Dynamic Trees

- Goal: maintain a forest of rooted trees with costs on vertices.
  - Each tree has a root, every edge directed towards the root.
- Operations allowed:
  - link(v, w): creates an edge between v (a root) and w.
  - cut(v, w): deletes edge (v, w).
  - findcost(v): returns the cost of vertex v.
  - findroot(v): returns the root of the tree containing v.
  - findmin(v): returns the vertex w of minimum cost on the path from v to the root (if there is a tie, choose the closest to the root).
  - addcost(v, x): adds x to the cost every vertex from v to root.

Dynamic Trees

- An example (two trees):

Dynamic Trees

- Obvious Implementation
  - A node represents each vertex;
  - Each node x points to its parent p(x):
    - link, split, findcost: constant time.
    - findroot, findmin, addcost: linear time on the size of the path.
  - Acceptable if paths are small, but O(n) in the worst case.
  - Cleverer data structures achieve O(log n) for all operations.
Simple Paths

- We start with a simpler problem:
  - Maintain set of paths subject to:
    - split: cuts a path in two;
    - concatenate: links endpoints of two paths, creating a new path.
- Operations allowed:
  - \( \text{findcost}(v) \): returns the cost of vertex \( v \);
  - \( \text{addcost}(v, x) \): adds \( x \) to the cost of vertices in path containing \( v \);
  - \( \text{findmin}(v) \): returns minimum-cost vertex path containing \( v \).

Simple Paths as Lists

- Natural representation: doubly linked list.
  - Constant time for \( \text{findcost} \).
  - Constant time for \( \text{concatenate} \) and \( \text{split} \) if endpoints given, linear time otherwise.
  - Linear time for \( \text{findmin} \) and \( \text{addcost} \).

Can we do it \( O(\log n) \) time?

Simple Paths as Binary Trees

- Alternative representation: balanced binary trees.
  - Leaves: vertices in symmetric order.
  - Internal nodes: subpaths between extreme descendants.

Compact alternative:
- Each internal node represents both a vertex and a subpath:
  - subpath from leftmost to rightmost descendant.

Simple Paths: Maintaining Costs

- Keeping costs:
  - First idea: store \( \text{cost}(x) \) directly on each vertex;
  - Problem: \( \text{addcost} \) takes linear time (must update all vertices).
  - Better approach: store \( \Delta \text{cost}(x) \) instead:
    - Root: \( \Delta \text{cost}(x) = \text{cost}(x) \)
    - Other nodes: \( \Delta \text{cost}(x) = \text{cost}(x) - \text{cost}(p(x)) \)
### Simple Paths: Maintaining Costs

- **Costs:**
  - addcost: constant time (just add to root)
  - Finding cost(x) is slightly harder: \(O(\text{depth}(x))\).

<table>
<thead>
<tr>
<th>actual costs</th>
<th>difference form</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 (v_4)</td>
<td>9 (v_4)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>4 (v_1)</td>
</tr>
<tr>
<td>3 (v_3)</td>
<td>6 (v_3)</td>
</tr>
<tr>
<td>6 (v_1)</td>
<td>4 (v_1)</td>
</tr>
<tr>
<td>2 (v_2)</td>
<td>6 (v_2)</td>
</tr>
<tr>
<td>3 (v_5)</td>
<td>6 (v_5)</td>
</tr>
<tr>
<td>7 (v_3)</td>
<td>3 (v_3)</td>
</tr>
<tr>
<td>9 (v_1)</td>
<td>3 (v_1)</td>
</tr>
<tr>
<td>3 (v_6)</td>
<td>4 (v_6)</td>
</tr>
</tbody>
</table>

**Costs:** 6 2 3 4 7 9 3

---

### Simple Paths: Structural Changes

- Concatenating and splitting paths:
  - Join or split the corresponding binary trees;
  - Time proportional to tree height.
  - For balanced trees, this is \(O(\log n)\).
    - Rotations must be supported in constant time.
    - We must be able to update \(\Delta \text{min}\) and \(\Delta \text{cost}\).

---

### Simple Paths: Finding Minima

- Still have to implement findmin:
  - Store \(\min\text{cost}(x)\), the minimum cost on subpath with root \(x\).
  - findmin runs in \(O(\log n)\) time, but addcost is linear.

