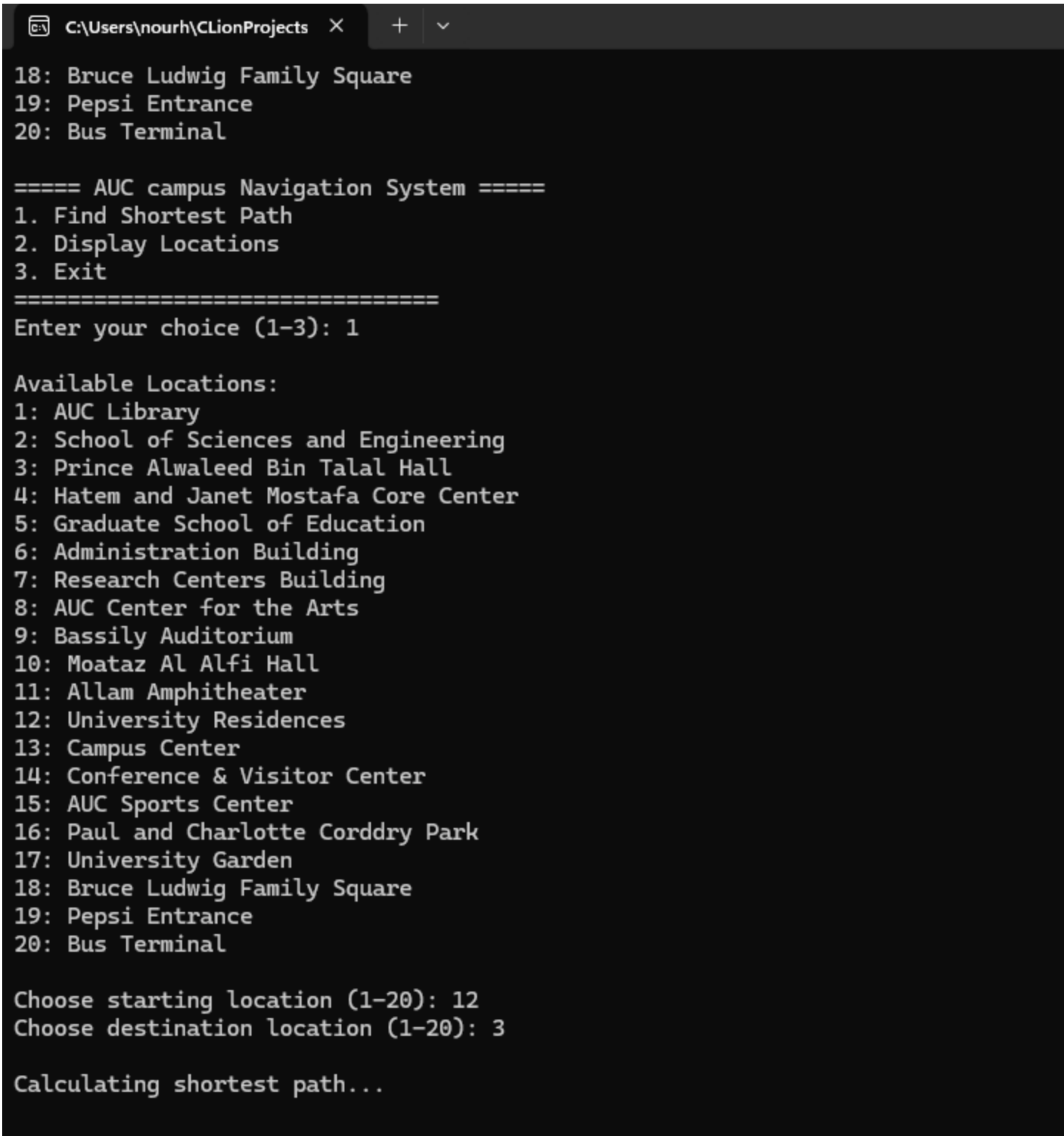
Application interface:

Below, I will provide screenshots for the application's design, and how the output looks like

