Lecture 6: List contraction, tree contraction, and symmetry breaking

1 Work-efficient list prefix

In the previous lecture, we saw an elegant "pointer jumping" algorithm for computing a prefix operation on a linked list, with a good critical path of $\Theta(\lg n)$ on an n-element list. Unfortunately, the pointer jumping approach requires $\Theta(n \lg n)$ work for a job that can easily be done in linear time on a serial machine. This section presents a work-efficient randomized algorithm for prefix computation on a list, with an expected critical path of $O(\lg n)$ and expected work of $O(n)$. This algorithm also gives us a work-efficient method for computing node depths of a binary tree, since the Euler-tour method seen in class reduces the computation of depths to a list prefix.

1.1 List contraction algorithm

The key idea is "list contraction." We are able to reduce the size of the prefix calculation and then recursively compute the "contracted" list. Specifically, given an associative binary operator $\otimes$ and a list of n elements, numbered from 1 to n and with values $a_1, a_2, \ldots, a_n$, we follow the following procedure, which we call RANDOMIZED-LIST-PREFIX:

1. **Choose victims.** We randomly choose a set of elements for elimination from the list, by first independently (in the probablity sense) labeling each element either HEAD or TAIL, and then picking any HEAD-labeled element which is not followed immediately by a HEAD-labeled element. (In this scheme, the last element of the list is always chosen if it is a HEAD.) The victims form an independent set (that is, a subset of the graph in which no two nodes are adjacent), since if an element is a victim, its successor is a TAIL and its predecessor is followed by a HEAD (the victim itself) and so neither neighbor can also be a victim.

2. **Splice out victims.** The victims are then spliced out of the list by jumping the pointers of their predecessors over them. As each victim i is removed from the list, its value a_i is pushed into its successor by setting the successor's value to be $a_i \otimes a_{i+1}$. (Independence of victims guarantees that the successor is not also a victim.) This computation maintains the following invariant: if element i has not been spliced out of the list, then the product (with respect to $\otimes$, and in order) of all the values stored in all elements before i still in the list is equal to $[1, i]$, which is the desired final value for element i. (Recall that $[j, k]$ where $j \leq k$ is shorthand notation for $a_j \otimes a_{j+1} \otimes \ldots \otimes a_k$.)

3. **Recursively compute.** We now recursively call RANDOMIZED-LIST-PREFIX on the contracted list (which in fact may be the same list, if by chance no elements were eliminated). Due to the invariant maintained above, the prefix computation on the contracted list gives the right answer for the original list for non-eliminated elements. (The base case of recursion is a single element.)

4. **Splice in.** Now we want to splice in the eliminated elements into the contracted list (which now has the property that element i contains the value $[1, i]$) by restoring the pointers (which, of course, needed to have been saved somewhere) and setting the value of each spliced-in element i to $a_{i-1} \otimes a_i = [1, i-1] \otimes [i, i] = [1, i]$. At this point, the prefix computation is done.

Figure 1 shows how RANDOMIZED-LIST-PREFIX runs on a list of 6 elements.

1.2 Analysis of work

Intuitively, the reason that list contraction does less work than pointer jumping is that the eliminated elements incur no further work once they are spliced in and out, but in pointer jumping, every element continues to incur work at every step (until the base case is reached). We can show that the expected length of the list with which RANDOMIZED-LIST-PREFIX is called is reduced by a constant fraction at every contraction.

Let n_k be the length of the list after k contractions. Given a list of n elements, for any element i,

$$\Pr\{i \text{ is eliminated}\} = \begin{cases} \Pr\{i \text{ is HEAD}\} \Pr\{i\text{'s successor is TAIL}\} = \frac{1}{4} & \text{if } i < n \ , \\ \Pr\{i \text{ is HEAD}\} = \frac{1}{2} & \text{if } i = n \ , \end{cases}$$

and thus, after contraction, $\mathrm{E}[n_k] \leq \frac{3}{4} n_{k-1}$. Substituting recursively, and using the fact that $\mathrm{E}[\mathrm{E}[n_k]] = \mathrm{E}[n_k]$, we have that

$$\begin{aligned} \mathrm{E}[n_k] &\leq \left(\frac{3}{4}\right)^k n_0 \\ &= \left(\frac{3}{4}\right)^k n \ . \end{aligned}$$

This directly leads to a geometric series that upper bounds the amount of work done. We get a work which is linear in the length of the list:

$$\sum_{k=1}^{\infty} \mathrm{E}[n_k] = \Theta(n) \ .$$

Therefore the list contraction algorithm is work-efficient.

