Team:
Alessandro junior Parli U47375394
Jude Mersinger U21950507

Report of Final Project COP4530

Implementation of Dijkstra's Algorithm to Find the Shortest Path

Introduction

In this report, Dijkstra's Algorithm will be researched, analyzed, and executed to find the shortest path between two vertices. Including the creation of an undirected weighted graph Abstract Data Type (ADT). Adjacency lists, adjacency matrices, or incidence matrices can be used to build the implemented graph. In this case, the adjacency lists will be used in this programming project. It will offer methods to add, and remove edges and/or vertices, including Dijkstra's algorithm to find the shortest path between two points.

There are two main objectives in this project which are, implementing Dijkstra's algorithm and the graph ADT, and second is demonstrating how they work with examples and code. By accomplishing these objectives, we hope to have a deeper understanding of the inner workings of Dijkstra's shortest path-finding algorithm. This hands-on experience with graph data structures and algorithms will help us with these results.

The internal workings and functionalities of the implemented graph ADT will be examined in detail in this paper. Next, we will do the same to Dijkstra's algorithm and describe its operation and relationship to the graph. Lastly, the report will discuss the project's results and offer examples. Finalizing this report, all these requirements for COP 4530's Programming Project 4 will be fulfilled.

Problem statement and Approach

The main focus of this project is to solve a basic graph theory problem, which is to find the shortest path between two vertices in a network that is represented as a graph. For example, imagine a network of roads in which every road segment has a weight (distance). Within this network, the objective is to find the path between two distinct locations (vertices) which has the lowest cumulative weight/distance. Dijkstra's algorithm is the main tool for solving this issue. It's an effective tool for quickly determining the weighted graph's shortest path. To accomplish this, the aim of our project was to integrate Dijkstra's algorithm with an undirected weighted graph ADT. This is how we worked with the problem:

1. ADT Graph Design: The fundamental component of our implementation is the Graph class, which serves as the primary Abstract Data Type (ADT) for modifying the graph structure. We began by creating a Graph ADT with the features outlined in the project requirements. This included commands to add and remove edges and vertices, making sure that every vertex has an identifiable label for simple identification. This flexibility allowed us to adapt the graph as needed throughout the program's execution. Every vertex in the graph has a distinct label (usually a string) that acts as

its identity. This label-based system makes it easier to retrieve and manipulate particular vertices in an efficient manner, by quickly referring to and working with specific graph elements when they have unique labels. We also used Dijkstra's algorithm to determine the shortest path between two vertices, which will be explained later in the paper. For efficient management of vertices and edges, the Graph class internally utilizes two maps. Whereas the edge map uses integers (possibly unique identifiers) as keys to access Edge objects, the vertex map uses string labels as keys to access corresponding Vertex objects. Furthermore, the Graph class has a destructor to ensure the correct deallocation of memory resources when the graph object is not in use. This keeps the system stable and stops memory leaks. The Graph class will serve as the root for working with graphs and implementing algorithms like Dijkstra's shortest path finding. We'll go more into the implementation details in the following sections, examining how it works with the previously mentioned Vertex and Edge classes to accomplish these features. We'll also take a closer look at the given code portion for the Graph class in order to get a better idea of the functions it implements.

