CS 224: Advanced Algorithms Spring 2017

Lecture 26 — April 18, 2017

Prof. Jelani Nelson *Scribe: Stefan Gramatovici*

1 Overview

Today we will use link-cut trees to give a $O(mn \log n)$ solution to the max flow with general capacities. In our analysis we will actually show an amortized time complexity $O(m \log^2 n)$ per blocking flow leading to an overall complexity of $O(mn \log^2 n)$ for max flow. Using a slightly improved analysis, the exact same algorithm achieves a complexity of $O(mn \log^2 n)$, but we will not prove this in class.

2 Link Cut Trees

Remember from last time that when using DFS to solve max flow we need to remember paths that were already explored but still have positive capacity. It is easy to see that such paths form a forest.

We will remember such paths using link cut trees, due to Sleator and Tarjan [1]. Link cut trees will maintain a collection of vertex-disjoint rooted trees on n vertices subject to:

1. $fintRoot(v)$: return root of the tree containing v
2. $makeTree(v)$: create a singleton tree consisting of v
3. $link(v, w, x)$(assumes v is root of its tree): Set v parent in the tree to w with capacity x.
4. $cut(v)$: removes edge from v to its parent
5. $findMin(v)$: return edge on root-to-v path of min capacity. Breaks ties by returning the edge closer to root.
6. $subtract(v, x)$: subtract x from the capacity of all edges on v-root path.

3 Blocking Flow

Figure 1 is a pseudocode implementation to find a blocking flow, using Link Cut trees in a black box way.

Algorithm 1 Algorithm for finding a blocking flow. L denotes the level graph with source s and sink t, and V, E are its vertices and edges, respectively. $u(e)$ is the capacity of edge e.

```
1: procedure BlockingFlow(L)
2:     Clone L' to be equal to L
3:     for v ∈ V do
4:         makeTree(v)
5:     end for
6:     while True do
7:         v ← findRoot(s)
8:         if v = t then
               // Add flow along the path from s to t
               // (z, parent(z)) is the min capacity edge of capacity x
9:             (z, x) ← findMin(s)
10:            subtract(s, x)
11:            while True do
                   // remove all edges on this path with capacity 0
                   // if s is a singleton tree, assume y is null
12:                (a, y) ← findMin(s)
13:                if y = 0 then
14:                    cut(a)
15:                    delete edge (a, parent(a)) from L
16:                else
17:                    break
18:                end if
19:            end while
20:        else
               // try to advance
21:            if v has an outgoing edge (v, w) ∈ E then
22:                link(v, w, u((v, w)))
23:            else
24:                if v = s then
                       // we're done: no more possibilities to explore
25:                    break
26:                else
                       // (aggressive) retreat
27:                    for each child c of v do
28:                        cut(c)
29:                        delete edge (c, v) from graph L
30:                    end for
31:                end if
32:            end if
33:        end if
34:    end while
35:    f ← 0                                   ▷ the blocking flow to be returned
36:    for e ∈ E do
37:        f_e ← u_{L'}(e) − u_L(e)
38:    end for
39:    return f
40: end procedure
```

4 Implementation of Link Cut Tree

We focus on one tree.

Definition 1. *For node v, its preferred child is*

- *v itself, if v was most recently accessed node in its subtree*
- *otherwise, it'sthe child leading to the most recently accessed descendant of v*

Definition 2. (u, v) *is a preferred edge if u is the preferred child of v.*

4.1 Preferred path representation

Consider some tree that we are trying to represent (from now on referred to as the **represented tree**). Note that preferred edges form continuous disjoint paths that cover all points of our tree. Collapsing the chains into single point and keeping the incidence of the initial tree, we obtain another tree called the **preferred path decomposition**. This secondary tree will be stored in a link cut tree.

Each preferred path will be sored as an splay tree keyed on depth in the represented tree. A tree storing a preferred path will be called **aux tree**. The root of the aux tree will have a pointer to the vertex in the represented tree that starts the path(has smallest depth in the represented tree).

