Graphs
Lesson 7

Dijkstra’s shortest path algorithm

Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a *SPT (shortest path tree)* with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, another set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.

Algorithm

1) 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.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While *sptSet* doesn’t include all vertices

….**a)** Pick a vertex u which is not there in *sptSet* and has a minimum distance value.

….**b)** Include u to *sptSet*.

….**c)** Update 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 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.

```
vector <int> dijkstra(int V, vector<vector<int>> adj[], int S)
  {
      vector<vector<int>> graph(V, vector<int>(V, 0));

      for(int i=0;i<V;i++)
      {
         for(int j=0;j<adj[i].size();j++)
         {
            graph[i][adj[i][j][0]]=adj[i][j][1];
         }
      }

      vector<int>distance(V,INT_MAX);
      distance[S]=0;

      queue<pair<int,int>>q;
      q.push({S,0});

      while(!q.empty())
      {
         int x=q.front().first;
         q.pop();

         for(int i=0;i<V;i++)
         {
            if(graph[x][i]!=0 && distance[i] > distance[x] +graph[x][i])
            {
               distance[i]=distance[x]+graph[x][i];
               q.push({i,distance[i]});
            }
         }
      }
      return distance;
  }
```

Bellman–Ford Algorithm

Given a graph and a source vertex *src* in the graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed Dijkstra's algorithm for this problem. *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. But the time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*

Algorithm

1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in a given graph.
…..**a)** Do following for each edge u-v
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]
………………….dist[v] = dist[u] + weight of edge uv
3) This step reports if there is a negative weight cycle in the graph. Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"

The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

How does this work?

Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 time. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

```
int isNegativeWeightCycle(int n, vector<vector<int>>edges)
        {
           int cost[n]={INT_MAX};
           cost[0]=0;
           for(int i=0;i<n-1;i++)
           {
              bool flag=true;
              for(int j=0;j<edges.size();j++)
              {
                 int u=edges[j][0];
                 int v=edges[j][1];
                 int x=edges[j][2];
                 if(cost[u] != INT_MAX && cost[v]>cost[u]+x)
                 {
                    cost[v]=cost[u]+x;
                    flag=false;
                 }
              }
              if(flag==true)
                 break;
           }
           for(int j=0;j<edges.size();j++)
      {
         int u=edges[j][0];
         int v=edges[j][1];
         int x=edges[j][2];

         if(cost[v]>cost[u]+x)
         {
            return 1;
         }
      }
      return 0;
        }
```

Floyd Warshall

The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph. The Graph is represented as an adjacency matrix, and the matrix denotes the weight of the edges (if it exists) else -1. **Do it in-place.**

Input:

```
graph[][] = { {0,   5,  INF, 10},
              {INF,  0,  3,  INF},
              {INF, INF, 0,   1},
              {INF, INF, INF, 0} }
```

```
         10
  (0)------->(3)
   |          /|\
 5 |           |
   |           | 1
  \|/          |
  (1)------->(2)
         3
```

Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

Output:
Shortest distance matrix

```
   0      5      8      9
  INF     0      3      4
  INF    INF     0      1
  INF    INF    INF     0
```

Expected Time Complexity: O(n^3)

Expected Space Complexity: O(1)

Constraints:

1 <= n <= 100

Algorithm

1. We initialize the solution matrix as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex.
2. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
3. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

    1) k is not an intermediate vertex in the shortest path from i to j. We keep the value of dist[i][j] as it is.
    2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

```
void shortest_distance(vector<vector<int>>& matrix)
    {
      int V=matrix.size();
      for(int k=0;k<V;k++)
      {
        for(int i=0;i<V;i++)
        {
          for(int j=0;j<V;j++)
          {
            if(matrix[i][k]==INF || matrix[k][j]==INF )
           continue;
            else if(matrix[i][j]==INF)
           matrix[i][j]=matrix[i][k]+matrix[k][j];
            if(matrix[i][j]>(matrix[i][k]+matrix[k][j]))
            {
              matrix[i][j]=min(matrix[i][j],matrix[i][k]+matrix[k][j]);
            }
          }
        }
      }
}
```

Johnson’s algorithm for All-pairs shortest paths

The problem is to find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative. Johnson’s algorithm uses both Dijkstra and Bellman-Ford as subroutines.

