Faster dijkstra on special graphs

Himanshu Jaju

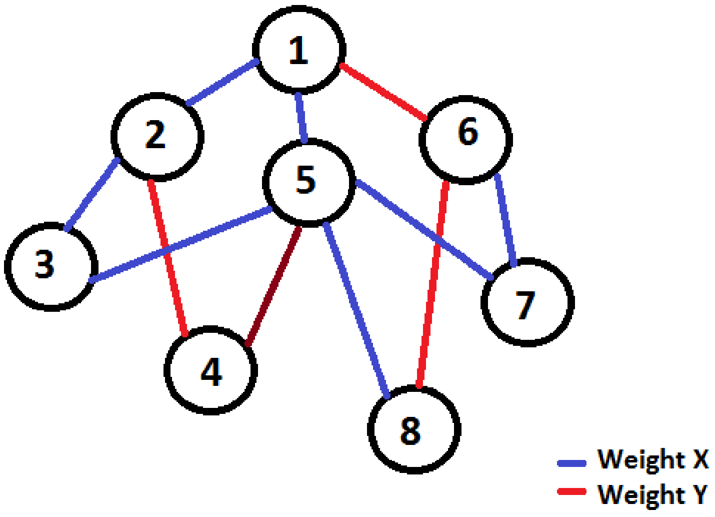
March 2, 2016

Pre Requisites

Dijkstra's algorithm

Motivation Problem

You are given a weighted graph G with V vertices and E edges. Find the shortest path from a given source S to all the vertices, given that all the edges have weight $W \in \{X, Y\}$ where $X, Y >= 0$



Naive Solution

Most people when given this question would solve it using Dijkstra's algorithm which can solve this efficiently in $O(E * logV)$ (Let's not take Fibonacci heap implementation into consideration now). It turns out that this solution, though it will work in most cases well under the time, can be improved further to a linear time algorithm.
Before moving into the next section, try to think about how you could optimise Dijkstra's algorithm. (**Hint :** There are only two possible values of the edge weight.)

Optimised Solution

Let us see why we have the $O(logV)$ factor in the original Dijkstra : **priority queue**! So, if we can just find a way to keep the queue sorted by non-decreasing distance in $O(1)$ always, then we can replace the priority queue with a normal queue and find the shortest distance from source to all vertices.

Let us keep two different queues QX and QY. Suppose we are at an arbitrary node u and we are able to reduce the distance to node v by travelling over an edge from u. If this travelled edge from u to v has a weight X then push v to the queue QX, else push it on the queue QY. In short, QX stores information of all nodes whose current minimum is achieved by last travelling over an X weighted edge and QY stores the information of nodes having last weighted edge as Y.

Our entire algorithm is almost the same as that of Dijkstra. One change is to use two queues as mentioned above instead of the priority queue to try and remove the log factor. Another change is that in case of Dijkstra we always take the top node as the next node, but in this algorithm we take the less distant node of the two queue heads. If we can prove that the queues are sorted, we can prove that the answer produced cannot be different from Dijkstra since we are always taking the minimum distant node for calculations and also keeping the distance queues sorted.

Claim : "The queues QX and QY are always sorted."

Proof : Let us assume that the first inversion has just occurred in QX. Let the last element be V and the second last element be U. So, $dist(U) > dist(V)$. Now, since both of them last travelled an edge with weight X, we can subtract this quantity from both the distances. Thus, $dist(U) - X > dist(V) - X$. Let $pre(a) = dist(a) - X$. Thus, $pre(U) > pre(V)$.
Now, $pre(U)$ and $pre(V)$ must have come from one of the two queues. They cannot come from the same queue, since we assumed the two queues were always sorted before this moment of time. So they must be from two different queues. But according to our algorithm, we take the minimum of the head of the two queues and so if $pre(U)$ was taken before $pre(V)$, then $pre(U) <= pre(V)$. This contradiction proves that QX is always sorted. We can use a similar argument for the other queue.

Once the claim is proved, the whole algorithm should look pretty trivial from implementation point of view. Since we have removed the log factor, this runs in linear time. We can also extend this to a bigger number of distinct edges and use a set for finding minima. You can refer to the pseudocode in the next section in case you have implementation doubts.

Pseudo Code

```
queue $QX$ , $QY$
push source $S$ to $QX$
while one of the two queues in not empty:
    $u$ = pop minimal distant node among the two queue heads
    for all edges $e$ of form $(u, v)$:
        if $dist(v) > dist(u) + cost(e)$:
            $dist(v) = dist(u) + cost(e)$;
            if $cost(e) == X$:
                $QX$.push($dist(v)$,$v$);
            else:
                $QY$.push($dist(v)$,$v$);
```

While finding the minimal distant node, you need to check if anyone of the queue is empty. Otherwise, everything else is same as Dijkstra when it comes to implementation.

Problems to practice

I don't have any specific problems for this, but all problems that can be solved by 0-1 BFS can also be solved by this. Refer here for some questions.

Conclusion

I hope you learnt something new! Feel free to point out mistakes. There is a rare chance that you would need this algorithm anytime, but it might turn out useful in optimisation problems. This post was largely inspired by a comment on my 0-1 BFS tutorial.

Happy Coding!