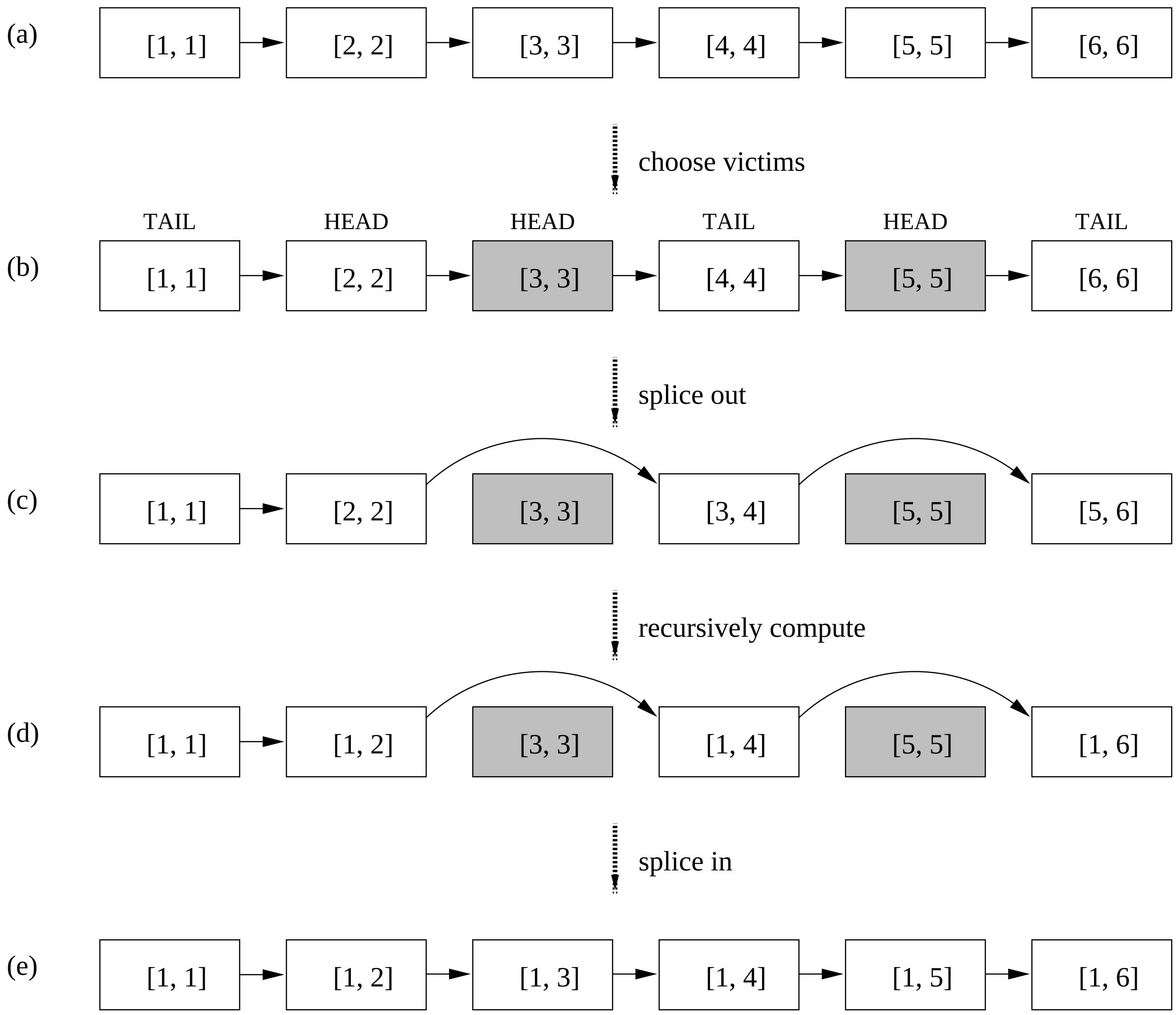

Figure 1: RANDOMIZED-LIST-PREFIX run on a list of 6 elements. This sequence shows a single contraction. **(a)** The input list. **(b)** Elements are assigned HEAD or TAIL and then victims are chosen accordingly. **(c)** Victims are spliced out, with their values being pushed into successors. **(d)** Prefix of the contracted list is recursively computed. **(e)** Victims are spliced back in.

1.3 Analysis of critical path

The critical path is linear in the number of contractions needed, i.e. the number of contractions needed to reach the base case of a one-element list. Central to our probabalistic analysis is Markov's inquality: for any nonnegative random variable v and constant t, $\Pr\{v \geq t\} \leq \mathrm{E}[v]/t$. First we observe that $\Pr\{n_k \geq 1\} \leq \mathrm{E}[n_k] \leq \left(\frac{3}{4}\right)^k n$. This bound is so weak that we need $k \geq \log_{4/3} n$ just to beat the trivial bound $\Pr\{v \geq t\} \leq 1$. Nevertheless, if we set $k = r \log_{4/3} n$ for some constant r, we have

$$\begin{aligned}
\Pr\left\{n_{r \log_{4/3} n} \geq 1\right\} &\leq \mathrm{E}\left[n_{r \log_{4/3} n}\right] \\
&= n\left(\frac{3}{4}\right)^{r \log_{4/3} n} \\
