CSCE 2211 Fall 2023

Term Project Report

Dr. Amr Goneid

Navigation program : AUCMaps

Ahmed Fawzy Abdelkader 900222336

Joudy El Gayar 900222142

Malek Mahmoud 900222057

Department of Computer Science and Engineering, AUC

Abstract :

Navigation and mapping are one of the most nuanced and elegant problems in modern computer science, and the solutions to these types of problems tend to be complex. Related to this, college campuses can be enormous and confusing, AUC being in the spotlight. Despite this though, the campus still shares a beauty to those who understand how to navigate and live in it. With these points in mind, we chose to create AUCMaps, a mapping and navigation application tailored to the AUC's campus using graphs, tries, and Dijkstra's algorithm. We first learned about the current solutions: Google maps and AUC Mobile App. Then we decided to go on our own path given the valuable knowledge and resources. In this report, we define the problem that AUCMaps solves, provide an in-depth recollection of the development process, talk in detail about the data structures and algorithm used, specify the input data and user experience and how we overcame challenges related to that, and finally reflect on our application and methodologies.

Keywords: graph, trie, mapping, Google maps, AUC, shortest path, greedy algorithms, A* algorithm, Dijkstra's Algorithm, heaps, data structures, algorithms

Outline:

1. Introduction

The goal of the project this semester is to incorporate data structures and algorithms to produce a working application. We decide to explore a problem deeply invested in data structures and algorithms, and one which is highly related to our campus and colleges. Our main goal is to find the shortest paths within a complex map. The American University in Cairo (AUC) campus area goes up to 1 million square meters. Such a huge sample is not easy to navigate. Moreover, we do not only want to navigate, we want to travel efficiently. To do this we use several data structures which will be discussed and a core algorithm for finding the shortest path.

2. Problem Definition

One of the biggest challenges facing students, especially newer ones, at the American University in Cairo's(AUC) expansive and uniquely designed campus is navigation. The complexity of the campus layout, characterized by its large size, vaguely named buildings, and the necessity for students to attend classes in various buildings, presents a daunting task for both new and existing students. One of our group members, Ahmed, has hands-on experience with this. He notes that, It is a specialized job on campus to learn the best routes and proper naming, as used in AUC Banner and on the AUC's website. This is where the need for AUCMaps becomes evident.

Existing solutions, such as Google Maps, lack indoor navigation capabilities, and while the AUC Mobile App offers some outdoor and indoor navigation, it suffers from limited usage by students. Moreover, the existing AUC on campus maps are not only outdated but also add to the confusion with their complex layouts and lack of space awareness. Our approach to tackling this problem involved applying elegant and sophisticated algorithms and data structures. We utilized Dijkstra's algorithm for efficient pathfinding and incorporated data structures like a Trie and a Graph to manage complex mapping data. Furthermore, the Graph and Trie are crucial for searching in pathfinding and simple classroom search. The user interface of AUCMaps, developed using QT, is tailored specifically for the AUC campus. It includes features like an AUC Student login page, zoom functionality, lists, and auto-complete functionalities to enhance user experience. Additionally, we integrated Google Earth's satellite imagery to provide realistic and accurate representations of the campus layout.

The program we created is not just a map; it's an intelligent navigation system. AUCMaps offers the shortest paths using known routes combined with “student know-how” shortcuts to various buildings and classrooms. Moreover, while you can not navigate inside of the buildings, similarly to Google Maps, due to the vast number of classrooms and intricate hallways which would require us to collaborate with the university, which we discuss later; what you can do is move through the floors accessing classrooms on any floor and even the roof. The distances, measured using Google Earth, provide a realistic idea of travel time within the campus which is not offered in the AUC Mobile App.

This can help students better understand the distances they are traveling within the campus consequently promoting better time management. Additionally, by analyzing the travel patterns of students, AUCMaps can generate new data, potentially uncovering underlying campus issues and aiding in future solution development. This innovative approach positions AUCMaps as a vital tool for every AUC student, simplifying campus navigation and enriching the overall campus experience for new and veteran students.

In this report we will be discussing the development process of AUCMaps,the methodology and development process, the algorithms and data structures utilized, input data specifications, the challenges we faced when developing the app and how we overcame them, and finally analysis and critique of our methods and application.

3. Methodology

In the beginning of this project, the plan was to implement a hash table, which would store the data regarding all rooms on campus. Acting as a dictionary for AUC. On the other side, we would have a graph structure representing the buildings. However, after meeting with Eng. Hany, he pointed out that such implementations would seem unprofessional. The reason is that the two data structures are separate from each other. He recommended we join them together to make the project more applauding and polished.

Since each node represents a building. It was decided in order to implement the data structures together to put a hash table in each node in the graph, that would represent the rooms in the building. Moreover, after careful research, the implementation of a trie instead of a hash table seemed more convincing. As it is efficient when searching for keys that share common prefixes. Which is the case, because we have multiple rooms with the same prefixes.

In the development of AUCMaps, our methodology focused on ensuring a solution tailored to the AUC campus's navigation needs. Initially, we contemplated a 3D modeling approach for visualizing the campus, utilizing planes to represent floors, lines for paths, and volumetric shapes for buildings. The model initially was to be laid out on a grid, with blocked areas for inaccessible zones, and buildings and paths mapped accordingly with satellite imagery. Such a system would have allowed for navigation on a coordinate plane, aligning with traditional mapping techniques. However, considering the content and tools available to us this semester, we opted for a more pragmatic approach. We decided to visualize the campus using a weighted undirected graph. This choice was influenced by the nature of campus navigation because any point you can walk to, you should be able to walk back from. We implemented an adjacency list instead of an adjacency matrix to represent our graph, as the sparser structure of the list provided a more memory-efficient and faster solution for our application, where nodes represented key locations on campus, and edges represented accessible paths between them. Each node in our structure was created with classroom searching and shortest path finding in mind, containing the name and weight and an associated trie object to hold classroom information where applicable. The visual representation of the campus, as shown in Figure 1, was visualized by this graph data structure, which also formed the foundation for our coding of the adjacency list. The choice of a data structure to store and search classrooms was iterative, and we believe we made the best choice. The plan was to implement a hash table, which would store the data regarding all rooms on campus. Acting as a dictionary for AUC. On the other side, we would have a graph structure representing the buildings. However, after meeting with Eng. Hany, he pointed out that such implementations would seem unprofessional. The reason is that the two data structures are separate from each other. He recommended we join them together to make the project more applauding and polished. Since each node represents a building. It was decided in order to implement the data structures together to put a hash table in each node in the graph, that would represent the rooms in the building. Moreover, after careful research, the implementation of a trie instead of a hash table seemed more convincing. As it is efficient when searching for keys that share common prefixes. Which is the case, because we have multiple rooms with the same prefixes. When it came to pathfinding, we initially considered the A* search algorithm, known for its efficiency in solving shortest path problems. However, the heuristic function of A* needs a coordinate plane for heuristic distance estimates, but the graph structure does not inherently support this. We also thought of putting the graph data structure on a coordinate plane, but when it came to applying this we found it to be redundant and not necessary since we can use a different algorithm while still solving the shortest path problem. The code we had is shown in

Appendix H. Upon this realization we transitioned to Dijkstra's algorithm since simply it can be thought of as an A* algorithm without the heuristic function. This algorithm while slower than A*, provided a reliable solution without the need for heuristics. We facilitated the interaction between our data structures, the pathfinding algorithm, and Trie searching by using Enums to represent the names of the nodes and mapping their corresponding nodes using an index for ease of programing, which was essential in the adjacency list traversal and Dijkstra's implementation. For the GUI development, we used the QT C++ framework, which our team was most experienced with, to design and implement the interface. Through a series of iterative meetings, we drafted the layout and functionality of the app, as illustrated in Figure 2. The GUI was designed with an emphasis on user experience, incorporating features such as zoom functionality, lists, and an auto-complete feature to facilitate ease of use. While we desired to include a more advanced interactive guide, waypoint to waypoint navigation,we encountered limitations due to the constraints of the QT framework and we could not find such implementations with our current tools. Nevertheless, the resulting application provided a functional and user-friendly interface that effectively met the core navigation requirements from the navigation of the AUC campus .

