CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 5 — January 31, 2023

Prof. Jelani Nelson *Scribe: Jonathan Pei and Vibhav Athreya*

1 Overview

Today, we are going to finally talk about link-cut trees [1], the data structural problem they solve, how we can use them to implement blocking flow, and analyze their runtime.

The main things we're covering in this lecture are how to implement link-cut trees and analyze them to show that they are fast. There is a way to implement link-cut such that each operation takes worst case $O(\log n)$ time, but today we'll be talking about how to use splay trees to implement these operations in $O(\log^2 n)$ amortized time. You will show how to optimize this to $O(\log n)$ time in a later problem set.

2 Problem that Link-Cut Trees Solves

Let's first discuss the problem that link-cut trees solve. We visualize each iteration of BlockingFlow as a level graph as seen in Figure 1 below:

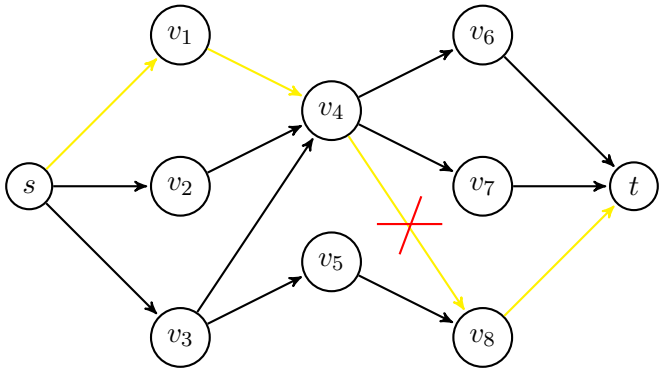

Figure 1: Level Graph with 1 Augmented Path

Recall that in BlockingFlow, we iteratively make a sequence of advances and retreats to augment paths, leaving behind saturated edges. An example of an augmented path is depicted in the above graph, where the advances are indicated in yellow, and the saturated edge is crossed about with a red X. In future augmentations, we may end up revisiting the same vertex many times. Thus, to optimize path augmentations, we want to somehow memoize the explored path fragments (e.g. $\{(s, v_1), (v_1, v_4)\}, \{(v_8, t)\}$). This way, we can decrease the cost of advances by jumping "up" the graph through these explored path fragments.

For example, suppose for our next augmentation, we want to take the path

$$s \to v_1 \to v_4 \to v_7 \to t.$$

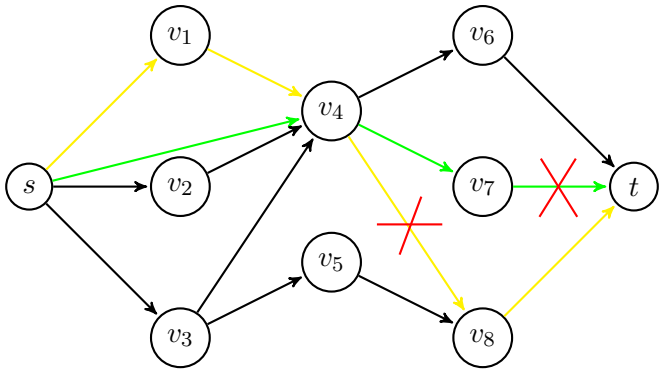

Figure 2: Level Graph with 2 Augmented Paths

In doing so, we would want to be able to jump from s directly to v_4 when advancing, instead of having to traverse through v_1. We indicate this optimized augmenting traversal in green in Figure 2.

In the later parts of this lecture, we discuss how to implement an efficient data structure that memoizes the explored tree fragments and allows us to make these jumps.