&= n^{1-r} .
\end{aligned}$$

Now we analyze the expected number of contractions k. Since for an nonnegative integral random variable v, $\mathrm{E}[v] = \sum_{i=0}^{\infty} \Pr\{v = i\} \cdot i$, we have the following upper bound on k, by grouping the elements of the expectation summation:

$$\begin{aligned}
\mathrm{E}[k] &\leq \Pr\left\{0 \log_{4/3} n < k \leq 1 \log_{4/3} n\right\} \cdot 1 \log_{4/3} n \\
&\quad + \Pr\left\{1 \log_{4/3} n < k \leq 2 \log_{4/3} n\right\} \cdot 2 \log_{4/3} n \\
&\quad + \Pr\left\{2 \log_{4/3} n < k \leq 3 \log_{4/3} n\right\} \cdot 3 \log_{4/3} n \\
&\quad + \dots \\
&= \sum_{r=0}^{\infty} \Pr\left\{r \log_{4/3} n < k \leq (r+1) \log_{4/3} n\right\} \cdot (r+1) \log_{4/3} n \\
&\leq \sum_{r=0}^{\infty} \Pr\left\{k \geq r \log_{4/3} n\right\} \cdot (r+1) \log_{4/3} n \\
&\leq \sum_{r=0}^{1} \Pr\left\{k \geq r \log_{4/3} n\right\} \cdot (r+1) \log_{4/3} n \\
&\quad + \sum_{r=2}^{\infty} \Pr\left\{k \geq r \log_{4/3} n\right\} \cdot (r+1) \log_{4/3} n \\
&\leq \log_{4/3} n + 2 \log_{4/3} n + \sum_{r=2}^{\infty} n^{1-r} \cdot (r+1) \log_{4/3} n \\
&= 3 \log_{4/3} n + O(1) \\
&= \Theta(\lg n) .
\end{aligned}$$

Thus the critical path of RANDOMIZED-LIST-PREFIX is $O(\lg n)$.

