DATA STRUCTURES FINAL PROJECT REPORT: DIJKSTRA'S ALGORITHM

Made by :

- **Avariq Fazlur Rahman / 2502043002**
- **Bernard Choa / 2502022414**
- **Chellshe Love Simrochelle / 2502043040**

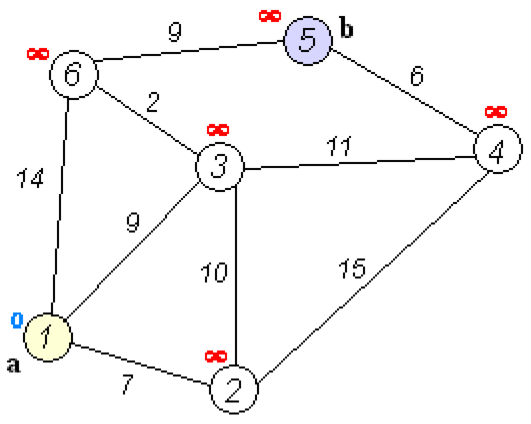
- **Project Description**

  · What is Dijkstra’s Algorithm?

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph. With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. In particular, you can **find the shortest path from a node (called the "source node") to all other nodes in the graph**, producing a shortest-path tree.

This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, especially in domains that require modelling networks. Dijkstra's Algorithm can only work with graphs that have **positive** weights. This is because, during the process, the weights of the edges have to be added to find the shortest path. If there is a negative weight in the graph, then the algorithm will not work properly. Once a node has been marked as "visited", the current path to that node is marked as the shortest path to reach that node. And negative weights can alter this if the total weight can be decremented after this step has occurred.

A simple rendition of Dijkstra’s Algorithm can be found like this :



  · Our Project

From this, we decided to make a visualization of Dijkstra’s Algorithm by implementing it inside a pathfinding program that helps find the shortest path from a starting node to an end node.

- Data Structure Used and Alternatives

In this project, we used Priority Queue as our main data structure method. Specifically, we used the priority queue, better known as “Heap”. The priority queue selects the next vertex so as to (eventually) ensure the shortest paths in a weighted graph. If you use a FIFO queue instead, you will not be able to account for arbitrary edge weights. This will essentially be a breadth-first search which only guarantees to find the shortest paths in unweighted graphs. The use of a priority queue comes at the cost of greater runtime - usually by a log factor.

While using a priority queue in Dijkstra's algorithm makes sense for sparse graphs, you can get away without an explicit container for dense graphs (without a slowdown).

- Analysis of Data Structure

Our project mainly uses Queues to form it. A Queue is a linear structure that follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

However, a simple queue wouldn’t work due to it being a FIFO order, so our project mainly uses a priority queue or usually is also called “Heap”. A priority Queue is an abstract data type, which is similar to a queue, however, in the priority queue, every element has some priority. The priority of the elements in a priority queue determines the order in which elements are removed from the priority queue. Therefore all the elements are either arranged in an ascending or descending order.

Last but not least, the Graph data structure is also used, since it is the main data structure that the algorithm will operate on.

- Program Manual

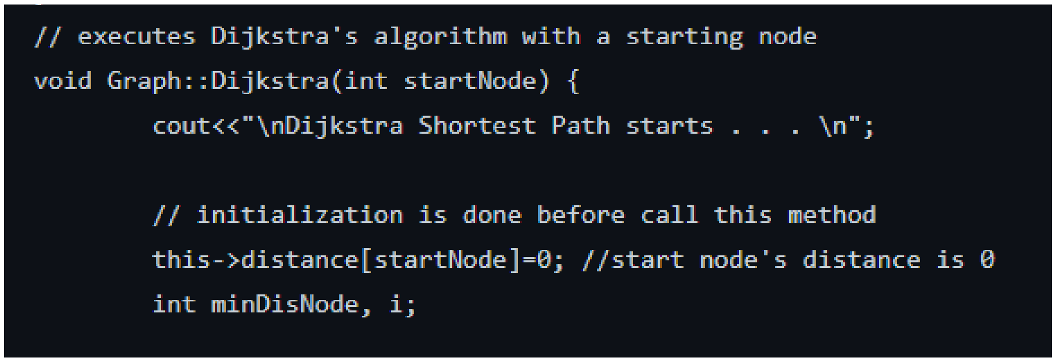
The program contains code for the Graph class with 5 parameters:

