IMPLEMENTATION OF A POLY-LOG DYNAMIC CONNECTIVITY ALGORITHM

By

Nicholas Pilon

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF COMPUTER SCIENCE (HONORS)
AT
DALHOUSIE UNIVERSITY
HALIFAX, NOVA SCOTIA
SEPTEMBER 2004

© Copyright by Nicholas Pilon, 2004

DALHOUSIE UNIVERSITY
FACULTY OF
COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Computer Science for acceptance a thesis entitled "**Implementation of a Poly-Log Dynamic Connectivity Algorithm**" by **Nicholas Pilon** in partial fulfillment of the requirements for the degree of **Bachelor of Computer Science (Honors)**.

Dated: September 2004

Supervisor: ______________________
Dr. Ernst Grundke

Readers: ______________________
Dr. Jason Brown

DALHOUSIE UNIVERSITY

Date: **September 2004**

Author: **Nicholas Pilon**

Title: **Implementation of a Poly-Log Dynamic Connectivity Algorithm**

Faculty: **Computer Science**

Degree: **BSCs** Convocation: **October** Year: **2004**

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

____
Signature of Author

THE AUTHOR RESERVES OTHER PUBLICATION RIGHTS, AND NEITHER THE THESIS NOR EXTENSIVE EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

THE AUTHOR ATTESTS THAT PERMISSION HAS BEEN OBTAINED FOR THE USE OF ANY COPYRIGHTED MATERIAL APPEARING IN THIS THESIS (OTHER THAN BRIEF EXCERPTS REQUIRING ONLY PROPER ACKNOWLEDGEMENT IN SCHOLARLY WRITING) AND THAT ALL SUCH USE IS CLEARLY ACKNOWLEDGED.

Contents

List of Figures vi

Abstract vii

Acknowledgements viii

1 Motivation 1

2 Background 3

3 The Algorithm 5

3.1 Insertion 6

3.2 Deletion 6

3.3 Replacement 6

3.4 Number of Vertices 7

4 Infrastructure 8

4.1 Splay Trees 8

4.2 Euler Tour Trees 9

4.2.1 Edge Deletion 11

4.2.2 Rerooting 12

4.2.3 Insertion 13

4.3 Bit Fields 13

5 Implementation **15**

5.1 The BiGraph 15

5.2 Splay Trees 16

5.3 Euler Tour Trees 17

5.4 ConnectivityGraph 18

6 Performance **19**

6.1 Code Statistics 20

7 Application to Mobile Ad-Hoc Networks **27**

8 Conclusions and Future Work **29**

References **31**

List of Figures

4.1 Converting a tree to an Euler tour representation, with intermediate step showing the extra edges. 10
4.2 The three fundamental operations on Euler tour trees. 11
4.3 Construction of an Euler tour tree from a disconnected graph by adding one edge at a time. 12

6.1 Time per operation plotted against number of nodes for a test case with 100 nodes, 50 runs per number of nodes, steps of 2 in number of nodes. 21
6.2 The ratio of times in figure 6.1 to $\log^2 n$ plotted against number of nodes. 22
6.3 Time per operation plotted against number of nodes for a test case with 500 nodes, 500 runs per # of nodes, steps of 10 in # of nodes. . 23
6.4 The ratio of times in figure 6.3 to $\log^2 n$ plotted against number of nodes. 24
6.5 Time per operation plotted against number of nodes for a test case with 1000 nodes, 200 runs per # of nodes, steps of 20 in # of nodes. 25
6.6 The ratio of times in figure 6.5 to $\log^2 n$ plotted against number of nodes. 26

Abstract

We implement and test a fully-dynamic connectivity algorithm for bigraphs proposed by Holm, de Lichtenberg, and Thorup. For a graph with n nodes, the algorithm achieves $O\left(\log^2 n\right)$ time complexity by maintaining a spanning tree over each component of the graph. The spanning trees are stored by placing an Euler tour over the tree in a splay tree. The implementation was written in C++. Analysis of the implementation's performance in randomized trials indicate that it meets this theoretical bound.

Acknowledgements

Donald Morrison, for helping me debug assorted parts of the code and proof-reading this thesis.
Gavin Carothers, for providing computer time for running simulations and helping me debug assorted parts of the code.
Keith Slade, for incorporating the MAR code into NS2 version 2.27.
Peter Spierenburg, for passing on the dalthesis.sty file used to format this thesis.
Dr. Jason Brown, for reading this thesis and helping me find the Holm et al. algorithm and work out how Euler tour trees worked.
Dr. Michael McAllister, for refereeing the presentation of this thesis.
Dr. Ernst Grundke, for supervising me and providing lots of good advice.

Chapter 1

Motivation

The Ant Colony research group [3] is examining routing in mobile ad-hoc networks (MAnets) by means of agents. For our purposes, agents are active pieces of data that move about the network bearing routing information and are updated as they pass through nodes. The agents, known as ants, the Ant Colony are examining, carry routing information from node to node in the network, wandering through it at random. When devising and implementing the specifics of these protocols, the researchers of the Ant Colony need to be able to compare them to each other and to other wireless routing protocols. Much of the comparison has been, and will be, done by way of network simulations run using the `ns2` network simulator [1].

Although MANets, being wireless networks, do not have physical links between their nodes as traditional wired networks do, they still have conceptual links between the nodes. These links, which we represent as edges in a graph of the network, reach from each node to all nodes within radio range of it. As the nodes in the network are mobile, these links can be made and broken very often compared to the mostly-static links of wired networks.

