Linked Lists and Trees

How efficiently can we

- find the root of a tree,
- solve the prefix problem, if elements are given not by an array, but by a linked list?

  For instance, which time is sufficient to determine a maximal element in a linked list?
- parallelize the traversal of a possibly deep tree?
- evaluate arithmetic expressions?

We encounter important algorithmic methods:

- the doubling technique (pointer jumping),
- Euler tours
- and tree contraction.

Pointer Jumping

A forest F is represented by

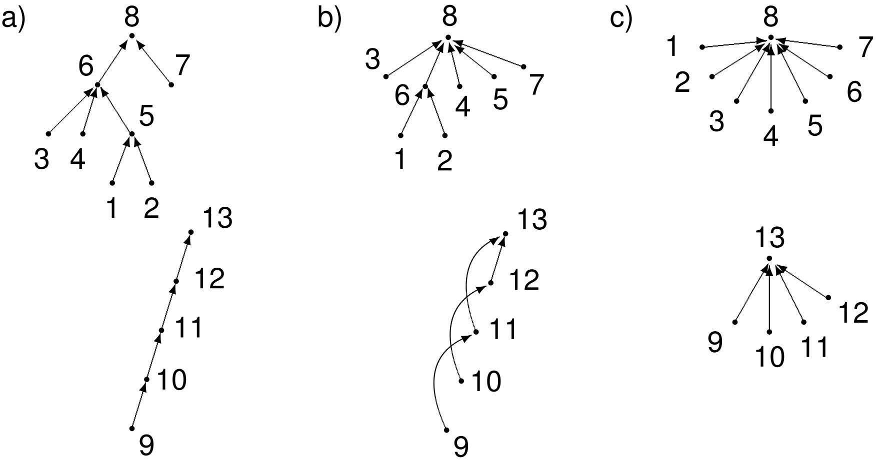
- its set of nodes $V = \{1, \ldots, n\}$
- and by an array "parent": parent[i] is the parent of node i. (If i is a root, then parent[i] $= i$.)

Determine the respective root for all nodes in parallel.

Finding the root via pointer jumping:

```
for $i = 1$ to $n$ pardo
    while $\mathrm{parent}[i] \neq \mathrm{parent}\big[\mathrm{parent}[i]\big]$ do
    // parent[$i$] is different from the root.
        $\mathrm{parent}[i] = \mathrm{parent}\big[\mathrm{parent}[i]\big]$;     // We climb upwards.
    Output ($i$,parent[$i$]);
```

Pointer Jumping: Examples



Pointer Jumping: The Analysis

- Pointer jumping is a CREW algorithm, since the parent array is concurrently read, but only exclusively modified.
- The invariant: if node i becomes a child of node j in round t, then i and j have distance 2^t in the original forest, provided j is not a root.
    - The base case: the claim is true for $t = 0$.
    - The inductive step: If i is a child of k in round t and becomes a child of j in round $t + 1$, then
        - i and k as well as k and j have distance 2^t in F.
        - hence i and j have distance 2^{t+1} in F.

- Pointer jumping is a CREW-PRAM algorithm.
- If F is a forest with n nodes and depth d, then the root is determined for any node in time $O(\log_2 d)$ with n processors.

List Ranking

- A singly linked list of n nodes is represented by a shared array S. $S[i]$ is the successor of i, resp. $S[i] = 0$ if i has no successor.
- Each node i has a value $V[i] = a_i$.

Determine all suffix sums $a_n, a_{n-1} * a_n, \dots, a_1 * a_2 * \cdots * a_n$ for an associative operation $*$.

- Applications:
  - If $V[i] = 1$ for all i, then $a_i * \cdots * a_n$ is the distance (plus one) of the ith list element from the end of the list.
  - For $x * y = \max\{x, y\}$ the "sum" $a_1 * \cdots * a_n$ is the maximum of $\{a_1, \dots, a_n\}$.
- In comparison to the prefix problem:
  - we now determine suffix sums instead of prefix sums.
  - The crucial difference is the restricted access to list elements, instead of the random access for the prefix problem.

List Ranking Via Pointer Jumping

```
for $i = 1$ to $n$ pardo
    $W[i] = V[i]$; $T[i] = S[i]$;    // Save values and pointers.
    while ($T[i] \neq 0$) do
    // We have not reached the end of the list.
        $W[i] = W[i] * W[T[i]]$;    // Values are added.
        $T[i] = T[T[i]]$;    // We perform pointer jumping.
```

- Correctness:

  Before each iteration $W(i)$ is the sum of all list elements, beginning in list element i and ending before list element $T(i)$.
- Speed: $\Theta(\log_2 n)$ with n processors for a list of length n, provided the operation $*$ can be evaluated in time $O(1)$.
- The algorithm can be implemented on a EREW-PRAM.