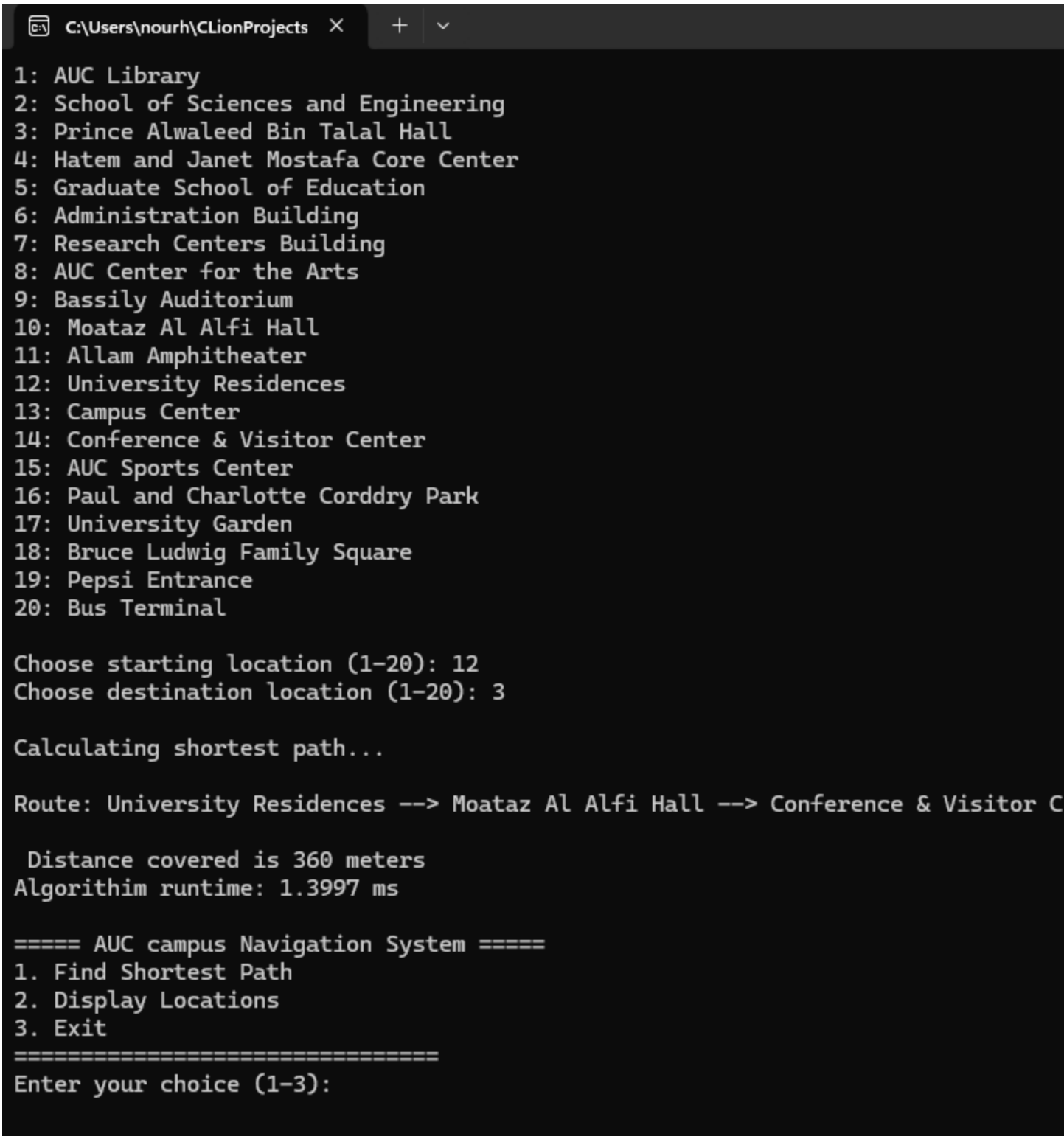
1. If you are unfamiliar with the places, choose option 2 to display the building names:



2. Choose 1 to find the shortest path. The user will be prompted to enter two numbers, the number corresponding to the source and the number corresponding to the destination:



3. The output is the distance in meters between the two points and the path



4. The user is prompted to either continue finding the shortest paths or if they want to exit they press 3.

Summary of results:

We verified the accuracy of the algorithm using online sources and our own tracing of the file throughout 10 test runs, all of which correctly found the shortest path. The average runtime we calculated using these 10 tests was 3.32391 milliseconds. Each test was carried out by randomly choosing a source and a destination.