3 Blocking Flow

When implementing BlockingFlow, we maintain a rooted forest (using a link-cut tree data structure) subject to the following operations:

- `maketree`(v): make singleton tree just with node v
- `link`(v, w, x): make w the parent of v with residual capacity x on (v, w)
- `cut`(v): remove edge $(v, \text{parent}(v))$ and return the residual capacity
- `findRoot`(v): return root of v's tree
- `findMin`(v): return edge of min capacity from v to its root, breaking ties by returning the one furthest from the root
- `addFlow`(v, x): subtract x from u_e for all e on the path from v to its root

Now that we've defined these operations, we can write out BlockingFlow in Algorithm 1.

Algorithm 1 Perform a single iteration of BlockingFlow

```
procedure BlockingFlow(G = (V, E), s, t)
    for all v ∈ V do
        makeTree(v)
    end for
    while True do
        v ← findRoot(s)
        if v = t then                                              ▷ Augment
            (s, x) ← findMin(s)
            addFlow(s, x)
            while (z, parent(z)) ← findMin(s) has remaining capacity 0 do
                cut(z)
                del((z, parent(z)))
            end while
        else                                                       ▷ Advance?
            if v has outgoing edge (v, w) ∈ E then
                link(v, w, u_(v,w))
            else
                if v = s then                                      ▷ Done
                    break
                else                                               ▷ Mass Retreat
                    for all child w of v do
                        cut(w)
                        del((w, v))
                    end for
                end if
            end if
        end if
    end while
```

4 Implementing Link-Cut Trees Using Splay Trees

Now, we discuss how to actually implement link-cut trees using splay trees for BlockingFlow. We first introduce the idea of accessing a vertex, which we can currently think of as "touching a vertex v." We denote the operation as `access(v)`.

4.1 Preferred Path Decomposition

Definition 4.1 (Preferred Child)**.** Every vertex v will have at most one **preferred child**, defined as follows:

$$\texttt{prefChild}(v) = \begin{cases} \text{null, if } v \text{ was the most recent access in its subtree} \\ \text{child towards most recent access, otherwise} \end{cases}$$

Definition 4.2 (Preferred Edge)**.** A **preferred edge** is defined as any edge that leads to a preferred child.

Definition 4.3 (Preferred Path)**.** A **preferred path** is defined as a maximal contiguous sequence of preferred edges.

Using a preferred path decomposition, we can visualize a tree fragment in the level graph below:

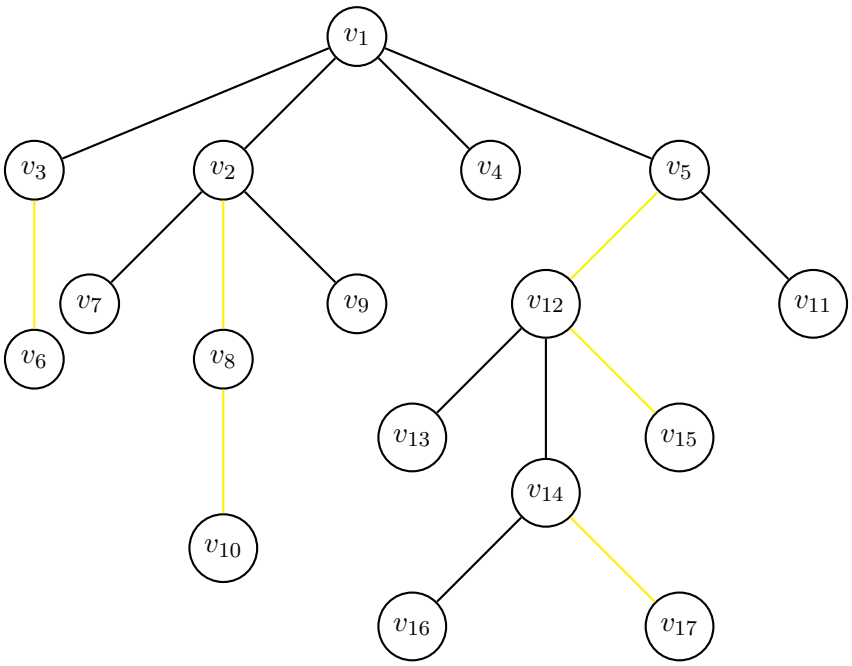

Figure 3: Represented Tree (i.e. Tree Fragment of Level Graph)

In this tree fragment, we indicate 4 preferred paths (in yellow)

$$\{v_3, v_6\},\ \{v_2, v_8, v_{10}\},\ \{v_5, v_{12}, v_{15}\},\ \{v_{14}, v_{17}\}$$

and as part of the link-cut data structure, we store each of these paths as a splay tree. Note that preferred paths consisting of one vertex are stored as singletons.

