Dynamic Trees

- Motivation (Online MSTs)
- Problem Definition
- A Data Structure for Dynamic Paths
- A Data Structure for Dynamic Trees
- Extensions

Online Minimum Spanning Trees

- The online minimum spanning trees problem:
  - Input: a sequence of edges (with costs), one at a time.
  - Goal: keep the minimum spanning forest of the graph.
- An algorithm:
  - For each new edge $(v, w)$:
    - If $v$ and $w$ belong to different components, insert the edge.
    - If $v$ and $w$ are in the same component:
      - insert $(v, w)$ into the solution; and
      - remove the most expensive edge on the cycle created.
Online Minimum Spanning Trees

edge cost
(a,g) 6
(a,h) 7
(a,d) 6
(a,e) 5
(a,b) 7
(a,f) 5
(a,y) 8
(a,b) 5
→(a,x) 2
(a,x) 4
→(a,y) 4
→(a,b) 5
→(a,e) 6
→(a,g) 6

Dynamic Trees

Online Minimum Spanning Trees

edge cost
(f,g) 6
(f,h) 7
(a,d) 6
(a,e) 5
(a,b) 7
(a,f) 5
(b,y) 8
(b,h) 5
→(b,x) 2
→(b,y) 4
→(b,g) 4
→(b,e) 5
→(b,d) 6
→(b,h) 6

Dynamic Trees

Online Minimum Spanning Trees

edge cost
(f,g) 6
(f,h) 7
(a,d) 6
(a,e) 5
(a,b) 7
(a,f) 5
(b,y) 8
(b,h) 5
→(b,x) 2
→(b,y) 4
→(b,g) 4
→(b,e) 5
→(b,d) 6
→(b,h) 6

Dynamic Trees

Online Minimum Spanning Trees

edge cost
(f,g) 6
(f,h) 7
(a,d) 6
(a,e) 5
(a,b) 7
(a,f) 5
(b,y) 8
(b,h) 5
→(b,x) 2
→(b,y) 4
→(b,g) 4
→(b,e) 5
→(b,d) 6
→(b,h) 6

Dynamic Trees

Dynamic Trees
Dynamic Trees

Online Minimum Spanning Trees

- How fast is the algorithm?
  - How fast can we find the most expensive edge of a cycle?
    - $O(\log n)$, with the right data structure.
  - Total running time: $O(m \log n)$ (m edges, n vertices)

Dynamic Trees - Problem Definition

- 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)`: deletes edge (v, p(v)) (where p(v) is v’s parent).
  - `findcost(v)`: returns the cost of vertex v.
  - `findroot(v)`: returns the root of the tree containing v.
  - `findmin(v)`: returns the minimum-cost vertex w on the path from v to the root.
- A possible extension:
  - `evert(w)`: makes w the root of its tree.

Dynamic Trees

- Motivation (Online MSTs)
- Problem Definition
- A Data Structure for Dynamic Paths
- A Data Structure for Dynamic Trees
- Extensions

Dynamic Trees

- An example (two trees):
Dynamic Trees

Applications
- Used as a building block of several graph algorithms:
  - online minimum spanning trees
  - dynamic graphs
  - directed minimum spanning trees
  - network flows (e.g., maximum flow)
  - ...
Obvious Implementation of Dynamic Trees

- Each node represents a vertex.
- Each node $x$ points to its parent $p(x)$:
  - cut, link, findcost: constant time.
  - findroot, findmin: time proportional to path length.
- Acceptable if paths are small, but $O(n)$ in the worst case.
- We can get $O(\log n)$ for all operations.

Dynamic Trees

- Motivation (Online MSTs)
- Problem Definition
- A Data Structure for Dynamic Paths
- A Data Structure for Dynamic Trees
- Extensions

Dynamic Trees

- We start with a simpler problem:
- Maintain set of paths subject to the following operations:
  - split: removes an edge, cutting a path in two;
  - concatenate: links endpoints of two paths, creating a new path.
- Operations allowed:
  - findcost($v$): returns the cost of vertex $v$;
  - findmin($v$): returns minimum-cost vertex on the path containing $v$.

Simple Paths as Lists

- Natural representation: doubly-linked list:
  - Path characterized by two endpoints.
  - findcost: constant time.
  - concatenate: constant time.
  - split: constant time.
  - findmin: linear time (not good).
- Can we do it $O(\log n)$ time?