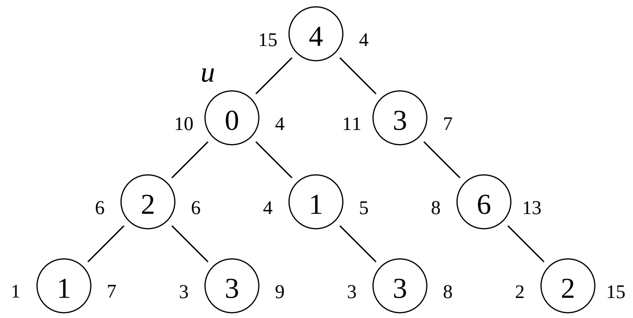

Figure 2: Example of treefix computations with the addition operator. The number inside each node is the value of the element stored there, the number to the left of each node is the result of the leaffix operation, and the number to the right is the result of the rootfix operation. Consider node u. Its leaffix result is the in-order sum of all elements stores in the tree rooted at u $(1 + 2 + 3 + 0 + 1 + 3 = 10)$, and its rootfix result is the prefix scan from the root $(4 + 0 = 4)$.

2 Treefix computation

Suppose we have a binary tree of elements and an associative binary operator $\otimes$. In this section, we consider a class of "treefix" computations: a "rootfix" operation computes the $\otimes$-scan of every simple path from root to leaf, and a "leaffix" operation computes, for every node u, the result of the in-order $\otimes$-product described by the tree rooted at u. Figure 2 shows an example of rootfix and leaffix with the addition operator.

2.1 Rootfix computation with tree contraction

To perform rootfix with the addition operator, we can use the Euler-tour method: for each node with value v, set the incoming A value to be v, the middle B value to be 0, and the outgoing C value to be $-v$. Unfortunately, this method does not generalize to any associative binary operator, since it requires an inverse, so that the values computed on the way down (the A values) can be undone on the way up (the C values). Maximum, for instance, could not be computed with an Euler-tour.

For a general rootfix computation, we use an method invented by Miller and Reif [2] called "tree contraction," which is analogous to list contraction. Though a deterministic algorithm with the same bounds exists, we consider a randomized algorithm that is both simpler to analyze and generally faster in practice. This algorithm, RANDOMIZED-ROOTFIX, is analogous to RANDOMIZED-LIST-PREFIX and works as follows:

1. **Choose victims.** To choose nodes to be eliminated, we randomly choose pairs of adjacent nodes in the binary tree so that all pairs are independent (i.e. no pairs share nodes), and select the child of each pair to be a victim. To do this, we make each node independently choose one of its neighbors (either parent or children) based on a

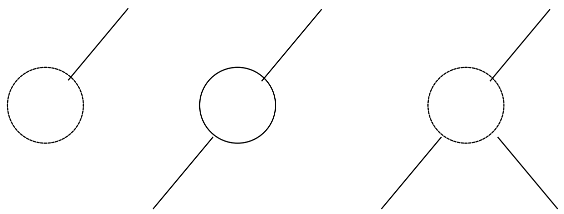

Figure 3: The probability distribution of choosing a neighbor depends on the number of children. **(a)** Leaves always choose parents. **(b)** One-child nodes choose between neighbors uniformyly. **(c)** Nodes with two children never choose their parent, guaranteeing that the child in any pair never has two children, which guarantees that a binary tree remains binary after pairs are combined.

probability distribution that depends on how many children the node has. If two nodes choose each other, they are paired. If a node has no children, i.e. it is a leaf, then it always chooses its parent. If a node has one child, it randomly chooses its parent or child uniformly. (In the special case of a root with one child, we can make it always choose its child.) If a node has two children, it randomly choose one of its children uniformly, never choosing its parent. Figure 3 shows the three types of nodes with the various probabilities.

2. **Splice out.** After victims are selected, we contract the tree by splicing them out, setting the parent's child pointer (whichever one pointed to the victim) to the victim's own child, if one exists. We know that no victim will have two children, since nodes with two children never choose their parent. This property guarantees that a binary tree remains binary after contraction. As a node is spliced out, its value is pushed into its child, if one exists; specifically if u is a victim and it has a child v, then v's value is set to $u \otimes v$. This maintains the invariant that the $\otimes$-product of the sequence of nodes from the root to any non-eliminated node is the correct final value for that node.

3. **Recursively compute.** We recursively compute the rooffix of the contracted tree by calling RANDOMIZED-ROOTFIX, with a single node as base case. The invariant maintained above insures that the values computed by the rootfix of the contracted tree are the correct values for the original tree.