The connectivity of this graph of the network is of great interest to us when examining a protocol. Knowing how many components the graph of the network has and what sizes those components are allows us to evaluate data from our simulations more accurately. Since connectivity in a wireless network with mobile nodes can

change in a matter of seconds, data from simulation results that seems indicative of a protocol problem may, in fact, simply indicate that the network has become disconnected.

In a test with more than a handful of nodes, or which runs for more than a couple of seconds, checking connectivity by hand is simply not feasible. Using a brute-force algorithm is also not desirable, as it must be run many times over the course of the simulation. Recalculating the graph's connectivity from scratch each time is computationally intensive.

To avoid this expensive operation, we instead decided to employ a dynamic connectivity algorithm. Instead of reanalyzing the graph every time the connectivity information is requested, these algorithms store the connectivity information and update it after each insertion or deletion of a graph edge or vertex.

Chapter 2

Background

One of the earliest ancestors to the algorithm used in this work (see section 3) was devised in 1983 by Tarjan and Vishkin. It was subsequently updated in later papers [11]. This algorithm was designed for parallel implementation, and, for a graph with n nodes and m edges, only has an $O(n)$ sequential implementation. It can be implemented to run in $O(\log n)$ time and $O(n+m)$ space on $O(n+m)$ processors, or $O\left(\frac{n^2}{p}\right)$ time and $O(n^2)$ space for any $p < \frac{n^2}{\log^2 n}$ processors. The technique employed by Tarjan and Vishkin uses a block-finding algorithm on any spanning tree, as opposed to an earlier linear-time algorithm proposed by Tarjan[10], which required a depth-first tree. They also introduce the Euler tour tree, and use it in their algorithm to compute a wide variety of information.

The next significant development on the road to our chosen algorithm was made in 1995 by Henzinger and King [4]. Their paper reintroduced the Euler tour tree, and introduced the idea of storing it in a balanced b-ary binary search tree [9]. Henzinger and King also developed the general technique used by Holm et al., of partitioning the edges of the graph into $O(\log n)$ levels. Their algorithm is randomized and, over a sequence of $\Omega(m_0)$ operations, has an expected time of $O\left(u \log^3 n\right)$ for the updates needed for connectivity with u updates and a worst-case query time of $O\left(\frac{\log n}{\log \log n}\right)$.

The splay trees that provide the best times for the operations needed by Euler tour trees were devised in 1985 by Sleator and Tarjan [9]. They sought to develop

a self-adjusting binary tree structure. Their goal was to reduce the time taken by a sequence of operations by adjusting the tree structure to optimize typical accesses. The basic operations of splay trees are not, as they are with most binary search trees, inserting, deleting, and finding nodes but, instead, accessing nodes, joining trees, and splitting trees. Other tree operations are defined in terms of these basic ones. For a discussion of the basic splay tree operations, see section 4.1.

The algorithm we have implemented was developed in 1998 by Holm, de Lichtenberg, and Thorup [5]. Their paper builds on work by Frederickson [2], Henzinger and King [4]. The work by Henzinger and King is used to develop a dynamic deterministic algorithm for connectivity and minimum spanning tree, while Frederickson's work is used for 2-edge connectivity. The algorithm of Holm et al. achieves $O\left(\log^2 n\right)$ per update and $O\left(\frac{\log n}{\log\log n}\right)$ per query with n vertices and m edges. Extensions to their basic connectivity algorithm also allow for a minimum spanning forest of a graph and the 2-edge connectivity of the graph to be calculated in $O\left(\log^4 n\right)$ time, and biconnectivity in $O\left(\log^5 n\right)$ time (corrected in [6]). Storage complexity is $O\left(m + n\log n\right)$.

The algorithm Holm et al. devised was further refined by Thorup in 2000 [12]. Instead of building on generic underlying data structures, like Euler tour trees, Thorup devised a custom data structure, the "structural forest" to store information about the spanning trees. His improvements result in a storage complexity of $O\left(m\right)$ and fully-dynamic 2-edge and biconnectivity results of $O\left(\left(\log^3 n\right)\log\log n\right)$. The bound for biconnectivity could be incorrect in light of the corrections to the original 1998 paper given in [6].

Chapter 3

The Algorithm

The graph connectivity algorithm that we have chosen to implement was developed by Holm et al. [5] and employs the techniques devised by Henzinger and King [4]. The algorithm can be extended to test for minimum spanning tree, biconnectivity, and 2-edge connectivity, but we focus on the simple version that just implements connectivity.

As with most dynamic connectivity algorithms, the Holm et al. algorithm (henceforth, "the algorithm" or "the connectivity algorithm") tracks the connectivity of the graph by managing a forest of spanning trees of the components of the graph.

Each edge e of a graph with m edges and n vertices has an integer level $l(e) \leq L = \lfloor \log_2 n \rfloor$ associated with it. These levels effectively split the forest into multiple nested forests, such that F_i is the forest induced by edges of level i, which gives us $F = F_0 \subseteq F_1 \subseteq F_2 \subseteq \ldots$. We use these levels to maintain the following two invariants. First, F is a maximal subtree with regard to edge levels. Secondly, that $n(F_i) \leq \left\lfloor \frac{n}{2^i} \right\rfloor$ holds, where $n(G)$ denotes the number of vertices in a graph G. Unlike the 1995 Henzinger and King randomized algorithm[4], we never decrease the level of an edge. Increasing a non-tree-edge's level corresponds to discovering that its endpoints are "close together" in F and, thus, that it is a better choice for replacing any deleted tree-edges.

3.1 Insertion

Insertion is the simplest of the three operations of the connectivity algorithm. Newly-inserted edges always have a level of zero. The only operation we need to do is to check if the endpoints are in different trees and, if they are, join the two trees.

