COMS E6998: Advanced Data Structures (Spring'19) Apr 25, 2019

Lecture 13: Dynamic Graph Data Structures

Instructor: *Omri Weinstein* Scribes: *Mia Saint Clair, Qingtian Gong*

1 Introduction

1.1 Overview

Today we will start the last topic of the course, which is data structures for graph problems. In this lecture we will discuss Link-Cut Trees and the "Heavy-Light" decomposition. Next week we will discuss Euler-Tour Trees and $O(\lg^2 n)$ operations for general undirected graphs.

1.2 Motivation

- Online networks (algorithms). Most online networks are highly dynamic. We want to maintain the structures, properties, summaries, or statistics of a network, such as the spanning forests, distance information, paths in the networks, or density of components.
- Most basic: reachability/connectivity. We want to maintain the information about connectivity of vertices, while the graph is changing with insertion/deletion of edges.
- (Approximate) distance of shortest path. Shortest path problem is a challenging problem, especially when the graph is dynamic.

1.3 Distance Oracles [Thorup-Zwick '95]

Given a graph, a distance oracle is a data structure to maintain the (approximate) distance of the shortest path between two vertices u and v.

Given a graph, preprocess it so that you can quickly answer whether two vertices are in the same component. A basic algorithm for this problem is union-find. Now I want to speed up not just whether they are in the same component, but also speed up getting the distances. A naive solution for that is to store all $O(n^2)$ answers and query in constant time.

Distance oracle can solve this problem statically. It uses super-linear space $s = n^{1+o(1/k)}$ for k-approximate distances in constant time $t = O(1)$. Its main idea is to compress the graph into subgraphs that preserve distances between vertices in k-approximate. Such compressed graphs only exist in undirected graphs. Thus this data structure fails in directed graphs, because there is no such graphs.

2 Link-Cut Trees

Next, we are going to consider dynamic problems in undirected graphs undergoing a sequence of edge insertions/deletions. Our goal is to get $O(\lg n)$ time for forests. Why it is non-trivial? The trees could be very unbalanced, and finding the root may take $\Omega(n)$ time.

Today we will see how to solve this unbalanced problem. We will see two ways to do this. The first way is Link-Cut Trees. Link-Cut Trees solve connectivity and many other statistics in $O(\lg n)$ time. Its goal is to maintain a collection of rooted trees (forest) under the following operations:

- MakeTree(): To make a tree with single node.
- Link(v,w): To make v a child of w. It is an insertion of an edge between v and w, like in a social network, one person becomes friend with another one. We assume v is always the root of its tree.
- Cut(v): To disconnect v from its parent.
- FindRoot(v): Returns the root of the tree that vertex v is a node of. This operation is interesting because path to root can be very long. The operation can be used to determine if two nodes u and v are connected
- PathAggregate(v): To keep aggregating node by node and return any statistics/query like min/max/sum along the path from the root to v. The major motivation here is to solve network flow problems.

Link-Cut Trees were developed by Sleator and Tarjan. They achieve logarithmic amortized cost per operation for all operations. Link-Cut Trees are similar to Tango trees in that they use the notions of preferred child and preferred path. They also use splay trees for the internal representation.

3 "Heavy-Light" Decomposition

3.1 Definition

The main concept of Link-Cut Trees is very similar to Tango Trees. The key idea is to maintain represented trees (forest) via decomposition to preferred paths. The represented trees are the original trees.

A preferred child (PC) is a unique child of the last access. Its definition is:

$$PC(v) := \begin{cases} \phi, if\ last\ accessed\ node\ in\ T_v\ was\ v \\ w, if\ w \in T_v\ was\ the\ last\ accessed\ in\ T_v \end{cases}$$

where T_v is v's subtree. A preferred edge is an edge between preferred child and its parent. A preferred path is a chain of preferred edges to its maximal length.

Link-Cut Trees then store each preferred path of the represented tree T in an auxiliary tree, which is a Splay Tree. Nodes in each auxiliary (Splay) tree are keyed by their depth (in the represented tree). The use of Splay Tree is crucial for analysis. For each node in the auxiliary trees, the left subtree stores the nodes higher than v in the represented tree, while the right subtree stores the nodes lower than v. So the root of the preferred path will be the leftmost node of its auxiliary tree.

We also want to store path parent pointers. A parent pointer points to the parent of the preferred path's topmost node in the represented tree. Each auxiliary tree has one path parent pointer, and it is stored in the root of the auxiliary tree. The structure of Link-Cut Trees is illustrated below:

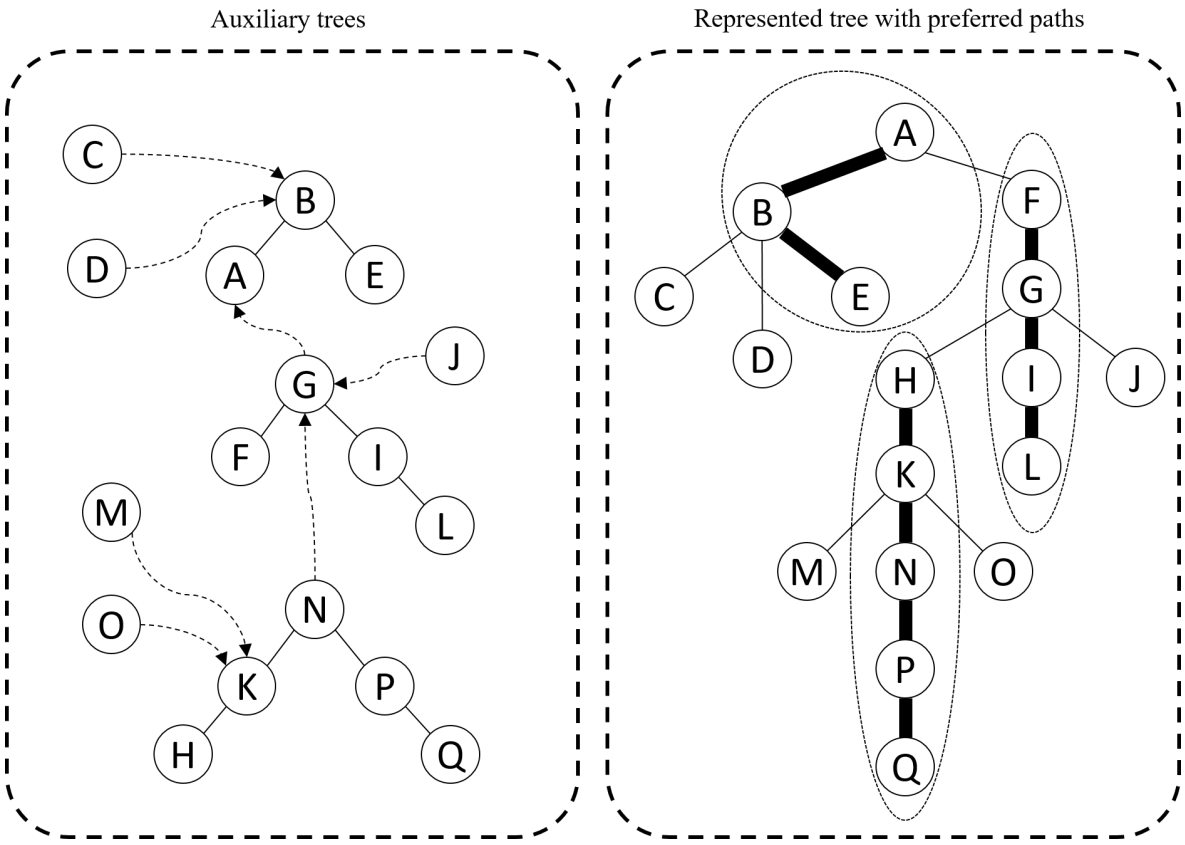

Figure 1: Auxiliary trees and the corresponding represented tree

3.2 Operations

3.2.1 Access

Every operation begins with the subroutine $Access(v)$. The point of this subroutine is to restructure the entire tree. $Access(v)$ causes the preferred paths to change. The main point is to make the path from root to v preferred.

We need to "clip-off" all the nodes lower than v. The natural way to do this is to splay v to the root of its auxiliary tree. Thus all the nodes higher than v are in the left subtree, and all the nodes lower than v are in the right subtree. Then we can "clip-off" v's right child. Then we set the path parent pointer of v's right child to be pointing to v. By definition of preferred child, v has no preferred child then. So the right child of v should be null.

We then use a loop to conduct other preferred child changes. What we do in the loop is to "clip-off" the previous preferred child and switch its preferred child to the new one. So we first set w to be the path parent pointer of v, and then we splay w to the root. After that we "clip-off" the previous preferred child of w and switch it to v.

Pseudocode:

Access(v)

- Splay v in its auxiliary tree and then "clip-off" right child
  - path-parent(right(v))$\leftarrow v$
  - parent(right(v))$\leftarrow \phi$
  - right(v)$\leftarrow \phi$
- Loop until we reach the root:
  - $w \leftarrow$ path-parent(v)
  - Splay w
  - Switch w's preferred child:
    - path-parent(right(w))$\leftarrow w$
    - parent(right(w))$\leftarrow \phi$
    - right(w)$\leftarrow v$
    - parent(v)$\leftarrow w$
    - path-parent(v)$\leftarrow \phi$
  - $v \leftarrow w$

3.2.2 Cut

To cut an edge between v and its parent, we first run Access(v) and then disconnect it with its left child.

Pseudocode:

Cut(v)

- Access(v)
- parent(left(v))$\leftarrow \phi$
- left(v)$\leftarrow \phi$

3.2.3 Path Aggregate

We run Access(v) first. And then since each node maintains the values such as min, max, and sum, we can easily return the needed value directly.

Pseudocode:

PathAggregate(v)

- Access(v)
- return v.subtree-sum

3.2.4 Link

We run Access() in both trees and make w a left child of v.

