SHORTEST PATH ALGORITHMS

BY MNNIT COMPUTER CODING CLUB

SINGLE SOURCE SHORTEST PATH ALGORITHMS

Dijkstra Algorithm

Introduction

Dijkstra’s Algorithm is a generalisation of BFS for weighted graphs. In other words what BFS is to unweighted graphs, Dijkstra is to weighted graph.

It is a greedy algorithm that uses a priority_queue (or a heap) instead of a regular queue to relax the distances of vertices. The upcoming slides discuss the working and the proof of the algorithm.

Algorithm

Let us take an array **d**, where d[s] = the minimum distance of node s from the source. Also maintain a boolean array **visited**.

Before we begin the algorithm assign d[x] = oo if x != source else d[source] = 0. We add the source vertex to our priority queue claiming that as of now we can only relax the distances based on the edges coming out of the source.

Now the algorithm starts and continues till there is some node remaining in the priority queue. We pop off the node with the minimum distance currently present in the priority queue and if we haven't already used it to relax the distances we use it and mark it visited, otherwise we just continue.

After a node **v** is taken from the priority queue, we relax the distances using the edges coming out of that node **v** as follows:

d[to]=min(d[to],d[v]+len)

If **d[to]** is indeed relaxed from the edge(v , to) then we add vertex '**to**' to our priority queue as a potential candidate to relax distances.

This algorithm runs for N iterations, wherein in each iteration one of the nodes is selected and the edges are relaxed accordingly

Note that when a vertex is being used for relaxing other distances it is assumed that the shortest path length was that vertex is already reached.

For implementation details refer to Codes directory.

Restoring the shortest path

For this we'll use the same trick we learnt in DP class 3 under the topic "Tracing the Actual Answer Back".

Just like we did for DP problems, we'll maintain another array pre[N+1] which stores the penultimate node in the shortest path from source to current node. In other words it pre[to] stores the node v such that last relaxation that was done on d[to] was through the edge (v , to).

Now we can trace back the path for any vertex using this pre array. For more details refer to the content of DP class 3.

```
vector<int> restore_path(int s, int t, vector<int> const& pre) {
    vector<int> path;

    for (int v = t; v != s; v = pre[v])
        path.push_back(v);
    path.push_back(s);

    reverse(path.begin(), path.end());
    return path;
}
```

Proof

Correctness of Dijkstra's algorithm can be proved by induction.

Assume that till now every node that has been marked has got its shortest path. Let's say for the current iteration the algorithm chose node v, i.e., the current d[v] is least among all the nodes in the priority queue. Now let's say the shortest path from source to v can be divided into two parts. P1 consists of marked nodes ending at q and P2 consists of unmarked nodes starting at p (which is currently in the queue). Now since for the last iteration we relaxed p using (q ,p) hence d[p] is the shortest path till p. So d[p] <= d[v]. But the algorithm chose v over p so d[v] <= d[p]. So d[v] = d[p] and thus the current marked node has also reached its shortest path.

Complexity Analysis

If we use a priority queue then the algorithm runs in **O((N+M) * log(N))**. This is because there are at max O(N) iterations for the algorithm wherein the edges are relaxed using the edges from the current node being used. Also all the edges of the graph are used hence the +M part.

Notice that though this complexity is good enough for sparse graphs, we can do better for dense graphs simply by avoiding priority queue altogether. For sparse graphs we can achieve a complexity of **O(N^2 + M)** which is better if M is roughly equal to N^2.

Problem

https://codeforces.com/contest/20/problem/C

Problem

E - Rush Hour 2

Drawbacks

Dijkstra Algorithm cannot be used for graphs with negative edge weights. Since then the greedy selection of paths won’t work.

Bellman Ford Algorithm

Solution to negative weight Problem

As we saw in the case of Dijkstra's Algorithm, the negative weights lead to the failure of a greedy algorithm, so instead we have to resort to using a Dynamic Programming solution.

That DP solution is Bellman Ford Algorithm. It is an algorithm that works fine in case of negative edges as well. Also if the graph contains any negative cycles it is able to detect them as well.

Also note that Bellman Ford algorithm works only for directed graphs. (It can work for undirected graphs only when all the edges have positive weights, but for that scenario we have much faster Dijkstra's).

Algorithm

Let’s create an array d, where d[v] represents the minimum distance of node v from the source. Initially d[source] = 0 and for rest v d[v] = oo.

The algorithm runs in various phases. In each phase we iterate over ALL the edges and try to relax the the node at the receiving end of the edge.

In at most (n-1) phases we’ll get the required minimum distances.

Proof of the Algorithm

Let us now prove the following assertion: After the execution of **ith** phase, the Bellman-Ford algorithm correctly finds all shortest paths whose number of edges does not exceed **i**.