We now need to implement access(v): make root-to-v path preferred so that the aux tree containing v becomes root(in the tree of aux trees) and v becomes the root vertex of this root tree. We will do the following operations:

4.2 access(v)

- $splay(v)$ (in the aux tree containing v)
- $v.right.pathparent = v$
- $v.right.auxparent = none$
- $v.right = none$ until $v.pathparent = none$:
  - $w = v.pathparent$
  - $splay(w)$
  - $w.right.pathparent = w$
  - $w.right.auxparent = none$
  - $w.right = v$
  - $v.auxparent = w$
  - $splay(v)$

Once we have $access(v)$, the other operations needed for a link cut tree can be implemented as follows:

4.3 findRoot(v)

- $access(v)$
- return the min keyed element of $v's$ aux tree, then access it(this maintains good amortized time due to the properties of splay trees.)

4.4 findMin(v)

Note: each splay tree will be an augmented splay tree with capacity information that allows additional $\log n$ queries regarding the values(not keyes) stored in a tree such as findMinValue.

- $access(v)$
- return the min capacity value in v's tree(every node stores min capacity in its subtree.[1])

4.5 cut(v)

- $access(v)$
- $v.left.pathparent = none$
- $v.left.auxparent = none$
- $v.left = none$

4.6 link(v,w,x)

- $access(v)$
- $access(w)$
- $v.auxleft = w$
- $w.auxparent = v$

5 Analysis

Let m, n be the number of edges and vertices. Running time of blocking flow is $O(m \cdot (\text{ sum of operations }) + n \cdot t_{makeTree})$. Thus we will need to analyze the amortized complexity of all the operations above.

First note that in all implementations above the only operation that we need to analyze with respect to time cost is $access(v)$. The other operations consist of $access(v)$ and some constant time.

Claim 1. *Over the course of T accesses, total runtime is $O((T + PCC) \cdot \log n)$, where PCC is the number of preferred child changes.*

[1] the capacity of a node is the capacity of the edge to its parent

Proof of Claim.. Note that in an access, the most expensive operations are splays. We make $O(PCC)$ splays and n accesses. Based on the properties of splay trees, this leads to a time complexity of $O(\log n(T + PCC))$.[2]

We are left to bound PCC by $O(n + T \log n)$. We will use heavy-light decomposition.

5.1 Heavy-light decomposition

Consider the initial tree(represented tree). Let $s(v)$ be the size of the subtree rooted at v. We will label the edges as follows:

- (u, v) is heavy if $s(u) > \frac{1}{2}s(v)$
- (u, v) is light otherwise

When a preferred child changes from (v, u) to (v, w), we will say that (v, u) is a preferred edge destruction and (v, w) is a preferred child creation.

Observation. Every edge is the represented tree can be of four types:

$$\{\text{light, heavy}\} \times \{\text{preferred, not preferred}\}$$

Lemma. $PCC \leq 2LPCC + n + T$, where $LPCC$ stands for the number of light preferred child creations.

Proof. Suppose (u, v) is a PCC. If (u, v) is light, it counts for both sides of the inequality so there is nothing to prove. If (u, v) is heavy, consider two sub cases

- old pref. child of v was null. This case is somewhat unlikely and it can only happen if the query is the first access in v's subtree (happens at most n times), or if last access in v's subtree was to v. We charge this to the last access on v, giving us the $m + n$ term on the RHS
- old pref. child of v is not null. As we create a light preferred child edge, this means that the previous preferred edge of w was light(as we can only have only light and one heavy edge per vertex). We can charge this to the creation of old preferred light edge, giving us the factor of $2LPCC$ in the bound

We are left to bound LPCC. Let's see how much each operation contributes to $LPCC$. We claim that all operations contribute to LPCC with at most $O(\log n)$. Again, we only consider $access(v)$. For the other operations the proof is follows shortly.

Note that any path there are at most $\log n$ light edges. This is due to the fact that as we go down the path, we leave more than half of the points in trees outside of the path that hang from the heavy edge. Thus any access will create at most $\log n$ LPCC.

Now using the lemma, we obtain $PCC \leq O(T \log n + n)$, giving us a final bound of $O(T \log^2 n + n \log n)$ across T operations. As $T \leq m$, this gives us a bound of $O(m \log^2 n + n \log n)$ for our algorithm.

[2] to get $O(m \log n)$ amortized time per blocking flow we would need to show that total runtime $O(T \log n + PCC)$. This can be achieved by a different weight analysis of the contained splay tree

References

[1] Daniel D. Sleator and Robert Endre Tarjan. *A Data Structure for Dynamic Trees*. Journal of Computer and System Sciences, Vol. 26, No. 3, 1983.