4.2 Storing Preferred Paths

Each preferred path will be stored as a splay tree with the following specifications:

- We will store preferred path decomposition explicitly.
- The items in the splay tree will be the vertices in the preferred path.
- The key corresponding to each vertex will be its depth, prioritized by increasing depth (i.e. higher up in the tree means less priority)

Note that the tree depicted in Figure 3 depicts just a single tree (containing a bunch of preferred path trees) in a forest of trees represented by the link-cut data structure.

Next, to distinguish between a preferred path tree and its corresponding splay tree, we define the following terms:

Definition 4.4 (Represented Tree)**.** A **represented tree** is defined as an entire tree fragment from the level graph (e.g. (s, v_1, v_4) from Figure 1, or the entire tree in Figure 3).

Definition 4.5 (Auxiliary Tree)**.** An **auxiliary tree** is defined as a splay tree that represents a preferred path within the represented tree. Note that a represented tree corresponds to a tree of auxiliary trees.

Thus, we may represent the represented tree in Figure 3 as a forest of auxiliary trees, as shown in Figure 4 below:

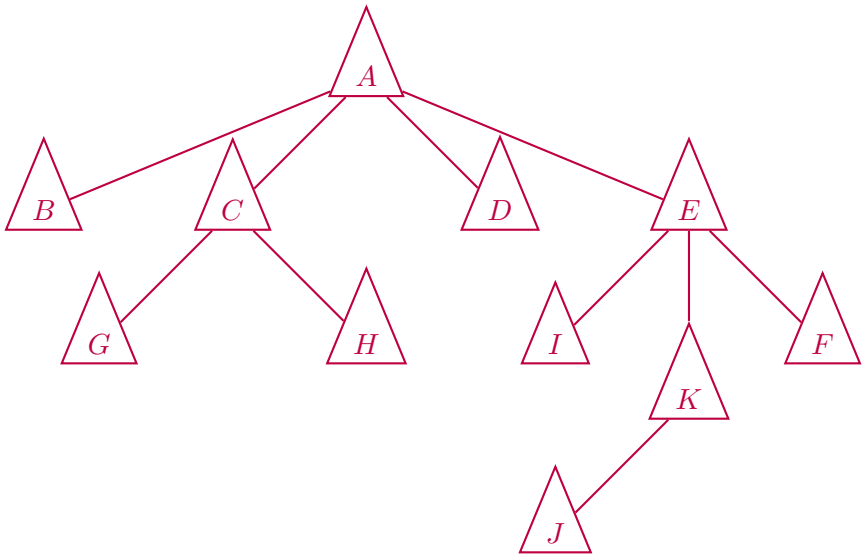

Figure 4: Auxiliary Tree Forest

Here, the preferred paths are

$$\begin{array}{llll} A:\{v_1\}, & B:\{v_3,v_6\}, & C:\{v_2,v_8,v_{10}\}, & D:\{v_4\}, \\ E:\{v_5,v_{12},v_{15}\}, & F:\{v_{11}\}, & G:\{v_7\}, & H:\{v_9\}, \\ I:\{v_{13}\}, & J:\{v_{16}\}, & K:\{v_{14},v_{17}\} & \end{array}$$

Now that we've defined the preferred path decomposition and the auxiliary tree forest representation, we discuss what `access` actually does. Anytime we call `access` on a vertex v, the preferred path from the root node (of the root auxiliary tree) to v will be changed and v will be moved to the root auxiliary tree. Then, after this `access`, we want to splay v to become the root of all the auxiliary trees. This way, we indicate that v is the most recently accessed node.

