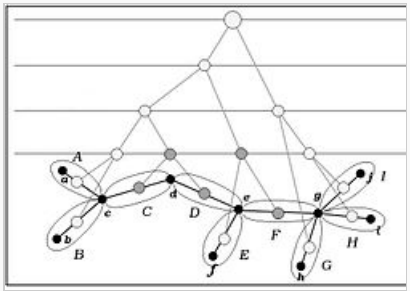
Top tree

From Wikipedia, the free encyclopedia

A **top tree** is a data structure based on a binary tree for unrooted dynamic trees that is used mainly for various path-related operations. It allows simple divide-and-conquer algorithms. It has since been augmented to maintain dynamically various properties of a tree such as diameter, center and median.

A top tree $\mathfrak{R}$ is defined for an *underlying tree* $\mathcal{T}$ and a set ∂T of at most two vertices called as External Boundary Vertices



An image depicting a top tree built on an underlying tree (black nodes)A tree divided into edge clusters and the complete top-tree for it. Filled nodes in the top-tree are path-clusters, while small circle nodes are leaf-clusters. The big circle node is the root. Capital letters denote clusters, non-capital letters are nodes.

Glossary [edit]

Boundary Node [edit]

See Boundary Vertex

Boundary Vertex [edit]

A vertex in a connected subtree is a *Boundary Vertex* if it is connected to a vertex outside the subtree by an edge.

External Boundary Vertices [edit]

Up to a pair of vertices in the top tree $\mathfrak{R}$ can be called as External Boundary Vertices, they can be thought of as Boundary Vertices of the cluster which represents the entire top tree.

Cluster [edit]

A *cluster* is a connected subtree with at most two Boundary Vertices. The set of Boundary Vertices of a given cluster $\mathcal{C}$ is denoted as ∂C. With each cluster $\mathcal{C}$ the user may associate some meta information $I(\mathcal{C})$, and give methods to maintain it under the various internal operations.

Path Cluster [edit]

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Српски / srpski
Edit links

If $\pi(\mathcal{C})$ contains at least one edge then $\mathcal{C}$ is called a *Path Cluster*.

Point Cluster [edit]

See Leaf Cluster

Leaf Cluster [edit]

If $\pi(\mathcal{C})$ does not contain any edge i.e. $\mathcal{C}$ has only one Boundary Vertex then $\mathcal{C}$ is called a *Leaf Cluster*.

Edge Cluster [edit]

A Cluster containing a single edge is called an *Edge Cluster*.

Leaf Edge Cluster [edit]

A Leaf in the original Cluster is represented by a Cluster with just a single Boundary Vertex and is called a *Leaf Edge Cluster*.

Path Edge Cluster [edit]

Edge Clusters with two Boundary Nodes are called *Path Edge Cluster*.

Internal Node [edit]

A node in $\mathcal{C} \setminus \partial C$ is called an *Internal Node* of $\mathcal{C}$.

Cluster Path [edit]

The path between the Boundary Vertices of $\mathcal{C}$ is called the *cluster path* of $\mathcal{C}$ and it is denoted by $\pi(\mathcal{C})$.

Mergeable Clusters [edit]

Two Clusters $\mathcal{A}$ and $\mathcal{B}$ are *Mergeable* if $\mathcal{A} \cap \mathcal{B}$ is a singleton set (they have exactly one node in common) and $\mathcal{A} \cup \mathcal{B}$ is a Cluster.

Introduction [edit]

Top trees are used for maintaining a Dynamic forest (set of trees) under link and cut operations.

The basic idea is to maintain a balanced Binary tree $\Re$ of logarithmic height in the number of nodes in the original tree $\mathcal{T}$ (i.e. in $\mathcal{O}(\log n)$ time) ; the **top tree** essentially represents the recursive subdivision of the original tree $\mathcal{T}$ into clusters.

In general the tree $\mathcal{T}$ may have weight on its edges.

There is a one to one correspondence with the edges of the original tree $\mathcal{T}$ and the leaf nodes of the top tree $\Re$ and each internal node of $\Re$ represents a cluster that is formed due to the union of the clusters that are its children.

The top tree data structure can be initialized in $\mathcal{O}(n)$ time.

Therefore the top tree $\Re$ over ($\mathcal{T}, \partial T$) is a binary tree such that

- The nodes of $\Re$ are clusters of ($\mathcal{T}, \partial T$);
- The leaves of $\Re$ are the edges of $\mathcal{T}$;
- Sibling clusters are neighbours in the sense that they intersect in a single vertex, and then their parent cluster is their union.
- Root of $\Re$ is the tree $\mathcal{T}$ itself, with a set of at most two External Boundary Vertices.

