Dijkstra

Dijkstra’s algorithm is a popular shortest-path algorithm used to find the minimum distance between a starting node and all other nodes in a weighted graph with non-negative edge weights. It works by gradually exploring the graph, always choosing the node with the smallest known distance and updating the distances of its neighboring nodes. The algorithm uses a priority queue to efficiently select the next closest node to process. As it continues, Dijkstra’s algorithm builds the shortest-path tree, ensuring that once the shortest distance to a node is finalized, it never changes. It is widely used in network routing, navigation systems, and various optimization problems because of its efficiency and accuracy in computing shortest paths.

Advantages / Benefits

- Efficiently finds the shortest path from a single source to all vertices.
- Guarantees optimal paths in graphs with non-negative edge weights.
- Widely used in network routing, GPS navigation, and mapping applications.
- Can be efficiently implemented using a priority queue or min-heap for large graphs.
- Works for both dense and sparse graphs when appropriate data structures are used.
- Once a vertex is processed, its shortest distance is final, simplifying path reconstruction.

Limitations

- Cannot handle negative edge weights, unlike Bellman-Ford.
- Time complexity with a simple array is O(V2), which can be inefficient for large graphs.
- Even with a priority queue, time complexity is O((V + E) log V), which may be slow in some cases.
- Does not detect negative weight cycles, so it is unsuitable for graphs containing them.

• For dynamic graphs where edges change frequently, the algorithm may need to be rerun entirely.

Steps of Dijkstra’s Algorithm

1. **Initialize distances**
    - Set the distance of the **source node to 0**.
    - Set the distance of **all other nodes to infinity (∞)**.
2. **Mark all nodes as unvisited**
   Keep a set or array to track which nodes have been processed.
3. **Use a priority queue (min-heap)**
   This helps pick the node with the **smallest current distance** efficiently.
4. **Insert the source node into the priority queue**
   The priority queue stores pairs like (distance, node).
5. **Repeat until the priority queue becomes empty:**
    - Extract the node with the **smallest distance** (the current node).
    - If that node is already visited, skip it.
    - Mark the current node as visited.
6. **Relax (update) the distances of all neighbors**
   For each neighbor of the current node:
    - Compute a new possible distance:
      newDist = distance[current] + weight(current, neighbor)
    - If newDist < distance[neighbor]:
        - Update distance[neighbor] = newDist
        - Push (newDist, neighbor) into the priority queue.
7. **Continue until all reachable nodes are processed**
   Once a node’s shortest distance is confirmed, it will not change again.

8. **End with shortest distances**
   The distance array now contains the minimum distance from the source to every other node.

Pseudocode:

```
Dijkstra(graph, start):
  create a distance array dist[] and initialize all values to INF
  create a visited array visited[] and initialize all to false

  dist[start] = 0

  priority_queue< pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>> > pq

  pq.push({0, start})

  while pq is not empty:
    (currentDist, node) = pq.top()
    pq.pop()

    if visited[node] == true:
      continue

    visited[node] = true

    for each (neighbor, weight) in graph[node]:
```

```
    if dist[node] + weight < dist[neighbor]:
        dist[neighbor] = dist[node] + weight
        pq.push({dist[neighbor], neighbor})
```

Time Complexity:

Time Complexity: O((V + E) log V)

Where:

V = Number of Vertices/Nodes

E = Number of Edges

Dijkstra Code:

https://github.com/Hazra32/Algorithm-Problem/blob/main/Dijkstra/Basic/basic.cpp