Analysis and critique:

In this section, we will discuss the advantages and disadvantages of our project.

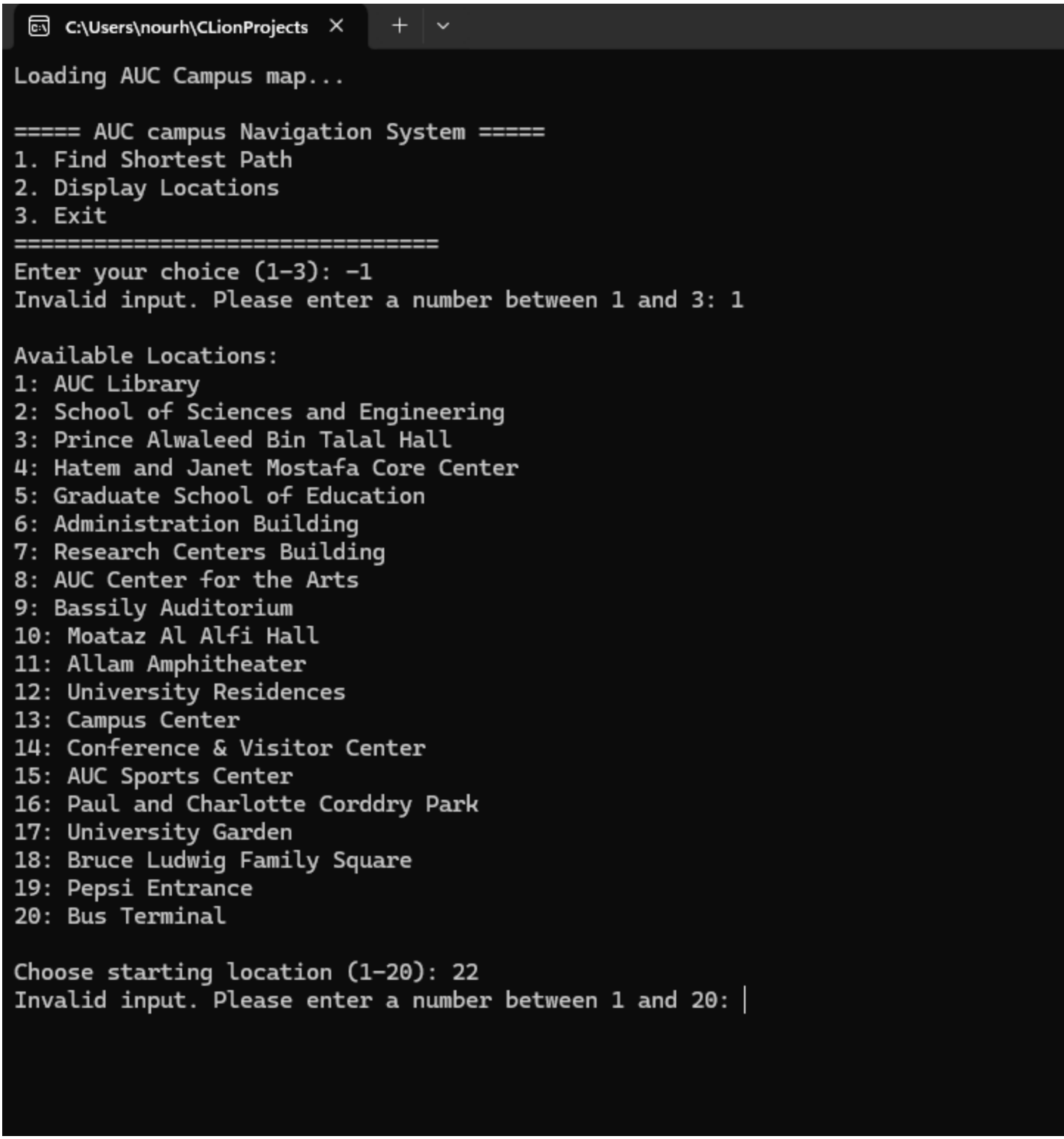
Advantages:

1.Handling of edge cases

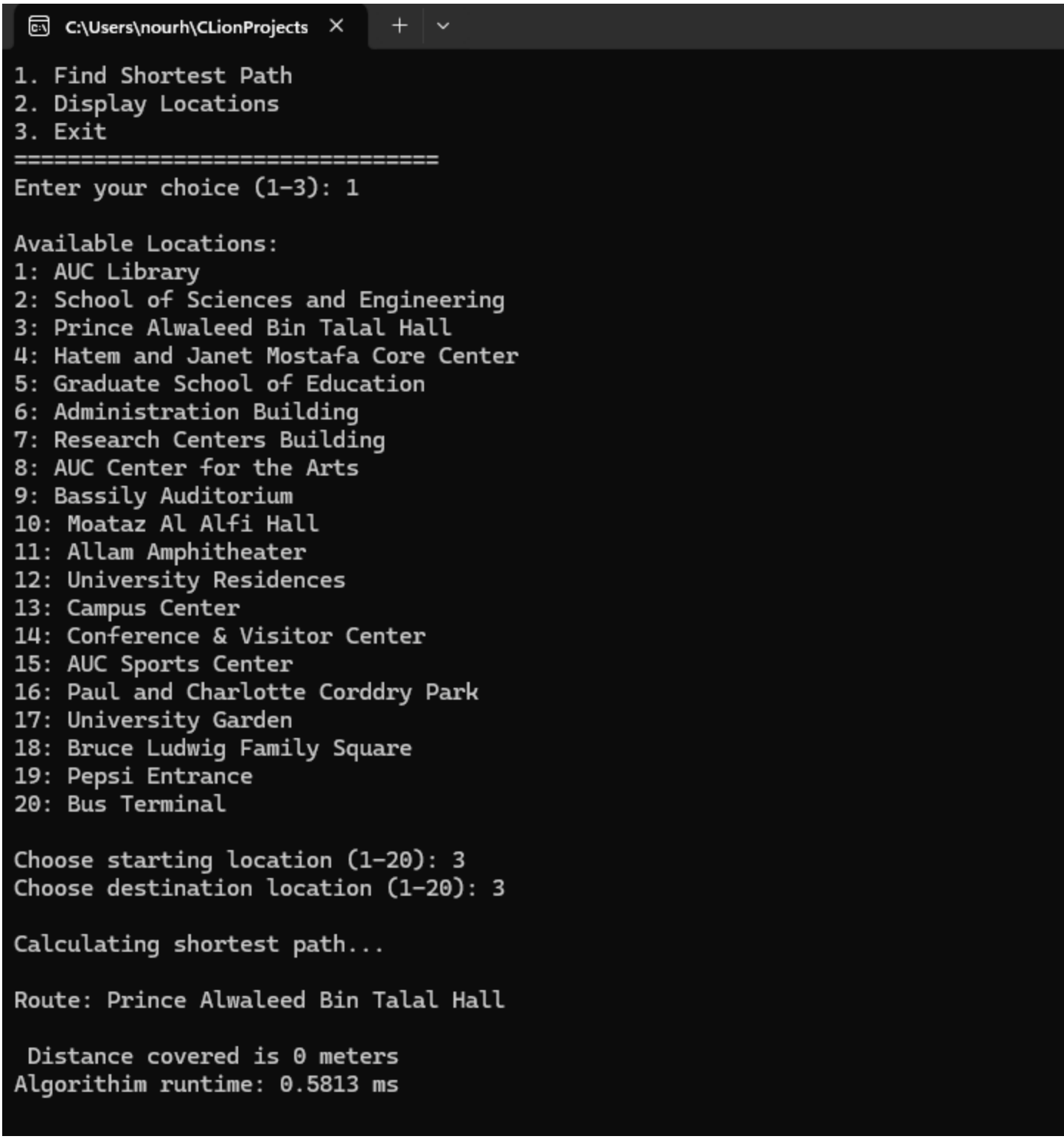
a. The case when two nodes are not connected to each other. Here, we removed any connection between AUC library and Huss building. The code gracefully deals with this and gives an appropriate message to the user

```
13: Campus Center
14: Conference & Visitor Center
15: AUC Sports Center
16: Paul and Charlotte Corddry Park
17: University Garden
18: Bruce Ludwig Family Square
19: Pepsi Entrance
20: Bus Terminal

Choose starting location (1-20):1

Choose destination location (1-20):3



Calculating shortest path...
No path from AUC Library to Prince Alwaleed Bin Talal Hall
Algorithim runtime: 1.9759 ms
```

b. Incorrect input by the user: The user is prompted to re-enter a valid input



c. The source and the destination are the same: The code correctly gives the trivial route and 0 distance.