Simple Paths as Binary Trees

- Alternative representation: balanced binary tree.
  - 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
• We store cost(x) directly in each node.
  • Problem: findmin still takes linear time (must visit all vertices).
  Actual costs:
  \[
  \begin{array}{ccccccc}
  v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 \\
  6 & 2 & 3 & 4 & 7 & 9 & 3 \\
  \end{array}
  \]

Dynamic Trees

Simple Paths: Finding Minima
• Also store mincost(x), minimum cost in subpath with root x.
  • findmin(x) now runs in \(O(\log n)\) time.
  Actual costs:
  \[
  \begin{array}{ccccccc}
  v_1 & v_2 & v_4 & v_5 & v_3 & v_6 & v_7 \\
  6 & 2 & 3 & 4 & 7 & 9 & 3 \\
  \end{array}
  \]

Dynamic Trees

Simple Paths: Data Fields
• Final version:
  • Stores mincost(x) and cost(x) for every vertex x.
  Actual costs:
  \[
  \begin{array}{ccccccc}
  v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 \\
  6 & 2 & 3 & 4 & 7 & 9 & 3 \\
  \end{array}
  \]

Dynamic Trees

Simple Paths: Structural Changes
• Concatenating and splitting paths:
  • Join or split the corresponding binary trees;
  • Time proportional to tree height.
  • For balanced trees (AVL, red-black, etc.), this is \(O(\log n)\):
    • Rotations must be supported in constant time.
    • We must be able to update mincost, but that's easy:

\[
\text{mincost}'(v) = \min \{\text{cost}(v), \text{mincost}(a), \text{mincost}'(w)\}
\]

Dynamic Trees

Splaying
• Simpler alternative to balanced binary trees: splaying.
  • Trees may be unbalanced in the worst case.
  • Guarantees \(O(\log n)\) amortized access.
  • Much simpler to implement.
  • Basic characteristics:
    • Maintains no balancing information.
    • On an access to \(x\):
      • Moves \(x\) to the root;
      • Roughly halves the depth of other nodes in the access path.
    • Primitive operation: rotation.
  • All operations (insert, delete, join, split) use splaying.
**Amortized Analysis**

- Bounds the running time of a sequence of operations.
- Potential function $\Phi$ maps configurations to real numbers.
- Amortized time to execute each operation:
  - $a_i = t_i + \Phi_i - \Phi_{i-1}$
  - $t_i$: actual time to execute the operation;
  - $\Phi_i$: potential after the $i$-th 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 $x$);
  - $r(x)$: rank of node $x$, defined as $\log s(x)$;
  - $\Phi$: potential of the data structure (twice the sum of all ranks);
  - At most $2 n \log n$, 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 idea:
  - \(r(x)\) = rank of \(x\) after the \(i\)-th splay step;
  - \(a_i\) = amortized cost of the \(i\)-th splay step;
  - \(a_i = \log(r(x) - r_{x_i}(x)) + 1\) (for the zig step, if any)
  - \(a_i = \log(r(x) - r_{x_i}(x))\) (for each zig-zig or zig-zag step)
  - Total amortized time for all \(k\) steps:
    \[\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} [6(r(x) - r_{x_i}(x))] + [6(r(x) - r_{x_i}(x)) + 1] = 6r(x) - 6r(x) + 1\]

Proof of Access Lemma: Splaying Step

- Zig-zig:
  - Claim: \(a = 6(r(x) - r(y)) + 1\)
  - \(t + \Phi - \Phi \leq 6(r(x) - r(y))\)
  - \(2 + 2(r(y) - r(x)) \leq 6(r(x) - r(y))\)
  - \(r(y) - r(x) \leq 3(r(x) - r(y))\)

Proof of Access Lemma: Splaying Step

- Zig:
  - Claim: \(a = 6(r(x) - r(y))\)
  - \(t + \Phi - \Phi \leq 6(r(x) - r(y))\)
  - \(2 + 2(r(y) - r(x)) \leq 6(r(x) - r(y))\)
  - \(r(y) - r(x) \leq 3(r(x) - r(y))\)

Splaying

- Summing up:
  - No rotation: \(a = 1\)
  - Zig: \(a = 6(r(x) - r(y))\)
  - Zig-zig: \(a = 6(r(x) - r(y))\)
  - Zig-zag: \(a = 6(r(x) - r(y))\)
  - Total amortized time at most \(6(r(x) - r(y)) + 1 = O(\log n)\)
  - 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?

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

A vertex $v$ is exposed if:
- There is a solid path from $v$ to the root;
- No solid edge enters $v$.

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

Virtual Tree: An Example
Dynamic Trees

- Example: expose(y)
  - Take all edges on the path to the root, ...

- Uses splice operation.

Exposing a Vertex

- expose(y): makes the path from y to the root solid.
- Implemented in three steps:
  1. Splay within each solid tree on the path from x to root.
  2. Splice each dashed edge from x to the root.
     - splice replaces left solid child with dashed child;
  3. Splay on x, which will become the root.

- expose(y): (1) splay within each solid tree;
- Does not change the partition into solid paths.
Exposing a Vertex: An Example

- `expose(y)`: (2) splice on all vertices from `y` to the root.
  - Original exposed path: `(q l f c b a)`
  - New exposed path: `(y v u t s m j g d c b a)`

Dynamic Trees: Splice

- Additional restructuring primitive: `splice`.
  - Dashed child becomes solid, replaces left child.