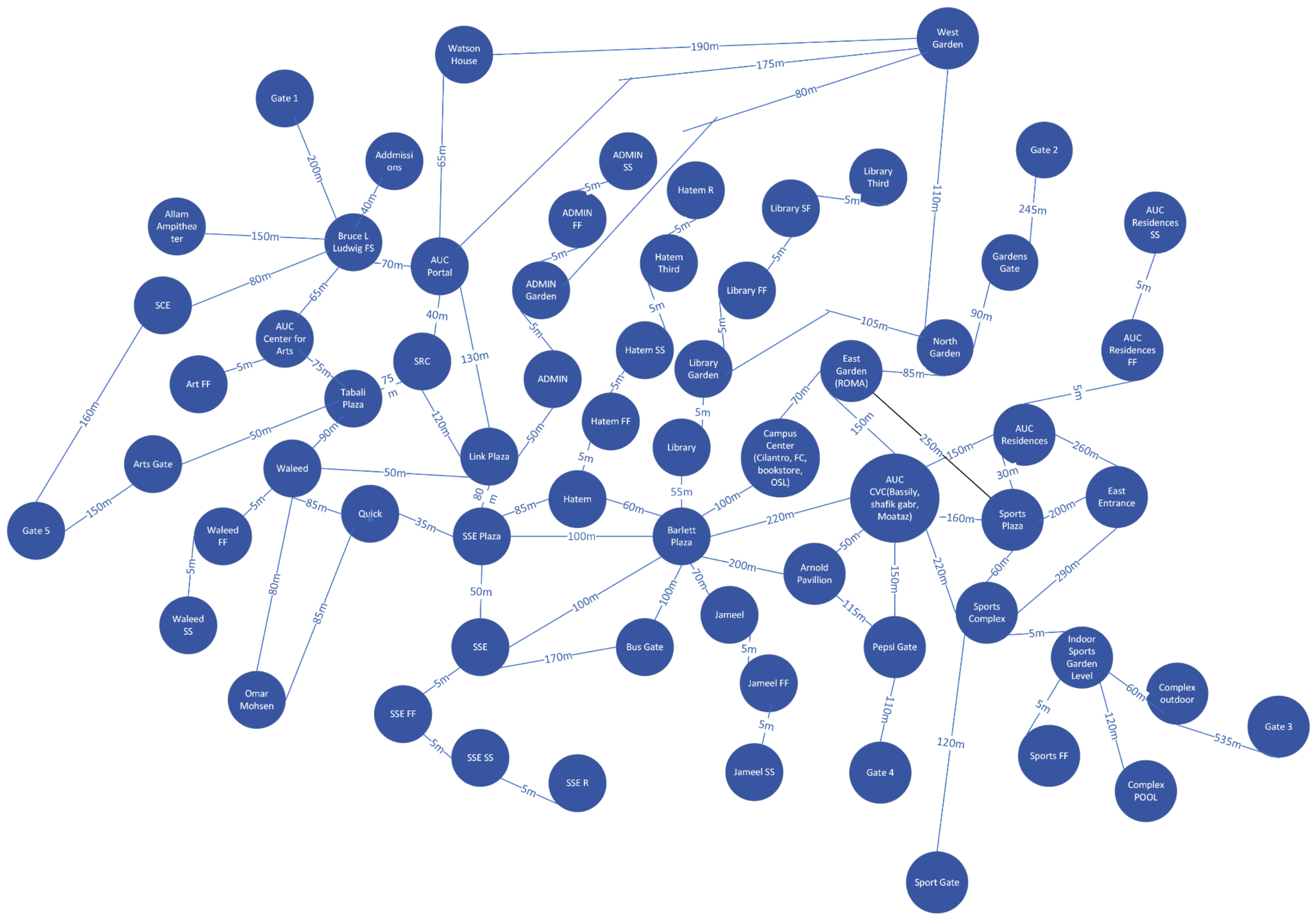


Figure 1: AUC’s graph

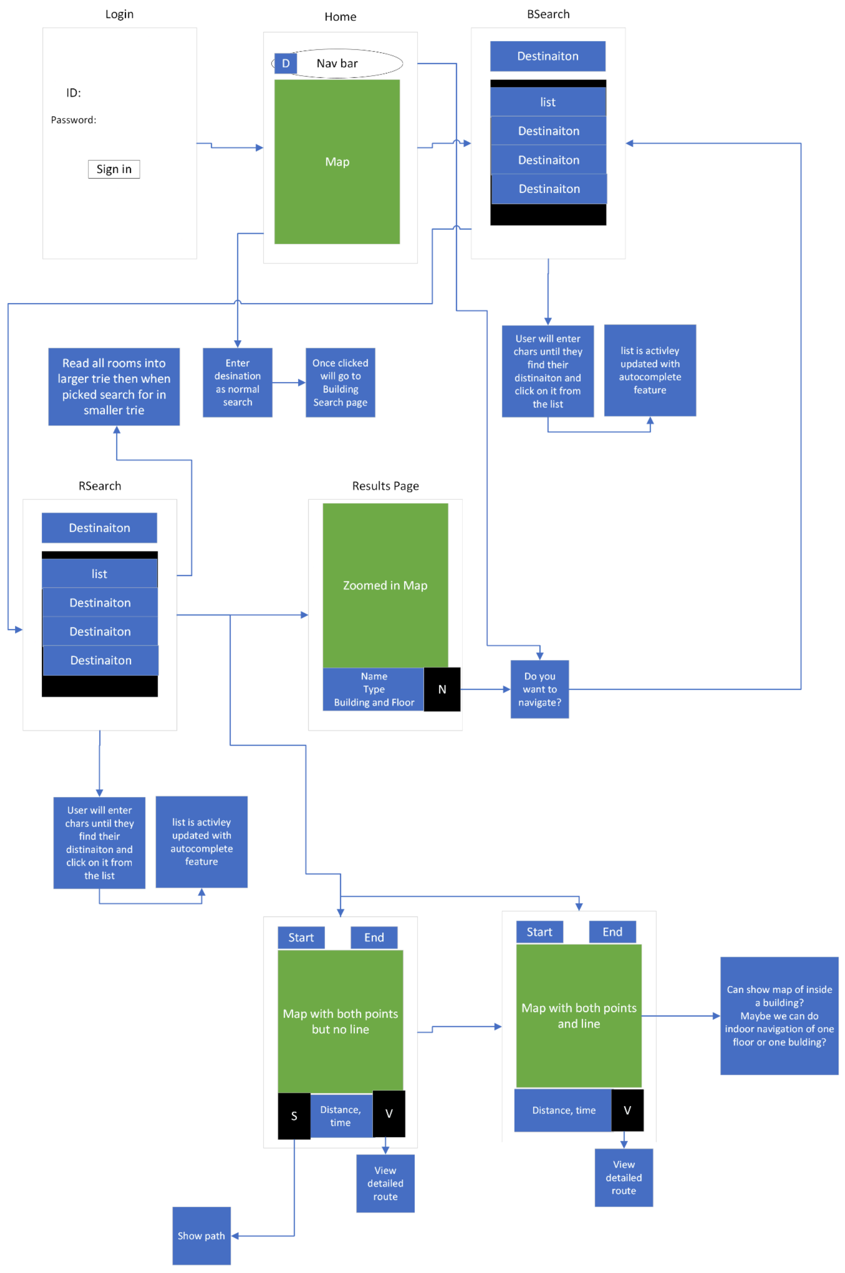

Login
ID:
Password:
Sign in
Home
D
Nav bar
Map
BSearch
Destinaiton
list
Destinaiton
Destinaiton
Destinaiton
Read all rooms into larger trie then when picked search for in smaller trie
Enter desination as normal search
Once clicked will go to Building Search page
User will enter chars until they find their distinaiton and click on it from the list
list is activley updated with autocomplete feature
RSearch
Destinaiton
list
Destinaiton
Destinaiton
Destinaiton
Results Page
Zoomed in Map
Name
Type
Building and Floor
N
Do you want to navigate?
User will enter chars until they find their distinaiton and click on it from the list
list is activley updated with autocomplete feature
Start
End
Map with both points but no line
S
Distance, time
V
View detailed route
Show path
Start
End
Map with both points and line
Distance, time
V
View detailed route
Can show map of inside a building? Maybe we can do indoor navigation of one floor or one bulding?