If we apply Dijkstra’s Single Source shortest path algorithm for every vertex, considering every vertex as source, we can find all pair shortest paths in O(V*VLogV) time. So using Dijkstra’s single source shortest path seems to be a better option than Floyd Warshell, but the problem with Dijkstra’s algorithm is, it doesn’t work for negative weight edges.The idea of Johnson’s algorithm is to re-weight all edges and make them all positive, then apply Dijkstra’s algorithm for every vertex.
The idea of Johnson’s algorithm is to assign a weight to every vertex. Let the weight assigned to vertex u be h[u]. We reweight edges using vertex weights. For example, for an edge (u, v) of weight w(u, v), the new weight becomes w(u, v) + h[u] – h[v]. The great thing about this reweighting is, all set of paths between any two vertices are increased by the same amount and all negative weights become non-negative. Consider any path between two vertices s and t, weight of every path is increased by h[s] – h[t], all h[] values of vertices on path from s to t cancel each other.A new vertex is added to the graph and connected to all existing vertices. The shortest distance values from new vertex to all existing vertices are h[] values.

Algorithm:

1) Let the given graph be G. Add a new vertex s to the graph, add edges from new vertex to all vertices of G. Let the modified graph be G'.

2) Run Bellman-Ford algorithm on G' with s as source. Let the distances calculated by Bellman-Ford be h[0], h[1], .. h[V-1]. If we find a negative weight cycle, then return. Note that the negative weight cycle cannot be created by new vertex s as there is no edge to s. All edges are from s.

3) Reweight the edges of original graph. For each edge (u, v), assign the new weight as "original weight + h[u] – h[v]".

4) Remove the added vertex s and run Dijkstra's algorithm for every vertex.
The following property is always true about h[] values as they are shortest distances.
h[v] <= h[u] + w(u, v)

The property simply means, shortest distance from s to v must be smaller than or equal to shortest distance from s to u plus weight of edge (u, v). The new weights are w(u, v) + h[u] - h[v]. The value of the new weights must be greater than or equal to zero because of the inequality "h[v] <= h[u] + w(u, v)".

Time Complexity: The main steps in the algorithm are Bellman Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is O(VE) and time complexity of Dijkstra is O(VLogV). So overall time complexity is O(V2log V + VE).
The time complexity of Johnson's algorithm becomes the same as Floyd Warshell when the graph is complete (For a complete graph E = O(V2). But for sparse graphs, the algorithm performs much better than Floyd Warshell.

Shortest Path in Directed Acyclic Graph

Given a Weighted Directed Acyclic Graph and a source vertex in the graph, find the shortest paths from given source to all other vertices.

For a general weighted graph, we can calculate single source shortest distances in O(VE) time using Bellman–Ford Algorithm. For a graph with no negative weights, we can do better and calculate single source shortest distances in O(E + VLogV) time using Dijkstra's algorithm. Can we do even better for Directed Acyclic Graph (DAG)? We can calculate single source shortest distances in O(V+E) time for DAGs. The idea is to use Topological Sorting.

We initialize distances to all vertices as infinite and distance to source as 0, then we find a topological sorting of the graph. Topological Sorting of a graph represents a linear ordering of the graph. Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent vertex using the distance of the current vertex.

Following is complete algorithm for finding shortest distances.

```
1) Initialize dist[] = {INF, INF, ….} and dist[s] = 0 where s is the source vertex.
2) Create a topological order of all vertices.
3) Do following for every vertex u in topological order.
………..Do following for every adjacent vertex v of u
………………if (dist[v] > dist[u] + weight(u, v))
………………………dist[v] = dist[u] + weight(u, v)
```

Time Complexity: Time complexity of topological sorting is O(V+E). After finding topological order, the algorithm processes all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is O(E). So the inner loop runs O(V+E) times. Therefore, the overall time complexity of this algorithm is O(V+E).