Design and Analysis of Data Structures for Dynamic Trees

Renato Werneck
Advisor: Robert Tarjan
Outline

⇒ The Dynamic Trees problem
  • Existing data structures
  • A new worst-case data structure
  • A new amortized data structure
  • Experimental results
  • Final remarks
The Dynamic Trees Problem

- Dynamic trees:
  - Goal: maintain an \( n \)-vertex forest that changes over time.
    * \textbf{link}(v, w): adds edge between \( v \) and \( w \).
    * \textbf{cut}(v, w): deletes edge \((v, w)\).
  - Application-specific data associated with vertices/edges:
    * updates/queries can happen in bulk (entire paths or trees at once).

- Concrete examples:
  - Find minimum-weight edge on the path between \( v \) and \( w \).
  - Add a value to every edge on the path between \( v \) and \( w \).
  - Find total weight of all vertices in a tree.

- Goal: \( O(\log n) \) time per operation.
Applications

- Subroutine of network flow algorithms
  - maximum flow
  - minimum cost flow
- Subroutine of dynamic graph algorithms
  - dynamic biconnected components
  - dynamic minimum spanning trees
  - dynamic minimum cut
- Subroutine of standard graph algorithms
  - multiple-source shortest paths in planar graphs
  - online minimum spanning trees
- ...

Application: Online Minimum Spanning Trees

- Problem:
  - Graph on \( n \) vertices, with new edges “arriving” one at a time.
  - Goal: maintain the minimum spanning forest (MSF) of \( G \).

- Algorithm:
  - Edge \( e = (v, w) \) with length \( \ell(e) \) arrives:
    1. If \( v \) and \( w \) in different components: insert \( e \);
    2. Otherwise, find longest edge \( f \) on the path \( v \cdots w \):
      * \( \ell(e) < \ell(f) \): remove \( f \) and insert \( e \).
      * \( \ell(e) \geq \ell(f) \): discard \( e \).
Example: Online Minimum Spanning Trees

- Current minimum spanning forest.
Example: Online Minimum Spanning Trees

- Edge between different components arrives.
Example: Online Minimum Spanning Trees

- Edge between different components arrives:
  - add it to the forest.
Example: Online Minimum Spanning Trees

- Edge within single component arrives.
Example: Online Minimum Spanning Trees

- Edge within single component arrives:
  - determine path between its endpoints.
Example: Online Minimum Spanning Trees

- Edge within single component arrives:
  - find longest edge on the path between its endpoints.
Example: Online Minimum Spanning Trees

- Edge within single component arrives:
  - if longest edge on the path longer than new edge, swap them.
Online Minimum Spanning Trees

- Data structure must support the following operations:
  - add an edge;
  - remove an edge;
  - decide whether two vertices belong to the same component;
  - find the longest edge on a specific path.

- Each in $O(\log n)$ time:
  - basic strategy: map arbitrary tree onto balanced tree.
Data Structures for Dynamic Trees

• Different applications require different operations.

• Desirable features of a data structure for Dynamic Trees:
  – low overhead (fast);
  – simple to implement;
  – intuitive interface (easy to adapt, specialize, modify);
  – general:
    * path and tree queries;
    * no degree constraints;
    * rooted and unrooted trees.
Outline

- The Dynamic Trees problem
- Existing data structures
  - A new worst-case data structure
  - A new amortized data structure
- Experimental results
- Final remarks
Main Strategies

- **Path decomposition:**
  - ST-trees [Sleator and Tarjan 83, 85];
    * also known as link-cut trees or dynamic trees.

- **Tree contraction:**
  - Topology trees [Frederickson 85];
  - Top trees [Alstrup, Holm, de Lichtenberg, and Thorup 97];
  - RC-trees [Acar, Blelloch, Harper, Vittes, and Woo 04].

- **Linearization:**
  - ET-trees [Henzinger and King 95, Tarjan 97];
  - Less general: cannot handle path queries.
Path Decomposition

- **ST-trees** [Sleator and Tarjan 83, 85]:
  - partition the tree into vertex-disjoint paths;
  - represent each path as a binary tree;
  - “glue” the binary trees appropriately.
Path Decomposition

- **ST-trees** [Sleator and Tarjan 83, 85]:
  - partition the tree into vertex-disjoint paths;
  - represent each path as a binary tree;
  - "glue" the binary trees appropriately.

- **Complexity:**
  - with ordinary balanced trees: $O(\log^2 n)$ worst-case;
  - with globally biased trees: $O(\log n)$ worst-case;
  - with splay trees: $O(\log n)$ amortized, but much simpler.