<table>
<thead>
<tr>
<th>actual costs</th>
<th>mincost</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 (v_4)</td>
<td>2 (v_4)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>3 (v_1)</td>
</tr>
<tr>
<td>5 (v_3)</td>
<td>6 (v_3)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>3 (v_1)</td>
</tr>
<tr>
<td>2 (v_2)</td>
<td>4 (v_2)</td>
</tr>
<tr>
<td>3 (v_5)</td>
<td>2 (v_5)</td>
</tr>
<tr>
<td>7 (v_3)</td>
<td>3 (v_3)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>3 (v_1)</td>
</tr>
<tr>
<td>6 (v_1)</td>
<td>4 (v_1)</td>
</tr>
</tbody>
</table>

**Costs:** 6 2 3 4 7 9 3

---

### Simple Paths: Data Fields

- Final version:
  - Stores \(\Delta \text{min}(x)\) and \(\Delta \text{cost}(x)\) for every vertex

<table>
<thead>
<tr>
<th>actual costs</th>
<th>((\Delta \text{cost}, \Delta \text{min}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 (v_4)</td>
<td>(3, 0)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>5 (v_3)</td>
<td>(4, 0)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>2 (v_2)</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>3 (v_5)</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>7 (v_3)</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>6 (v_1)</td>
<td>(1, 0)</td>
</tr>
</tbody>
</table>

**Costs:** 6 2 3 4 7 9 3

---

### Simple Paths: Finding Minima

- Store \(\Delta \text{min}(x) = \text{cost}(x) - \min\text{cost}(x)\) instead.
  - findmin still runs in \(O(\log n)\) time, addcost now constant.

<table>
<thead>
<tr>
<th>actual costs</th>
<th>(\Delta \text{min})</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 (v_4)</td>
<td>2 (v_4)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>3 (v_1)</td>
</tr>
<tr>
<td>5 (v_3)</td>
<td>6 (v_3)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>3 (v_1)</td>
</tr>
<tr>
<td>2 (v_2)</td>
<td>4 (v_2)</td>
</tr>
<tr>
<td>3 (v_5)</td>
<td>2 (v_5)</td>
</tr>
<tr>
<td>7 (v_3)</td>
<td>3 (v_3)</td>
</tr>
<tr>
<td>2 (v_1)</td>
<td>3 (v_1)</td>
</tr>
<tr>
<td>6 (v_1)</td>
<td>4 (v_1)</td>
</tr>
</tbody>
</table>

**Costs:** 6 2 3 4 7 9 3

---

### Simple Paths: Structural Changes

- Restructuring primitive: rotation.

- Fields are updated as follows (for left and right rotations):
  - \(\Delta \text{cost}'(v) = \Delta \text{cost}(v) + \Delta \text{cost}(uv)\)
  - \(\Delta \text{cost}'(u) = -\Delta \text{cost}(v)\)
  - \(\Delta \text{cost}'(b) = \Delta \text{cost}(v) + \Delta \text{cost}(b)\)
  - \(\Delta \text{min}'(u) = \max(0, \Delta \text{min}(b) - \Delta \text{cost}'(b), \Delta \text{min}(c) - \Delta \text{cost}(c))\)
  - \(\Delta \text{min}'(c) = \max(0, \Delta \text{min}(a) - \Delta \text{cost}(a), \Delta \text{min}(u) - \Delta \text{cost}'(u))\)
**Splaying**

- Simpler alternative to balanced binary trees: splaying.
  - Does not guarantee that trees are balanced in the worst case.
  - Guarantees $O(\log n)$ access in the amortized sense.
  - Makes the data structure much simpler to implement.
- Basic characteristics:
  - Does not require any balancing information;
  - On an access to $v$, splay on $v$:
    - Moves $v$ to the root;
    - Roughly halves the depth of other nodes in the access path.
  - Based entirely on rotations.
- Other operations (insert, delete, join, split) use splay.

**Three restructuring operations:**

- **zig**
  
- **zigzag** (only happens if $y$ is root)

---

**An Example of Splaying**

---

---

---

---
An Example of Splaying

A vertex is splayed by performing a zig-zag sequence at the root of the tree.

