CS216 Assignment4

Description of the subtree disassembly trick proposed by Tarjan

匡亮 (12111012)

May 1, 2023

Abstract

This is my report of CS216 Assignment4, including a description of the subtree disassembly trick proposed by Tarjan:

1. Its core idea and critical procedures.
2. Running time and space complexity analysis.
3. Its pseudocode.
4. How it speeds up SPFA.

Contents

1 Core idea and critical procedures of Tarjan's trick

For convenience, in the whole report we assume the terminus vertex t is reachable from every vertex in G, and our algorithm will immediately terminate if we find a negative ring (though some vertices cannot reach it, which means they still have a valid non-infinite minimum distance to the terminus vertex). Under these conditions, Tarjan's trick can significantly speed up SPFA without change the algorithm a lot.

Here I don't repeat what we have learned in our courses (like how SPFA works) and begin from SPFA.

In SPFA, we maintain an array $first$ for each vertex, and check whether there is a ring after n iterations to determine whether there is a negative ring. However, if we have found a ring during the iterations, we can immediately terminate the algorithm and report there is a negative ring. Therefore, the basic idea of this trick is try to detect whether there is a ring when modifying $first$ with an acceptable extra cost.

Obviously, before we find a ring, $(v, first[v]), \forall v \neq t$ is a tree with root t. When we set $first[v] \leftarrow w$, we can do DFS in v's subtree and check whether w is in v's subtree, if so, we can terminate our algorithm. However, this DFS will cost an extra $O(n)$ time, so we should expect it to do more for us. We can further observe that before the modification, the minimum distance of all vertices in v's subtree (call them x) is evaluated based on the minimum distance of v, which has been just modified. Therefore, the minimum distance of all of them are out of date. We can set them **dormant**, then they will not be used to update others' minimum distance before their own minimum distance are updated. At the same time we can remove them from the tree (set $first[x] \leftarrow null$), so that they will not be set dormant by other ancestors again. They will add themselves back to the tree automatically when their minimum distance are updated (because they will update $first[x]$ at that time).

2 Running time and space complexity analysis

Based on SPFA, we just store one more tree with $O(1)$ space per vertex, so the space complexity is still $O(n)$.

Add/Remove a vertex to/from the tree can be easily done in $O(1)$ time with linked-list. Adding only happens when $first[v]$ are updated for some v, and removing happens no more than adding because vertices will not be removed twice, so the time complexity is still $O(nm)$.

3 Pseudocode

Algorithm 1: SPFA Improved by Tarjan's Trick

Data: G, t

n = number of nodes in G;
Array $M[V]$;
Initialize $M[t] = 0$ and $M[v] = \infty$ for all other $v \in V$;
Initialize $first[t] = t$ and $first[v] = null$ for all other $v \in V$;
Initialize a tree with only root vertex t;
$foundNegativeRing$ = False;
for $i = 1, ..., n-1$ **do**
    for $w \in V$ *where* w *is not dormant* **do**
        if $M[w]$ *has been updated in the previous iteration* **then**
            for *all edges* (v, w) **do**
                $M[v] = \min(M[v], c_{vw} + W[w])$;
                if *this change the value of* $M[v]$ **then**
                    $first[v] = w$;
                    for *all vertex* x *in* v*'s subtree* **do**
                        if $x == w$ **then**
                            $foundNegativeRing$ = True;
                            End the algorithm;
                        end
                        Set x dormant;
                        Remove x from the tree;
                    end
                    Set v not dormant;
                    Add v to the tree as w's child;
                end
            end
        end
    end
    if *no value changed in this iteration* **then**
        End the algorithm;
    end
    /* If $foundNegativeRing$ is true, then users should ignore the returned $M, first$. */
end

Result: $foundNegativeRing, M, first$

4 How it speeds up SPFA

If there is a negative ring, the algorithm will terminate in advance, so it is speeded up without a doubt.

Even if there is no negative rings, this trick can still speed SPFA up. The picture below shows a classic example:

In this example, the terminus vertex is on the leftmost, and the orange numbers are weights. If we update all vertices from left to right, we will always first update the vertex on the lower left corner of a triangle, and then the vertex on the upper angle, which will

update the vertex on the lower left corner in the next iteration. In short, the vertices on the lower layer have been always misleading in the whole procedure. Actually, in the ath round for $a < \frac{n}{2}$, there are $2a - 1$ vertices being updated in the last iteration. However, if we use Tarjan's trick, there will always be $O(1)$ vertices need to be operated in each iteration. Thouth the misleading still exists, it will not bring extra cost to us because we can always immediately detect it and set the misled vertices dormant.

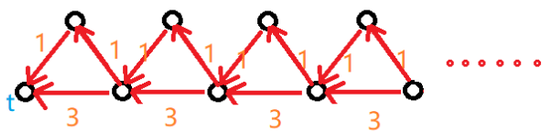

Figure 1: An example forcing SPFA to run for $\Omega(nm)$ time.

We can think in another way. Recall that SPFA calculates the minimum distance from v to t with no more than i edges for all vertices v in the ith iteration. In this example, the ith lower layer vertex from right has i different minimum distances, each of them use different numbers of edges, so its value will be changed for i times during the procedure. The picture below shows an example for $i = 3$. But if we use Tarjan's trick, we will keep focusing on minimizing the final answer and ignore those useless local answer.

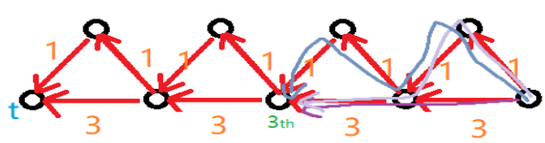

Figure 2: An explanation for why SPFA runs slowly on this example.

References

[Jon Kleinberg / Éva Tardos(2005)] Algorithm Design (P.304 - 307)

[Stefan Lewandowski(2010)] Shortest Paths and Negative Cycle Detection in Graphs with Negative Weights - I: The Bellman-Ford-Moore Algorithm Revisited https://d-nb.info/1014960916/34