Depth-first Search

- How to traverse a graph with a parallel depth-first search?
  - If we have to determine whether node u is visited before node v in a depth-first search started in node s, then there are in all likelihood no good parallel algorithms!
  - Parallelizations of depth-first search exist, but they are inefficient.

- We will see that tree traversals can be efficiently parallelized, namely computing the ordering of nodes according to a preorder, postorder or level order traversal.

- How does depth-first search work when applied to trees?
  - Starting at the root, dfs follows the tree edges to reach a leaf.
  - After reaching a leaf, the traversed edges are traversed again, but now in backwards direction.
  - Depth-first search stops, when all edges are traversed exactly once in either direction.

Euler Tours

$T = (V, E)$ is an (undirected) tree.

- $\mathrm{Euler}(T) := (V, \{(i,j) \mid \{i,j\} \in E\ \}$ is the "Euler graph" of T.
- An Euler tour is a path in $\mathrm{Euler}(T)$ which traverses all edges of $\mathrm{Euler}(T)$ exactly once and returns to its starting point.

- Euler tours of Euler(T) correspond to a depth-first search traversal of T and vice versa.
- How to compute Euler tours fast?
  - If T is given by its adjacency list representation, then the linked list $N[v]$ collects all neighbors of a node v.
  - Each linked list orders neighbors according to their position within the list.
  - If we have already constructed a partial Euler tour with last edge (u, v): with which edge should we continue?

Constructing an Euler Tour Edge by Edge

- If (u, v) is the last edge of a partial Euler tour and
- if w is the right circular neighbor of u in the list $N[v]$, then

continue the tour with the successor edge (v, w).

We verify correctness by induction on the number of nodes.

- Let l be a leaf with parent v. $N[v] = (\ldots, u, l, w, \ldots)$ is the list of v.
- Remove l and we obtain the tour $T = (\ldots, u, v, w, \ldots)$.
- What does the recipe require for the original graph?
  - after edge (u, v) continue with edge (v, l).
  - The list $N[l]$ consists only of v:
    since v is its own right circular neighbor, continue with edge (l, v).
  - w is the right neighbor of l in $N[v]$: the next edge is (v, w).
- Thus we get the tour $T = (\ldots, u, v, l, v, w, \ldots)$ in the original tree.

The recipe works.

Input Representation

We assume that a tree is given as an adjacency list with cross references:

- the adjacency list $N[v]$ of a node v is given as a circular list and
- for any element w in $N(v)$ there is a link to element v in $N(w)$.

- What is the successor of edge (u, v)?
  - Determine v in the list $N[u]$.
  - Follow the cross reference from v (in $N[u]$) to get to u (in $N[v]$).
  - Then determine the right neighbor w of u in $N[v]$ and (v, w) is the successor of (u, v).
- The list of an Euler tour can be determined in constant time with $2 \cdot |E| = 2(n - 1)$ processors.
  (The adjacency list contains $2(n - 1)$ elements, since each edge occurs twice.)

Traversing Trees in Parallel

Let $T = (V, E)$ be an undirected rooted tree which is presented as an adjacency list with cross references. The following problems can be solved on an EREW-PRAM in time $O(\log_2 |V|)$ with $\frac{|V|}{\log_2 |V|}$ processors:

1. Determine the parent "parent(v)" for each node $v \in V$.
2. Determine a postorder numbering.

   Postorder visits the children first and then the parent.
3. Determine a preorder numbering.

   Preorder visits the parent first and then the children.
4. Determine a level-order numbering: assign to each node its depth.
5. Determine the number of descendants for every node $v \in V$.

The Euler Tour Algorithm

1. Determine an Euler tour beginning in the root.
2. Assign weights $w(e)$ to edges e of Euler(T).
   // The weight assignment is problem dependent.
3. Apply list ranking, with addition as operation, to the reversed list of the Euler tour, i.e., compute prefix sums.
4. Evaluate the prefix sum value(e) for each edge e.

Ressources

If steps (2) and (4) run in time $O(1)$, then list ranking is the most expensive step.