A tree with a single vertex has an empty top tree, and one with just an edge is just a single node.

These trees are freely augmentable allowing the user a wide variety of flexibility and productivity without going into the details of the internal workings of the data structure, something which is also referred to as the *Black Box*.

Dynamic Operations [edit]

The following three are the user allowable Forest Updates.

- **Link(v, w):** Where v and w are vertices in different trees $\mathcal{T}_1$ and $\mathcal{T}_2$. It returns a single top tree representing $\Re_v \cup \Re_w \cup (v, w)$

- **Cut(v, w)**: Removes the edge (v, w) from a tree $\mathcal{T}$ with top tree $\mathfrak{R}$, thereby turning it into two trees $\mathcal{T}_v$ and $\mathcal{T}_w$ and returning two top trees $\mathfrak{R}_v$ and $\mathfrak{R}_w$.
- **Expose(S)**: Is called as a subroutine for implementing most of the queries on a top tree. S contains at most 2 vertices. It makes original external vertices to be normal vertices and makes vertices from S the new External Boundary Vertices of the top tree. If S is nonempty it returns the new Root cluster $\mathcal{C}$ with $\partial C = S$. **Expose({v,w})** fails if the vertices are from different trees.

Internal Operations [edit]

The Forest updates are all carried out by a sequence of at most $\mathcal{O}(\log n)$ Internal Operations, the sequence of which is computed in further $\mathcal{O}(\log n)$ time. It may happen that during a tree update, a leaf cluster may change to a path cluster and the converse. Updates to top tree are done exclusively by these internal operations.

The $I(\mathcal{C})$ is updated by calling a user defined function associated with each internal operation.

- **Merge**$(\mathcal{A}, \mathcal{B})$: Here $\mathcal{A}$ and $\mathcal{B}$ are *Mergeable Clusters*, it returns $\mathcal{C}$ as the parent cluster of $\mathcal{A}$ and $\mathcal{B}$ and with boundary vertices as the boundary vertices of $\mathcal{A} \cup \mathcal{B}$. Computes $I(\mathcal{C})$ using $I(\mathcal{A})$ and $I(\mathcal{B})$.
- **Split**$(\mathcal{C})$: Here $\mathcal{C}$ is the root cluster $\mathcal{A} \cup \mathcal{B}$. It updates $I(\mathcal{A})$ and $I(\mathcal{B})$ using $I(\mathcal{C})$ and than it deletes the cluster $\mathcal{C}$ from $\mathfrak{R}$.

Split is usually implemented using **Clean**$(\mathcal{C})$ method which calls user method for updates of $I(\mathcal{A})$ and $I(\mathcal{B})$ using $I(\mathcal{C})$ and updates $I(\mathcal{C})$ such that it's known there is no pending update needed in its children. Than the $\mathcal{C}$ is discarded without calling user defined functions. **Clean** is often required for queries without need to **Split**. If Split does not use Clean subroutine, and Clean is required, its effect could be achieved with overhead by combining **Merge** and **Split**.

The next two functions are analogous to the above two and are used for base clusters.

- **Create**(v, w): Creates a cluster $\mathcal{C}$ for the edge (v, w). Sets $\partial C = \partial(v, w)$. $I(\mathcal{C})$ is computed from scratch.
- **Eradicate**$(\mathcal{C})$: $\mathcal{C}$ is the edge cluster (v, w). User defined function is called to process $I(\mathcal{C})$ and than the cluster $\mathcal{C}$ is deleted from the top tree.

Non local search [edit]

User can define **Choose**$(\mathcal{C})$: operation which for a root (nonleaf) cluster selects one of its child clusters. The top tree blackbox provides **Search**$(\mathcal{C})$: routine, which organizes **Choose** queries and reorganization of the top tree (using the Internal operations) such that it locates the only edge in intersection of all selected clusters. Sometimes the search should be limited to a path. There is a variant of nonlocal search for such purposes. If there are two external boundary vertices in the root cluster $\mathcal{C}$, the edge is searched only on the path $\pi(\mathcal{C})$. It is sufficient to do following modification: If only one of root cluster children is path cluster, it is selected by default without calling the **Choose** operation.