- **Main features:**
  - relatively low overhead (one node per vertex/edge);
  - adapting to different applications requires knowledge of inner workings;
  - tree-related queries require bounded degrees (or ternarization).
Contractions: Rake and Compress

- Proposed by Miller and Reif [1985] (parallel setting).

- **Rake:**
  - eliminates a degree-one vertex;
  - edge combined with successor:
    * assumes circular order.

- **Compress:**
  - eliminates a degree-two vertex;
  - combines two edges into one.

- Original and resulting edges are clusters:
  - cluster represents both a path and a subtree;
  - user defines what to store in the new cluster.
Contractions

- Contraction:
  - series of rakes and compresses;
  - reduces tree to a single cluster (edge).

- Any order of rakes and compresses is “right”:
  - final cluster will have the correct information;
  - data structure decides which moves to make:
    * just “asks” the user how to update values after each move.

- Example:
  - work in rounds;
  - perform a maximal set of independent moves in each round.
Contractions: Example
Contractions: Example
Contractions: Example
Contractions: Example
Contractions: Example
Contractions: Example
Contractions: Example

e -> i

i -> n
Contractions: Example
Top Trees: Example
Top Trees: Example
Top Trees: Example
Top Trees: Example
Top Trees: Example
**Contractions: Updating Values**

- To find the minimum-cost edge on the tree:
  - rake: $C \leftarrow \min\{A, B\}$
  - compress: $C \leftarrow \min\{A, B\}$

- To find the maximum edge on a path:
  - rake: $C \leftarrow B$
  - compress: $C \leftarrow \max\{A, B\}$

- To find the length of a path:
  - rake: $C \leftarrow B$
  - compress: $C \leftarrow A + B$
Contractions: Example with Values
Top Trees: Path Lengths
Top Trees

- Top tree embodies a contraction:
  - top tree root represents the entire original tree;
  - \texttt{expose}(v, w) ensures that path $v \cdots w$ is represented at the root;
  - interface only allows direct access to the root.

- Other operations:
  - \texttt{link}: joins two top trees;
  - \texttt{cut}: splits a top tree in two;
  - Goal: make \texttt{link}, \texttt{cut}, and \texttt{expose} run in $O(\log n)$ time.

- Known implementation [Alstrup et al. 97]:
  - interface to topology trees:
    * variant where clusters are vertices;
    * degree must be bounded.
Outline

- The Dynamic Trees problem
- Existing data structures
  ⇒ A new worst-case data structure
  - A new amortized data structure
- Experimental results
- Final remarks
Tree Contraction

- Contraction scheme:
  - Work in rounds, each with a maximal set of independent moves.

- Lemma: there will be at most $O(\log n)$ levels.
  - At least half the vertices have degree 1 or 2.
  - At least one third of those will disappear:
    * a move blocks at most 2 others.
  - No more than 5/6 of the original vertices will remain.
Online MSF on Augmented Stars

- Corollary: $\text{expose}(v, w)$ can be implemented in $O(\log n)$ time.
  - Temporarily eliminate all clusters with $v$ or $w$ as internal vertices:
    * at most two per level: $O(\log n)$. 
Online MSF on Augmented Stars

- Corollary: expose\((v, w)\) can be implemented in $O(\log n)$ time.
  - Temporarily eliminate all clusters with $v$ or $w$ as internal vertices:
    * at most two per level: $O(\log n)$.
  - Build a temporary top tree on the remaining root clusters.
  - Restore original tree.
Tree Contraction

Update scheme:

- Goal: minimize “damage” to original contraction after link or cut.
- For each level (bottom-up), execute two steps:
  1. replicate as many original moves as possible;
  2. perform new moves until maximality is achieved.
- Step 1 is implicit.
Updates: Example
These moves cannot be replicated

New move
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Updates: Example
Theorem: each level can be updated in $O(1)$ time.

Only need to worry about the **core**:
- connected subgraph induced by the new (active) clusters;
- remaining clusters: inactive subgraph.
Running Time

- Theorem: each level can be updated in $O(1)$ time.

- Only need to worry about the core:
  - connected subgraph induced by the new (active) clusters;
  - remaining clusters: inactive subgraph.
Running Time

- Theorem: each level can be updated in $O(1)$ time.

- Only need to worry about the core:
  - connected subgraph induced by the new (active) clusters;
  - remaining clusters: inactive subgraph.

- Suffices to prove that core size $s$ is bounded by a constant.
  - If core were a free tree, would shrink by $1/6$ between rounds.
  - Each point of contact will add $O(1)$ clusters to the core.
  - Claim: there are at most four points of contact.
  - Constant initial size + multiplicative decrease + additive increase:
    $\Rightarrow$ constant size.
Contraction-based Data Structure

- Positive aspects:
  - general (top tree interface);
  - conceptually simple algorithm;
  - direct implementation (does not use topology trees);
  - $O(\log n)$ worst case.