C:\Users\nourh\CLionProjects
1. Find Shortest Path
2. Display Locations
3. Exit
==========
Enter your choice (1-3): 1
Available Locations:
1: AUC Library
2: School of Sciences and Engineering
3: Prince Alwaleed Bin Talal Hall
4: Hatem and Janet Mostafa Core Center
5: Graduate School of Education
6: Administration Building
7: Research Centers Building
8: AUC Center for the Arts
9: Bassily Auditorium
10: Moataz Al Alfi Hall
11: Allam Amphitheater
12: University Residences
13: Campus Center
14: Conference & Visitor Center
15: AUC Sports Center
16: Paul and Charlotte Corddry Park
17: University Garden
18: Bruce Ludwig Family Square
19: Pepsi Entrance
20: Bus Terminal
Choose starting location (1-20): 3
Choose destination location (1-20): 3
Calculating shortest path...
Route: Prince Alwaleed Bin Talal Hall
Distance covered is 0 meters
Algorithim runtime: 0.5813 ms

2. Efficient Shortest Path Calculation:

In this implementation of Dijkstra's algorithm, we are using a priority queue (min-heap), which is efficient for graphs with non-negative edge weights. Priority queue improves performance over naive implementations that scan all nodes every iteration. The worst-case complexity for using the priority queue is O((V + E) log V), where V is the number of vertices and E is the number of edges, while that of the simple linear search is $O(V^2)$ (Barbehenn, 2002)

3. Supports Directed and Undirected Graphs, in addition to weighted and unweighted graphs:

By checking graph type flags, it correctly handles the 4 kinds of graphs and updates the adjacency matrix accordingly.

4. User friendly interface

The process of choosing a source and destination is straightforward, in addition to an easily interpretable output of the shortest path.

Disadvantages:

1.No Handling of Negative Edges:

Dijkstra's algorithm does not handle negative edge weights; the code does not check or warn about this, which might cause incorrect results if the input graph has negative weights.

2. No Early Exit in Dijkstra:

The algorithm runs through all reachable nodes even if the destination is reached earlier. It could be optimized to stop once the shortest path to the destination is found.

3. Limited Scalability and Extensibility:

The adjacency matrix and linear vertex scanning restrict scalability for larger graphs. The interface is minimal, with little support for other graph algorithms, dynamic updates, or edge deletion. Location names are hardcoded in a static vector inside the print location function.

Conclusion:

In conclusion, our implementation for Dijkstra algorithm, despite its limitations to the general problem of finding the shortest path, is well suited for the problem at hand. In our case where we want to guide people through AUC campus, a full representation of AUC campus with more details will include roughly 50 nodes which are densely connected to each other. This type of graph is optimal for our simple code optimized through the use of an adjacency matrix and priority queues in the Dijkstra algorithm.

Overall, this project lays a solid foundation for understanding and applying shortest path algorithms in practical settings and could be easily combined with further extension to include additional graph algorithms, dynamic graph updates, and enhanced user interface features. Through an already flexible graph representation that supports both directed and undirected as well as weighted and unweighted graphs, the system can be adapted to various real-world applications beyond campus navigation.

Appendix:

The code we used in the Implementation is provided in this GitHub repository.

References:

Barbehenn, M. (2002). A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices. IEEE transactions on computers, 47(2), 263.

Huang, Yizhen & Yi, Qingming & Shi, Min. (2013). An Improved Dijkstra Shortest Path Algorithm. 10.2991/iccsee.2013.59.

Singh, H., & Sharma, R. (2012). Role of adjacency matrix & adjacency list in graph theory. International Journal of Computers & Technology, 3(1), 179-183.