Examples of non local search [edit]

Finding i-th edge on longer path from v to w could be done by $\mathcal{C}$=**Expose({v,w})** followed by **Search(**$\mathcal{C}$**)** with appropriate **Choose**. To implement the **Choose** we use global variable representing v and global variable representing i. Choose selects the cluster $\mathcal{A}$ with $v \in \partial A$ iff length of $\pi(\mathcal{A})$ is at least i. To support the operation the length must be maintained in the I.

Similar task could be formulated for graph with edges with nonunit lengths. In that case the distance could address an edge or a vertex between two edges. We could define Choose such that the edge leading to the vertex is returned in the latter case. There could be defined update increasing all edge lengths along a path by a constant. In such scenario these updates are done in constant time just in root cluster. **Clean** is required to distribute the delayed update to the children. The **Clean** should be called before the **Search** is invoked. To maintain length in I would in that case require to maintain unitlength in I as well.

Finding center of tree containing vertex v could be done by finding either bicenter edge or edge with center as one endpoint. The edge could be found by $\mathcal{C}$=**Expose({v})** followed by **Search(**$\mathcal{C}$**)** with appropriate **Choose**. The choose selects between children $\mathcal{A}, \mathcal{B}$ with $a \in \partial A \cap \partial B$ the child with higher maxdistance(a). To support the operation the maximal distance in the cluster subtree from a boundary vertex should be maintained in the I. That requires maintenance of the cluster path length as well.

Interesting Results and Applications [edit]

A number of interesting applications originally implemented by other methods have been easily implemented using the top tree's interface. Some of them include

- ([SLEATOR AND TARJAN 1983]). We can maintain a dynamic collection of weighted trees in $\mathcal{O}(\log n)$ time per link and cut, supporting queries about the maximum edge weight between any two vertices in $O(\log n)$ time.
  - Proof outline: It involves maintaining at each node the maximum weight (max_wt) on its cluster path, if it is a point cluster then max_wt($\mathcal{C}$) is initialsed as $-\infty$. When a cluster is a union of two clusters then it is the maximum value of the two merged clusters. If we have to find the max wt between v and w then we do $\mathcal{C}$ = Expose(v, w), and report max_wt$(\mathcal{C})$.
- ([SLEATOR AND TARJAN 1983]). In the scenario of the above application we can also add a common weight x to all edges on a given path $v \cdots w$ in $\mathcal{O}(\log n)$ time.
  - Proof outline: We introduce a weight called extra($\mathcal{C}$) to be added to all the edges in $\pi(\mathcal{C})$. Which is maintained appropriately ; split($\mathcal{C}$) requires that, for each path child $\mathcal{A}$ of $\mathcal{C}$, we set max_wt(A) := max_wt($\mathcal{A}$) + extra($\mathcal{C}$) and extra($\mathcal{A}$) := extra($\mathcal{A}$) + extra($\mathcal{C}$). For $\mathcal{C}$:= join($\mathcal{A}, \mathcal{B}$), we set max_wt($\mathcal{C}$) := max {max_wt($\mathcal{A}$), max_wt($\mathcal{B}$)} and extra($\mathcal{C}$) := 0. Finally, to find the maximum weight on the path $v \cdots w$, we set $\mathcal{C}$:= Expose(v, w) and return max_wt($\mathcal{C}$).
- ([GOLDBERG ET AL. 1991]). We can ask for the maximum weight in the underlying tree containing a given vertex v in $\mathcal{O}(\log n)$ time.
  - Proof outline: This requires maintaining additional information about the maximum weight non cluster path edge in a cluster under the Merge and Split operations.
- The distance between two vertices v and w can be found in $\mathcal{O}(\log n)$ time as length(Expose(v, w)).
  - Proof outline:We will maintain the length length($\mathcal{C}$) of the cluster path. The length is maintained as the maximum weight except that, if $\mathcal{C}$ is created by a join(Merge), length($\mathcal{C}$) is the sum of lengths stored with its path children.
- Queries regarding diameter of a tree and its subsequent maintenance takes $\mathcal{O}(\log n)$ time.
- The Center and Median can me maintained under Link(Merge) and Cut(Split) operations and queried by non local search in $\mathcal{O}(\log n)$ time.
- The graph could be maintained allowing to update the edge set and ask queries on edge 2-connectivity. Amortized complexity of updates is $O(\log^4 n)$. Queries could be implemented even faster. The algorithm is not trivial, $I(\mathcal{C})$ uses $\Theta(\log^2 n)$ space ([HOLM, LICHTENBERG, THORUP 2000]).
- The graph could be maintained allowing to update the edge set and ask queries on vertex 2-connectivity. Amortized complexity of updates is $O(\log^5 n)$. Queries could be implemented even faster. The algorithm is not trivial, $I(\mathcal{C})$ uses $\Theta(\log^2 n)$ space ([HOLM, LICHTENBERG, THORUP 2001]).
- Top trees can be used to compress trees in a way that is never much worse than DAG compression, but may be exponentially better. [1]