4. **Splice in.** Now we splice in the previously eliminated nodes. As a node is spliced in, its proper rootfix value is calculated from its parent's value (which has been recursively computed). Specifically, if a node u is the parent of a node v which is being spliced in, the value of v is set to $u \otimes v$, which will be the correct final value for v.

Figure 4 shows how this method works with the addition operator.

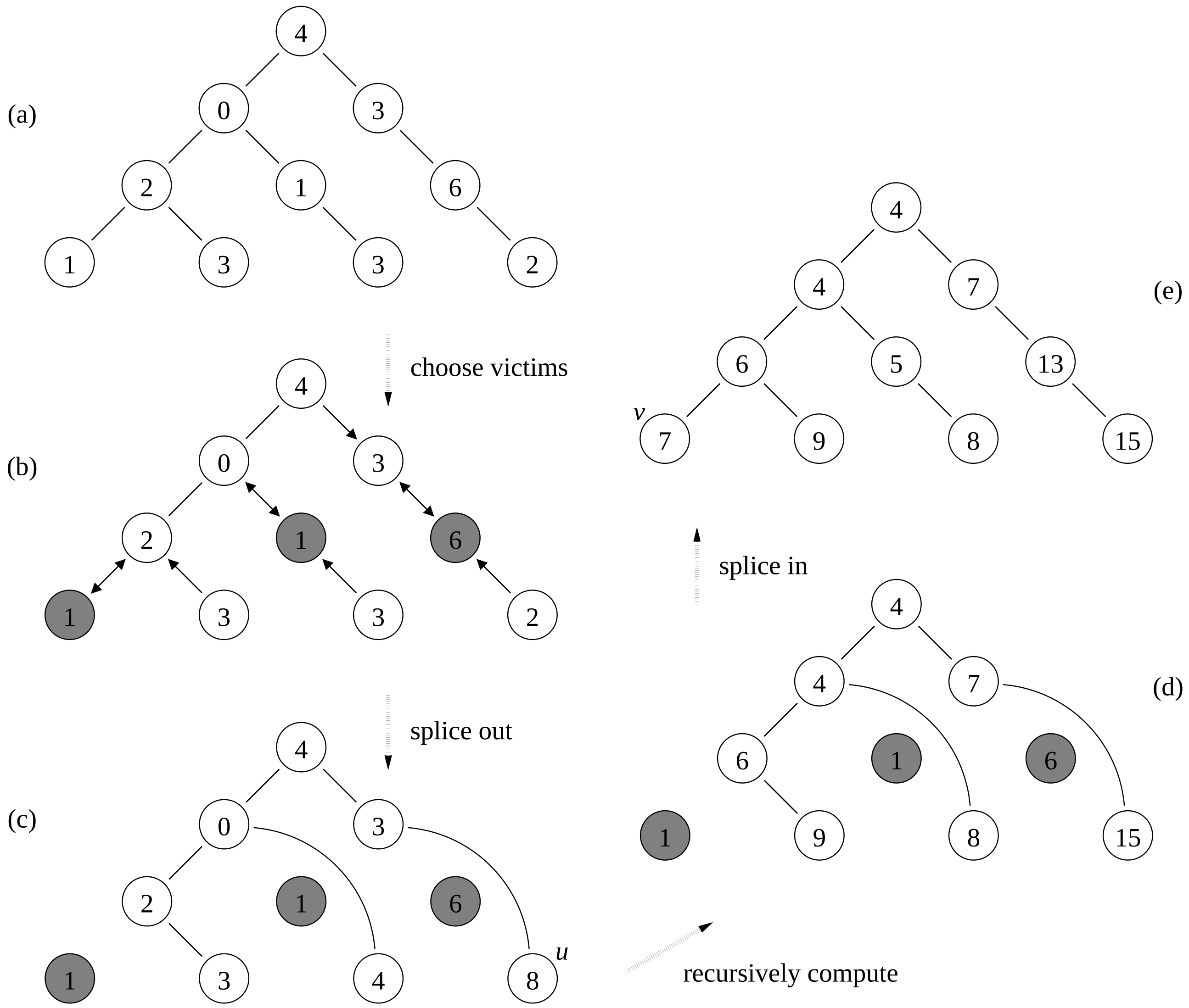

Figure 4: Example of rootfix with the addition operator. **(a)** The initial binary tree. **(b)** Each node chooses a neighbor, indicated by outgoing arrows, and then children of pairs (nodes which choose each other) become victims, which are shaded. **(c)** Victims are spliced out, with their values being pushed to their children. For example, node u becomes $6+2=8$. **(d)** The rootfix of the contracted tree is computed recursively. Notice these are the correct final values for the nodes in the contracted tree. **(e)** Victims are spliced back in, with their values being computed from parents. For example, node v becomes $6+1=7$.

2.2 Analysis of work and critical path

Let n_k be the number of nodes in the tree after k contractions. We want to show that the expected number of nodes in the tree decreases by a constant fraction at each contraction, and then the analysis is the same as that of RANDOMIZED-LIST-PREFIX.

Consider a binary tree with n nodes. Let c_i be the number of nodes in the tree with i children, so $n = c_0 + c_1 + c_2$. Now we consider that expected number of each type of node that is eliminated during a contraction. Since a node is eliminated if it chooses its parent and its parent also chooses it and these choices are independent, the probability that a given node is eliminated is the probability that it chooses its parent times the probability that its parent choose it. For leaves, this product is $1/2$, and for nodes with one child, it is $1/4$. Thus, the expected number of leaves eliminated is $c_0/2$ and the expected number of one-child nodes eliminated is $c_1/4$.

Now recall that a binary tree has the property that $c_0 = c_2+1$, and thus $c_0/4 = c_2/4+1/4$.[1] Then the expected total number of eliminated nodes is at least

$$
\begin{aligned}
\frac{c_0}{2} + \frac{c_1}{4} &= \frac{c_0}{4} + \frac{c_0}{4} + \frac{c_1}{4} \\
&= \frac{c_0}{4} + \left(\frac{c_2}{4} + \frac{1}{4}\right) + \frac{c_1}{4} \\
&\geq \frac{(c_0 + c_1 + c_2)}{4} \\
&= \frac{n}{4} \,.
\end{aligned}