3.2 Deletion

The deletion operation itself is also fairly simple. Non-tree-edges are simply deleted. Tree-edges are deleted and the subtrees they were joining are separated. Once this is done, the algorithm must check to see if there's an edge in the graph that could be used to reconnect the two components. This is done using the replacement operation, starting at the level of the edge we just deleted.

3.3 Replacement

When a tree-edge is deleted from our graph, the tree it was part of becomes disconnected. To maintain our connectivity information, we must search through the non-tree-edges adjacent to these two trees to try and find one that reconnects them. To maintain our $O\left(\log^2 n\right)$ per operation bound, we must attempt to "trim" the selection of edges we consider. To do this, we search for edges of level i that reconnect the two trees, were i initially equals the level of the deleted edge. If we don't find any level i non-tree-edges that can replace the deleted tree-edge of level i, we consider level $i-1$ edges. If $i=0$ and no replacement edges are found, we conclude that there is no non-tree-edge joining the two newly-created trees, and that the deletion has really disconnected the graph.

When the replacement algorithm determines an edge isn't usable, we must increase the level to "pay" for considering the edge in our amortized analysis. To do this, let us consider the trees T_v and T_w our newly-disconnected vertices, v and w are in. As we can assign the labels arbitrarily, let us say that $|T_v| \leq |T_w|$. Given this, we start

by increasing the level of all level i tree-edges in T_v. While iterating through the level i non-tree-edges adjacent to T_v, any we test that do not connect to a vertex in T_w have their level increased by one.

3.4 Number of Vertices

For the purposes of analysis, the algorithm is treated as if the graph has a fixed number of vertices. This does not have to be true in practice. The number of nodes can be dynamic, and the bounds are as if n was equal to the largest number of vertices ever present in the graph at once.

We can confirm this quite simply. Let us assume that we have an n-node graph with some edges in it, and wish to delete some k vertices from the graph. This involves removing all edges from the vertices, effectively isolating them. As long as we do not add edges to them again, they do not affect the connectivity of the rest of the graph in any way, and can be discarded. Our bounds on the number of levels will still be true for n, but not necessarily for $n - k$. While the bound on the number of levels may remain true in the smaller graph for some cases that involve removing a number of very high-level edges, it will not hold in general.

Likewise, when adding a vertex, it is added with no edges connecting it to the rest of the graph. Thus, we can simply treat it as if it had been there as an isolated vertex all along without affecting anything else we had done to the graph.

By combining these two results, we have shown that we can freely add vertices to and delete vertices from the graph without affecting the algorithm or its performance.

Chapter 4

Infrastructure

The connectivity algorithm uses a number of underlying data structures to efficiently store the information it needs to do its work. Here, we provide a high-level theoretical overview and analysis of these structures. The details of the implementation are given in section 5.

4.1 Splay Trees

Splay Trees are a type of efficient binary search tree. Unlike most balanced binary search trees Splay Trees are a self-adjusting data structure and, as such, do not employ a strict balancing algorithm. Instead, splay trees attempt to optimize the structure of the tree to produce a good amortized time over a sequence of operations. The connectivity algorithm does not use splay trees directly. Instead, it uses them to implement Euler tour trees [9].

The basic operations of a Splay Tree are as follows. Further details, and more in-depth complexity analysis, can be found in [9]. As with all binary search trees, these operations assume that there is some kind of ordering among the elements stored in the tree.

`access-node(`n`)`: This moves the node n to the root of the tree through a series of rotations, referred to as splaying. This is an $O(\log n)$ operation.

`join-tree(`T_1`, `T_2`)`: This joins trees T_1 and T_2. All elements in T_2 must be greater than all elements in T_1. This is an $O(\log n)$ operation.

`split-tree(`T_1`, `i`)`: Splits the tree T_1 into two trees. One of the resulting trees contains all the elements less than or equal to the element i and things greater than i. This is also an $O(\log n)$ operation.

Splay Trees also support a number of standard binary search tree operations. For convenience, we define those here that are used later in our description of the algorithm.

`tree-prev(`T`, `i`)`: Given a tree T and a node of that tree i, this finds the node of T that comes immediately before i in the ordering used to order the nodes of the tree. This is an $O(\log n)$ operation.

`tree-smallest(`T`)`: Given a tree T, this returns the smallest element of that tree. All this operation does is walk down the path from the root to the leftmost node of the tree. It is an $O(\log n)$ operation.

`tree-largest(`T`)`: Given a tree T, this returns the largest element of that tree. Like `tree-smallest`, this is an $O(\log n)$ operation.

4.2 Euler Tour Trees

In addition to the the standard vectors, sets, etc. used by the algorithm, it uses a data structure known as an Euler tour tree. An Euler tour tree represents a tree as a sequence of vertices encountered when traversing that tree while on an Euler tour, as shown in figure 4.1.

First, we replace each of the undirected edges of the tree with two directed edges, one forward and one back. Starting at the root, we then traverse these edges, adding each node to the sequence each time we visit it. This gives us a representation of the tree with $2n-1$ instances of nodes, which allows us to perform certain operations on the tree very easily.