Figure 2: GUI Plan

4. Specification of Data Structures used

In the development of AUCMaps, our approach required the selection and implementation of data structures that would offer the best performance for the graph-based navigation system, and with our current data. The core of our application uses an adjacency list representation of a graph as an efficient and effective handling of navigation data. We chose this over an adjacency matrix or a compact list since the adjacency list provides a memory-efficient approach to sparse graph representation(O(V+E) where V is vertices and E is edges), which aligns with the layout of the AUC campus, which is relatively sparse concerning the capabilities of the graph data structure. Furthermore, it was best fit without an OOP plan. The structure of our nodes within the graph is composed of multiple elements using a struct: a name (string), which provides a GUI identifier for each location; a weight (int), which quantifies the cost or distance from a node to that node; and an associated trie object, to hold classroom information. This since we do not need new insert by the user or deletions by the user so the only time complexities that concerns AUCMaps with regards to the graph is the complexity for checking if there is an edge between two vertices which in the worst case is O(V) where V is vertices in the graph, and the complexity for lookup by index which is constant due to our implementation using a vector of vectors of type node. This trie object is important for the quick retrieval of classroom data, allowing the application to support fast searches. The trie is a tree data structure. Its implementation here is done by an unordered map. The Reason behind this is that it is more flexible than arrays. Meaning it does require a fixed size in order to work. Moreover, maps function similar to a hashtable, therefore, dealing with better complexities. In this implementation each node here represents a letter. Furthermore, each character in the string corresponds to a level in the trie. The root node does not represent any letters, but it contains a map named children that stores the characters of the first letters that are in the next level. Meaning if the root is at level one then the first character of a five word letter would be at level two. Furthermore, the last character would be stored at level six. In this project our nodes in a trie have four private members. First is the unordered_map. Second, a boolean variable in order to tell if this letter is the last letter of a word or not. Every node is initialized with a false boolean variable. Third and fourth, two string types in order to store the building of which this room is in and what type of room it is. While our trie class has a pointer to the trie nodes. For the representation of campus buildings and the correlation of their names to specific nodes within our graph, we used an enumeration (enum) data structure. This allowed us to map each building name to an integer value, creating an effective indexing system which we later use in Dijkstra. Pathfinding in AUCMaps utilizes Dijkstra's algorithm, a solid and proven method for finding the shortest paths between nodes in a graph. We implement this algorithm using priority queues and vectors, and it is based on the implementation of GeeksForGeeks which will be further discussed in the algorithm specification section. The priority queue deals with the order of node exploration based on cumulative weights, ensuring the algorithm always proceeds along the shortest path at that time. The priority queue has an insertion and deletion time complexity of O(log n). Additionally, vectors are used to store the names and types of classrooms, allowing for the dynamic storage of our data as more information becomes available we can easily expand. Further they provide constant look up times by index. With this in mind, we have used dynamic data structures for the majority of our data structures in this project since it has allowed for easier debugging and would allow us to easily expand on AUCMaps. Lastly, we have implemented the reading of data from a text file for the initial setup of the adjacency list. This decision was because we needed flexibility and speed of implementation. A text file allows for easy updates to the data structure without the need to recompile the code, a crucial feature for maintaining the application as we expand our app.

5. Specification of Algorithms used

The underlying data structure of AUCMaps is a graph, specifically an adjacency list, which is used for the insertion of nodes and the management of their edges. Each node insertion inserts a node type which includes: the name of the building, weight to the building (to itself is 0), and trie. We also indexed as mentioned previously using Enums. The custom code for the insertion and reading from the file can be found in Appendix A. The specific format of the text file is shown in the next section. With this we need to talk about the Trie. The Trie structure has two main functions: insert and search. The insert function's purpose in this project is to input classes into every building's trie. The insert function receives a string of the class name, its location, and the type of classroom. To insert we traverse the root's children array. If the current character we are working with does not exist, we create a new node. However, if it does already exist we use that current node's children map and proceed to do the same procedure. Lastly, when we reach the last character we mark it as the last one by turning its boolean variable to true. This translates to that this is where the word is complete. Moving on, the search function's objective is designed to allow us to search for a room in a building. It functions by searching for the first character of the string in the map of the root node.Then it looks for the second character in the children map of that of the first character. After, it goes down to the next level and repeats the procedure. This continues until it reaches the last character that should have the boolean variable set as true. Refer to both insertion and search functions code in appendix D&E below. Then we discuss our implementation of Dijkstra's algorithm. Our implementation of Dijkstra's algorithm is used to find the shortest path between locations on the AUC campus. It is inspired by the GeeksforGeeks implementation mentioned in the references, and by also learning the algorithm to fit it for our requirements. The algorithm starts by initializing distances from the start node to all others as infinitely large, except for the start node itself, which is set to zero. A priority queue manages the order of node traversal, based on the smallest cumulative distance, thus the most immediate node is prioritized for exploration. As the algorithm runs, each node's adjacent connections are examined, updating the shortest distance and predecessor when a more efficient path is discovered. This process iterates until all nodes have been visited, culminating in a comprehensive map of the shortest paths from the starting point to every other accessible location. The execution of Dijkstra's algorithm within our application is designed with development and debugging in mind. This is why the console currently outputs the date getting updated at the most suitable times as seen in the code in Appendix B. With this, we can revisit the code at any time and be able to work effectively. We also use a path reconstruction function that uses reverse indexing to return the path of nodes taken to reach the destination node so we can return it to the user along with the total distance. We considered proving the user with the distance to each node, and the total distance traveled but we found that to be confusing for the user and unnecessary. The total time complexity of the implication of Dijkstra is O((V+E)log(V)) where V is the number of vertices and E is the number of edges. This is because of the priority queue, relaxation procedure, and our reconstruct path function. Despite the existence of more complex algorithms, the straightforward and elegance of Dijkstra's without the need for heuristics proved most compatible with our graph's structure and the application's requirements. Moreover, it has proven highly effective for accurately finding the shortest path from node to node or from node to classroom despite a bug that we believe is due to an indexing issue that we have not found yet and which will be discussed in the analysis and critique section.

6. Data Specifications

This project did indeed require a lot of data in order to function properly. Reasoning is, the aim of it is to create a map for the university. Containing buildings, offices, rooms, plazas etc. Thus, we needed a source to obtain such data from. The main sources who helped build the base for this project were the AUC mobile application, google earth, and google maps. The AUC app helped gain the names of the rooms and buildings. Corresponding to that, google earth provided us with a picture of the university which is used on the front end to resemble a map of the university. Lastly, google maps was crucial to measure distances between buildings, as this is one of the main objectives of this project. To find the shortest distance between the source and the destination. Through google maps and AUC app we were able to write a text file. This text file's content was the backbone of our to create the adjacency list. Each line represents a building and where one can go from that building, with distance in mind. Hatem 0,SSEPlaza 85,BarlettPlaza 60,HatemFirstFloor 5,. This is an example of a line in our text file. This line represents a node for Hatem, whose distance to itself is zero. The distance to SSE Plaza is 85m.

Furthermore, the AUCMaps application is designed to interact with users through an input-output process to facilitate search and navigation across the campus. When it comes to data input, users provide data for two primary functions: searching and navigation. For search purposes, the user specifies the node, which is a building, then they can choose to specify a room within the building or just the building itself. For navigation, the user must input both the source and the destination nodes, which may be a choice between reaching a specific classroom or merely the building itself. Once processed, AUCMaps delivers comprehensive data back to the user. In search mode, it provides the specific location within the building, denoted by floor. In navigation mode, the application displays the total distance of the journey, the expected time taken, and a step by step path guide node by node. This detailed feedback ensures users can find their way efficiently and with confidence, enhancing their campus experience by minimizing uncertainty and optimizing travel time. Furthermore, they can use the dynamic satellite image map, which is labeled, to follow the shortest path.

7. Experimental Results

We would like to preface our results and documentation that the program works in both implementations, but we believe it is best as the GUI since it has more features and is much more user friendly.

Documentations:

GUI Program:

Displayed is an integrative map to help with navigation.

- To search for a classroom or location on campus click on the “Search AUCMaps” bar.
    - Choose the classroom or location you would like to search for from the list or type in the name and choose from the available choices.
    - Displayed is a map of the campus with waypoints for ease of navigation, and details about your search.
    - To navigate click on the “Navigate” button in the bottom left corner.
    - To close the application click the “Close” button in the bottom right corner.
- To navigate to a classroom or location on campus click on the “Directions” button or access after searching.
    - Displayed is an integrative map to help with navigation.
    - Click on the Source bar.
    - Choose the location you would like to start from in the list or type in the name and choose from the available choices.
    - Choose the classroom or location you would like to navigate to from the list or type in the name and choose from the available choices.

