# Dynamic Connectivity

**Daniel Angelo Esteves Lawand**

**Cristina Gomes Fernandes**

IME/USP

daniel.lawand@usp.br

## Objectives

Fully dynamic connectivity in graphs [1, 8] is the problem in which a graph can undergo insertion and removal of edges and one wants to efficiently perform queries of the type: decide if two vertices are in the same component of the graph. It is easy to solve this problem, keeping the graph with adjacency lists, and using breadth first search (BFS), for instance, to answer the queries. This however takes linear time per query. This work considers the particular case of the problem where the graph is a forest and describes a non-trivial solution that answers queries in logarithmic amortized time. The solution is the basis of a similarly efficient implementation for dynamic graphs.

## Materials and Methods

The study began with the reading of section 1.1 of the Dynamic Graphics book [1] about the basic concepts of dynamic graphs and the possible data structures used to solve the problem. After this reading, link-cut trees [2] were chosen as the data structure to be studied and implemented. Lecture notes [4, 5] and some articles [2, 6] were used to understand them. Link-cut trees are based on a simpler data structure, the binary search trees (BSTs). The BSTs adopted were the splay trees [3], and the Algorithms book [7] was used to understand how splay trees work.

The first part of the work was the implementation of the data structure for the fully dynamic connectivity problem in forests, that is, in an acyclic graph, in order to make the problem simpler. The second part would be the implementation of the more general case, that is, the data structure for the fully dynamic connectivity problem in graphs. After the study, the implementation of the data structures in C language began. The splay trees were the first to be implemented, then the link-cut trees and finally the dynamic forest. The GitHub [9] platform was used for version control. In addition, Bash scripts were implemented for automated tests.

## Results

The number of vertices of the underlying graph or forest will be denoted by $n$. The implemented library that handles dynamic forests is: the **dynamicForest(*n*)** creates a forest with $n$ vertices without edges; the **addEdge(*u*, *v*)** adds an edge between vertices $u$ and $v$; the **deleteEdge(*u*, *v*)** removes the edge between $u$ and $v$; and the **connected(*u*, *v*)** returns true if $u$ and $v$ are in the same forest component and false otherwise. The **dynamicForest** routine has time cost $O(n)$ and the other operations have $O(\lg n)$ amortized cost.

The link-cut tree is a data structure used to represent forests, and maintains the dynamic forest, its library is: the **maketree()** creates a tree with a single node; the **link(*u*, *v*)** adds an edge between $u$ and $v$, assuming that $u$ is the root of its link-cut tree; the **cut(*v*)** removes the edge between $v$ and its parent in the link-cut tree, assuming that $v$ is not the root of its link-cut tree; and the **findroot(*u*)** returns the root of the link-cut tree containing $u$. The **maketree** routine runs in $O(1)$ time and the other operations have an amortized cost of $O(\lg n)$.

The implementation of the dynamic forest library from the link-cut trees library is straightforward. The **connected(*u*, *v*)** operation uses **findroot** on $u$ and $v$ and checks if they return the same root. The **addEdge(*u*, *v*)** operation executes **link(*u*, *v*)**, while the **deleteEdge(*u*, *v*)** operation chooses the

deepest between *u* and *v* in the link-cut tree to perform the **cut** operation.
The splay tree library is: the **makeSplay()** creates a splay tree with a single node; the **join(***u, v***)** unites the splay trees with roots in *u* and *v*, assuming *u* has maximum key in its splay tree and its key is smaller than all keys in the *v's* splay tree; the **split(***u***)** breaks *u*'s splay tree in two: one with all nodes with a key less than or equal to *u* and another with nodes with a key greater than *u*; and **splay(***u***)** makes node *u* root of its splay tree. The **splay** routine has an amortized time cost of O(lg *n*) and the other operations have cost of O(1).
Table 1 summarizes the efficiency of each operation of the dynamic forest for the traditional and the studied implementation. Note the trade off: to improve the efficiency of the queries, the link-cut tree implementation spends more time to add and remove edges.

Table 1: Time complexity of a traditional implementation and a link-cut tree implementation.

| Operation | Adjacency lists + BFS | Link-cut trees (amortized time) |
|---|---|---|
| addEdge | O(1) | O(lg *n*) |
| deleteEdge | O(1) | O(lg *n*) |
| connected | O(*n*) | O(lg *n*) |

## Conclusions

This work is designed to solve the problem of dynamic connectivity in a context where the manipulated graph is huge and there will be many more queries than changes in the graph. Therefore, queries with linear cost are very undesirable in this situation. So it is essential to make queries more efficient, even if it makes the cost of inserting and removing edges in the graph a little more expensive. The result is a non-trivial but extremely efficient implementation for dynamic forests, the first step towards the more general case of dynamic graphs.

## References


[1] C. Demetrescu, I. Finocchi, and G. F. Italiano. Handbook of Data Structures and Applications, chapter Dynamic Graphs. Chapman and Hall/CRC, 2004.

[2] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. In Proc. of the 13th Annual ACM Symp. on Theory of Computing (STOC), 362–391, 1983.

[3] D. D. Sleator and R. E. Tarjan. Self adjusting binary search trees. Journal of the ACM, 32(3):652–686, 1985.

[4] E. Demaine, J. Holmgren (scriber), J. Jian (scriber), M. Stepanenko (scriber), and M. Ishaque (scriber). Link-cut trees problems. Lecture Notes in Advanced Data Structures, 2012.

[5] E. Demaine and K. Lai (scriber). Dynamic graph problems. Lecture Notes in Advanced Data Structures, 2007.

[6] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In Proc. of the 27th Annual ACM Symp. on Theory of Computing (STOC), 502–516, 1995.

[7] R. Sedgewick and K. Wayne. Algorithms. Addison-Wesley, 4 edition, 2011.

[8] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. Journal of the ACM, 48(4):723, 2001.

[9] D. Lawand. Dynamic connectivity implementation. GitHub, November 2022, https://github.com/danlawand/conexidade-dinamica.