The operations supported by Euler tour trees are:

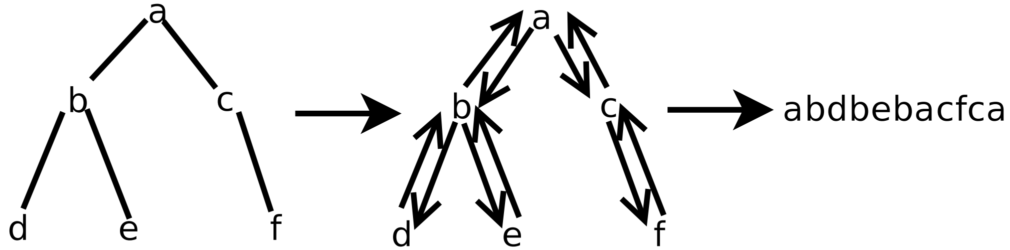

Figure 4.1: Converting a tree to an Euler tour representation, with intermediate step showing the extra edges.

1. `ett-delete-edge(`a`, `b`)`: this operation removes the edge $\{a, b\}$ from the tree, splitting it into two trees. The second tree is rooted at b.

2. `ett-join-edge(`a`, `b`)`: this operation merges two trees together by adding the edge $\{a, b\}$. As part of the operation, the second tree's root is changed to b.

3. `ett-change-root(`a`)`: this operation changes the root of the tree. The root of a tree is arbitrary, but our tour sequence differs depending on what the root is. This lets us take an existing sequence and change the root it started at.

For a visual representation of these operations, please see figure 4.2.

The connectivity algorithm uses these structures to represent spanning trees over the components of the graph. As such, we must be able to join them, split them, and search through them very quickly. If we store the sequence in a splay tree, with instances ordered by position in the sequence, then we can achieve an $O(\log n)$ bound [4, 9].

Euler tour trees stored in splay trees, as described, still fall short of the performance required for our algorithm. Explicitly storing the position of each instance in the sequence results in $O(n)$ record updates per operation.

Instead, we must order them *implicitly*. We do this by exploiting the splay tree join and split operations. When we perform an operation on an Euler tour tree, we

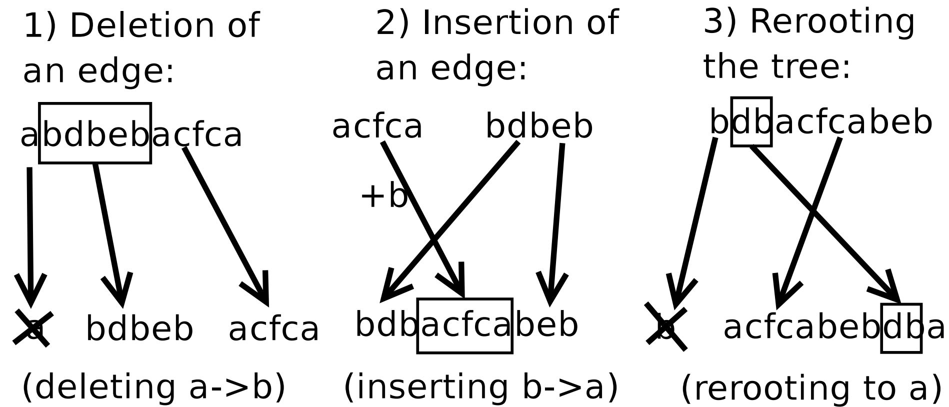

Figure 4.2: The three fundamental operations on Euler tour trees.

break apart the sequence using the `split-tree` operation and then rebuild it from left to right using `join-tree`. This implicitly orders the elements, as everything in the second tree of a join winds up being greater than everything in the first in a resulting tree.

This works if we already have trees with the necessary ordering, but is this good enough for the connectivity algorithm we are considering? As described in section 3, the algorithm starts with a completely disconnected graph and adds edges one by one. This means that our spanning trees all contain one vertex when the program starts, and are joined by repeated Euler tour `join-tree` operations, as shown in figure 4.3.

We now examine how, exactly, we can implement our Euler tour tree operations in terms of Splay Tree operations, and verify that they have the desired complexity [4].

4.2.1 Edge Deletion

Given references to a tree T_1, instances of a a_1 and a_2, and instances of b b_1, and b_2 when deleting $\{a, b\}$ from T_1, we can use the following series of operations:

T_2=`split-tree(`T_1`, ` a_1`)`

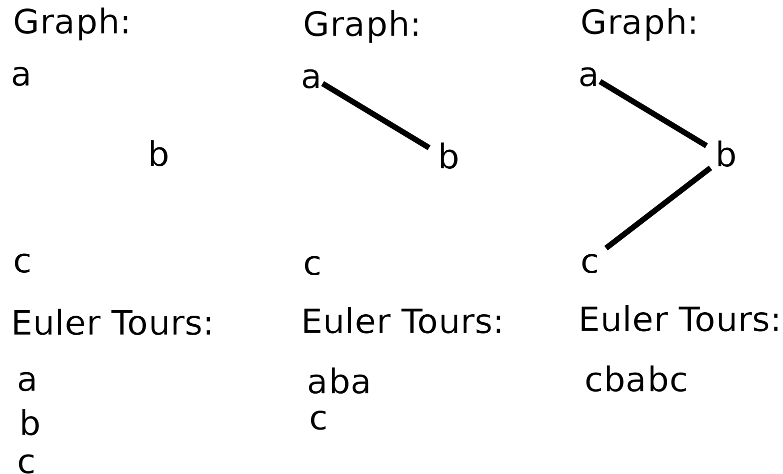

Figure 4.3: Construction of an Euler tour tree from a disconnected graph by adding one edge at a time.

T_3`=split-tree(`T_2`, `b_2`)`

T_4`=split-tree(`T_3`, `a_2`)`

`join-tree(`T_1`, `T_4`)`

This is clearly an $O(\log n)$ operation, as it involves four $O(\log n)$ operations. T_1 and T_2 are the resulting trees, with T_3 left containing only a_2, which is discarded, and T_4 being left empty by the `join-tree` operation.