- Click on the “Directions” button to get the path, distance, estimated time, and location.
- Displayed is a zoomed in map of the location for ease of navigation.

This implementation limits the errors that the user can make since they can not pick an incorrect room or node. Furthermore, the buttons are set to only be pressed once the one before is done to avoid segmentation fault errors due to missing data. For example, you can not pick your designation before you pick your source in the directions page.

Terminal Program:

Attached below in Appendix F, you will find our output of the program. We serve 3 services. First, Search for a room in a building and receive its details. Second, receive the shortest path from one building to another. Lastly, Input the class and its building that you would like to visit and we will provide you with the shortest path ". In addition we have tested them with our current room sample that you will find commented in our main file. To use the program you need to use the enums as the input as shown in Appendix G. In the terminal program we handle user undefined entries through returning an index of -1 for an incorrect building and checking the index.

8. Analysis and Critique

The core data structure of our application is an adjacency list, utilized for the effective insertion of nodes and the management of our edges. For graphs, the time complexity of node insertion and edge management is O(1) and O(V) for all the vertices, given the direct access to nodes and edges. Studying the trie's complexity there are mainly two functions used in this project: insert and search. The insertion's complexity is O(n) where n is the number of characters in the string. Therefore, in this case the worst and average cases are pretty much the same. Similarly, this also follows for the search function. The major algorithm of our application is the implementation of Dijkstra's algorithm, which is the core functionality of finding the shortest paths across the AUC campus. The time complexity of our customized Dijkstra's algorithm is O((V+E)log(V)), where 'V' represents the number of vertices and 'E' is the number of edges in the graph. This complexity accounts for the use of a priority queue which optimizes the node traversal order, and the edge relaxation procedure used for updating shortest paths. Additionally, the algorithm includes a path reconstruction function that further contributes to the complexity. We critique our complexities since had we used a different approach, any one of the previously mentioned, we would have a complexity of the A* algorithm which is known to be faster due to the heuristic function. However, we still believe that we made the best decision for our application in the present since it achieves our goal without getting a high time complexity due to the use of a priority queue in Dijkstra. Despite this we note that in the future we would likely pursue a different approach: using a grid and modeling to map the campus then using the A* algorithm to find the optimal paths.

9. Conclusions and Reflections

This project came across some difficulties which could not be resolved with the current resources and time available. One being, this project only deals with sample rooms. Meaning, only the main buildings in the university will have classes inserted within them. This is due to constraints in time and resources, ideally we would collaborate with the university to get the database of the rooms.

Reflecting on the development of AUCMaps, we recognize that while our program serves its primary purpose well, there is always room for improvement. With additional resources and expertise, we want to enhance several aspects of the application. A more traditional mapping method could have significantly reduced search times, providing users with faster results. This would be especially beneficial for indoor navigation, which remains a challenge with our current mapping application. Our current model excels in guiding users to and from buildings, but navigating within the buildings themselves is an area that needs development.

Moreover, a more advanced set of GUI features for non GUI specialized programmers, like us, would greatly improve user interaction. While our current interface is user friendly and provides some advanced features we would like to implement lines that guide user's paths and GPS tracking which would be present in a more advanced or more programmer friendly program. Ultimately, AUCMaps stands as a successful navigation app for AUC, but we are aware of the potential that lies ahead. As we continue to refine our skills and expand our resources through our degree, we are excited about the possibilities to evolve AUCMaps or a similar application with advanced GUI and mapping, and perhaps artificial intelligence.

In summary, this project's main objective is to provide the user with the shortest path possible from one building to another. In order to avoid the consumption of lost routes or long ones. This is done by the utilization of data structures. Mainly, trie and graph. Moreover, with the use of sufficient algorithms such as the dijkstra. As the project might have come short handed in some areas like using only room samples, nevertheless, it still sufficiently accomplished its goal. In addition, a lot of learning lessons have been transcended to the team by facing these blocks. The team does not only hope to navigate the university, but it anticipates to continue to navigate also the endless evolution of technology and utilize future findings as catalyst in order to build breakthroughs in this significant field.

Acknowledgements

Eng. Mohammed Hany - Guidance and several meeting sessions

References:

“Trie: (Insert and Search).” *GeeksforGeeks*, GeeksforGeeks, 20 Feb. 2023, www.geeksforgeeks.org/trie-insert-and-search/.

“Dijkstra’s Algorithm for Adjacency List Representation: Greedy Algo-8.” *GeeksforGeeks*, GeeksforGeeks, 1 June 2023, www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/.

Appendix

Listing of all Relevant Implementation Codes

Appendix A: Reading from file and inserting into adjacency list

```
vector<string> Graph::readFile(string &theFile) {

   string ret;
   vector<string> str;
   str.resize(size);
   string line;
   ifstream file("Data.txt");
   int i =0;
   if(!file.is_open())
   {
       cerr<<"File is not open"<<endl;
   }
   else
   {
       while(getline(file, line))
       {
           str[i] += line + '\n';
           i++;
       }

   }

   file.close();

   //theFile = ret;

   return str;

}


//read file in createGraph and send it
string Graph::createSub(int &index, string &theFile, Buildings ind) {
   vector<string> ret = readFile(theFile);
   string s;
   int i = index;
   while(i < ret[ind].length() && (ret[ind])[i] != ',') {
       s += (ret[ind])[i];
```

```
        i++;
    }

    if (i == ret[ind].length()) {
        index = 0;
    } else {
        index = i + 1;
    }

    return s;
}



void Graph::getLoop(int commaCount[]) {
    ifstream file("Data.txt");
    string line;
    int i = 0;

    if (file.is_open()) {
        while(getline(file, line)) {
            commaCount[i] = count(line.begin(), line.end(), ',');
            i++;
        }
        file.close();
    } else {
        cerr << "Unable to open file" << endl;
    }

}



void Graph::createGraph(Buildings ind) {
    ifstream file("Data.txt");
    string line;
    string theFile;

    int commaCount[100];

    getLoop(commaCount);

    int j = 0;

    if (!file.is_open()) {
        cerr << "Unable to open file" << endl;
    }



    int index = 0; //be careful

    //the next index reads from the same area in the text file not the next line
    while (j<commaCount[ind]) {
        //index of substring to get name and weight separately
        int i = 0;
        string sub = createSub(index, theFile, ind);
        string number= {};
        int weight =0;
        Trie t;
        Node *p = new Node;
        while(sub[i] != ' ')
        {
            p->name += sub[i];
            i++;
        }
        while(sub[i] != '\0')
        {
            if(sub[i]!= ' ')
            {
                number += sub[i];}

            i++;

        }
        p->trie = t;
        p->weight = stoi(number);
        //test number of loops
        j++;
        adjList[ind].push_back(*p);

        }
}
```

