Lecture 17 — Graph Contraction I: Tree Contraction

In this lecture, we will explore tree contraction, yet another powerful construct based on the idea of contraction that forms the basis of many fast parallel algorithms on trees and graphs. As you might expect, working with trees is generally simpler than working with general graphs since trees have a lot of nice structural properties. Let’s start with a simple example.

1 Leaffix Scan, Rootfix Scan, Tree Size, and Depth

Suppose we’re given a tree (not necessarily balanced) and we’re interested in finding the size of the subtree rooted at each node or finding the depth of every node in the tree. Can we design an algorithm that computes these fast in parallel?

In many cases, we might also be interested in computing something slightly more general. We’ll look at Leaffix and Rootfix. Leaffix is a play on the word “prefix” as in the prefix sum (plus scan) operation covered in previous lectures. The Leaffix operation scans a tree from bottom to top, storing the sum of the children into their parent, up to the root. An example of Leaffix is given below:

```
7 / \ 24
  5 9 ==> 10 0
  / \          / \
 3 6 1 0 0 0
```

Rootfix is a corresponding operation but it operates in the opposite direction: from top to bottom. An example of Rootfix is given next:

```
7 / \ 0
  5 9 ==> 7 7
  / \          / \
 3 6 1 12 12 12
```

Indeed, Leaffix and Rootfix generalize depth and size computations. Notice that by assigning the value 1 to every node, the Leaffix operation will find the size of the subtree rooted at each node while the Rootfix operation will find the depth of every node in the tree.

**Goal:** We want to support these operations in $O(n)$ work and $O(\log^2 n)$ span.

To get some intuition, we’ll consider the problem of computing the tree size for every node in the tree. First, we’ll look at the following two extremes:

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.
Balanced binary tree. In this case, the depth of the tree is \( \log n \), so intuitively, it shouldn't be difficult to perform these operations in, say, \( O(\log^2 n) \).

**Exercise 1.** Design an algorithm for computing the size of each subtree of a tree \( T \) that has \( O(n) \) work and \( O(\log^2 n) \) span assuming \( T \) is \( O(\log n) \) deep.

Linked list. At the other end of the spectrum, we have linked lists. A linked list is really a tree where every node has only one child. What can we do in this case? On the one hand, this seems hopeless because a linked list has a long chain of dependencies. On the other hand, what we aim to compute is just a prefix sum and if this were represented as a sequence, we would already have a linear-work algorithm to do that.

1.1 Linked List

In the `scan` algorithm, elements are paired up into disjoint pairs, where each pair is then combined into a single element, forming a new sequence half the original length. The key idea there was that we knew exactly which elements to pair together: each even-numbered element is paired with the odd-numbered element to the right of it. *How can we apply this idea to the case of linked lists?*

To answer this question, we’ll take a closer look at `scan` to see what we can learn from it. The crux of the `scan` algorithm is a pairing that shrinks the sequence by half. The beauty of it was that these pairs were easy to construct (`scan` uses “position” information to help decide how to pair up elements). Furthermore, these pairs are easy to contract: they are disjoint and straightforward to combine the elements. These two features are going to be the goals of our pairing algorithm:

**Many Easy Pairs:** Ideally, we would like to pair up every element (node) in the tree, like in `scan`. But as the scan recurrence suggests, we would be happy with any constant fraction, as long as the pairing is easy to construct (e.g., can be done in \( O(n) \) work and \( O(\log n) \) span).

**Disjointness:** Like in `scan`, we want to create non-overlapping pairs. Intuitively, this means we could contract these pairs simultaneously in parallel without a problem.

Whereas `scan` could easily decide the pairing using position information, the main difficulty for us is the apparent symmetry in the problem—from the point of view of a vertex in the linked list, it doesn’t know whether it is an odd element or an even element. Worse yet, it is unclear whether a vertex should stay put or attach herself to the right or to the left to get many disjoint pairs. And even if we want to come up with new rules, it is unclear what to do: all vertices pretty much look alike.