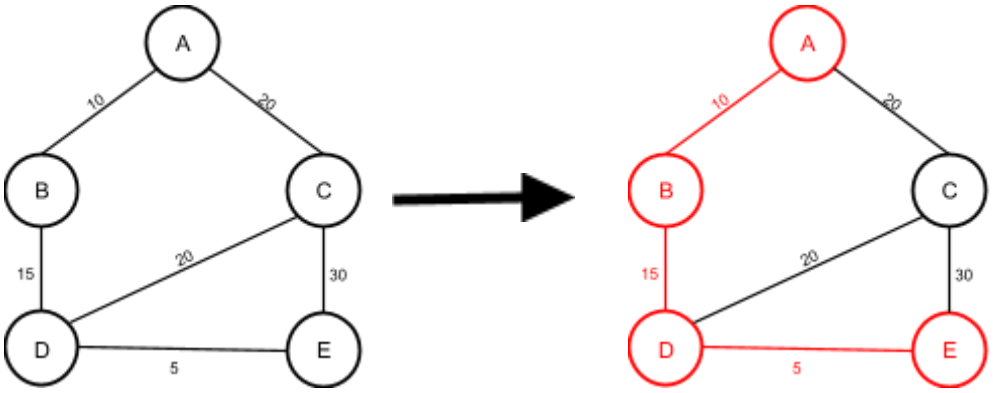
2. Implementation Details**:** We started by implementing the Graph class in order to make the design implementation possible. We used an adjacency list representation in order to store edges efficiently. Basically, an adjacency list is an array of lists. Every position in the array refers to a vertex in the graph, and the labels of all vertices that are immediately connected to it are stored in the list at that position. Since it does not store empty entries, as an adjacency matrix representation would, this structure conserves space for limited graphs, and makes the code more effective. Furthermore, an adjacency list makes it quicker to retrieve a vertex's neighbors, or connected vertices. Instead of iterating through a whole row in an adjacency matrix, we can just access the corresponding list in the array. Lets take for example a social network where users can communicate with friends. An array of lists, where each position in the array represents a user and the list at that position stores the usernames of their friends. Where "A" is a user and "B" and "C" are the neighbors, then "A’ s " adjacency list entry would be a list listing "B" and "C". By doing this, the connections between users are effectively captured without wasting space on non-existent neighbors. So compared to an adjacency matrix, which would store connection information for every user pair regardless of their relation, this method provides faster access by storing edges connected to each vertex in a separate list. Because of its efficiency, it can be used to implement Dijkstra's algorithm, which finds the shortest path within our graph. In the following sections, we'll go deeper into the details of the Graph class implementation and discuss the creation of distinct classes for edges and vertices to improve the code. The edge class will hold data regarding the connection between two vertices. This will include the weight attached to the edge (unsigned long in the project example) and the two vertex labels it connects (references to Vertex objects). It also includes methods to access and modify its data, just like the Vertex class does. As for the vertex class, it acts as the foundation for individual points within the graph. It typically stores a unique label (often a string) for easy

identification and potentially other data relevant to the specific problem being modeled. Additionally, it will have methods to access and modify this data.

3. Dijkstra's Algorithm Integration: The Dijkstra algorithm's implementation in the Graph ADT is the project's essential component. The shortestPath function is a crucial feature for this class. As mentioned before, the role of this function is in storing the traversed vertex labels in a given vector, and in returning the total weight connected to the shortest path. This makes it possible for us to reconstruct the best possible route between the selected points. The vertices nearest to the starting point are given priority as this algorithm iteratively examines the graph. As it goes along, it updates distances dynamically and finds the shortest path between the selected start and end vertices. For example, as mentioned before, consider a complicated network of roads where each road segment has a weight. Dijkstra's algorithm finds the shortest path between two points, enabling you to move through this network with efficiency. The algorithm works repeatedly beginning at a chosen vertex and gradually expanding its understanding of the graph. Here is a summary of its main steps:
    1. Assign a current distance of 0 to the source node and infinity to the remaining nodes.
    2. Assign the current node to the non-visited node that is currently the closest.
    3. The current node N adds the weight of the edge connecting 0 to 1 to the current distance of each neighboring node. Set it as the new current distance of N if it is less than the current distance of Node.
    4. Designate node 1 as visited at this time.
    5. In the event that nodes remain unvisited, proceed to step 2.

   This algorithm successfully determines the shortest path between each vertex in the graph and the starting point at the conclusion of this process. Furthermore, it will be able to reconstruct the actual shortest path. For instance, consider a road network where A, B, C, D, and E are locations. The distances between these locations are represented in the diagrams below.