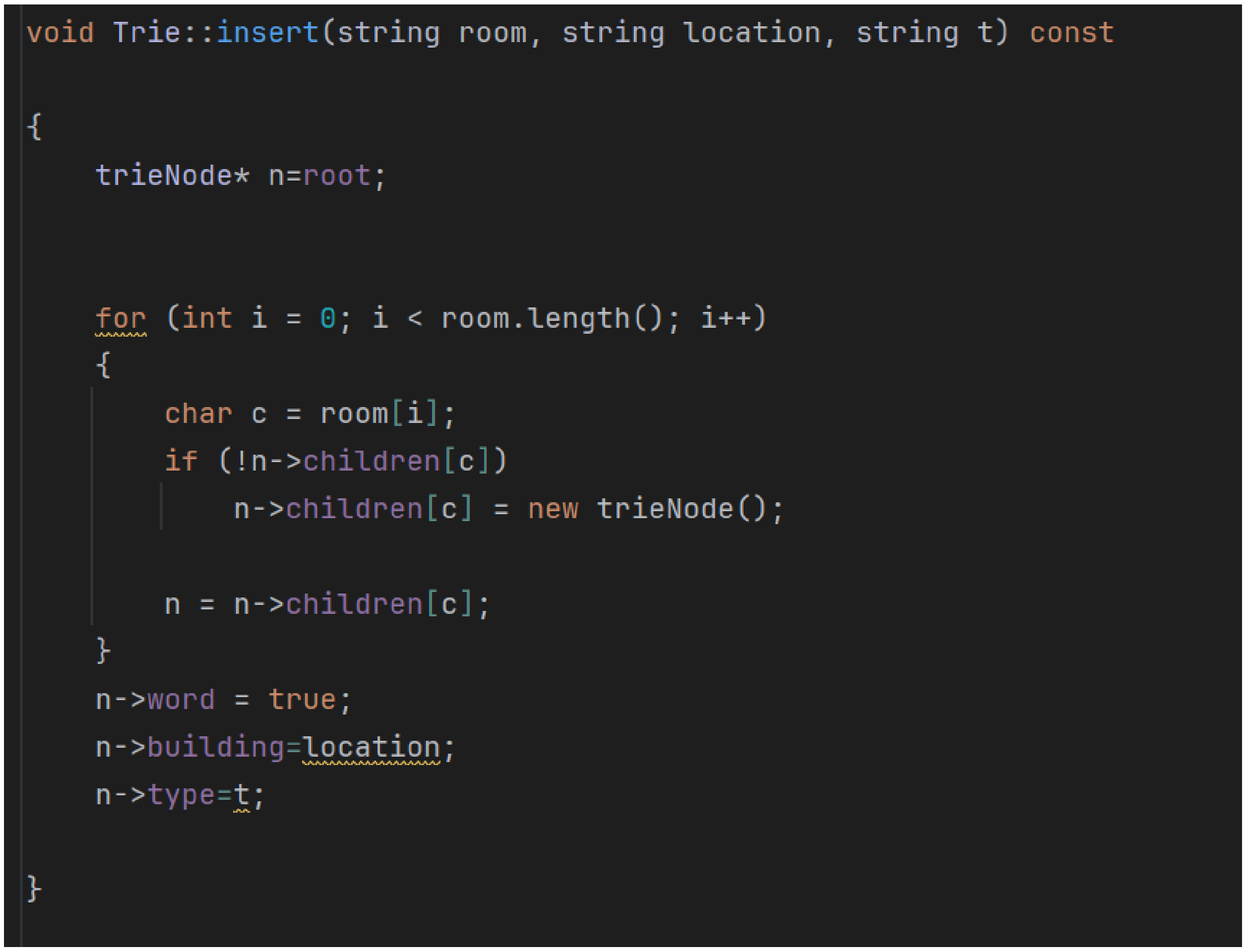
Appendix B: Dijkstra Algorithm

```
void Graph::dijkstra(int startNode, vector<string>& p, string endNode, int &x)
{
   vector<int> dist(size, INT_MAX); // Distance of all nodes from startNode
   priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
   vector<int> pred(size, -1);

   dist[startNode] = 0;
   pq.push(make_pair(0, startNode)); // (distance, node)

  cout << "Starting Dijkstra's algorithm from node " << startNode << endl;

   while (!pq.empty()) {
       int distance = pq.top().first;
       int currentNode = pq.top().second;
       pq.pop();

       cout << "Visiting node " << currentNode << " with distance " << distance
<< endl;

       bool isFirstNode = true;

       // Iterate through all the adjacent nodes of currentNode
       for (auto& adjNode : adjList[currentNode]) {
           if (isFirstNode) {
               isFirstNode = false;
               continue;
           }

           int nodeIndex = nodeToIndexMap[adjNode.name]; // Find the index of
the adjacent node
           int nodeDistance = distance + adjNode.weight;

           if(nodeIndex == 0){cout<<"LOOK HERE: "<<adjNode.name <<endl;}

           cout << "Checking adjacent node " << adjNode.name << " (Index: " <<
nodeIndex << ") with edge weight " << adjNode.weight << endl;

           // Check if a shorter path is found
           if (nodeDistance < dist[nodeIndex]) {
               dist[nodeIndex] = nodeDistance;
               pred[nodeIndex] = currentNode;
               pq.push(make_pair(nodeDistance, nodeIndex));
                cout << "Updating distance of node " << adjNode.name << " to "
<< nodeDistance << endl;
           }
       }


   }

   getPath(startNode, intNodeIndex(endNode), pred, p);
   x=dist[intNodeIndex(endNode)];



}
```

Appendix C: Reconst Path

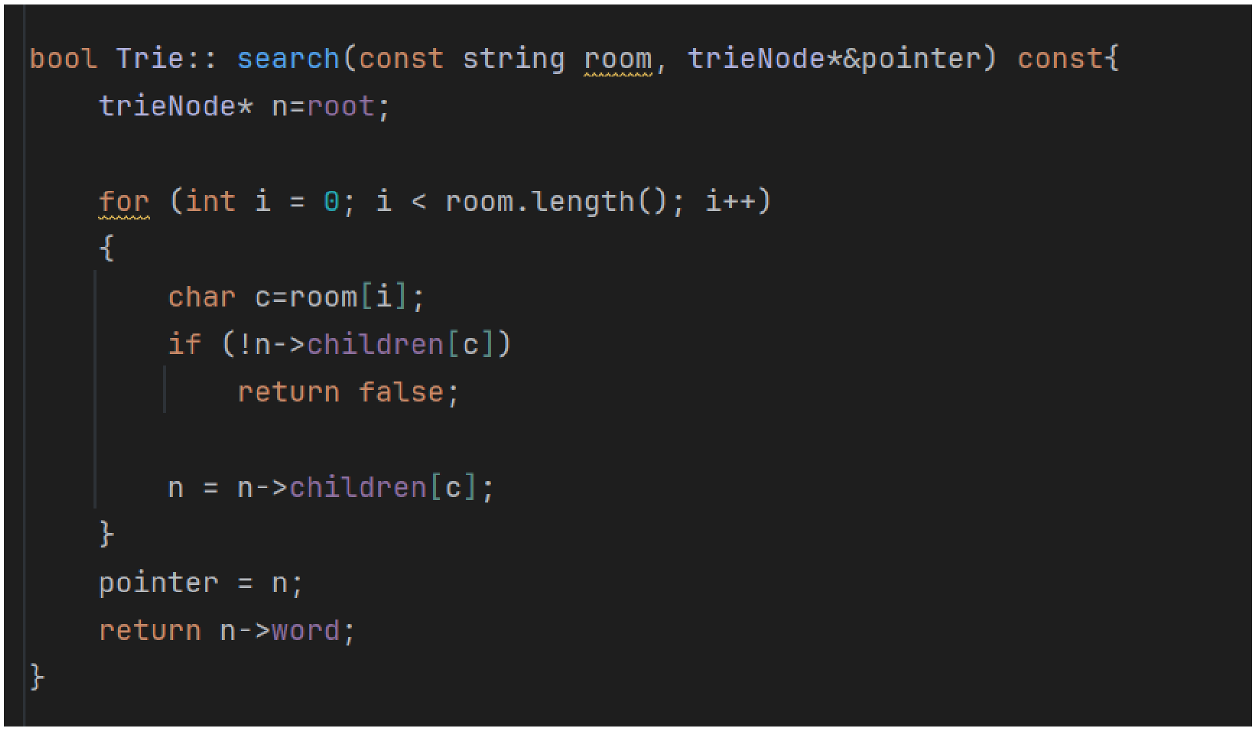
```
void Graph::getPath(int startNode, int endNode, const vector<int>& pred,
vector<string>& p)
{
   if (endNode == startNode)
   {
       p.push_back(indexToNodeMap[startNode]);
       return;
   }
   if (pred[endNode] == -1)
       return;

   getPath(startNode, pred[endNode], pred, p);
   p.push_back(indexToNodeMap[endNode]);
}
```