- adjMatrix: The adjacency matrix representation of the graph object
- vCount: Amount of vertices
- key: An array that contains either 0s or 1s, 0 means the node of the graph is not part of the minimum spanning tree yet, 1 means the node is part of the minimum spanning tree
- distance: Distance of the node in question from the starting node in the minimum spanning tree
- parent: Parent node of the node in question

The Graph class has a constructor that takes in an integer as the vCount of the Graph object, as well as the following class methods:

- addEdge(): Adds an edge and its weight (int cost) which connect two nodes (int i, int j)
- removeEdge(): Removes the edge which connects two nodes (int i, int j)
- isEdge(): Checks if there is an edge that connects two nodes (int i, int j)
- display(): Prints the adjacency matrix of the graph
- initState(): Initializes the graph object before performing the Dijkstra algorithm
- showBasicInfo(): Prints all the nodes of the graph and the attributes of each node, such as its key, distance, and parent
- isAllKeyTrue(): Checks if all nodes possess a key of 1
- findMinDisNode(): Finds the node with the smallest distance from the MST

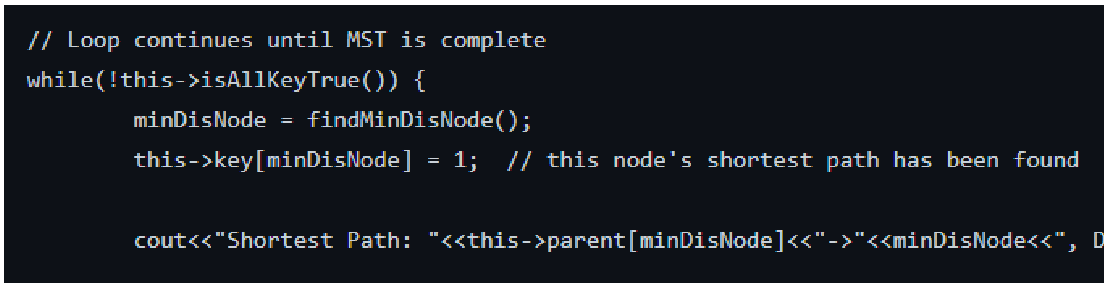
Last but not least, there is also the implementation of the algorithm itself, Dijkstra()



```
// executes Dijkstra's algorithm with a starting node
void Graph::Dijkstra(int startNode) {
        cout<<"\nDijkstra Shortest Path starts . . . \n";

        // initialization is done before call this method
        this->distance[startNode]=0; //start node's distance is 0
        int minDisNode, i;
```

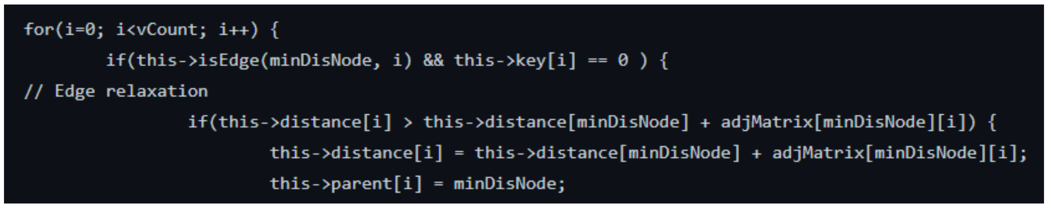
Before the algorithm itself can operate, there is a preparatory step that initializes the distance of the starting node as 0 and creates the dummy variable minDisNode.



```
// Loop continues until MST is complete
while(!this->isAllKeyTrue()) {
        minDisNode = findMinDisNode();
        this->key[minDisNode] = 1;  // this node's shortest path has been found

        cout<<"Shortest Path: "<<this->parent[minDisNode]<<"->"<<minDisNode<<", D
```