Note that calling `access` does not change the represented tree.

4.3 Implementing Link and Cut

Before we describe how to implement `access` in detail, we first walk through how to implement `link` and `cut`.

4.3.1 Link

Algorithm 2 Link v and w with edge weight x

procedure LINK(v, w, x)

`access`(v)

`access`(w)

v`.pathparent` $\leftarrow w$

w`.right` $\leftarrow v$

Note that the `.pathparent` pointer point from the root of a given auxiliary tree to some node in its parent auxiliary tree (e.g. $(rv_B \in B) \rightarrow (v_A \in A)$ in Figure 4).

Let's visualize what we're doing here. v is in the root tree of auxiliary trees, and has nothing to its right. w is in a similar situation, higher up in the auxiliary tree forest compared to v.

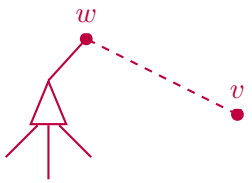

Figure 5: Auxiliary Tree Forest after Link Operation

Then, in the represented trees, w is somewhere in a represented tree, and v is the root of its own represented tree (we only call `link` from a root).

Note that we first call `access` on v and then on w so that w remains in a separate auxiliary tree because it has become the most recently accessed node. Now, the path from w up to its root (indicated in Figure 6) is preferred. Also, since v has no preferred child, it becomes a singleton auxiliary tree.

4.3.2 Cut

Note that the `.left.parent` and `.left` pointers are within an auxiliary tree. Let's visualize what we're doing here. Cutting v basically cuts off everything descendant of v in the represented tree

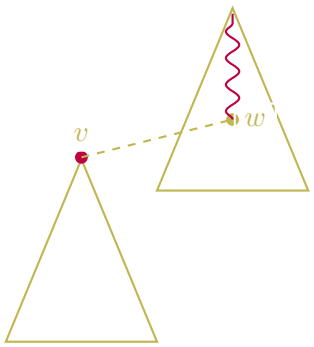

Figure 6: Represented Tree after Link Operation

Algorithm 3 Cut the edge $(v, \text{parent}(v))$

procedure CUT(v)

    `access`access(v)

    v`.left.parent` $\leftarrow$ null

    v`.left` $\leftarrow$ null

(and thus everything on the left of v in its auxiliary tree).

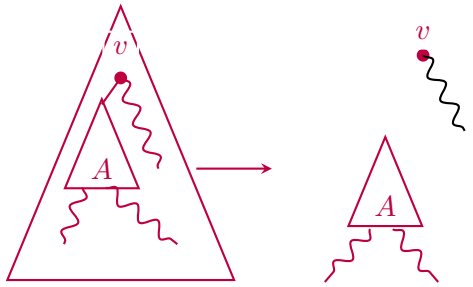

Figure 7: v's Auxiliary Tree Pre-Cut/ Auxiliary Forest Post-Cut

4.4 Implementing Access

Finally, we implement `access(v)`. We start with our forest of auxiliary trees with v somewhere down below the root auxiliary tree. This is depicted in Figure 8 below:

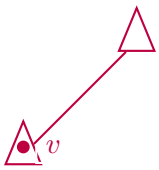

Figure 8: Original tree of aux trees (some aux trees may be omitted)

First, we splay v to put it at the root of its auxiliary tree. Then, since v has no preferred child anymore, everything on the right of v in its auxiliary tree becomes a separate auxiliary tree with a parent pointer to v. v used to be part of some preferred path, and now it's part of another preferred path.

Then, we look at the the highest vertex in the preferred path represented by v's aux tree (the bottom left node in the aux tree), which we shall call t. Now, t has a parent pointer to the deepest node in the parent auxiliary tree (the bottom right node in the aux tree), which we will call w. This entire process is depicted in Figure 9 below.

