Recitation 9 — Tree Contraction

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

March 19, 2012

Today's Agenda:
- Announcements
- Linked List Prefix Sum
- Tree Contraction

1 Announcements

- HW 4 is being returned today
- HW 5 is being graded
- HW 6 is due next Tuesday
- Questions about homework, class, life, universe?

2 Linked List Prefix Sum

We have a doubly linked list M of nodes; each node i has some data D_i. For each node, we want to compute $S_i = \sum_{\text{j after i in M}} d_j$. Let $R[i]$ be the right-neighbor of node i and $L[i]$ be the left-neighbor of node i. (or $NONE$ if the corresponding elements don't exist)

This looks like the familiar prefix sum problem, but harder: we don't know the index of each element, just the left and right neighbors.

But we can use the same idea: combine pairs of adjacent nodes, recurse on the smaller list, and then reconstruct the original list.
Q: How do we stop a node from ending up in two pairs?
A: Restrict each node to only being the left half of a pair or the right half of a pair. Then each node only has one pair it could be a part of.

Q: How do we mark the nodes? We don't have much to go on...
A: Try random (random is usually worth a try).

Let's compute the probability that node i is the left half of a pair. This happens with probability $1/4$ (we must have marked i L and $i+1$ R, each of which occurs indepedently with probability $1/2$). By linearity of expectation, we get $(n-1)/4$ pairs in expectation (the last node can't start a pair). This is 2x better than lecture!

Now it's just details: we're going to splice out the second node from each pair. Let (x, y) be a pair. We are deleting y from the list, so we need to set $R'[x] = R[y], L'[R[y]] = x$. x gets $y's$ data: $D'[x] = D[x] + D[y]$. Now recurse on the new list (the old list minus all the $y's$). Everything that wasn't a y has the right sum D (by IH): for each y, set $D'[y] = D[R[x]] + D[y]$ (drawing a picture makes this step clear).

Let's do an example: (keep track of L, R, D, I at each step; I is the set of active indices)
$L = \{1, NONE, 0, 4, 2\}$

$R = \{2, 0, 4, NONE, 3\}$
$D = \{2, 3, 5, 1, 4\}$
$I = \{0, 1, 2, 3, 4\}$

Let's say we get $LLRRL$ on our first set of flips. Students should show the steps for the first step of the algorithm, do the recursive call, and then show how to reconstruct the list. (hand back HW while they are working this example)

Review answer:
$(0, 2)$ and $(4, 3)$ pair.

So we splice out $2, 3$ and get:
$L = \{1, NONE, 0, 4, 0\}$
$R = \{4, 0, 4, NONE, NONE\}$
$D = \{7, 3, 5, 1, 5\}$
$I = \{0, 1, 4\}$

Now we recursively compute $S[0] = 12, S[1] = 15, S[4] = 5$
Now we can compute $S[2] = S[4] + S[2] = 10$ and $S[3] = 0 + S[3] = 1$.

Let's write some code:

```
fun bit = /* returns 0 or 1 at random */
fun rank succ pred data =
    let
        fun rank' S P D I =
            if (length I) <= 2 then D
            else
            let
                fun independent_set X =
                let
                    val bits = tabulate (fn _ => bit()) (length X)
                    fun ok i = (nth pred i)!=NONE andalso (nth bits i)=0 andalso (nth bits (nth
                in
                    filter (fn i => ok i) X
                end
                val I' = independent_set I
                val US = tabulate (fn i => (nth P i, nth S i)) I'
                val UD = tabulate (fn i => (nth P i, (nth D i) + (nth D (nth P i)))) I'
                val UP = tabulate (fn i => (nth S i, nth P i)) I'
                val R = rank' (inject US S) (inject UP P) (inject UD D)
                              (difference I I')
                val UD' =
                    tabulate (fn i => (i, (nth D i) + (nth D (nth S i)))) next_I
            in
                inject UD' R
            end
    in
        rank' succ pred data (tabulate (fn i => i) (length S))
    end
```

3 Tree Contraction

OK, on to the real deal. Same problem as before, but for arbitrary binary trees.

The same strategy applies: we want to pair up adjacent nodes (edges of the tree) without using the same node in more than one edge. Once we've picked the pairs, splicing out the second node in each pair and propogating information is almost the same as the linked list case. Picking pairs, however, is tricky.

There are three types of vertices: leaves, nodes with one child, and nodes with two children. Let V_0 be the set of leaves, V_1 be the set of nodes with one child, and V_2 be the set of nodes with two children.

Lemma 3.1. $|V_0| = |V_2| + 1$

Proof. Every path down from the root ends in a leaf. Whenever we see a node with one child along a path, the number of paths doesn't change. Every time we see a node with two children along a path, the path splits and the total number of paths increases by 1. We start out (before we see the root) with one path. □

Because leaves only have one neighbor, they are a good candidate to be in our independent set. In fact, we can definitely get at least half of the leaves in each step (each leaf conflicts with at most one other leaf: its sibling). Terminology: the process of collecting leaves into our independent set is called "rake". Just to be explicit, an algorithm for doing this: we can arbitrarily number the leaves and take all the leaves which do not have a sibling which is a lower-numbered leaf.

So now we have at least $|V_0|/2$ nodes, which takes care of getting a constant fraction of $|V_0| + |V_2|$ (specifically, $1/4$). Unfortunately, $|V_1|$ might be arbitrarily large compared to $|V_0| + |V_2|$ (consider the linked list!), so we need a way to get some of those nodes too.

We can do pretty much the same thing as last time: As before, choose L or R at random for each node in V_1. We can take any node labeled R which is the child of a node in V_2, or any parent-child pair labeled (L, R) where neither L nor R is a parent of a leaf. The step of collecting nodes from V_1 into our independent set is called "compress".

There are two cases to consider to make sure we get enough nodes this way:
If $V_1 \leq 2V_0$, we might end up not being able to make pairs out of any of the V_1 nodes (neither the parent nor the grandparent of a leaf can start a pair). However, we still get $V_0/2$ pairs from the leaves, and there are at most $|V_0| + |2V_0| + |V_0| + 1$ nodes in the tree, so we get about a quarter of the nodes total.

Otherwise, we have $V_1 > 2V_0$, and we get $(V_1 - 2V_0)/4$ nodes from compress (every eligible node has a $1/4$ chance of starting a pair; each leaf eliminates both its parent and grandparent from being eligible) and $V_0/2$ nodes from rake, which is just $V_1/4$ nodes total. However, we $|V_1| + |V_0| + |V_0| + 1 < 2|V_1|$ nodes in the tree, so we get about an eighth of the nodes total.

We could do a more careful analysis, but we definitely get a constant fraction of the nodes in the tree on each rake-compress step. Since we remove a constant fraction of the nodes at each step, we expect to contract the whole tree in logarithmically many steps.

Lemma 3.2. *We remove an independent set of nodes in each rake-compress step*

Proof. No leaf pairs intersect because no two leaves we pick have a common parent. No V_1 pairs intersect by construction (as before, each V_1 node can only be part of at most 1 possible pair). No V_1 pair intersects with a leaf pair because we don't allow V_1 pairs to use the parents of leaves. □

This is a general framework, which can solve a variety of problems depending on how we update the data during rake, compress, and after the recursive call.

3.1 Subtree Size

We want to compute the size of the subtree rooted at each node. Initialize $size[v] = 1$ for every v (if we wanted to compute a sum-of-descendents at each node like before, we would just change the initializations here and everything would work).

```
rake(v):
    size[parent[v]] += size[v]

compress(v):
    size[parent[v]] += size[v]

post-recursive-call(v):
    size[v] += size[child[v]]
```