Algorithms

Rank: **957**

View Leaderboard

Topics: Shortest Path Algorithms

Shortest Path Algorithms

TUTORIAL PROBLEMS

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

This problem could be solved easily using **(BFS)** if all edge weights were (1), but here weights can take any value. Three different algorithms are discussed below depending on the use-case.

Bellman Ford's Algorithm:

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most $n - 1$ edges, because the shortest path couldn't have a cycle.

So why shortest path shouldn't have a cycle ?
There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

Algorithm Steps:

- The outer loop traverses from $0 : n - 1$.
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, the

?

update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Implementation:

Assume the source node has a number (0):

```
 vector <int> v [2000 + 10];
 int dis [1000 + 10];

 for(int i = 0; i < m + 2; i++){

      v[i].clear();
      dis[i] = 2e9;
 }

for(int i = 0; i < m; i++){

      scanf("%d%d%d", &from , &next , &weight);

      v[i].push_back(from);
      v[i].push_back(next);
      v[i].push_back(weight);
}

 dis[0] = 0;
 for(int i = 0; i < n - 1; i++){
      int j = 0;
      while(v[j].size() != 0){

           if(dis[ v[j][0]  ] + v[j][2] < dis[ v[j][1] ] ){
                dis[ v[j][1] ] = dis[ v[j][0]  ] + v[j][2];
           }
           j++;
      }
 }
```

A very important application of Bellman Ford is to check if there is a negative cycle in the graph,

Time Complexity of Bellman Ford algorithm is relatively high $O(V \cdot E)$, in case $E = V^2$, $O(E^3)$. Let's discuss an optimized algorithm.

Dijkstra's Algorithm

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Implementation:

Assume the source vertex = 1.

```
#define SIZE 100000 + 1

vector < pair < int , int > > v [SIZE];   // each vertex has all the connected
vertices with the edges weights
int dist [SIZE];
bool vis [SIZE];

void dijkstra(){
                                                  // set the vertices distances
as infinity
    memset(vis, false , sizeof vis);             // set all vertex as unvisited
    dist[1] = 0;
    multiset < pair < int , int > > s;           // multiset do the job as a min-
priority queue

    s.insert({0 , 1});                            // insert the source node with
distance = 0

    while(!s.empty()){

        pair <int , int> p = *s.begin();         // pop the vertex with the
minimum distance
        s.erase(s.begin());

        int x = p.s; int wei = p.f;
```

```
        if( vis[x] ) continue;                    // check if the popped vertex is
visited before
         vis[x] = true;

        for(int i = 0; i < v[x].size(); i++){
            int e = v[x][i].f; int w = v[x][i].s;
            if(dist[x] + w < dist[e]  ){             // check if the next vertex
distance could be minimized
                dist[e] = dist[x] + w;
                s.insert({dist[e],  e} );            // insert the next vertex
with the updated distance
            }
        }
    }
}
```

Time Complexity of Dijkstra's Algorithm is $O(V^2)$ but with min-priority queue it drops down to $O(V + E\ log\ V)$.

However, if we have to find the shortest path between all pairs of vertices, both of the above methods would be expensive in terms of time. Discussed below is another alogorithm designed for this case.

Floyd–Warshall's Algorithm

Floyd–Warshall's Algorithm is used to find the shortest paths between between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative. The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in $O(V^3)$, where V is the number of vertices in a graph.

The Algorithm Steps:

For a graph with N vertices:

- Initialize the shortest paths between any 2 vertices with Infinity.
- Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all N vertices as intermediate nodes.
- Minimize the shortest paths between any 2 pairs in the previous operation.
- For any 2 vertices (i, j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: $min(dist[i][k] + dist[k][j], dist[i][j])$.

$dist[i][k]$ represents the shortest path that only uses the first K vertices, $dist[k][j]$ represents the shortest path between the pair k, j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

```
for(int k = 1; k <= n; k++){
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );
        }
    }
}
```


Time Complexity of Floyd–Warshall's Algorithm is $O(V^3)$, where V is the number of vertices in a graph.

Contributed by: omar khaled abdelaziz abdelnabi