Appendix D: Trie insert function



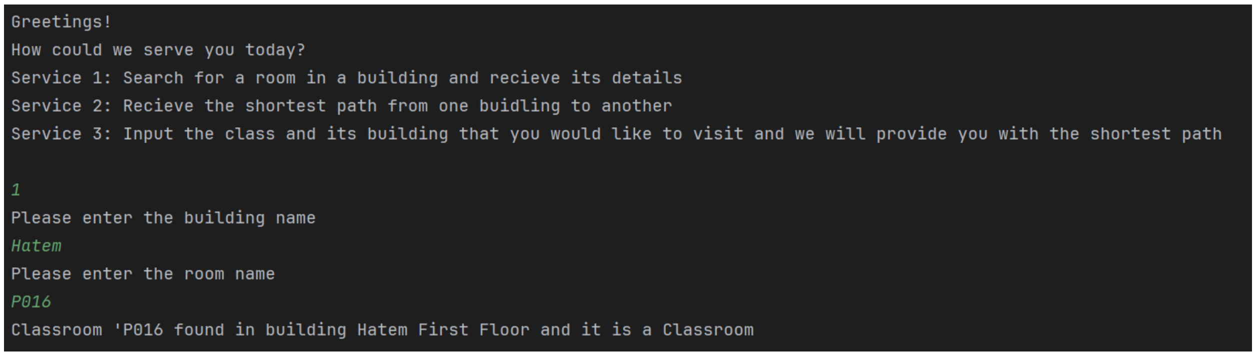
```
void Trie::insert(string room, string location, string t) const

{
    trieNode* n=root;


    for (int i = 0; i < room.length(); i++)
    {
        char c = room[i];
        if (!n->children[c])
            n->children[c] = new trieNode();

        n = n->children[c];
    }
    n->word = true;
    n->building=location;
    n->type=t;

}
```

Appendix E: Trie Search function

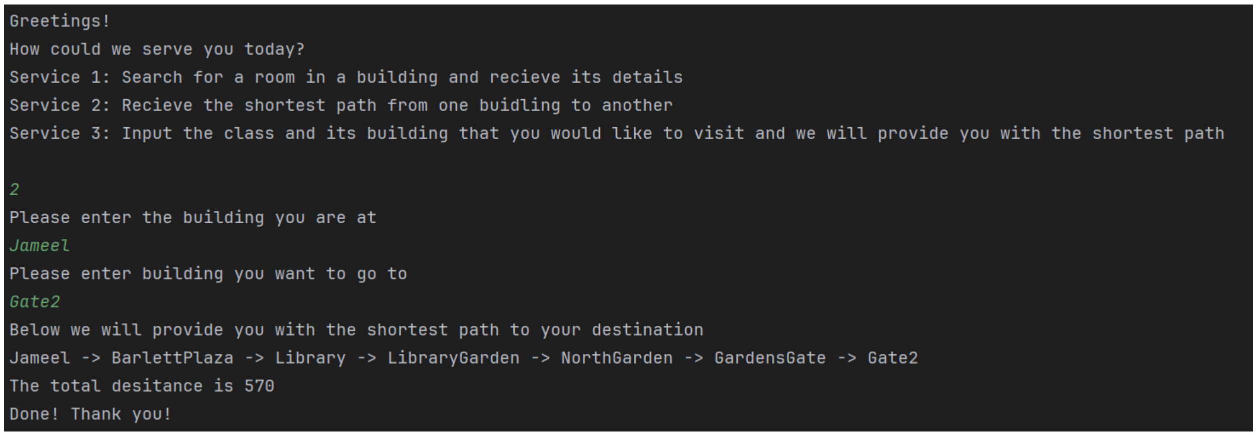


```
bool Trie:: search(const string room, trieNode*&pointer) const{
    trieNode* n=root;

    for (int i = 0; i < room.length(); i++)
    {
        char c=room[i];
        if (!n->children[c])
            return false;

        n = n->children[c];
    }
    pointer = n;
    return n->word;
}
```

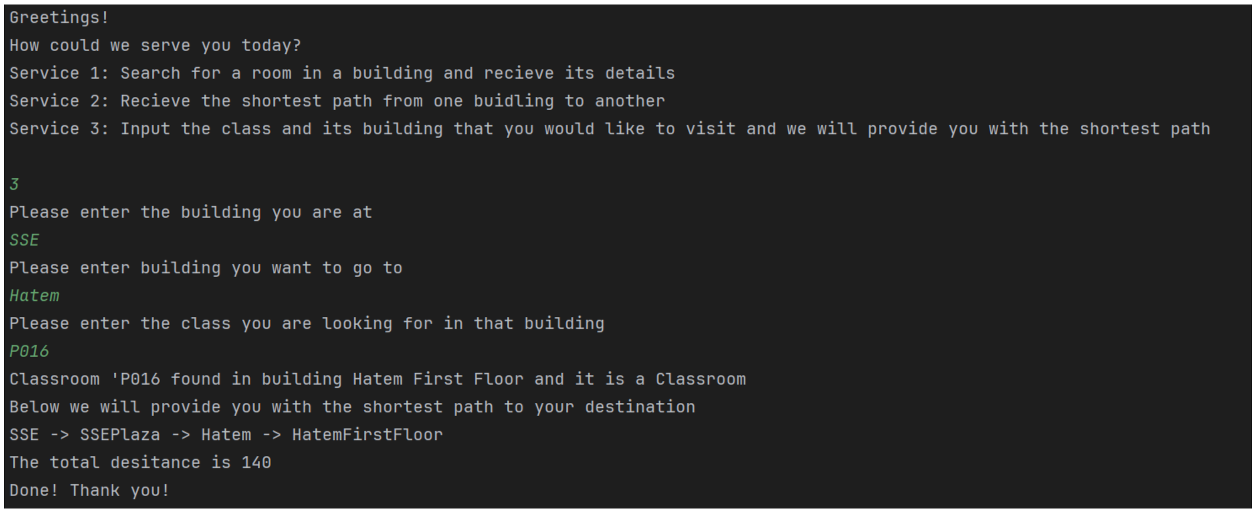
Appendix F: Sample output of the terminal program



```
Greetings!
How could we serve you today?
Service 1: Search for a room in a building and recieve its details
Service 2: Recieve the shortest path from one buidling to another
Service 3: Input the class and its building that you would like to visit and we will provide you with the shortest path

1
Please enter the building name
Hatem
Please enter the room name
P016
Classroom 'P016 found in building Hatem First Floor and it is a Classroom
```

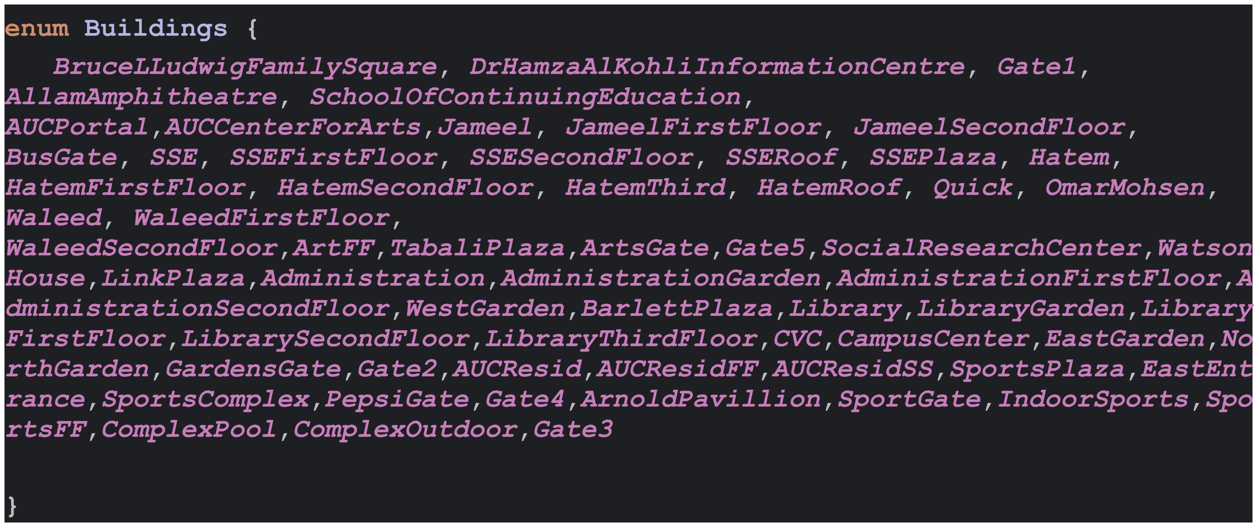


```
Greetings!
How could we serve you today?
Service 1: Search for a room in a building and recieve its details
Service 2: Recieve the shortest path from one buidling to another
Service 3: Input the class and its building that you would like to visit and we will provide you with the shortest path

2
Please enter the building you are at
Jameel
Please enter building you want to go to
Gate2
Below we will provide you with the shortest path to your destination
Jameel -> BarlettPlaza -> Library -> LibraryGarden -> NorthGarden -> GardensGate -> Gate2
The total desitance is 570
Done! Thank you!
```