End result:

- The splayed vertex is brought to the root of the tree.

Amortized Analysis

- Bounds the running time of a sequence of operations.
- Potential function $\Phi$ maps each configuration to a real number.
- Amortized time to execute each operation:
  - $a_i = t_i + \Phi_i - \Phi_{i-1}$
  - $\Phi_i$: potential after the $i$-th operation;
  - $t_i$: actual time to execute the operation;
- Total time for $m$ operations:
  \[ \sum_{i=1}^{m} a_i = \sum_{i=1}^{m}(t_i + \Phi_i - \Phi_{i-1}) = \Phi_m - \Phi_1 + \sum_{i=1}^{m} t_i \]

Amortized Analysis of Splaying

- Definitions:
  - $s(x)$: size of node $x$ (number of descendants, including $x$);
  - At most $n$, by definition.
  - $r(x)$: rank of node $x$, defined as $\log s(x)$;
  - At most $\log n$, by definition.
  - $\Phi$: potential of the data structure (twice the sum of all ranks).
  - At most $O(n \log n)$, by definition.
- Access Lemma [ST85]: The amortized time to splay a tree with root $t$ at a node $x$ is at most
  \[ 6(r(t) - r(x)) + 1 = O(\log(s(t)/s(x))). \]

Proof of Access Lemma

- Access Lemma [ST85]: The amortized time to splay a tree with root $t$ at a node $x$ is at most
  \[ 6(r(t) - r(x)) + 1 = O(\log(s(t)/s(x))). \]

Proof of Access Lemma: Zig-zig

- Claim:
  \[ a_i = t_i + \Phi_i - \Phi_{i-1} \]
  \[ \Phi_i = \Phi_{i-1} + \log(n) + [6(r_i(x) - r_{i-1}(x)) + 1] \]
  \[ = 6r_i(x) - 6r_{i-1}(x) + 1 \]

Proof of Access Lemma: Zig-zag

- Claim:
  \[ a_i = t_i + \Phi_i - \Phi_{i-1} \]
  \[ \Phi_i = \Phi_{i-1} + \log(n) + (6r_i(x) - 2r_{i-1}(x)) \]
  \[ = 6r_i(x) - 2r_{i-1}(x) + 1 \]

Proof of Access Lemma: Splaying Step

- Zig-zig:
  \[ a_i = t_i + \Phi_i - \Phi_{i-1} \]
  \[ \Phi_i = \Phi_{i-1} + \log(n) + (6r_i(x) - 2r_{i-1}(x)) \]
  \[ = 6r_i(x) - 2r_{i-1}(x) + 1 \]
  TRUE because $s(x)/s'(x) < 1$: both ratios are smaller than 1, at least one is at most $1/2$.

Proof of Access Lemma: Zig

- Claim:
  \[ a_i = t_i + \Phi_i - \Phi_{i-1} \]
  \[ \Phi_i = \Phi_{i-1} + \log(n) + (6r_i(x) - r_{i-1}(x)) \]
  \[ = 6r_i(x) - r_{i-1}(x) + 1 \]
  TRUE because $s(x)/s'(x) < 1$: both ratios are smaller than 1, at least one is at most $1/2$. 
Splaying

• To sum up:
  • No rotation: \( a = 1 \)
  • Zig: \( a \leq 6 (r'(x) - r(x)) + 1 \)
  • Zig-zig: \( a \leq 6 (r'(x) - r(x)) \)
  • Zig-zag: \( a \leq 4 (r'(x) - r(x)) \)
  • Total amortized time at most \( 6 (r(t) - r(x)) + 1 = O(\log n) \)

• Since accesses bring the relevant element to the root, other operations (insert, delete, join, split) become trivial.

Dynamic Trees

• We know how to deal with isolated paths.

• How to deal with paths within a tree?

Dynamic Trees

• Main idea: partition the vertices in a tree into disjoint solid paths connected by dashed edges.