$$

Since the expected number of nodes in the tree decreases by at least a fourth during each contraction, we know by the analysis seen for RANDOMIZED-LIST-PREFIX that the expected work of RANDOMIZED-ROOTFIX is $O(n)$ and its expected critical path is $O(\lg n)$. (In fact, prefix computation on a list is a special case of rootfix, and RANDOMIZED-LIST-PREFIX performs a special case of RANDOMIZED-ROOTFIX's functionality, with the HEAD and TAIL labels being equivalent to a string of one-child nodes choosing between parent and child.)

2.3 Leaffix computation with tree contraction

To compute the leaffix on a tree, we can use a tree contraction algorithm which uses the same basic approach as RANDOMIZED-ROOTFIX and achieve bounds of $O(n)$ expected work and $O(\lg n)$ expected critical path. However, we have to be more clever, since the in a leaffix

[1]This is easily proved by induction: given a binary tree, remove a leaf, and then $c_0 = c_2 + 1$ holds by assumption for the new tree. Restore the removed leaf: if it is added to a leaf, then c_0 is unchanged since the new leaf replaces the leaf it is added to, and c_2 is also unchanged, and if it is added to a node with one child, then the new leaf increases c_0 by one and the node it was added to changes into a node with two leaves, also increasing c_2 by one.

computation the final value depends on an entire subtree of elements as opposed to a list of ancestors. Whereas in the rootfix calculation we directly modified the element values as nodes were spliced in and out, in our leaffix calculation we have each node keep track of a triple of values and then modify these triples according to some invariant.

Specifically, our algorithm RANDOMIZED-LEAFFIX keeps track, at each node, a triple of constants $\langle a, b, c\rangle$, which represents the following computation: $a \otimes l \otimes b \otimes r \otimes c$, where l and r are variables for the leaffix values of the left and right children, or, if a child does not exist, the identity I of $\otimes$.[2] For each node, we initialize the triple by setting a to be the element value and b and c to be I. RANDOMIZED-LEAFFIX maintains the invariant that the final leaffix value of a node can be obtained by recursively evaluating the computation represented by that node's triple (i.e. finding the values for l and r based on the subtrees and then computing $a \otimes l \otimes b \otimes r \otimes c$). For convenience, we call this the "triple invariant."