4.2.2 Rerooting

Given references to T_1 and s (some instance of the node we want to be the new root):

p `= tree-prev(`T_1`, `s`)`

r_1 `= tree-smallest(`T_1`)`

T_r `= split-tree(`T_1`, `p`)`

r_r `= tree-largest(`T_r`)`

`if (`p `!=` r_1`)`

```
        T_l = split-tree(T_1, r_1)
        access-node(r_r)
        join-tree(T_r, T_l)
    e = tree-largest(T_r)
    access-node(e)
    join-tree(T_r, tree(newinstance(s)))
```

All of the operations employed above are $O(\log n)$, so the operation as a whole must be $O(\log n)$. The resulting tree is T_r.

4.2.3 Insertion

If we are inserting $\{a, b\}$, then given references to T_1, T_2, a_1 (some instance of a) and b_1 (some instance of b), we can employ the following algorithm:

```
ett-change-root(T_2, b)
T_1^1 = split-tree(T_1, a_1)
join-tree(T_1, T_2)
join-tree(T_1, tree(newinstance(s)))
join-tree(T_1, T_1^1)
```

Again, this is a $O(\log n)$ operation, as it is composed of a constant number of $O(\log n)$ operations. The resulting tree is in T_1.

4.3 Bit Fields

To trim the space explored by the replacement operation, we add one more data structure on top of our splay trees. Each node of the tree holds two bit fields. One tells if there are any tree-edges of a given level in the subtree rooted at this instance, the other if there are any non-tree-edges adjacent to any instances in that subtree.

To store this, we keep two other bit fields, describing the edges adjacent to the vertex this instance represents. To conserve space and further narrow the search, we do not do this for every instance of a vertex. See section 5.3 for details of how our implementation handles this.

Updating these four bit fields when we change the edges adjacent to a node requires walking the path from a node in the tree to the root, to update the information about edges in the subtree under each node. As we are using splay trees, this is a $O(\log n)$ operation. Updating these records adds one more $O(\log n)$ operation to all tree operations, which preserves their overall complexity. It also adds a constant number of $O(\log n)$ operations (updating the bit fields) to the Euler tour tree insert-edge and delete-edge operations, which again preserves their overall complexity. As the levels of the edges are bounded by $\lfloor \log_2 n \rfloor$, a machine with a 32-bit integer gives $n = 2^{32} = 4294967296$ vertices. If we accept this upper bound on the number of vertices in our graph, we can update the bit fields in constant time using bitwise boolean operations [5].

Chapter 5

Implementation

The implementation of the algorithm is divided into two distinct layers of objects. The lower layer implements the data structures used by the connectivity algorithm, splay trees and bigraphs, in a generic manner. The upper layer extends these classes, adding the data and methods used by the connectivity algorithm.

The connectivity algorithm must be able to update and access the data records in the nodes of the splay trees and vertices and edges of the bigraphs. Virtual function "hooks" are provided in the parent classes and called when standard operations are performed. The subclasses then override these hooks, allowing them to perform the updates they need to when an operation is performed on their parent. Through this, all the necessary update operations for the connectivity tracking algorithm wind up completely transparent to the user of the ConnectivityGraph class.

5.1 The BiGraph

The BiGraph implementation has classes representing both edges and vertices. Code that uses the BiGraph creates appropriate vertex classes as children of the vertex classes the BiGraph uses. These are deleted when removed from the graph, as a vertex without a graph does not make sense. While this might seem like a burdensome limitation on programmers using the BiGraph class, in practice it can be resolved by

using composition instead of inheritance to add user data to vertices.

Edge objects are created and deleted by the BiGraph they are part of; they have no independent existence. Instead of using templates to specify the edge class to instantiate, we hard-code it into the class by way of a virtual method that is called to create edges. While this does require a subclass if the user of the classes wants to change the edge class, it could easily be replaced with a factory class or even a template.

The graph, edge, and vertex objects allow for all the standard queries to be made: edge endpoints, vertex neighborhoods, vertex degrees, etc. The connectivity algorithm does not make use of many of these, but any application code using the ConnectivityGraph class would find them useful.

5.2 Splay Trees

The Splay Tree implementation gives each node in the tree its own object, with a parent pointer and two child pointers. The implementation does not employ any kind of edge object in the splay tree, as it has no need for it.

We implement only the `access-node`, `join-tree`, and `split-tree` operations. `access-node` is implemented on the tree's node objects, and moves a node to the root through a standard series of rotations; `join-tree` merges two splay trees together, and `split-tree` splits them apart [9]. See section 4.1 for theoretical analysis of the algorithms. Because of the implicit ordering we use (see section 4.2), we employ a slight variation of the `split-tree` operation. Instead of splitting after a given element, which may or may not be in the tree, we split at the root by breaking the right subtree off into its own tree. We can still split at any node, we just have to access it before we perform the operation.

5.3 Euler Tour Trees

Our implementation of the Euler tour trees is close to a direct translation from pseudocode. (See section 4.2 for the pseudocode.) The additional code comes from the need to have the references to certain parts of the structure that we use achieve the complexity bounds of the Euler tour tree operations.

As each vertex in the graph can have more than one instance in the Euler tour tree that represents the component it is in, we need a quick way of finding an instance of a vertex. We can not search through the binary search tree as we usually would, as the nodes of the tree are ordered by position in the Euler tour sequence, not by any property of the vertices they correspond to. Instead, for each vertex in the graph, we maintain a pointer to an instance of that vertex, which we call the "active" instance. This allows us to find an instance given a vertex in constant time, and adds a constant-time update operation to the Euler tour tree edge deletion and rerooting operations, which can each destroy at most once instance.