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: vertex splayed on the i-th solid tree.
     - amortized cost of i-th splay: `6(\delta(x_i) - r(x_i)) + 1` (Access Lemma)
     - `\delta(x_i)`, `r(x_i)`: as the sum over all steps telescopes.
     - amortized cost first of pass: `6(\delta(x_k) - r(x_k)) + 6 \log n + k`.
  2. Splice dashed edges:
     - no rotations, no changes in potential: amortized cost is zero.
  3. Splay on `w`:
     - amortized cost is at most `6 \log n + k`.
     - `w` 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 `mincost` to walk down from `v` to `u`, the rightmost minimum-cost node in the solid subtree containing `v`;
  - splay at `w` and return `w`.
Implementing Dynamic Tree Operations

- 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;
  - make p(right(v)) = null and right(v) = null;
  - set mincost(v) = \min\{cost(v), mincost(left(v))\}.

Alternative Implementations

- Total time per operation depends on path representation:
  - Splay trees: \(O(\log n)\) amortized [Sleator and Tarjan, 85].
  - Balanced search trees: \(O(\log^2 n)\) amortized [ST83].
  - Locally biased search trees: \(O(\log n)\) 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 dashed subtrees.
  - Much more complicated than splay trees.

Dynamic Trees

- Motivation (Online MSTs)
- Problem Definition
- A Data Structure for Dynamic Paths
- A Data Structure for Dynamic Trees
- Extensions

Network Flow Applications

- Augmenting path:
  - path from source to sink with positive residual capacity \(C\).

Network Flow Applications

- Augmenting path:
  - path from source to sink with positive residual capacity \(C\).
  - Flow can be sent along this path (as much as \(C\)).
  - Residual capacity of each arc decreases by \(C\).
### Dynamic Trees

#### Network Flow Applications
- Augmenting path:
  - path from source to sink with positive residual capacity $C$.
- Flow can be sent along this path (as much as $C$).
- Residual capacity of each arc decreases by $C$.
- Maximum flow algorithms usually maintain only a tree.
- $\text{findmin}(s)$ can determine the residual capacity $C$.

How can we decrease the capacities?

#### Extension: Adding Costs
- $\text{addcost}(v,x)$: adds $x$ to the cost of each vertex on the path from $v$ to the root.

### Adding Costs to Dynamic Paths
- Corresponding operation on dynamic paths:
  - $\text{addcost}(v,x)$: adds $x$ to the cost of vertices in path containing $v$;
  - current representation takes linear time.

#### Adding Costs to Dynamic Paths
- Better approach is to store $\Delta \text{cost}(x)$ instead (difference form):
  - Root: $\Delta \text{cost}(x) = \text{cost}(x)$
  - Other nodes: $\Delta \text{cost}(x) = \text{cost}(x) - \text{cost}(p(x))$

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

#### Adding Costs to Dynamic Paths
- Still have to implement $\text{findmin}$:
  - Cannot store $\text{mincost}(x)$ explicitly ($\text{addcost}$ would be linear).

### Dynamic Trees

<table>
<thead>
<tr>
<th>Network Flow Applications</th>
<th>Extension: Adding Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Augmenting path:</td>
<td>- \text{addcost}(v,x): adds $x$ to the cost of each vertex on the path from $v$ to the root.</td>
</tr>
<tr>
<td>- Flow can be sent along this path (as much as $C$).</td>
<td></td>
</tr>
<tr>
<td>- Residual capacity of each arc decreases by $C$.</td>
<td></td>
</tr>
<tr>
<td>- Maximum flow algorithms usually maintain only a tree.</td>
<td></td>
</tr>
<tr>
<td>- $\text{findmin}(s)$ can determine the residual capacity $C$;</td>
<td></td>
</tr>
<tr>
<td>- How can we decrease the capacities?</td>
<td></td>
</tr>
</tbody>
</table>

#### Adding Costs to Dynamic Paths
- Corresponding operation on dynamic paths:
  - $\text{addcost}(v,x)$: adds $x$ to the cost of vertices in path containing $v$;
  - current representation takes linear time.

#### Adding Costs to Dynamic Paths
- Better approach is to store $\Delta \text{cost}(x)$ instead (difference form):
  - Root: $\Delta \text{cost}(x) = \text{cost}(x)$
  - Other nodes: $\Delta \text{cost}(x) = \text{cost}(x) - \text{cost}(p(x))$
Dynamic Trees

Adding Costs to Dynamic Paths

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

- \text{actual costs}:

\[
\begin{array}{cccccccc}
2 & 7 & 3 & 3 & 2 & 3 & 4 & 2 \\
\end{array}
\]

- \text{mincost}:

\[
\begin{array}{cccccccc}
5 & 7 & 5 & 7 & 2 & 7 & 5 & 7 \\
\end{array}
\]

- \text{costs}:

\[
\begin{array}{cccccccc}
6 & 2 & 3 & 4 & 7 & 9 & 3 & 2 \\
\end{array}
\]

Adding Costs to Dynamic Paths: Operations

- \text{findcost}(v):
  - expose \( v \), return \( \Delta \text{cost}(v) \).