   Dijkstra's algorithm would give priority to examining neighboring vertices that are closest to A (B with weight/distance 10), in order to determine the shortest path from A to E. After that, it would investigate routes from C and B before figuring out that the quickest path from A to E is A → B → D → E, which is a total of 30. To figure out this result, we are applying the steps of the algorithm given before. Dijkstra's algorithm integration gives the Graph class the ability to navigate and analyze weighted graphs with ease.



The next sections of this report go more deeply into the details of the Graph ADT implementation, the selected data structures, and the thorough justification of Dijkstra's algorithm in this particular context. We will also go over the project's results and show off some code samples.

Code Implementation:

The Header file (Graph.hpp) will include all the necessary libraries and the classes, including the vertex, edge, and graph classes. Basic functionalities such as input/output (iostream), vectors (vector), strings (string), algorithms (algorithm), and limits (climits) are included. To help with the effective storage and retrieval of vertices and edges within the graph, it also has headers for maps (map). In a graph, an edge (connection) between two vertices is represented by the Edge class. It has private member variables to store the starting vertex label (start_of_edge), ending vertex label (end_of_edge), and the weight associated with the edge (weight_of_edge). The class also includes public member functions as you may see in the code. Within the graph, a vertex (node) is represented by the Vertex class. It has a private member variable called label that holds the vertex's distinct string identifier. It also has a connecting_edges map that stores pointers to connected Vertex objects as values and uses strings as keys, which could be vertex labels. This map offers effective edge management associated with a specific vertex. The class also includes public member functions as you may see in the code. Next we will analyze the .cpp file to check the actual procedure and operations made to make this program achieve its goal efficiently and successfully. The next code will implement all the functionalities declared in the earlier .hpp file for the Vertex and Edge classes.

Vertex Class:

- “addVertex” creates a new edge by adding a key-value pair to the “connecting_edges” map. The key is the neighboring vertex's label, and the value is a pointer to the connected “Vertex” object.
- “removeVertex” removes an edge by erasing the entry in the “connecting_edges” map that corresponds to the provided vertex label.

- “adjacents” provides a constant reference to the “connecting_edges” map, allowing access to the vertex's neighboring vertices.
- “getLabel” retrieves the vertex's unique identifier stored in the label member variable.

```
class Vertex
{
    private:
        std::string label; // Label of the vertex
        std::map<std::string, Vertex*> connecting_edges; // Map to store connecting vertices
        std::string getLabel() const; // Getter function for the label

    public:
        friend class Graph; // Granting access to the Graph class
        void addVertex(std::string label, Vertex* v); // Add a vertex to the connecting edges
        void removeVertex(std::string label); // Remove a vertex from the connecting edges
        std::map<std::string, Vertex*> adjacents() const; // Get adjacent vertices

        Vertex(std::string input)
        {
            this->label = input; // Initializing the label of the vertex
        }
};
```

Edge Class:

- “isStart” functions determine if a given label is the starting point of the edge. They check if the label matches the “start_of_edge” member variable. The two-argument version allows for potentially directed edges where starting and ending points differ.
- “isEnd” functions behave similarly to “isStart” but check if a label is the ending point of the edge by comparing it to the “end_of_edge” member variable.

```
class Edge
{
    private:
        std::string start_of_edge; // Starting point of the edge
        std::string end_of_edge; // Ending point of the edge

    public:
        friend class Graph; // Granting access to the Graph class
        unsigned long weight_of_edge; // Weight of the edge
        bool isStart(std::string label_1, std::string label_2); // Check if the given labels are the start of this edge
        bool isStart(std::string label_1); // Overloaded function to check if the given label is the start of this edge
        bool isEnd(std::string label_1, std::string label_2); // Check if the given labels are the end of this edge
        bool isEnd(std::string label_1); // Overloaded function to check if the given label is the end of this edge

        Edge(std::string s, std::string e, unsigned long w)
        {
            start_of_edge = s; // Initializing the start of the edge
            end_of_edge = e; // Initializing the end of the edge
            weight_of_edge = w; // Initializing the weight of the edge
        }
};
```