In other words, for any vertex **a** let us denote the **k** number of edges in the shortest path to it (if there are several such paths, you can take any). According to this statement, the algorithm guarantees that after **kth** phase the shortest path for vertex **a** will be found.

Proof: Consider an arbitrary vertex **a** to which there is a path from the starting vertex **v**, and consider a shortest path to it (p0=v,p1,…,pk=a). Before the first phase, the shortest path to the vertex p0=v was found correctly. During the first phase, the edge (p0,p1) has been checked by the algorithm, and therefore, the distance to the vertex p1 was correctly calculated after the first phase. Repeating this statement **k** times, we see that after kth phase the distance to the vertex pk=a gets calculated correctly, which we wanted to prove.

The last thing to notice is that any shortest path cannot have more than n−1 edges. Therefore, the algorithm sufficiently goes up to the (n−1)th phase. After that, it is guaranteed that no relaxation will improve the distance to some vertex.

Implementation details

For implementation of Bellman Ford it is convenient to store the graph as a vector of edges. The actual code can be found in the Codes directory.

Negative Cycle Detection

After (n−1)th phase, if we run algorithm for one more phase, and it performs at least one more relaxation, then the graph contains a negative weight cycle that is reachable from v, otherwise, such a cycle does not exist.

This is because we have already proven that at most (n-1) phases are required to find the required distances.

Implementation details for Negative Cycle Detection

For this, it is sufficient to remember the last vertex **x** for which there was a relaxation in nth phase. This vertex will either lie in a negative weight cycle, or is reachable from it. To get the vertices that are guaranteed to lie in a negative cycle, starting from the vertex **x**, pass through to the predecessors **n** times. Hence we will get the vertex **y**, namely the vertex in the cycle earliest reachable from source. We have to go from this vertex, through the predecessors, until we get back to the same vertex **y** (and it will happen, because relaxation in a negative weight cycle occur in a circular manner).

Complexity Analysis

Time complexity : O(N * M) because we have to iterate through at max (N-1) phases and in each phase we consider all the M edges.

SPFA is an optimisation for Bellman Ford but still its asymptotic worst case complexity is O(N*M). For further reading refer here.

All source shortest path algorithm.

Floyd Warshall Algorithm

Introduction

Till now we have only seen the algorithms that find the minimum distance from a FIXED source to all other nodes. But Floyd Warshall finds d[i][j] -> the minimum distance between the nodes i and j for all pairs of i,j.

It works for both directed as well as undirected graphs.

Algorithm

This algorithm also attempts to relax the distance just like the other two. However, unlike the other two algorithms it doesn't use edges rather complete nodes to relax the distance.

We start off by creating a 2D array d , where d[i][j] = a[i][j] if a[i][j] > 0 else d[i][j] = oo.

Before k-th phase (k=1…n), d[i][j] for any vertices i and j stores the length of the shortest path between the vertex i and vertex j, which contains only the vertices 1,2,...,k−1 as internal vertices in the path.

Then we try to relax d[i][j] = min(d[i][j] , d[i][k] + d[k][j]) for each k from 1 to n.

Proof

Suppose now that we are in the k-th phase, and we want to compute the matrix d[][] so that it meets the requirements for the (k+1)-th phase. We have to fix the distances for some vertices pairs (i,j). There are two fundamentally different cases:

The shortest way from the vertex i to the vertex j with internal vertices from the set 1,2,…,k coincides with the shortest path with internal vertices from the set 1,2,…,k−1.

In this case, d[i][j] will not change during the transition.

The shortest path with internal vertices from 1,2,…,k is shorter.

This means that the new, shorter path passes through the vertex k. This means that we can split the shortest path between i and j into two paths: the path between i and k, and the path between k and j. It is clear that both this paths only use internal vertices of 1,2,…,k−1 and are the shortest such paths in that respect. Therefore we already have computed the lengths of those paths before, and we can compute the length of the shortest path between i and j as d[i][k]+d[k][j].

Combining these two cases we find that we can recalculate the length of all pairs (i,j) in the k-th phase in the following way:

dnew[i][j]=min(d[i][j],d[i][k]+d[k][j])

Retrieving the shortest path

For doing so we can store additional information in the form of another 2D array p, where p[i][j] stores the last phase number in which d[i][j] was relaxed on. Using p the shortest path can be generated recursively as follows:

path(i , j) = path(i , k) + (remove extra k) + path(k , j);

Complexity

Time Complexity : O(N^3)

If the graph doesn’t contain negative edges then obviously using dijkstra for all sources individually would be better since the complexity in that case would be O(N * (N + M) logN) which is better for the case of sparse graphs. However for dense graphs and/or negative weights Floyd Warshall is a better choice.

Problem

Shortest Routes II

Problem

Problem - 295B : A wonderful problem for Floyd Warshall.