This helps when inserting an edge and rerooting a tree, but what about when we delete an edge? In this case, we need the instances encountered when traversing the edge. To accomplish this, we have each edge keep a record of the instances created by traversing it when creating the euler tour. There are three of these instances if it is not a leaf, four if it is. We also have each instance keep a record of the one (if it is the first or last instance of the root) or two edges that were passed through to enter and leave it. We can update the "incident edge" records of the instances in constant time, but we must take $O(\log n)$ time to update the "incident instances" records in the edges, as they must be ordered by tree-index for the delete operation, and tree-index takes $O(\log n)$ time to compute.

This preserves our efficiency on the Euler tour tree operations, as this is just another $O(\log n)$ operation at the end of each [7].

To further narrow the search space for the replacement operation, and reduce the number of instances we need to update when we make changes to edges, we adopt the following convention. Tree-edges are recorded in the bit set of the active instance

of the vertices they are adjacent to. Non-tree edges are recorded in the bit-sets of the three or four instances that were generated by traversing the edge.

5.4 ConnectivityGraph

To implement the ConnectivityGraph, we extend all the components of the BiGraph: the graph itself, vertices, and edges.

To the vertex class, we add very little. The vertex must keep track of which of its edges are at which level, even though the edges themselves also track this. This allows us to easily retrieve all the edges of a given level for a node. Although we use a Standard Template Library (STL) `map` for this operation, which has logarithmic access times, we could easily switch to an STL `hashmap`, which has constant time accesses on average [8]. Code for managing the active instance pointers is also added.

The ConnectivityEdge class also does not add much to the BiGraphEdge. The bulk of the logic for increasing an edge level is located here, as that is an operation on an edge. Based on whether or not it is a tree-edge, the edge decides what other objects need to be notified of the increase and performs the necessary method calls.

Thanks to the extensive code in the other classes, the additions to the BiGraph class itself are minimal. We add the new operations necessary to track connectivity and the data structure to store the set of spanning trees. All the other code is in the classes the ConnectivityGraph uses [5].

Chapter 6

Performance

The first tests run on the implementation were small ones involving a mere handful of nodes. These thoroughly unremarkable tests verified the basic functionality of the algorithm, and helped verify its correctness.

The algorithm was tested using a random test program, to more easily produce and execute as many test cases as possible. This program generates a specified number of nodes, and then performs a specified number of operations on them. It chooses which operation to perform by randomly choosing a pair of nodes and checking whether or not they are adjacent. If they are, the edge joining them is deleted. If they are not, it is inserted.

While this proved invaluable for uncovering unusual problems that the rather simplistic manually-constructed test, there was one other issue. To properly debug a problem when it was encountered, the exact test that demonstrated it had to be rerun. To address this, a third parameter was added to the test program: the seed for the random number generator. To enable easy reproduction, the first thing output during a trial run is the command to re-execute that run.

During the process of debugging the algorithm, the contents of the Euler tour trees being operated on were dumped to `stderr`. This allowed us to convince ourselves of the correctness of the algorithm by checking through them at random and examining records in the course of debugging.

6.1 Code Statistics

The final source files contain 3431 lines of text. 992 of these lines are found in header files, 2439 in C++ source files. Using some simple `grep` commands to filter out blank lines and lines that contain only comments, the source code contains approximately 2180 lines of actual code.

To verify that the implementation achieved the theoretical $O\left(\log^2 n\right)$ bound, we ran repeated tests using the mass connectivity testing program (see section 6) to measure the per-operation run time. The tests were run on a dual 1 Ghz G4 machine with 1.5 gigabytes of RAM running MacOS 10.3.4. `gettimeofday()` was used to measure operation execution times, which were recorded in milliseconds. `awk` scripts were used to calculate the average run time for each run, then the average and standard deviation of these values for a given number of nodes. We do not evaluate adding and deletion of edges separately. As amortized analysis was employed by the authors of the algorithm, we must evaluate the two together.

As can be seen in figure 6.2, the ratio approaches a constant value as n increases. But this is a fairly small number of nodes, and a fairly small number of tests for each number of nodes. To verify for larger numbers of nodes, we ran test sequences with 500 nodes and 500 operations per test, and 1000 nodes with 200 operations per test. The results can be seen in figures 6.3 and 6.5.

As figures 6.4 and 6.6 show, the values very rapidly approach a constant ratio, despite an initial large bump. The wide standard deviation is somewhat startling, but is expected. Deletions take much more time than additions on average, and the exact time taken depends on the precise layout of the graph and other operations the machine was performing at the same time. To minimize the influence of the second factor, runs for each number of nodes were not clumped but, instead, were spread out. So for a test with 100 nodes, running 5 tests for each number of nodes, we had run a test with 2, 3, 4, ..., 100 nodes, then repeat five times.