After triples are initialized, RANDOMIZED-LEAFFIX works as follows:

1. **Choose victims.** Randomly choose node for elimination as in RANDOMIZED-ROOTFIX.

2. **Splice out.** If u is the parent of a victim v, we push v's triple up into u's triple as v gets spliced out. Relying on the fact that v has at most one child, we adjust the values of v's triple so that the triple invariant is maintained. Specifically, remembering that the values in the triple represent a computation, we effectively substitute the computation for v into the computation for u, and then combine adjacent constants with $\otimes$. This will result in a computation that can be represented by a triple, since v has at most one child. For example, if u's triple is $\langle a_u, b_u, c_u\rangle$, v's triple is $\langle a_v, b_v, c_v\rangle$, v is u's right child and v itself has a left child, then u's triple becomes $\langle a_u, b_u \otimes a_v, b_v \otimes I \otimes c_v \otimes c_u\rangle$. If instead v has a right child, for example, then u's triple becomes $\langle a_u, b_u \otimes a_v \otimes I \otimes b_v, c_v \otimes c_u\rangle$. The key observations are that this adjustment of u's triple takes constant time — because of the constant size of the triples, there is a constant number of ways the parent's triple can be adjusted— and that the triple invariant is maintained.

3. **Recursively compute.** We recursively compute the leaffix operation on the contracted tree with RANDOMIZED-LEAFFIX, without needing to initialize the triples as in the top-level call. The base case is when a single node, in which case the final leaffix value is simply $a \otimes I \otimes b \otimes I \otimes c$ where $\langle a, b, c\rangle$ is the node's triple. After the recursive call, the values computed for the nodes in the contracted tree will be the correct leaffix values for those nodes, due to the triple invariant.

4. **Splice in.** As a node is spliced back into the tree, its leaffix value is computed in constant time by evaluating the computation represented by its triple: $\langle a \otimes l \otimes b \otimes r \otimes c\rangle$. This is possible because the value of the node's child (if one exists) will have been recursively computed, so both l and r are known (at least one will be I). At this point, every node contains the leaffix value and the algorithm terminates.

[2]This algorithm requires $\otimes$ to have an identity, which is not an additional constraint on $\otimes$ since a token identity can always be created and the operator modified to recognize it, if a natural identity does not exist.

Because we have ensured that splicing in and out take constant time, the work and critical path analysis for Randomized-Leaffix is the same as that for Randomized-Rootfix, and thus we have expected work of $O(n)$ and expected critical path of $O(\lg n)$.

2.4 General expression evaluation with tree contraction

Randomized-Leaffix works by effectively keeping track of an expression at each node (represented by a triple) which depends on the expressions of that node's children. The key points are that a victim's expression can always be pushed up into its parent's expression in constant time during splicing out, that this pushing up results in an expression in the same form (so recursion works), that a victim's expression can be evaluated in constant time in the base case and during splicing in, and that the triple invariant is maintained. This poweful approach is in fact quite general and can be applied to many binary tree problems.

For example, consider a binary tree that describes an expression with the $+$ and $\times$ operators, where internal nodes each contain $+$ or $\times$ and each leaf contains a number. The value of the expression rooted at any given node can always be formulated as $(a \times l \times r) + (b \times l) + (c \times r) + d$ (and thus stored as a 4-tuple), where a, b, c, d are constants and l and r are variables that represent the values of the expressions rooted at the left and right subtrees. Substituting a victim's 4-tuple (which represents an expression with at most one non-identity-valued variable) into it's parent's 4-tuple results in an expression that can be represented by the same kind of 4-tuple. (This is easily seen by checking the possible cases.) The invariant that a node's 4-tuple represents the value of the expression rooted at that node is maintained, and base case and evaluation during splicing in are straightforward, and so we are doing essentially the same thing as Randomized-Leaffix with a different expression form at each node.

