**REBECCA ALINDA S23B23/056**

**OGWANG ANDREW S23B23/050**

**OPIFUDRIRA TIMOTHY S23B23/086**

**KATENDE DERRICK S23B23/024**

**ANGUYI NEWTON S23B23/071**

**AMAFEKU DAVID ZOLE S23B23/004**

**DIJKSTRA’S ALGORITHM**

Dijkstra’s algorithm is used to find the shortest path from the source to all vertices in the graph. It uses labels that are positive integers or real numbers which are totally ordered.

The idea is to generate a shortest path tree (SPT) with a given source as a root. We can use an Adjacency Matrix with two sets; one containing the vertices included in the shortest path-tree and the other containing vertices that are not included.

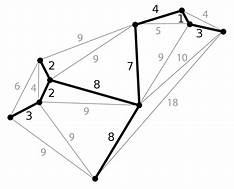
* Create a set **sptSet** (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
* Assign a distance value to all vertices in the input graph. Initialize all distance values as **INFINITE**. Assign the distance value as 0 for the source vertex so that it is picked first.
* While **sptSet** doesn’t include all vertices
  + Pick a vertex **u** that is not there in **sptSet**and has a minimum distance value.
  + Include u to **sptSet**.
  + Then update the distance value of all adjacent vertices of **u**.
    - To update the distance values, iterate through all adjacent vertices.
    - For every adjacent vertex **v,** if the sum of the distance value of **u** (from source) and weight of edge **u-v**, is less than the distance value of **v**, then update the distance value of **v**.

The sptSet[] represents a set of vertices that is included in SPT. If a value is true then it is included in the set, else it is not.

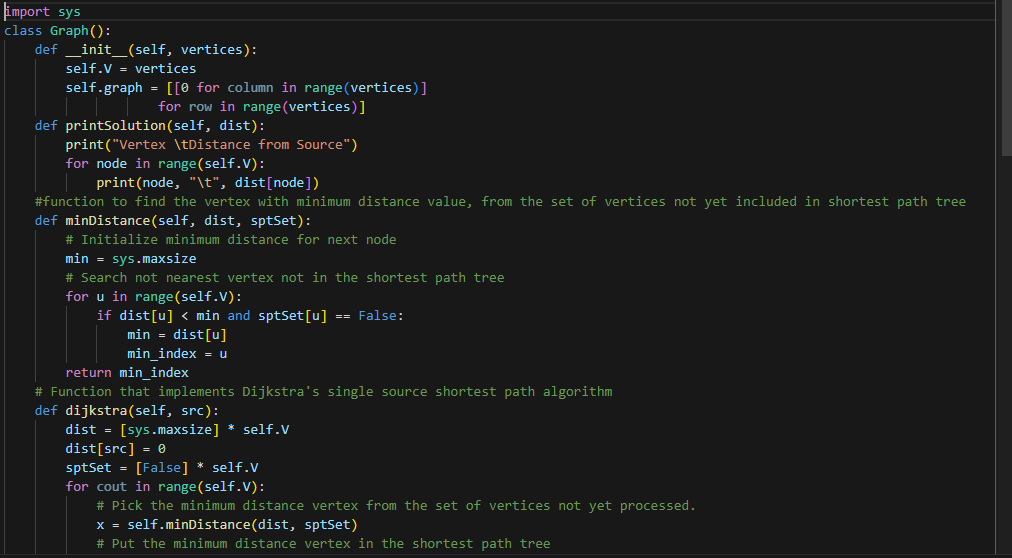
Dijkstra's algorithm uses a data structure for storing and querying partial solutions sorted by distance from the start. It does not use a min-priority queue. Its Pseudocode could be as follows;