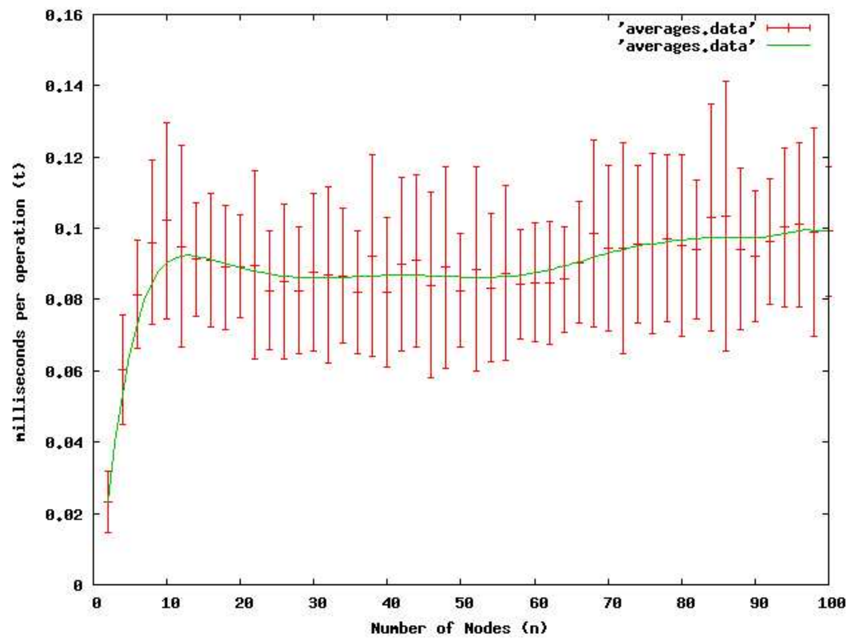

Figure 6.1: Time per operation plotted against number of nodes for a test case with 100 nodes, 50 runs per number of nodes, steps of 2 in number of nodes.

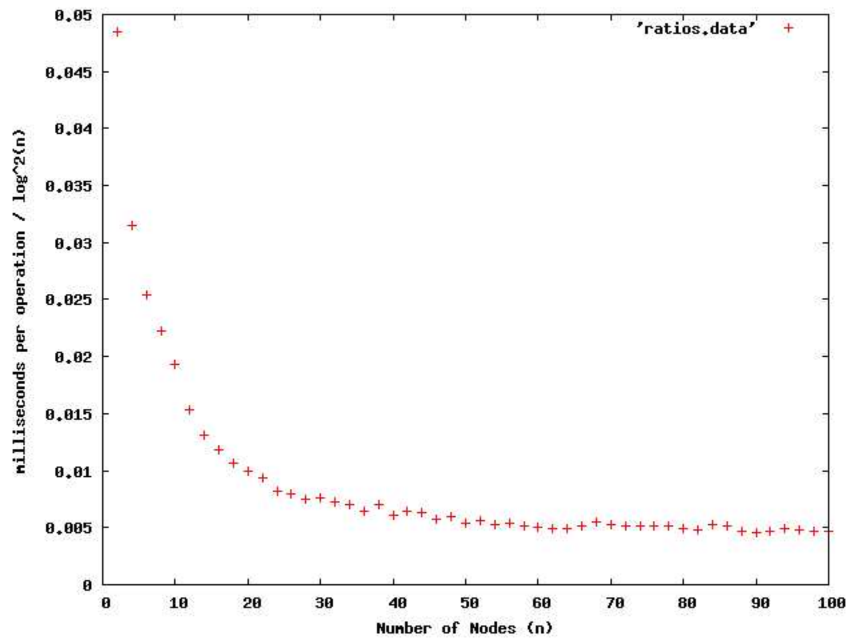

Figure 6.2: The ratio of times in figure 6.1 to $\log^2 n$ plotted against number of nodes.

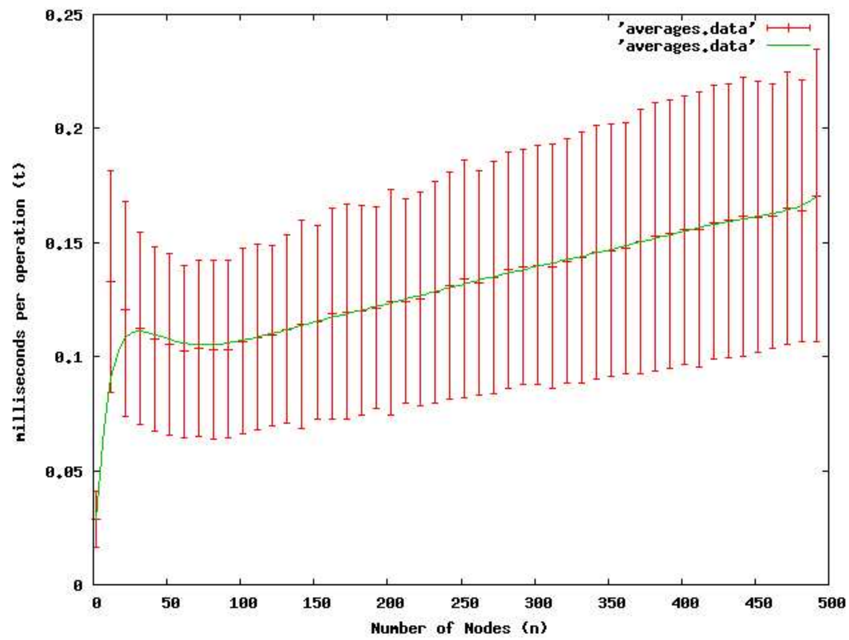

Figure 6.3: Time per operation plotted against number of nodes for a test case with 500 nodes, 500 runs per # of nodes, steps of 10 in # of nodes.