Pseudocode:

Link(v,w)

- Access(v)
- Access(w)
- left(v)$\leftarrow w$
- parent(w)$\leftarrow v$

4 Analysis

As one can see from the pseudo code, all operations are doing at most logarithmic work (amortized, because of the splay call in find root) plus an access. Thus it is enough to bound the run time of access. First we show an $O(\lg^2 n)$ bound.

4.1 An $O(\lg^2 n)$ Bound

From accesss pseudo code we see that its cost is the number of iterations of the loop times the cost of splaying. We already know from previous lectures that the cost of splaying is $O(\lg n)$ amortized (splaying works even with splits and concatenations). Recall that the loop in access has O(# preferred child changes) iterations. Thus to prove the $O(\lg^2 n)$ bound we need to show that the number of preferred child changes is $O(\lg n)$ amortized. In other words the total number of preferred child changes is $O(m \lg n)$ for a sequence of m operations. We show this by using the Heavy-Light Decomposition of the represented tree.

4.2 Heavy-light decomposition

The Heavy-Light decomposition is a general technique that works for any tree (not necessarily binary). It calls each edge either heavy or light depending on the relative number of nodes in its subtree.

Let $size(v)$ be the number of nodes in vs subtree (in the represented tree).

Definition *An edge from vertex parent(v) to v is called heavy if size(v) $> \frac{1}{2}$size(parent(v)), and otherwise it is called light.*

Furthermore, let light-depth(v) denote the number of light edges on the root-to-vertex path to v. Note that light-depth(v) $\lg n$ because as we go down one light edge we decrease the number of nodes in our current subtree at least a factor of 2. In addition, note that each node has at most one heavy edge to a child, because at most one child subtree contains more than half of the nodes of its parents subtree.

There are four possibilities for edges in the represented tree: they can be preferred or unpreferred and heavy or light.

4.3 Proof of the $O(\lg^2 n)$ Upper Bound

The amortized power of splay

To bound the number of preferred child changes, we do Heavy-Light decomposition on represented trees. Note that access does not change the represented tree, so it does not change the heavy or light classification of edges. For every change of preferred edge (possibly except for one change to the preferred edge that comes out of the accessed node) there exists a newly created preferred edge. So, we count the number of edges which change status to being preferred. Per operation, there are at most $\lg n$ edges which are light and become preferred (because all edges that become preferred are on a path starting from the root, and there can be at most $\lg n$ light edges on a path by the observation above). Now, it remains to ask how many heavy edges become preferred. For any one operation, this number can be arbitrarily large, but we can bound it to $O(\lg n)$ amortized. How come? Well, during the entire execution the number of events heavy edge becomes preferred is bounded by the number of events heavy edge become unpreferred plus $n-1$ (because at the end, there can be $n-1$ heavy preferred edges and at the beginning the might have been none). But when a heavy edge becomes unpreferred, a light edge becomes preferred. Weve already seen that there at most $\lg n$ such events per operation in the worst-case. So there are $\lg n$ events heavy edge becomes unpreferred per operation. So in an amortized sense, there are $\lg n$ events heavy edge becomes preferred per operation (provided $n-1/m$ is small, i.e. there is a sufficiently large sequence of operations).

4.4 The $O(\lg n)$ Bound

We prove the $O(\lg n)$ bound by showing that the cost of preferred child switch is actually $O(1)$ amortized. From accesss pseudo code one can easily see that its cost is

$$O(lgn) + (\textit{cost of preferred child switch} * \#\textit{preferred child switches}) \tag{1}$$

From the above analysis we already know that the number of preferred child switches is $O(\lg n)$, thus it is enough to show that the cost of preferred child switch is $O(1)$. We do it using the potential method.

Let $s(v)$ be the number of nodes under v in the tree of auxiliary trees. Then we define the potential function $\phi = \sum_v lg(s(v))$. From our previous study of splay trees, we have the Access theorem, which states that: $cost(splay(v)) \leq 3(\lg s(u) - \lg s(v)) + 1$ where u is the root of vs auxiliary tree.

Now note that splaying v affects only values of s for nodes in vs auxiliary tree and changing vs preferred child changes the structure of the auxiliary tree but the tree of auxiliary trees remains unchanged. Therefore, on access(v), values of s change only for nodes inside vs auxiliary tree. Also note that if w is the parent of the root of auxiliary tree containing v, then we have that $s(v) \leq s(u) \leq s(w)$. Now we can use this inequality and the above amortized cost for each iteration of the loop in access.

The summation telescopes and is less than

$$3(lg\ s\ (root\ of\ represented\ tree)) - lg(s(v)) + O(\#\ preferred\ child\ changes) \tag{2}$$

which in turn is $O(\lg n)$ since $s(root) = n$. Thus the cost of access is thus $O(\lg n)$ amortized as desired.

To complete the analysis we resolve the worry that the potential might increase more than $O(\lg n)$ after cutting or joining. Cutting breaks up the tree into two trees thus values of s only decrease and thus ϕ also decreases. When joining v and w, only the value of s at v increases as it becomes the root of the tree of auxiliary trees. However, since $s(v) \leq n$, the potential increases by at most $\lg\ s(v) = \lg n$. Thus increase of potential is small and cost of cutting and joining is $O(\lg n)$ amortized.