Determining Parents

- Assign the weight $w(e) = 1$ for all edges.
- $\text{parent}[v] = u$ iff $\text{value}(u, v) < \text{value}(v, u)$. Why?

$$
\begin{aligned}
u \text{ is the parent of } v \quad &\Leftrightarrow \quad \text{The Euler tour visits } u \text{ before } v \\
&\Leftrightarrow \quad \text{value}(u, v) < \text{value}(v, u).
\end{aligned}
$$

The Postorder Numbering

- First determine parent[u] for all nodes u.
- Then assign the weight $w(u, \mathrm{parent}(u)) = 1, \quad w(\mathrm{parent}(u), u) = 0.$
  - Only child→parent edges are counted.
  - If $(\ldots, u, \text{parent}[u], \ldots)$ is the Euler tour, then $\mathrm{value}(u, \mathrm{parent}(u))$ is the number of child→parent edges before and including edge $u \rightarrow \text{parent}[u]$.
  - Postorder visits a parent after all of its children ⇒
    u is visited right before the edge $u \rightarrow \text{parent}[u]$ is traversed.
- $\mathrm{post}(u) = \begin{cases} n & u \text{ is the root,} \\ \mathrm{value}(u, \mathrm{parent}(u)) & \text{otherwise} \end{cases}$ is a postorder numbering.

Preorder Numbering

- First determine parent[u] for all nodes u.
- Then assign the weight $w(u, \mathrm{parent}(u)) = 0, \quad w(\mathrm{parent}(u), u) = 1.$
  - Only parent→child edges are counted.
  - If $(\ldots, \text{parent}[u], u \ldots)$ is the Euler tour, then $\mathrm{value}(\mathrm{parent}(u), u)$ is the number of parent→child edges before and including edge parent[u] $\rightarrow u$.
  - Preorder visits a parent before it visits its children $\Rightarrow$ u is visited right after the edge parent[u] $\rightarrow u$ is traversed.
- $\mathrm{pre}(u) = \begin{cases} 1 & u \text{ is the root,} \\ \mathrm{value}(\mathrm{parent}(u), u) + 1 & \text{otherwise} \end{cases}$ is a preorder numbering.

Level-order Numbering

- First determine parent[u] for all nodes u.
- Then assign the weight $w(\text{parent}(u), u) = 1$ to capture that depth increases by one and $w(u, \text{parent}(u)) = -1$ to capture that we move back up to decrease depth by one.
    - If $(\ldots, \text{parent}[u], u \ldots)$ is the Euler tour, then $\text{value}(\text{parent}(u), u)$ is the depth of node u.
- Hence $\text{level}(u) = \begin{cases} 0 & u \text{ is the root,} \\ \text{value}(\text{parent}(u), u) & \text{otherwise} \end{cases}$ is the level-order numbering.

Counting the Number of Descendants

- First determine parent[u] for all nodes u.
- Set $w(u, \mathrm{parent}(u)) = 1, \quad w(\mathrm{parent}(u), u) = 0.$
  - Each node u is counted once by counting its “back edge” $u \to$ parent[u].
  - $\mathrm{value}(u, \mathrm{parent}[u]) - \mathrm{value}(\mathrm{parent}[u], u)$ counts all $v \to$ parent[v] edges traversed after visiting u for the first time and before leaving u for the last time.
- Hence $\mathrm{size}(u) =$

  $$\begin{cases} n & u \text{ is the root,} \\ \mathrm{value}(u, \mathrm{parent}(u)) - \mathrm{value}(\mathrm{parent}(u), u) & \text{otherwise} \end{cases}$$

  is the number of descendants of u.

Arithmetic Expression Trees

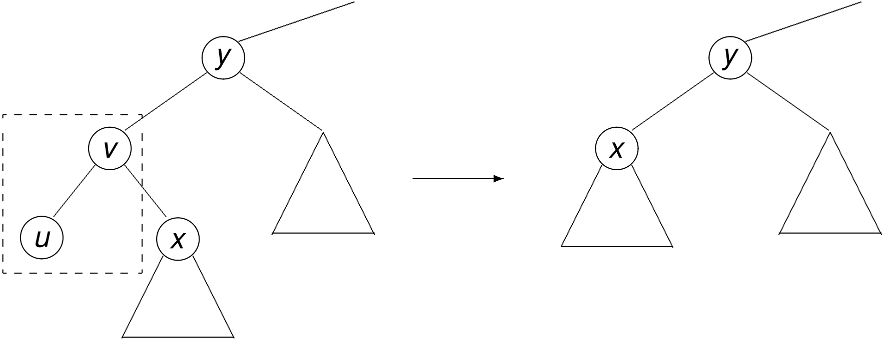
An expression tree $T = (V, E)$ is a binary tree:

- its inner nodes have degree exactly two and they are labeled with either $+, -, \cdot$ or $/$.
- Leaves are labeled by real numbers.