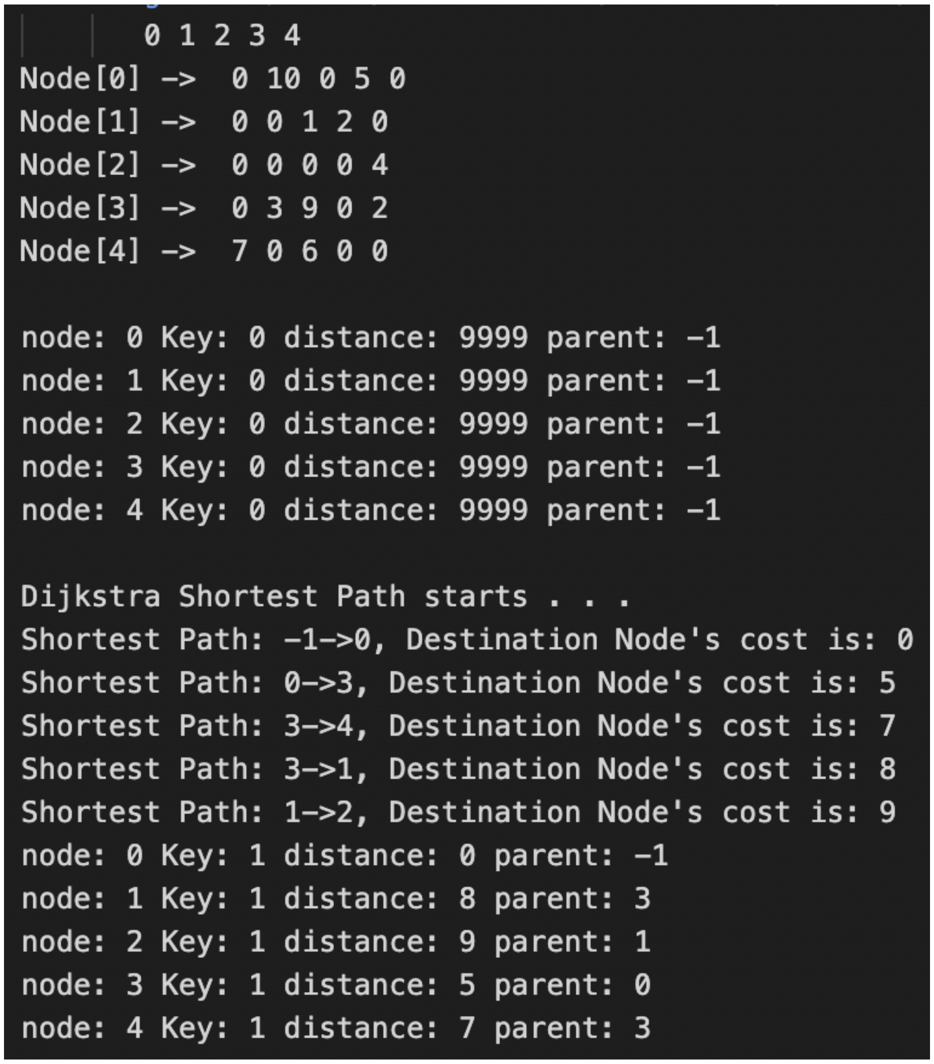
The next step is to find the minimum distance node by calling on the function findMinDisNode(). Once the minimum distance node has been found, the key of the node will be set to 1. So far, the procedure is rather standard. However, the only thing missing now is the edge relaxation component of the algorithm.



```
for(i=0; i<vCount; i++) {
        if(this->isEdge(minDisNode, i) && this->key[i] == 0 ) {
// Edge relaxation
                if(this->distance[i] > this->distance[minDisNode] + adjMatrix[minDisNode][i]) {
                        this->distance[i] = this->distance[minDisNode] + adjMatrix[minDisNode][i];
                        this->parent[i] = minDisNode;
```

The for loop begins by checking if there exists an edge between the minimum distance node and another node and also checking if the node has not been included in the MST. Once both conditions are true, the edge relaxation can be performed.

- Example of Working Program



```
        0 1 2 3 4
Node[0] ->  0 10 0 5 0
Node[1] ->  0 0 1 2 0
Node[2] ->  0 0 0 0 4
Node[3] ->  0 3 9 0 2
Node[4] ->  7 0 6 0 0

node: 0 Key: 0 distance: 9999 parent: -1
node: 1 Key: 0 distance: 9999 parent: -1
node: 2 Key: 0 distance: 9999 parent: -1
node: 3 Key: 0 distance: 9999 parent: -1
node: 4 Key: 0 distance: 9999 parent: -1

Dijkstra Shortest Path starts . . .
Shortest Path: -1->0, Destination Node's cost is: 0
Shortest Path: 0->3, Destination Node's cost is: 5
Shortest Path: 3->4, Destination Node's cost is: 7
Shortest Path: 3->1, Destination Node's cost is: 8
Shortest Path: 1->2, Destination Node's cost is: 9
node: 0 Key: 1 distance: 0 parent: -1
node: 1 Key: 1 distance: 8 parent: 3
node: 2 Key: 1 distance: 9 parent: 1
node: 3 Key: 1 distance: 5 parent: 0
node: 4 Key: 1 distance: 7 parent: 3
```