- \text{findroot}(v):
  - expose \( v \);
  - \text{find} \( w \), the rightmost vertex in the solid subtree containing \( v \);
  - \text{spay} at \( w \) and return \( w \).

- \text{findmin}(v):
  - expose \( v \);
  - use \( \Delta \text{cost} \) and \( \Delta \text{min} \) to walk down from \( v \) to \( u \), the last minimum-cost node in the solid subtree;
  - \text{spay} at \( u \) and return \( u \).

Adding Costs to Dynamic Paths: Updating Fields

- Updating fields during rotations:

  \[
  \begin{align*}
  \Delta \text{cost}(v) &= \Delta \text{cost}(v) + \Delta \text{cost}(u) \\
  \Delta \text{cost}(u) &= \Delta \text{cost}(v) \\
  \Delta \text{cost}(b) &= \Delta \text{cost}(v) + \Delta \text{cost}(b) \\
  \Delta \text{min}(u) &= \max(\text{null}, \Delta \text{min}(b) - \Delta \text{cost}(b), \Delta \text{min}(c) - \Delta \text{cost}(c)) \\
  \Delta \text{min}(v) &= \max(\text{null}, \Delta \text{min}(u) - \Delta \text{cost}(a), \Delta \text{min}(w) - \Delta \text{cost}(u))
  \end{align*}
  \]

- \text{Final version}:
  - Store \( \Delta \text{min}(x) \) and \( \Delta \text{cost}(x) \) on each node.

- \text{actual costs}:

\[
\begin{array}{cccccccc}
8 & 7 & 1 & 1 & 2 & 3 & 5 & 3 \\
\end{array}
\]

- \text{Final version}:
  - \text{Store} \( \text{actual costs}, \Delta \text{cost}, \Delta \text{min} \) on each node.

- \text{costs}:

\[
\begin{array}{cccccccc}
8 & 7 & 1 & 1 & 2 & 3 & 5 & 3 \\
\end{array}
\]

Adding Costs: Updating Fields

- Updating fields during \text{splice}:

  \[
  \begin{align*}
  \Delta \text{cost}(v) &= \Delta \text{cost}(v) - \Delta \text{cost}(v) \\
  \Delta \text{cost}(u) &= \Delta \text{cost}(v) + \Delta \text{cost}(a) \\
  \Delta \text{min}(u) &= \max(\text{null}, \Delta \text{min}(v) - \Delta \text{cost}(v), \Delta \text{min}(x) - \Delta \text{cost}(x))
  \end{align*}
  \]

  - Recall that \( w \) is always the root of a solid tree.

Dynamic Trees

Adding Costs: Operations

- \text{addcost}(v, x):
  - expose \( v \);
  - add \( x \) to \( \Delta \text{cost}(v) \), subtract \( x \) from \( \Delta \text{cost}(\text{left}(v)) \);
  - \text{link}(v, w):
    - expose \( v \) and \( w \) (they are in different trees);
    - set \( p(v) = w \) (that is, make \( v \) a middle child of \( w \)).

- \text{cut}(v):
  - expose \( v \);
  - add \( \Delta \text{cost}(v) \) to \( \Delta \text{cost}(\text{right}(v)) \);
  - make \( p(\text{right}(v)) = \text{null} \) and \( \text{right}(v) = \text{null} \).

  - set \( \Delta \text{min}(v) = \max(\text{null}, \Delta \text{min}(\text{left}(v)) - \Delta \text{cost}(\text{left}(v))) \)

Dynamic Trees
Another Extension: Change Root

- `evert(q)`: makes `q` the root of its tree
  - In the virtual tree: reverse left-right pointers:
    - This can be done implicitly with a reverse bit.
    - Must be stored in difference form (meaning depends on parents).

Another Extension: Change Root

- `evert(q)`: makes `q` the root of its tree
  - Make sure `q` is exposed, reverse solid path.

Other Extensions

- Associate values with edges:
  - Just interpret `cost(v)` as `cost(v, p(v))`.
- Other path queries (such as length):
  - Modify values stored in each node appropriately.
- Free (unrooted) trees: use `evert` to change root.
- Subtree-related operations:
  - Can be implemented, but parent must have access to middle children in constant time:
    - Tree must have bounded degree.
  - Approach for arbitrary trees: “ternarize” them:
    - [Goldberg, Grigoriadis and Tarjan, 1991]