Dynamic Trees

• A vertex \( v \) is exposed if:
  • There is a solid path from \( v \) to the root;
  • No solid edge enters \( v \).
Dynamic Trees

- Solid paths:
  - Represented as binary trees (as seen before).
  - Parent pointer of root is the outgoing dashed edge.
  - Hierarchy of solid binary trees linked by dashed edges: "virtual tree".
- "Isolated path" operations handle the exposed path.
  - The solid path entering the root.
  - Dashed pointers go up, so the solid path does not "know" it has dashed children.
- If a different path is needed:
  - expose(v): make entire path from v to the root solid.

Virtual Tree: An Example

- Example: expose(v)
  - Take all edges in the path to the root, ...
  - make sure there is no other solid edge incident into the path.
  - Uses splice operation.
Exposing a Vertex

- expose(x): makes the path from x to the root solid.
- Implemented in three steps:
  1. Splay within each solid tree in the path from x to root.
  2. Splice each dashed edge from x to the root.
     - splice makes a dashed become the left solid child;
     - If there is an original left solid child, it becomes dashed.
  3. Splay on x, which will become the root.

Dynamic Trees: Splice

- Additional restructuring primitive: splice.

  • Will only occur when x is the root of a tree.
  • Updates:
    - Δcost'(v) = Δcost(v) − Δcost(x)
    - Δcost'(u) = Δcost(u) + Δcost(x)
    - Δmin(x) = max(0, Δmin(v) − Δcost'(v), Δmin(x) − Δcost(x))

Exposing a Vertex: An Example

- expose(a)

Exposing a Vertex: Running Time

- Running time of expose(x):
  - proportional to initial depth of x;
  - x is rotated all the way to the root;
  - we just need to count the number of rotations;
    - will actually find amortized number of rotations: O(log n).
  - proof uses the Access Lemma.
    - s(x), r(x) and potential are defined as before;
    - In particular, s(x) is the size of the whole subtree rooted at x.
      - Includes both solid and dashed edges.

Exposing a Vertex: Running Time (Proof)

- k: number of dashed edges from x to the root t.
- Amortized costs of each pass:
  1. Splay within each solid tree:
     - x_i: vertex splayed on the i-th solid tree.
     - amortized cost of i-th splay: 6(r_i(x_i) − r(x_i)) + 1.
  2. Splice dashed edges:
     - no rotations, no potential changes: amortized cost is zero.
  3. Splay on x:
     - amortized cost is at most 6log n + 1.
     - x ends up in root, so exactly k rotations happen;
     - each rotation costs one credit, but is charged two;
     - they pay for the extra k rotations in the first pass.
- Amortized number of rotations = O(log n).

Implementing Dynamic Tree Operations

- findcost(v):
  - expose v, return cost(v).
- findroot(v):
  - expose v;
  - find w, the rightmost vertex in the solid subtree containing v;
  - splay at w and return w.
- findmin(v):
  - expose v;
  - use Δcost and Δmin to walk down from v to w, the last minimum-cost node in the solid subtree;
  - splay at w and return w.
Implementing Dynamic Tree Operations

- addcost(v, x):
  - expose v;
  - add x to ∆cost(v);
- link(v, w):
  - expose v and w (they are in different trees);
  - set p(v) = w (that is, make v a middle child of w).
- cut(v):
  - expose v;
  - add ∆cost(v) to ∆cost(right(v));
  - make p(right(v)) = null and right(v) = null.

Extensions and Variants

- Simple extensions:
  - Associate values with edges:
    - just interpret cost(v) as cost(v, p(v)).
  - other path queries (such as length):
    - change values stored in each node and update operations.
  - free (unrooted) trees.
    - implement ever operation, which changes the root.
- Not-so-simple extension:
  - subtree-related operations:
    - requires that vertices have bounded degree;
    - Approach for arbitrary trees: “ternarize” them:
      - [Goldberg, Grigoriadis and Tarjan, 1991]

Alternative Implementation

- Total time per operation depends on the data structure used to represent paths:
  - Splay trees: O(log n) amortized [ST85].
  - Balanced search tree: O(log² n) amortized [ST83].
  - Locally biased search tree: O(log n) amortized [ST83].
  - Globally biased search trees: O(log n) worst-case [ST83].
- Biased search trees:
  - Support leaves with different “weights”.
  - Some solid leaves are “heavier” because they also represent subtrees dangling from it from dashed edges.
  - Much more complicated than splay trees.