## Implementation of Dijkstra Algorithm through Graph Data Structure

set<pair<int, string>> cities;

* pair<int, string>: It is a standard template class in C++ that allows you to store a pair of values of different types. In this case, the pair consists of an int and a string.
* Each pair in the set represents a city and its associated distance.
* This allows you to efficiently retrieve the city with the smallest distance from the set, as the set automatically maintains the elements in the desired order.
* In C++, the set container automatically sorts its elements in ascending order using a comparison operator.

map<string, int> distances;

* While you could potentially use a set<pair<string, int>> to store city-distance pairs, it would not provide efficient direct access to the distances based on city names.
* Therefore, in this code, a map<string, int> is used to store the distances associated with cities because it provides efficient retrieval and key-value storage based on city names.

for (auto vertex : vertices) {

string city\_name = vertex.GetcityName();

distances[city\_name] = INT\_MAX;

previous[city\_name] = "";

}

* Initialize distances and previous: The code initializes all distances to infinity (INT\_MAX) and all previous vertices to an empty string. This is done using a loop that iterates over the vertices data structure.

while (!cities.empty()) {

// Get the city with the smallest distance from the set

string currentCity = cities.begin()->second; //

* Taking in the name of first city when the loop starts that is the second element of first Pair of set and storing it in the currentCity string.

cities.erase(cities.begin());

* After taking in the name of the city by string currentCity variable, remove it from the set to ensure no reprocessing.

// If we have reached the end city, break out of the loop

if (currentCity == endCity) {

break;

}

* If the first city is the destination then break the loop

// Process all edges of the current vertex

Vertex currentVertex = getVertex(currentCity);

list<Edge> edgeList = currentVertex.getEdgeList();

* Accessing the linked list of the city

for (auto edge : edgeList) {

string destination = edge.Getdest();

int weight = edge.Getweight();

totalDistance = distances[currentCity] + weight;

// If the total distance is less than the current distance to the destination, update the distance and previous

if (totalDistance < distances[destination]) {

cities.erase({distances[destination], destination});

distances[destination] = totalDistance;

previous[destination] = currentCity;

cities.insert({totalDistance, destination});

}

}

}

A is connected to B with a weight of 100.

A is connected to C with a weight of 5.

B is connected to D with a weight of 60.

C is connected to D with a weight of 30.

Let's dry run the code step by step:

Initialization:

distances map: {A: 0, B: INT\_MAX, C: INT\_MAX, D: INT\_MAX}

previous map: {A: "", B: "", C: "", D: ""}

cities set: {(0, A)}

Iteration 1:

Current city: A

Explore edges of A:

Edge: A -> B (weight: 100)

Total distance to B: 0 + 100 = 100

Since 100 < INT\_MAX, update:

distances map: {A: 0, B: 100, C: INT\_MAX, D: INT\_MAX}

previous map: {A: "", B: A, C: "", D: ""}

cities set: {(100, B)}

Edge: A -> C (weight: 5)

Total distance to C: 0 + 5 = 5

Since 5 < INT\_MAX, update:

distances map: {A: 0, B: 100, C: 5, D: INT\_MAX}

previous map: {A: "", B: A, C: A, D: ""}

cities set: {(5, C), (100, B)}

Iteration 2:

Current city: C

Explore edges of C:

Edge: C -> D (weight: 30)

Total distance to D: 5 + 30 = 35

Since 35 < INT\_MAX, update:

distances map: {A: 0, B: 100, C: 5, D: 35}

previous map: {A: "", B: A, C: A, D: C}

cities set: {(35, D), (100, B)}

Iteration 3:

Current city: B

Explore edges of B:

Edge: B -> D (weight: 60)

Total distance to D: 100 + 60 = 160

Since 160 > 35, no update is made.

All cities have been visited, and the algorithm terminates.

After the algorithm finishes, the resulting maps and sets will be:

distances map: {A: 0, B: 100, C: 5, D: 35}

previous map: {A: "", B: A, C: A, D: C}

cities set: {(35, D), (100, B)}

To find the shortest path from A to D, we can use the previous map:

Starting from D and following the previous cities: D -> C -> A

Reversing the path gives the shortest path from A to D: A -> C -> D

Therefore, the shortest path from A to D is A -> C -> D with a total distance of 35