Evaluate the corresponding arithmetic expression.

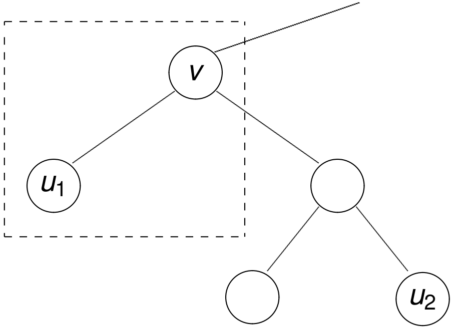
- We again assume that the tree is represented as an adjacency list with cross references.
- The approach: apply a sequence of contraction steps.
  - Each contraction removes one half of all leaves.
  - Combine evaluation with contraction
  - and we are done are logarithmically many contractions.

Contractions



- A contraction step for u:
  - Remove leaf u and its parent v.
  - Make the sibling x of u a child of grandparent y.
  - The sibling x “remembers” the value of u.
- Neither sibling x nor grandparent y may be removed by other contractions.

Collisions among Contractions



- We get collisions even if we only remove leaves in odd position.
  - if we remove leaf u_1, then v is removed as well.
  - if u_2 is removed, then grandparent v has to survive.
- The way out:
  - First remove left leaves in an odd position and then right leaves in an odd position. u_1 is removed first and afterwards u_2.
- How to determine leaves in odd position?

Finding Leaves in Odd Position

(1) Determine an Euler tour of Euler(T).

(2) A node is a leaf iff it has exactly one neighbor.

(3) Set $w(e) = \begin{cases} 1 & e = (u, l) \text{ for a leaf } l, \\ 0 & \text{otherwise;} \end{cases}$

(4) Determine the prefix sums value$(\mathrm{parent}[l], l)$ for all leaves l via list ranking.
// value$(\mathrm{parent}[l], l)$ is the number of leaves to the left of l.

(5) l is in odd position iff value$(\mathrm{parent}[l], l)$ is odd.

The Contraction Process

(1) Select all leaves in odd position.
(2) while there are more than three leaves do
  - Apply a contraction step to all left leaves in odd position and
  - then apply a contraction step to all right leaves in odd position.
    // We still have to worry about evaluating removed nodes.

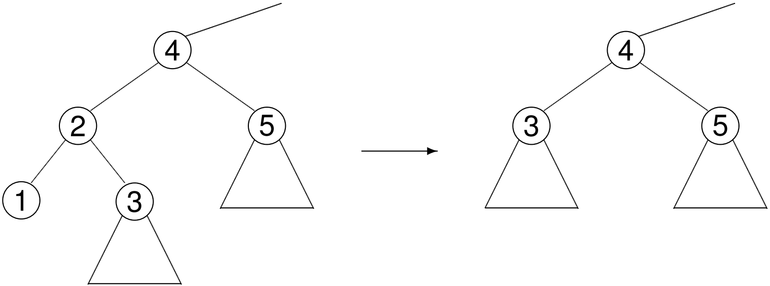
The analysis:

- After one contraction step only leaves in even position remain.
  - The number of remaining leaves is halved.
  - There are $O(\log_2 n)$ contraction steps.
- Leaves in odd positions have to be determined only once: just divide positions of the remaining leaves by two.
- The running time is bounded by $O(\log_2 n)$, if we use $\frac{n}{\log_2 n}$ processors. (Euler tours are now computed in time $O(\log_2 n)$.)

Combining Contraction and Evaluation

- All nodes compute rational functions.
- Assume that the original subtree of v computes the value x_v.
  - Throughout we represent the value of a node v by the rational function
    $$\frac{ax_v + b}{cx_v + d}.$$
  - Initially $a = d = 1$ and $b = c = 0$.
- Contraction steps change coefficients.

Combining Contraction and Evaluation: An Example



- We perform a contraction step removing leaf 1.
- The function $\frac{ax_3+b}{cx_3+d}$ of node 3 has to be modified. Node 1 computes the value *e*.
  - if node 2 divides: $\frac{e}{\frac{ax_3+b}{cx_3+d}} = \frac{(ec)x_3+(de)}{ax_3+b}$ is the new function.
  - if node 2 adds: $e + \frac{ax_3+b}{cx_3+d} = \frac{(a+ec)x_3+(b+ed)}{cx_3+d}$ is the new function.