https://codeforces.com/blog/entry/22276

Link To PDF version (Latex Formatted)

Topic : 0-1 BFS

Pre Requisites : Basics of Graph Theory , BFS , Shortest Path

Problem :

You have a **graph G** with **V vertices** and **E edges**. The graph is a weighted graph but the weights have a contraint that they can only be 0 or 1. Write an efficient code to calculate shortest path from a given source.

Solution :

Naive Solution — Dijkstra's Algorithm.

This has a complexity of O(E + VlogV) in its best implementation. You might try heuristics , but the worst case remains the same. At this point you maybe thinking about how you could optimise Dijkstra or why do I write such an efficient algorithm as the naive solution? Ok , so firstly the efficient solution isn't an optimisation of Dijkstra. Secondly , this is provided as the naive solution because almost everyone would code this up the first time they see such a question , assuming they know Dijkstra's algorithm.

Supposing Dijkstra's algorithm is your best code forward , I would like to present to you a very simple yet elegant trick to solve a question on this type of graph using Breadth First Search (BFS).

Before we dive into the algorithm, a lemma is required to get things crystal clear later on.

Lemma : "During the execution of BFS, the queue holding the vertices only contains elements from at max two successive levels of the BFS tree."

Explanation : The BFS tree is the tree built during the execution of BFS on any graph. This lemma is true since at every point in the execution of BFS , we only traverse to the adjacent vertices of a vertex and thus every vertex in the queue is at max one level away from all other vertices in the queue.

So let's get started with 0-1 BFS.

0-1 BFS :

This is so named , since it works on graphs with edge weights 0 and 1. Let's take a point of execution of BFS when you are at an arbitrary vertex "u" having edges with weight 0 and 1. Similar to Dijkstra , we only put a vertex in the queue if it has been relaxed by a previous vertex (distance is reduced by travelling on this edge) and we also keep the queue sorted by distance from source at every point of time.

Now , when we are at "u" , we know one thing for sure : Travelling an edge (u,v) would make sure that v is either in the same level as u or at the next successive level. This is because the edge weights are 0 and 1. An edge weight of 0 would mean that they lie on the same level , whereas an edge weight of 1 means they lie on the level below. We also know that during BFS our queue holds vertices of two successive levels at max. So, when we are at vertex "u" , our queue contains elements of level L[u] or L[u] + 1. And we also know that for an edge (u,v) , L[v] is either L[u] or L[u] + 1. Thus , if the vertex "v" is relaxed and has the same level , we can push it to the front of our queue and if it has the very next level , we can push it to the end of the queue. This helps us keep the queue sorted by level for the BFS to work properly.

But, using a normal queue data structure , we cannot insert and keep it sorted in O(1). Using priority queue cost us O(logN) to keep it sorted. The problem with the normal queue is the absence of methods which helps us to perform all of these functions :

1. Remove Top Element (To get vertex for BFS)
2. Insert At the beginning (To push a vertex with same level)
3. Insert At the end (To push a vertex on next level)

Fortunately, all of these operations are supported by a double ended queue (or deque in C++ STL). Let's have a look at pseudocode for this trick :

```
for all v in vertices:
        dist[v] = inf
dist[source] = 0;
deque d
d.push_front(source)
while d.empty() == false:
        vertex = get front element and pop as in BFS.
        for all edges e of form (vertex , u):
                if travelling e relaxes distance to u:
                        relax dist[u]
                        if e.weight = 1:
                                d.push_back(u)
                        else:
                                d.push_front(u)
```

As you can see , this is quite similar to BFS + Dijkstra. But the time complexity of this code is O(E + V) , which is linear and more efficient than Dijkstra. The analysis and proof of correctness is also same as that of BFS.

Before moving into solving problems from online judges , try these exercises to make sure you completely understand why and how 0-1 BFS works :

1. Can we apply the same trick if our edge weights can only be 0 and x (x >= 0) ?
2. Can we apply the same trick if our edge weights are x and x+1 (x >= 0) ?
3. Can we apply the same trick if our edge weights are x and y (x,y >= 0) ?

This trick is actually quite a simple trick, but not many people know this. Here are some problems you can try this hack at :

1. http://www.spoj.com/problems/KATHTHI/ — My implementation
2. https://community.topcoder.com/stat?c=problem_statement&pm=10337
3. Problem J of Gym

Div1 — 500 on topcoder are tough to crack. So congrats on being able to solve one of them using such a simple trick :). I will add more problems as I find.