Dijkstra's Algorithm:

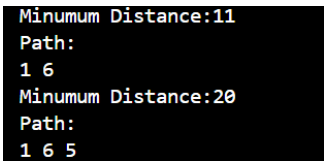
- Iterates through vertexQueue, checking the distance of all unvisited adjacent vertices for each vertex
- Uses "minDistance();" to find the vertex with the minimum distance from the current vertex
- Uses "comparator()" to compare the distances of adjacent vertices
- Removes the current vertex from the vertexQueue and iterates

```
// Dijkstra's algorithm
while (!vertexQueue.empty())
{
    std::string currentLabel = minDistance(vertexQueue);
    unsigned long currentDistance = vertexQueue.at(currentLabel);

    for (auto it : vertex.at(currentLabel).adjacents())
    {
        if (unvisited(it.first, vertexQueue))
        {
            unsigned long oldDistance = vertexQueue.at(it.first);

            // Comparing distances
            vertexQueue.at(it.first) = comparator(currentLabel, it.first, currentDistance, oldDistance);

            if (oldDistance != vertexQueue.at(it.first))
                visited.at(it.first) = currentLabel;
        }
    }
    vertexQueue.erase(currentLabel);
}
```

Results:

```
// Vertices and edges for the graph
std::vector<std::string> vertices1 = { "1", "2", "3", "4", "5", "6" };
std::vector<std::tuple<std::string, std::string, unsigned long>> edges1 = {
    {"1", "2", 7}, {"1", "3", 9}, {"1", "6", 14},
    {"2", "3", 10}, {"2", "4", 15},
    {"3", "4", 11}, {"3", "6", 2},
    {"4", "5", 6},
    {"5", "6", 9}
};
```

```
// Find the shortest path between vertices "1" and "6"
unsigned long x = g.shortestPath("1", "6", path1);

// Find the shortest path between vertices "1" and "5"
unsigned long y = g.shortestPath("1", "5", path2);
```



Conclusion:

Dijkstra's algorithm was successfully applied in this project to an undirected weighted graph (ADT). The report examined the integration of the algorithm, implementation specifics, and design decisions. For effective edge storage and retrieval, we employed adjacency lists, which makes the graph ADT appropriate for this project. The task of determining the shortest path between two vertices in a network was successfully handled by the functionalities that were implemented and the data structures that were selected. We were able to comprehend the capabilities and limitations of both the graph ADT and Dijkstra's algorithm better by exploring their inner workings. Our understanding of graph data structures and algorithms was strengthened by this project, which gave us an effective experience with these fundamental ideas in computer science. Overall, this project was successful in demonstrating how Dijkstra's algorithm works well for locating the shortest path within a weighted graph. The developed graph ADT is a useful tool for working with and examining graph structures, and the skills learned from this project can be used to solve a variety of computer science tasks involving graphs.

References:

"What Is Dijkstra's Algorithm?: Introduction to Dijkstra's Shortest Path Algorithm." GeeksforGeeks, GeeksforGeeks, 8 Mar. 2024, www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/#dijkstras-algorithm .

Riley, Sean. "Dijkstra's Algorithm - Computerphile." *YouTube*, YouTube, 4 Jan. 2017, www.youtube.com/watch?v=GazC3A4OQTE

Devadas, Srini, director. *Lecture 16: Dijkstra*. *Youtube Lecture 16: Dijkstra*, MIT OpenCourseWare, 14 Jan. 2013, https://www.youtube.com/watch?v=2E7MmKv0Y24&t=1661s. Accessed 14 Apr. 2024.

Navone, Estefania Cassingena. "Dijkstra's Shortest Path Algorithm - a Detailed and Visual Introduction." *freeCodeCamp.Org*, freeCodeCamp, 3 Feb. 2022, www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/.