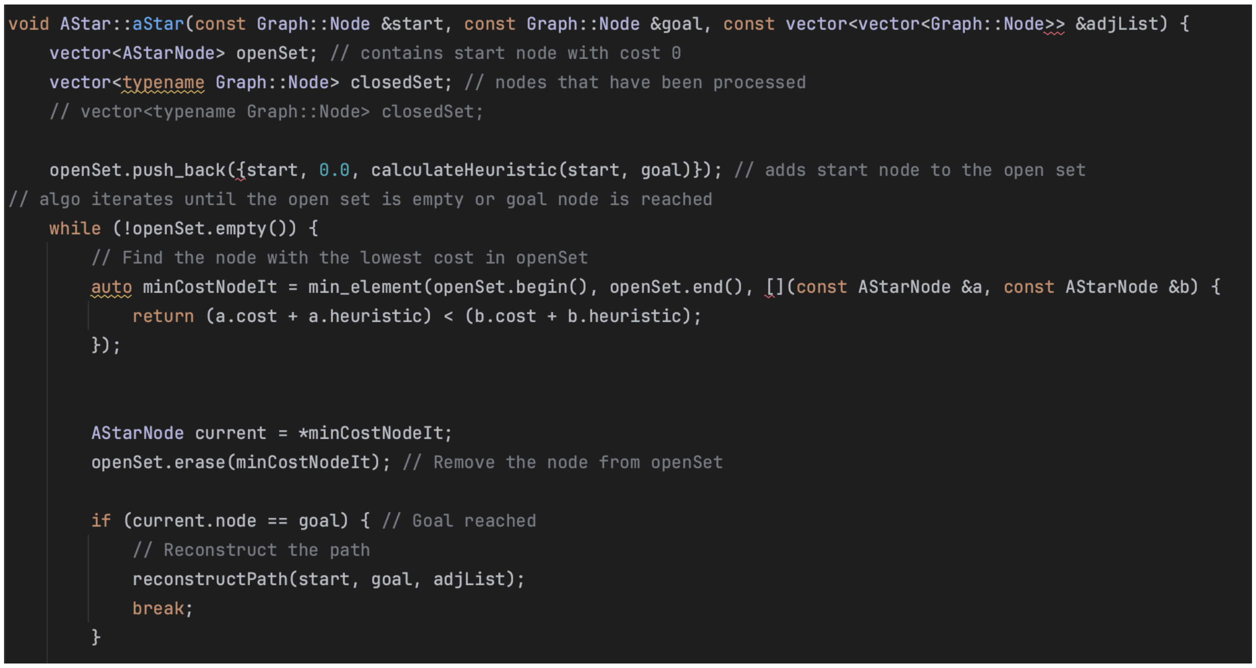
```
Greetings!
How could we serve you today?
Service 1: Search for a room in a building and recieve its details
Service 2: Recieve the shortest path from one buidling to another
Service 3: Input the class and its building that you would like to visit and we will provide you with the shortest path

3
Please enter the building you are at
SSE
Please enter building you want to go to
Hatem
Please enter the class you are looking for in that building
P016
Classroom 'P016 found in building Hatem First Floor and it is a Classroom
Below we will provide you with the shortest path to your destination
SSE -> SSEPlaza -> Hatem -> HatemFirstFloor
The total desitance is 140
Done! Thank you!
```

Appendix G: Enums used for indexing and terminal program

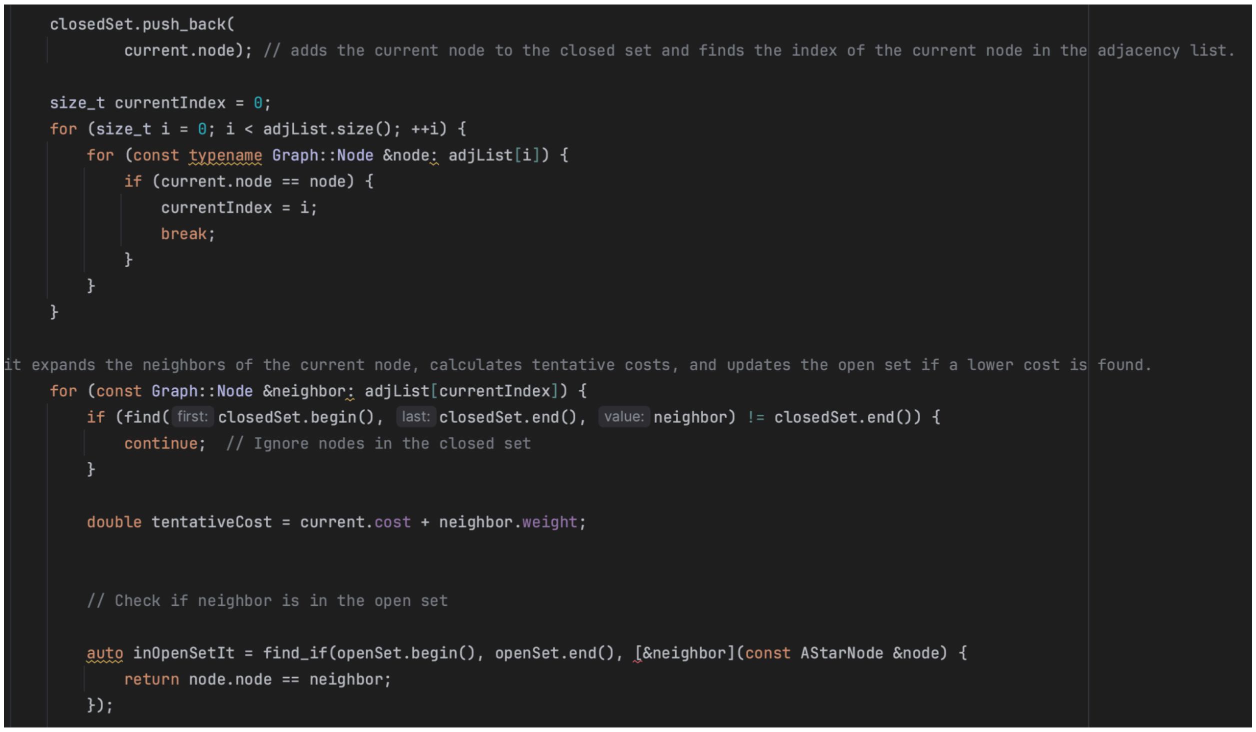


```
enum Buildings {
   BruceLLudwigFamilySquare, DrHamzaAlKohliInformationCentre, Gate1,
AllamAmphitheatre, SchoolOfContinuingEducation,
AUCPortal,AUCCenterForArts,Jameel, JameelFirstFloor, JameelSecondFloor,
BusGate, SSE, SSEFirstFloor, SSESecondFloor, SSERoof, SSEPlaza, Hatem,
HatemFirstFloor, HatemSecondFloor, HatemThird, HatemRoof, Quick, OmarMohsen,
Waleed, WaleedFirstFloor,
WaleedSecondFloor,ArtFF,TabaliPlaza,ArtsGate,Gate5,SocialResearchCenter,Watson
House,LinkPlaza,Administration,AdministrationGarden,AdministrationFirstFloor,A
dministrationSecondFloor,WestGarden,BarlettPlaza,Library,LibraryGarden,Library
FirstFloor,LibrarySecondFloor,LibraryThirdFloor,CVC,CampusCenter,EastGarden,No
rthGarden,GardensGate,Gate2,AUCResid,AUCResidFF,AUCResidSS,SportsPlaza,EastEnt
rance,SportsComplex,PepsiGate,Gate4,ArnoldPavillion,SportGate,IndoorSports,Spo
rtsFF,ComplexPool,ComplexOutdoor,Gate3


}
```