We will “break symmetry” using randomization. We will simulate the even/odd roles using coin flips. For this, we’ll flip a coin for each vertex: if it comes up tails, it is an “odd’ node—and if it comes up heads, it is an “even” node. This motivates the following algorithm\(^1\)

\[
\text{Pair}(T) = \\
1. \text{Each vertex flips a fair coin.} \\
2. \text{For each vertex } u \text{ (except the last vertex), if } u \text{ flipped tails and } \text{next}(u) \text{ flipped head, } u \text{ will be paired with } \text{next}(u). \\
\]

\(^1\)In class, we did a slightly version where we relied on the outcomes of both \( u \)'s neighbors; in the notes, we'll try to optimize for constants a little bit by deciding using only 2 flips.
Let’s see this in action:

In this example, we have 6 vertices. Suppose the coins turned up T, H, T, T, H, T. Then, according the rules above, we will pair up the first and the second vertex—and the 4th and the 5th.

Claim 1.1. The pairs are disjoint.

Claim 1.2. The number of pairs formed is expected \((n - 1)/4\).

Proof. First, notice that the number of pairs is exactly the number of nodes that flipped heads and their next successors flipped tails. Let \(X_v\) denote the event that \(v\) flipped heads and \(\text{next}(v)\) flipped tails. Then, \(E[X_v] = 1/4\). Summing across \(n - 1\) vertices (the last vertex doesn’t have a successor), we get using linearity of expectations that the expected number of pairs is \((n - 1)/4\). 

What do we do with these pairs? Just like in the scan algorithm, we contract them. For the purpose of solving the subtree size problem, each node \(v\) keeps a value \(s_v\). On the initial tree, \(s_v\) is 1 for all vertices. Throughout the algorithm, think of this as the weight of a vertex and we’re interested in finding for each vertex \(v\), the sum of the weights of all the vertices that come after \(v\) and of \(v\) itself. Eventually, this is going to turn into the size of the subtree. For this reason, when contracting a pair \(a \rightarrow b\), we add \(s_a\) and \(s_b\) together.

Repeating the pairing and contracting steps, we’ll eventually be in a situation where our tree simply consists of 2 vertices, which looks something like this:

At this point, we could determine that the sum of all the weights is \(5 + 1 = 6\). In a manner similar to the scan algorithm, we will expand out these contracted trees. As we do so, we derive the sum of the weights that we intend to compute. Starting with the 2-node example, we have
where the $t$ value ("total") of a node $v$ denotes the total weight of all the vertices to the right of $v$ and of $v$ itself. Continuing with our example, we know that inductively what's happening is

As we expand back out, we realize that we could readily derive the $t$ values of the head nodes by copying the corresponding $t$ values from the recursive solution. We derive the $t$ value of a tail vertex by adding its $s$ value to the $t$ value of the next node in the chain, similarly to the scan algorithm.

As shown in Claim 1.2, we expect $(n - 1)/4$ pairs to be formed in each recursive call. Notice that for each pair formed, the resulting tree has one less vertex (because it got contracted). Therefore, in expectation, after one round of contraction, the number of vertices $n'$ becomes

$$E[n'] \leq n - (n - 1)/4 \leq 7n/8$$

for $n \geq 2$. Here, we have just used the fact that for $n \geq 2$, $\frac{1}{4}(n - 1) \geq n/8$. In each recursive call, we could perform all the steps in $O(n)$ work and $O(\log n)$ span, so we have the following work/span recurrences:

$$W(n) = W(n') + O(n)$$
$$S(n) = S(n') + O(\log n)$$

where $0 \leq n' \leq n$ and as we just calculated, $E[n'] \leq 7n/8$. Hence, $E[W(n)] = O(n)$ and $E[S(n)] = O(\log^2 n)$, as desired.
General Trees. In some sense, the more “interesting” case is a mixture of the two, potentially unbalanced or list-like trees. These are the subject of the following section.