- Potential overheads:
  - pointers within and among levels:
    * need Euler tour of each level;
  - unmatched clusters are repeated (dummy nodes).

- Joint work with J. Holm, R. Tarjan, and M. Thorup.
Outline

- The Dynamic Trees problem
- Existing data structures
- A new worst-case data structure
  ⇒ A new amortized data structure
- Experimental results
- Final remarks
Representation

- Consider some unrooted tree:
Representation

- Make it a **unit tree**:
  - directed tree with degree-one root.
• Pick a root path:
  – starts at a leaf;
  – ends at the root.
Represent the root path as a binary tree:
- leaves: base clusters (original edges);
- internal nodes: compress clusters.
What if a vertex has degree greater than two?

- Recursively represent each subtree rooted at the vertex (at most two, because of circular order).
What if a vertex has degree greater than two?
  – Recursively represent each subtree rooted at the vertex.
  – Before vertex is compressed, rake subtrees onto adjacent cluster.
• Representation:
  – Up to four children per node (two proper ones, two foster ones)
  – Meaning: up to two rakes followed by a compress.
  – Example: $N_y = \text{compress}(\text{rake}(D, N_x), \text{rake}(E, yz)) = wz.$
How does the recursive representation work?
- Must represent subtrees rooted at the root path.
How does the recursive representation work?

- Must represent subtrees rooted at the root path.
How does the recursive representation work?

- Must represent subtrees rooted at the root path.
- Each subtree is a sequence of unit trees with a common root.
How does the recursive representation work?

- Must represent subtrees rooted at the root path.
- Each subtree is a sequence of unit trees with a common root.
- Represent each recursively.
How does the recursive representation work?
- Must represent subtrees rooted at the root path.
- Each subtree is a sequence of unit trees with a common root.
- Represent each recursively.
- Build a binary tree of rakes.
Representation

- Two different views:
  - User interface: tree contraction.
    * sequence of rakes and compresses;
    * a single tree (a top tree).
  - Implementation: path decomposition.
    * maximal edge-disjoint paths;
    * hierarchy of binary trees (rake trees/compress trees);
    * similar to ST-trees.
Full example:
Self-Adjusting Top Trees

- Topmost compress tree represents the root path:
  - determined by \texttt{expose}(v, w);
  - implementation similar to ST-trees;
  - basic operations:
    * \texttt{splay}: rebalances each binary tree;
    * \texttt{splice}: changes the partition into paths.

- Main result: \texttt{expose} takes $O(\log n)$ amortized time.

- Operations \texttt{link} and \texttt{cut} use \texttt{expose} as the main subroutine.

- Joint work with R. Tarjan [SODA’05].
Outline

• The Dynamic Trees problem
• Existing data structures
• A new worst-case data structure
• A new amortized data structure

⇒ Experimental results

• Final remarks
Experimental Results

- Data structures implemented (in C++):
  - **TOP-W**: worst-case top trees;
  - **TOP-S**: self-adjusting top trees;
  - **ET-S**: self-adjusting ET-trees;
  - **ST-V/ST-E**: self-adjusting ST-trees;
  - **LIN-V/LIN-E**: “obvious” linear-time data structure.

- Machine: 2.0 GHz Pentium IV with 1 GB of RAM.

- Applications:
  - random operations;
  - maximum flows;
  - online minimum spanning trees;
  - shortest paths.
Experimental Results: Executive Summary

- **ST-V**: fastest $O(\log n)$ algorithm;
  - **ST-E** and **ET-S**: up to twice as slow.

- **TOP-S** is 2–4 times as slow as **ST-V**.

- **TOP-W** vs. **TOP-S**:
  - **TOP-S** faster for links and cuts;
  - **TOP-W** faster for expose;
  - **TOP-S** benefits from correlated operations (splaying).

- **LIN-V/LIN-E**: fastest for paths with up to $\sim 1000$ vertices.
Experiment: Random Operations

- Setup:
  - Start with empty graph on $n$ vertices;
  - use $n - 1$ links to create a random spanning tree;
  - alternate cuts and links until $\#\text{ops} = 10n$. 

![Graph of microsseconds per operation vs. vertices for different algorithms]
Experiment: Maximum Flow

- Maximum flow:
  - Given: directed graph with $n$ vertices and $m$ edges, source $s$, sink $t$;
  - Goal: find maximum flow from $s$ to $t$.

- Algorithm:
  - Find shortest path from $s$ to $t$ with positive residual capacity;
  - send as much flow as possible, update residual capacities, repeat;
  - running time: $O(mn^2)$.