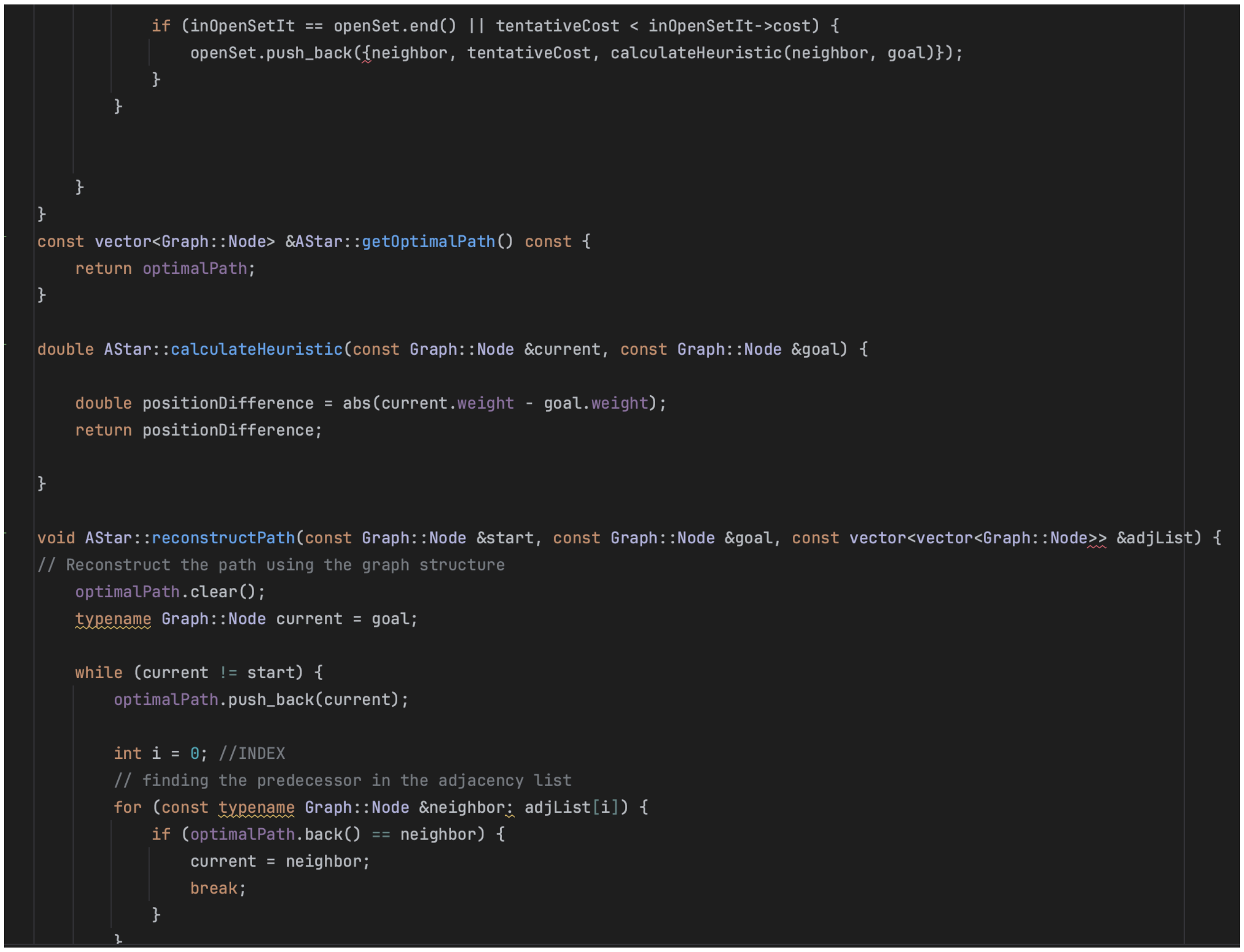
Appendix H: A Star Algorithm



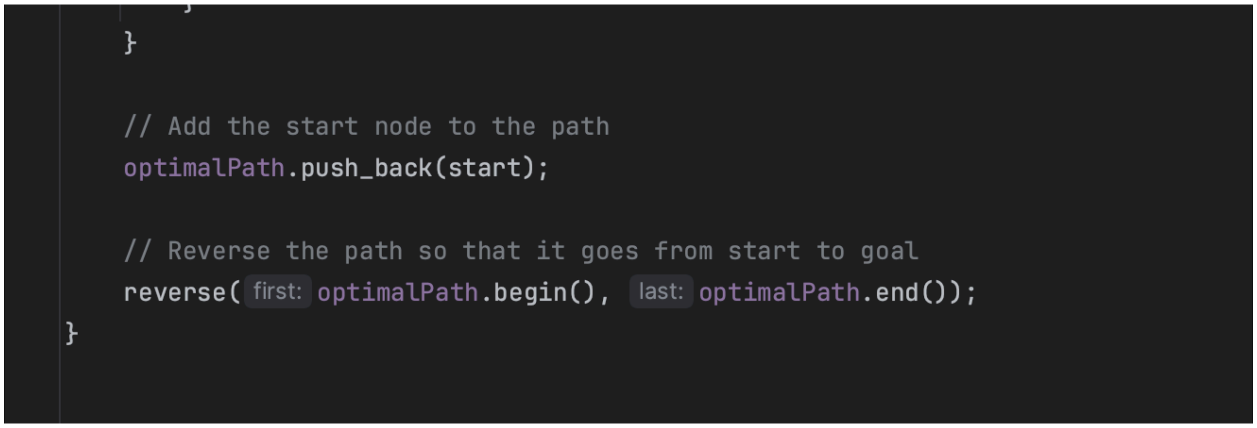
```cpp
void AStar::aStar(const Graph::Node &start, const Graph::Node &goal, const vector<vector<Graph::Node>> &adjList) {
    vector<AStarNode> openSet; // contains start node with cost 0
    vector<typename Graph::Node> closedSet; // nodes that have been processed
    // vector<typename Graph::Node> closedSet;

    openSet.push_back({start, 0.0, calculateHeuristic(start, goal)}); // adds start node to the open set
// algo iterates until the open set is empty or goal node is reached
    while (!openSet.empty()) {
        // Find the node with the lowest cost in openSet
        auto minCostNodeIt = min_element(openSet.begin(), openSet.end(), [](const AStarNode &a, const AStarNode &b) {
            return (a.cost + a.heuristic) < (b.cost + b.heuristic);
        });


        AStarNode current = *minCostNodeIt;
        openSet.erase(minCostNodeIt); // Remove the node from openSet

        if (current.node == goal) { // Goal reached
            // Reconstruct the path
            reconstructPath(start, goal, adjList);
            break;
        }
```



```cpp
    closedSet.push_back(
            current.node); // adds the current node to the closed set and finds the index of the current node in the adjacency list.

    size_t currentIndex = 0;
    for (size_t i = 0; i < adjList.size(); ++i) {
        for (const typename Graph::Node &node: adjList[i]) {
            if (current.node == node) {
                currentIndex = i;
                break;
            }
        }
    }

it expands the neighbors of the current node, calculates tentative costs, and updates the open set if a lower cost is found.
    for (const Graph::Node &neighbor: adjList[currentIndex]) {
        if (find( first: closedSet.begin(),  last: closedSet.end(),  value: neighbor) != closedSet.end()) {
            continue;  // Ignore nodes in the closed set
        }

        double tentativeCost = current.cost + neighbor.weight;


        // Check if neighbor is in the open set

        auto inOpenSetIt = find_if(openSet.begin(), openSet.end(), [&neighbor](const AStarNode &node) {
            return node.node == neighbor;
        });
```



```cpp
                if (inOpenSetIt == openSet.end() || tentativeCost < inOpenSetIt->cost) {
                    openSet.push_back({neighbor, tentativeCost, calculateHeuristic(neighbor, goal)});
                }
            }


        }
}
const vector<Graph::Node> &AStar::getOptimalPath() const {
    return optimalPath;
}

double AStar::calculateHeuristic(const Graph::Node &current, const Graph::Node &goal) {

    double positionDifference = abs(current.weight - goal.weight);
    return positionDifference;

}

void AStar::reconstructPath(const Graph::Node &start, const Graph::Node &goal, const vector<vector<Graph::Node>> &adjList) {
// Reconstruct the path using the graph structure
    optimalPath.clear();
    typename Graph::Node current = goal;

    while (current != start) {
        optimalPath.push_back(current);

        int i = 0; //INDEX
        // finding the predecessor in the adjacency list
        for (const typename Graph::Node &neighbor: adjList[i]) {
            if (optimalPath.back() == neighbor) {
                current = neighbor;
                break;
            }
```



```cpp
    }

    // Add the start node to the path
    optimalPath.push_back(start);

    // Reverse the path so that it goes from start to goal
    reverse( first: optimalPath.begin(),  last: optimalPath.end());
}
```