For other applications of this approach, see [1].

3 Deterministic symmetry breaking

Given a ring of elements (a circular linked list), how can we find a large independent set of elements, where by "large" we mean a constant fraction? In Randomized-List-Prefix, we have already seen a $O(1)$-critical path method for choosing an independent set with expected size of $n/4$, where n is the size of the list. In this section we present an algorithm for doing so deterministically with a critical path of $O(\lg^* n)$.[3] This is called symmetry breaking because if adjacent elements share a resource (the symmetry), choosing an independent set prevents competition (the breaking).

The algorithm Choose-Independent-Set works in two steps:

1. **Find 6-coloring.** First, we find a 6-coloring of the ring elements. This is done by starting with an n-coloring, where each element is assigned a distinct color $0, 1, \ldots, n-1$

[3] Recall that $\lg^* n$ is the minimum number of repeated applications of lg needed to reduce n to at most 1. It is a very slow function ($\lg^* 2^{65536} = 5$) and in practice can be considered a small constant.

based on its memory location (or processor index, if we have one processor per element), and interatively reducing the number of colors in the coloring (wihtout ever having same-colored neighbors) in the following way. Suppose the color indices of the ring are k bits long. Let $\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle$ be the color bits of a given element a, and $\langle b_{k-1}, b_{k-2}, \ldots, b_0 \rangle$ be the colors bits of that element's sucessor b. Since these two colors are different, we have some i where $a_i \neq b_i$. The number i, since it is less than k, can be written in $\lceil \lg k \rceil$ bits: $\langle i_{\lceil \lg k \rceil - 1}, i_{\lceil \lg k \rceil - 2}, \ldots, i_0 \rangle$. Assign $\langle i_{\lceil \lg k \rceil - 1}, i_{\lceil \lg k \rceil - 2}, \ldots, i_0, a_i \rangle$ to be a's new color, and assign new colors to all elements in the same way. Now we have a coloring with indices of $\lg k + 1$ bits, since if the first $\lceil \lg k \rceil$ bits of a element's new color are the same as the successor's first $\lceil \lg k \rceil$ bits, then by construction the last bit will differ ($a_i \neq b_i$).

This interative re-coloring will reduce the number of colors bits per element for all $k \geq 4$, since $\lceil \lg k \rceil + 1 < k$ for $k \geq 4$. When $k = 3$, the a element's color can differ with its successor's bit in location 0, 1, or 2, and so the new coloring will be one of the three 2-bit strings $\langle 0, 0 \rangle$, $\langle 0, 1 \rangle$, or $\langle 1, 1 \rangle$ concatenated with either 0 or 1, and thus the new coloring will be 3 bits representing one of six values. Hence the interative reduction of colors terminates at six colors, and since the number of bits per color is reduced by a lg at each interation, $O(\lg^* n)$ interations, each with constant critical path, are needed.

2. **Choose elements.** To determine the independent set of elements from after 6-coloring, we start will all elements marked as live and then iterate through each color $c = 0, 1, \ldots, 5$, killing any live neighbors of elements with color c as we go. There are at least $n/3$ live elements left at the end of this process, because it is impossible to have three consecutive dead elements: either the middle element kills its neighbors and thus stays alive, having no one left to kill it, or it is killed by one of its neighbors, and thus the neighbor stays alive. These remaining live elements comprise an independent set, since it is impossible to have two adjacent elements remain alive: one would have killed the other.[4] Thus this $O(1)$-critical path interation gives us an independent set of size $n/3$.

Overall, CHOOSE-INDEPENDENT-SET has $O(\lg^* n)$ critical path.

References

[1] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel graph algorithms. *Algorithmica*, pages 53–77, 1988.

[2] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. *Proceedings of the 26th Annual Symposium on Foundations in Computer Science*, pages 478–489, 1985.

[4]This shows that the set chose is a maximal independent set, meaning that we cannot revive any of the dead elements and continue to have an independent set. Note the distinction from having a *maximum* independent set, the independent set with the most elements; finding the maximum independent set is in fact NP-hard.