- With dynamic trees:
  - Keep “partial paths” between iterations as a forest;
  - allows augmentations in $O(\log n)$ time;
  - running time: $O(mn \log n)$. 
Layered networks:
- 4 rows, \( \lceil n/4 \rceil \) columns between \( s \) and \( t \);
- each vertex connected to 3 random vertices in the next column;
- augmenting paths have \( \Omega(n) \) arcs.
Maximum Flow on Square Meshes

- Square meshes:
  - full $\lfloor \sqrt{n} \rfloor \times \lfloor \sqrt{n} \rfloor$ grid between $s$ and $t$;
  - each grid vertex connected to all 4 neighbors;
  - augmenting paths have $\Omega(\sqrt{n})$ arcs.
**Experiment: Online Minimum Spanning Forest**

- **Online minimum spanning forest:**
  - Edges processed one at a time;
  - Edge \((v, w)\) inserted if
    * \(v\) and \(w\) in different components; or
    * \((v, w)\) is shorter than longest edge on path from \(v\) to \(w\):
      - longest edge is removed.
  - \(O(\log n)\) time per edge.

- **Kruskal** as reference algorithm:
  1. Quicksort all \(m\) edges (offline);
  2. Insert edge iff its endpoints are in different components:
     - use union-find data structure.
Online MSF on Random Graphs

- Random multigraphs:
  - $n = 4096$, edges with random endpoints and weights;
  - expected diameter: $O(\log n)$;
  - more edges $\Rightarrow$ greater percentage of exposes.
Online MSF on Augmented Stars

- Augmented stars:
  - $n = 65537$, varying number (and length) of spokes;
  - diameter: $O(65537/\#\text{spokes})$;
  - spoke edges, with length 1, processed first;
  - other edges (random), with length 2, processed later;
  - $10n$ edges in total.
Online MSF on Augmented Stars

- Augmented stars:
  - $n = 65537$, varying number (and length) of spokes;
  - diameter: $O(65537/\#\text{spokes})$. 

![Graph showing the microsseconds per edge for different number of spokes for various algorithms: TOP-S, TOP-W, ST-E, LIN-E, KRUSKAL. The graph illustrates the performance of these algorithms as the number of spokes increases.]
Online MSF and Cache Effects

- Procedure:
  - Partition vertices at random into $n/32$ components of size 32;
  - Random edges: pick random component, then random pair within it.
  - Total number of edges: $4n$. 

![Graph showing micro seconds per edge versus components]
**Experiment: Shortest Paths**

- Bellman’s single source shortest path algorithm:
  - Assign **distance label** to each vertex.
    * initially, $d(s) \leftarrow 0$ (source) and $d(v) \leftarrow \infty$ (remaining vertices).
  - In each iteration, process all arcs $(v, w)$ in fixed order:
    * if $d(w) < d(v) + \ell(v, w)$, relax $(v, w)$:
      * decrement $d(w)$ by $\Delta = d(v) + \ell(v, w) - d(w)$.
  - Stop when an iteration does not relax any arc.
  - Running time: $O(mn)$.

- Dynamic trees:
  - Maintain the current shortest path tree and current $d(\cdot)$;
  - When relaxing $(v, w)$, decrement $d(\cdot)$ for all descendants of $w$;
  - $O(mn \log n)$, but may require fewer iterations.
Experiment: Shortest Paths

- Random graph with Hamiltonian cycle:
  - $n$ arcs on Hamiltonian cycle with weight in $[1;10]$.
  - $3n$ random arcs with weight $[1;1000]$.
  - Edges processed in random order.
Experiment: Shortest Paths

- Random graph with Hamiltonian cycle:
  - Dynamic trees reduce #iterations by 60% to 75%...
  - ...but increases running times by a factor of at least 50.
Outline

- The Dynamic Trees problem
- Existing data structures
- A new worst-case data structure
- A new amortized data structure
- Experimental results

⇒ Final remarks
Summary

- Main contributions:
  - new worst-case data structure;
  - new self-adjusting data structure:
    * uses contraction and path decomposition.
  - experimental analysis.

- Future work and open problems:
  - Worst-case data structure: do we really need Euler tours?
  - Self-adjusting data structure: can we make it worst-case?
  - Hybrid data structure?
  - Extend top trees?
  - Generalize top trees (grids, planar graphs, ...)?
Thank You
# Main Approaches

<table>
<thead>
<tr>
<th></th>
<th>Top Trees</th>
<th>ST-trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>principle</td>
<td>tree contraction</td>
<td>path decomposition</td>
</tr>
<tr>
<td>general interface</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>unrestricted trees</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>representation</td>
<td>level-based</td>
<td>recursive</td>
</tr>
<tr>
<td>overhead</td>
<td>HIGH</td>
<td>LOWER</td>
</tr>
</tbody>
</table>