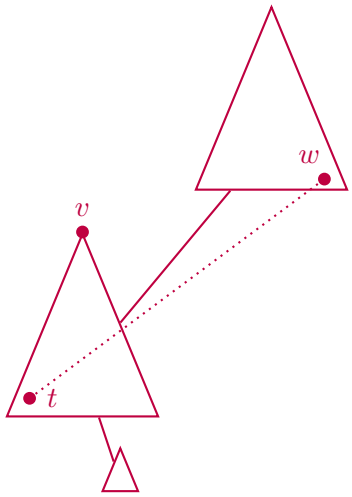

Figure 9: After splaying v

As a result of the `access`, we splay w so that it moves to the root of its auxiliary tree. Now, t is the preferred child of w and gets forked off into its own auxiliary tree. The auxiliary graph now looks like as follows:

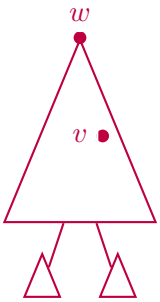

Figure 10: After splaying w, cutting off right tree, and merging v

Now, v is in its parent auxiliary tree with w. We splay v, and it gets moved to the root of its current auxiliary tree. Essentially, we just keep on repeating this process until we get v at the very top. We provide an overview of what `access`(v) does in Algorithm 4.

Algorithm 4 Accessing a node v

procedure ACCESS(v)

while v is not the root of the root auxiliary tree **do**

1. Splay v, cut off right tree as needed
2. Splay the appropriate node in v's parent auxiliary tree, cut off right tree as needed
3. Merge v into its parent auxiliary tree

end while

4.5 Runtime of Access

The cost of an `access` can be computed by finding the total cost of doing the preferred child change (PCC) splays.

Thus, the runtime of a single `access` is

$$O\left((\text{cost of splay}) \cdot (1 + (\#\text{ of PCC}))\right)$$

and runtime of m of them is

$$O\left((\text{cost of splay}) \cdot (m + (\#\text{ of PCC}))\right)$$

We know that a splay takes amortized $O(\log n)$ time, and in the next lecture we will show that the number of PCCs is $O(m \log n)$. Thus, our overall runtime is:

$$O\left(\log n \cdot (m + m \log n)\right) = O(m \log^2 n) \tag{1}$$

To prove this runtime, we use the analysis method called **Heavy-Light Decomposition**.

5 Heavy-Light Decomposition (HLD)

Definition 5.1 (size)**.** We define **size**(v) to be the number of nodes in v's subtree, including v.

In the HLD scheme, we categorize edge (v, w) from v to its parent w as **Heavy** or **Light** according to the following rule:

$$(v, w) \in \begin{cases} \text{heavy} & \text{if size}(v) > \frac{1}{2}\text{size}(w) \\ \text{light} & \text{if size}(v) \leq \frac{1}{2}\text{size}(w) \end{cases}$$

Then, HLD allows us to perform the following runtime analysis:

Suppose the preferred child of w switched from v to v'. Then, if we say that (v, w) is the preferred child destruction, then (v', w) is denoted as the preferred child creation (PCC).

Since we want to bound the number of preferred child creations (PCC), we want to prove the following two theorems:

Theorem 5.2. $(\#\text{ PCC}) \leq 2(\#\text{ LPCC}) + n + m$

Theorem 5.3. $(\#\text{ LPCC}) \leq O(m \log n)$

where LPCC stands for "Lightest Preferred Child Creation". Combining these two theorems together, we are able to show that

$$(\#\text{ PCC}) \leq O(m \log n)$$

and thus the runtime in Eq (1) directly follows.

Proofs for these theorems will be discussed in the next lecture. Stay tuned!

References

[1] Daniel Dominic Sleator and Robert Endre Tarjan. A Data Structure for Dynamic Trees. *J. Comput. Syst. Sci.*, 26(3):362-391, 1983