2 Tree Contraction

Indeed, the more challenging trees to contract are a mixture of nice balanced trees and chains (linked-list like paths). On the one hand, we have a random-pairing algorithm that excels at long chains. On the other, we know that an algorithm that “nibbles” the leaves will thrive on reasonably balanced trees.

What should we do on a mixture of the two? As we’ll soon see, we can build an efficient tree contraction algorithm out of two primitive operations, corresponding to the two extremes we just discussed. Let \( T = (V, E) \) be a tree. The most basic form of tree contraction involves two operations: rake and compress:

- **rake** \( (T) \) produces a tree \( T' \) by dropping all leaves (i.e. degree-1 vertices) of \( T \) unless that leaf is adjacent to another leaf, in which case only one leaf is dropped.

- **compress** \( (T) \) produces a tree \( T' \) by finding and contracting disjoint pairs. The goal of compress is to shorten a long path, so specifically, the pairs that compress finds have the property that each of them must involve at least one degree-2 vertex. For our purposes, we’ll further require that a pair be of the form \( a \rightarrow b \), where \( a \) is a degree-2 vertex (which is not a tree root) and \( b \) is any other vertex (of any degree).

For each of these pairs, we contract it. By requiring the pair to be of this form, we know that every pair in fact looks like \( v \rightarrow u \rightarrow w \), where \( u \) is the non-root degree-2 vertex, \( v = \text{parent}(u) \), and \( w = \text{next}(u) \). Thus, contracting it results in the two edges incident on \( u \) getting eliminated and a new edge connecting \( \text{parent}(u) \) and \( \text{next}(u) \) being formed. Pictorially (as shown below), the act of contracting this pair will create a “shortcut” from \( v \) to \( w \).

The tree contraction algorithm repeatedly applies these two operations alternately:

```plaintext
fun treeContract(T) =
    if T consists of only degree-0 vertices then return T
    else
        T' = compress(T)
        T'' = rake(T')
        return treeContract(T'')
```

For concreteness, we’ll assume the ArraySequence representation of the graph.
**How to pick pairs for `compress`?**  The algorithm above is underspecified. We didn’t say how we are going to pick the pairs when performing the `compress` operation. Let \( U(T) = \{ v \in V(T) : \text{deg}(v) = 2 \} \) be the set of all degree-2 vertices of \( T \) excluding the root. The intuition is that if there is a long path in \( T \), the vertices of this path will be present in \( U(T) \) (after the branches are pruned by `rake`).

We want to create a lot of pairs between vertices in \( U(T) \) and some other vertices, and we want the pairing to be non-overlapping. Inspired by our algorithm from the previous section, we’ll design an algorithm that forms \(|U(T)|/4 \) disjoint pairs.

This is a familiar situation from the linked-list case we just explored. We’ll flip a coin for each vertex in the graph, not just the \( U(T) \) vertices. Then, for each vertex \( u \in U(T) \), we’re only going to pair \( u \) up if \( u \) flipped tails. **Who should \( u \) be paired with?** Remember that \( u \) is a non-root vertex with degree 2, so it has exactly 1 parent and 1 successor (\( \text{next}(u) \) is well-defined). Like in the linked-list case, if \( \text{next}(u) \) flipped heads, we’ll pair \( u \) with it—otherwise, \( u \) just stays lonely. This is the following algorithm:

\[
\text{Pair}(T) =
\begin{align*}
1. & \quad \text{Each vertex flips a fair coin.} \\
2. & \quad \text{For each vertex } u \text{ in } U(T), \text{ if } u \text{ got a tail and } \text{next}(u) \\
& \quad \text{got a head, } u \text{ will be paired with } \text{next}(u). \\
\end{align*}
\]

One thing is clear: the pairs are disjoint. The following claim shows that this randomized algorithm creates \(|U(T)|/4 \) pairs:

**Claim 2.1.** The expected number of pairs is \(|U(T)|/4 \).

**Proof.** For a vertex \( u \in U(T) \), the probability that a vertex \( u \) is paired is exactly \( 1/4 \)—\( u \) must flip heads and its successor must flip tails. By linearity of expectations, we expect \(|U(T)|/4 \) pairs. \( \square \)

Let’s look at a small example to understand what the tree contraction algorithm really does.