Implementation [edit]

Top trees have been implemented in a variety of ways, some of them include implementation using a *Multilevel Partition* (Top-trees and dynamic graph algorithms Jacob Holm and Kristian de Lichtenberg. Technical Report), and even by using Sleator-Tarjan s-t trees (typically with amortized time bounds), Frederickson's Topology Trees (with worst case time bounds) (Alstrup et al. Maintaining Information in Fully Dynamic Trees with Top Trees).

Amortized implementations are more simple, and with small multiplicative factors in time complexity. On the contrary the worst case implementations allow speeding up queries by switching off unneeded info updates during the query (implemented by persistence techniques). After the query is answered the original state of the top tree is used and the query version is discarded.

Using Multilevel Partitioning [edit]

Any partitioning of clusters of a tree $\mathcal{T}$ can be represented by a Cluster Partition Tree CPT$(\mathcal{T})$, by replacing each cluster in the tree $\mathcal{T}$ by an edge. If we use a strategy P for partitioning $\mathcal{T}$ then the CPT would be CPT$_P$ $\mathcal{T}$. This is done recursively till only one edge remains.

We would notice that all the nodes of the corresponding top tree $\mathfrak{R}$ are uniquely mapped into the edges of this multilevel partition. There may be some edges in the multilevel partition that do not correspond to any node in the top tree, these are the edges which represent only a single child in the level below it, i.e. a simple cluster. Only the edges that correspond to composite clusters correspond to nodes in the top tree $\mathfrak{R}$.

A partitioning strategy is important while we partition the Tree $\mathcal{T}$ into clusters. Only a careful strategy ensures that we end up in an $\mathcal{O}(\log n)$ height Multilevel Partition (and therefore the top tree).

- The number of edges in subsequent levels should decrease by a constant factor.
- If a lower level is changed by an update then we should be able to update the one immediately above it using at most a constant number of insertions and deletions.

The above partitioning strategy ensures the maintenance of the top tree in $\mathcal{O}(\log n)$ time.

References [edit]

- Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup, *Maintaining information in fully dynamic trees with top trees*, ACM Transactions on Algorithms (TALG), Vol. 1 (2005), 243–264, doi:10.1145/1103963.1103966
- Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup, *Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity*, Journal of the ACM (JACM), Vol. 48 Issue 4(July 2001), 723–760, doi:10.1145/502090.502095
- Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4 . Section 2.3: Trees, pp. 308–423.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7 . Section 10.4: Representing rooted trees, pp. 214–217. Chapters 12–14 (Binary Search Trees, Red-Black Trees, Augmenting Data Structures), pp. 253–320.

1. ^ Tree Compression with Top Trees. BILLE, GOERTZ, LANDAU, WEIMANN 2013 arXiv:1304.5702 [cs.DS]

External links [edit]

- Maintaining Information in Fully Dynamic Trees with Top Trees. Alstrup et al
- Self-Adjusting Top Trees. Tarjan and Werneck, Proc. 16th SoDA, 2005

v · t · e	Tree data structures	[hide]
Search trees (dynamic sets/associative arrays)	2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · B^x · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic · (Left-leaning) Red-black · Scapegoat · Splay · T · Treap · UB · Weight-balanced	
Heaps	Binary · Binomial · Fibonacci · Leftist · Pairing · Skew · Van Emde Boas	
Tries	Hash · Radix · Suffix · Ternary search · X-fast · Y-fast	
Spatial data partitioning trees	BK · BSP · Cartesian · Hilbert R · *k*-d (implicit *k*-d) · M · Metric · MVP · Octree · Priority R · Quad · R · R+ · R* · Segment · VP · X	
Other trees	Cover · Exponential · Fenwick · Finger · Fusion · Hash calendar · iDistance · K-ary · Left-child right-sibling · Link/cut · Log-structured merge · Merkle · PQ · Range · SPQR · Top	

Categories: Binary trees