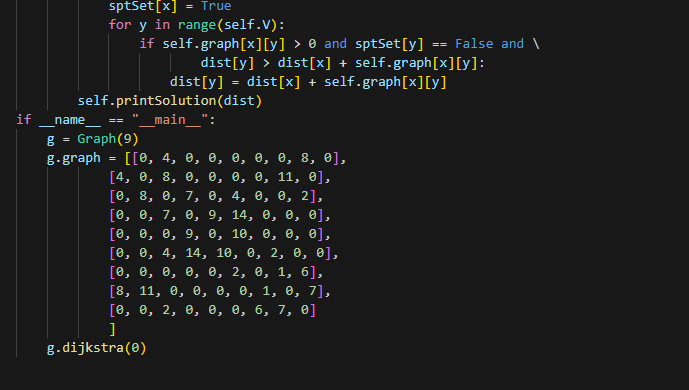
1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
2. Assign to every node a *distance from start* value: set it to zero for our starting node and to infinity for all other nodes. During the run of the algorithm, the distance of a node *N* is the length of the shortest path discovered so far between the node *N* and the *starting* node. Since initially no path is known to any other node than the starting node (which is a path of length zero), all other distances are initially set to infinity. Set the starting node as current.
3. For the current node, consider all of its unvisited neighbours and update their distances through the current node: Compare the newly calculated distance to the one currently assigned to the neighbour and assign it the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with its neighbour *B* has length 2, then the distance to *B* through *A* is 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8 (the path to B through A is shorter). Otherwise, keep its current distance (the path to B through A is not the shortest).
4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again. At this point, this visited node recorded distance is final and minimal because this node was selected to be the next to visit due to having the smallest distance from the *starting node*, cf. next step.
5. Review the unvisited nodes and select the one with the smallest known distance as the new "current node" and go back to step 3. If an unvisited node has an "infinity" distance, it means that it's not reachable (so far) and should not be selected. If there are no more reachable unvisited nodes, the algorithm has finished. If the new "current node" is the target node, it means we found a possible path to it, but not necessarily the shortest (we can't know before visiting all nodes). Depending on the application, it's possible to early exit here if the shortest path is not required or if we have ways to know it's the shortest path.
6. Once the loop exited, the shortest path can be extracted from the set of visited nodes by starting from the target node and picking its neighbour with the shortest distance, going back to start on an optimal path. If the target node recorded distance is infinite, it means no path exist.

AN ILLUSTRATION OF DIJKSTRA’S ALGORITHM

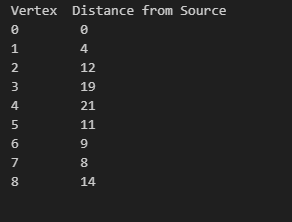


An example of use of djikstra’s algorithm;





The above code gives the following output;



**SPANNING TREE ALGORITHM**

A spanning tree is a subset of a graph such that all the vertices are connected using minimum possible number of edges. Thus a spanning tree does not have cycles and a graph can have more than one spanning tree.

* A Spanning tree does not exist for a disconnected graph.
* For a connected graph having **N** vertices then the number of edges in the spanning tree for that graph will be**N-1**.
* A Spanning tree does not have any cycle.
* We can construct a spanning tree for a complete graph by removing **E-N+1** edges, where **E** is the number of Edges and **N** is the number of vertices.
* **Cayley’s Formula:** It states that the number of spanning trees in a complete graph with N vertices is Nn-2.

Spanning tree algorithm can be applied in Dijkstra’s algorithm and A\* search algorithm.

It can be used in building a telecommunication network.

It can be used in image segmentation to break an image into distinguishable components.

It is used as a computer network routing protocol.

**The Minimum Spanning Tree Problem**

The Minimum Spanning Tree (MST) problem is about finding a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

There are several algorithms to find the MST, such as **Kruskal’s** and **Prim’s** algorithms. Both are greedy algorithms that build the spanning tree incrementally. The difference lies in the selection of the next edge; Kruskal’s algorithm chooses the next edge of least possible weight that doesn’t form a cycle, while Prim’s algorithm expands the tree from a starting vertex by attaching the next cheapest edge.

let’s consider a simple, connected, undirected graph with weighted edges. The goal is to connect all the vertices with the least total edge weight without forming any cycles.

Here’s a step-by-step illustration using Kruskal’s algorithm:

1. **Sort the Edges**: Arrange all edges in ascending order of their weights.
2. **Select the Edge**: Choose the edge with the smallest weight that doesn’t form a cycle when added to the growing spanning tree.
3. **Build the MST**: Keep adding edges following the above rule until all vertices are connected.

Let’s take a graph with 4 vertices and 5 edges:

Vertices: A, B, C, D Edges: AB(1), BC(2), CD(3), DA(4), AC(5)

**Step 1**: Sort the edges by weight: AB(1), BC(2), CD(3), DA(4), AC(5).

**Step 2**: Start with the smallest edge AB(1) and add it to the MST.

**Step 3**: Next, add edge BC(2) to the MST.

**Step 4**: Then, add edge CD(3). Now, all vertices are connected.

The edges DA(4) and AC(5) are not included as they would create cycles.

The resulting MST is a subgraph with edges AB, BC, and CD, connecting all vertices with a total weight of 1 + 2 + 3 = 6, which is the minimum possible.