**Draw picture.**

### 2.1 How fast is tree contraction?

We’ll now analyze the performance of the tree contraction algorithm. The plan is to write recurrences for work and span. To do this, we’ll need to analyze the effects that `rake` and `compress` have on the tree. Specifically, we’ll prove the following lemma which shows that applying `rake` and `compress` to a tree causes the number of vertices to go down by a constant fraction in expectation:

**Lemma 2.2.** Let \( T \) be a tree and \( T'' \) be the tree after applying `rake` and `compress` on \( T \). Then,

\[
\mathbb{E}[|V(T'')|] \leq \beta \cdot |V(T)|,
\]

where \( \beta = 23/24 \).
Proof. Let \( n_i \) be the number of vertices in \( V(T) \) with degree \( i \). Notice that rake gets rid of all degree-1 vertices and compress removes at least \( \frac{1}{4}|U(T)|/4 \) (by Claim 2.1). We also know that \( |U(T)| \geq n_2 - 1 \) — we don’t know the degree of the root node, so we’ll err on the safe side. Thus, in terms of \( n_2 \), we expect compress to remove at least

\[
\frac{1}{4}|U(T)| \geq \frac{n_2 - 1}{4} \geq \frac{n_2}{8}
\]

for \( n_2 \geq 2 \). Therefore, in expectation, the number of vertices that disappear after applying rake and compress is at least

\[
n_1 + \frac{n_2}{8} \geq \frac{1}{8}(n_1 + n_2),
\]

where we recall that compress doesn’t decrease the number of degree-1 vertices.

Now, how big is \( n_1 + n_2 \) compared to \( |V(T)| \)? As we will show next (Claim 2.3), \( n_1 + n_2 \) is at least \( 1/3 \) of the vertices, i.e.,

\[
n_1 + n_2 \geq \frac{|V(T)|}{3}.
\]

Hence, we expect to remove at least \( |V(T)|/24 \) vertices. Hence, in expectation, \( V(T’’) \) can have at most \( \frac{23}{24}|V(T)| \) vertices. \( \square \)

We now prove a claim that relates \( n_1 + n_2 \) to the size of the tree.

**Claim 2.3.** In any \( n \)-vertex tree \( T \), the number of degree-1 and 2 vertices is least \( n/3 \).

**Proof.** Suppose for a contradiction that the number of degree-1 and 2 vertices is less than \( n/3 \), so then there are at least \( 2n/3 \) vertices with degree 3 or more. This means \( \sum_{v \in V(T)} \deg(v) \) is at least \( 2n \), but this is a contradiction because \( \sum_{v \in V(T)} \deg(v) = 2(n - 1) < 2n \). In the last step, we used the fact that a tree has \( n - 1 \) edges. Notice that the argument here is similar to how you prove the Markov’s inequality. Indeed, we could have proved this claim using Markov’s inequality. \( \square \)

As a consequence of the lemma we just proved, we know that the work/span of the tree contraction algorithm is

\[
W(n) = W(n') + O(n) \quad S(n) = S(n') + O(\log n),
\]

where \( 0 \leq n' \leq n \) and \( E[n'] \leq \frac{23}{24} n \). These are familiar recurrences; we know they solve to \( E[W(n)] = O(n) \) and \( E[S(n)] = O(\log^2 n) \).

**Tree Contraction Examples:** In recitation tomorrow, you’ll see how to apply tree contraction to solve some interesting problems.