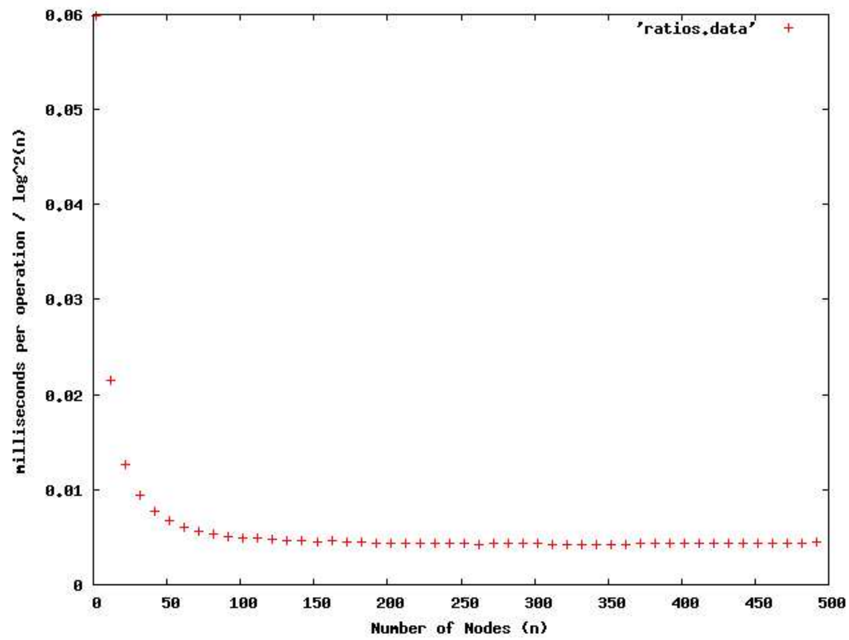

Figure 6.4: The ratio of times in figure 6.3 to $\log^2 n$ plotted against number of nodes.

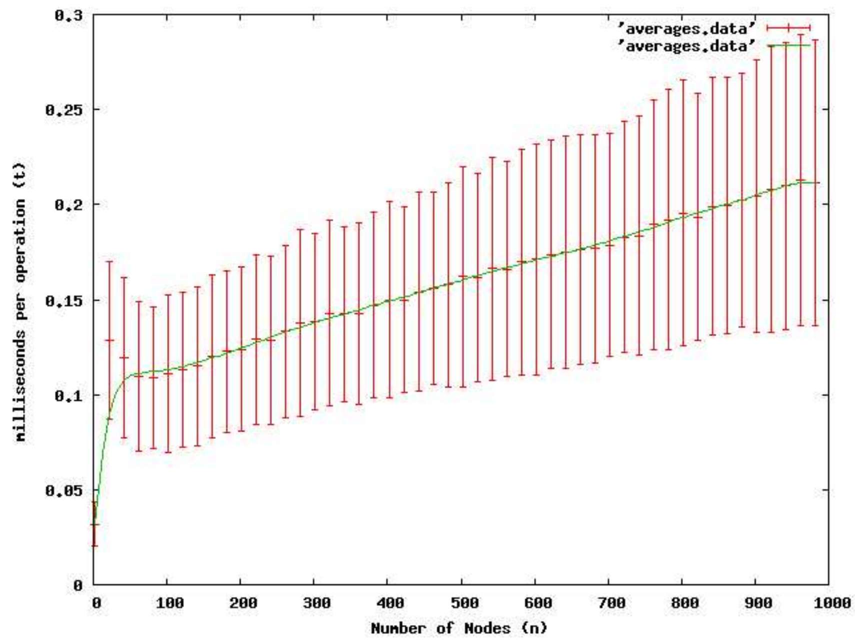

Figure 6.5: Time per operation plotted against number of nodes for a test case with 1000 nodes, 200 runs per # of nodes, steps of 20 in # of nodes.

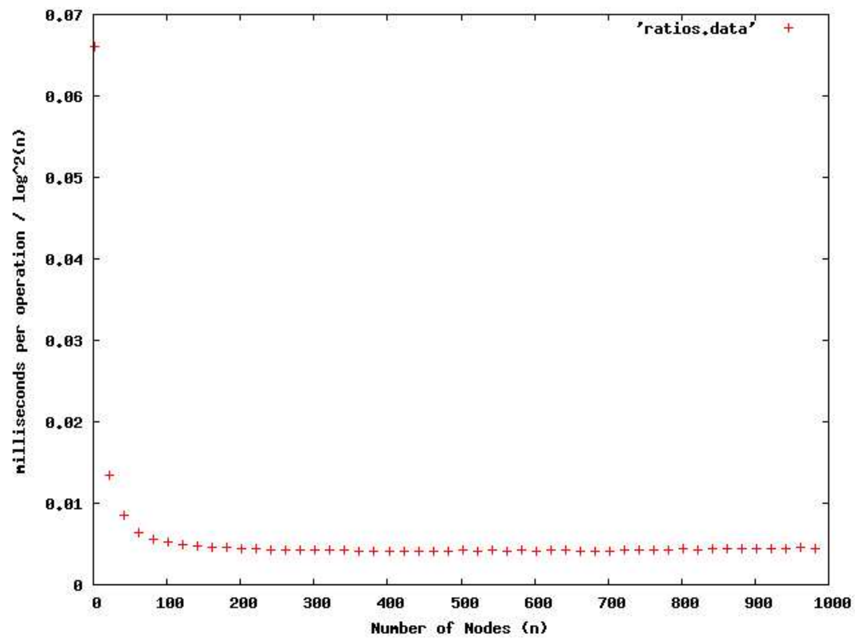

Figure 6.6: The ratio of times in figure 6.5 to $\log^2 n$ plotted against number of nodes.

Chapter 7

Application to Mobile Ad-Hoc Networks

The connectivity algorithm gives us a useful means of examining the network and allow for a more informed interpretation of other results. To do this, however, we must somehow output the status of the links between nodes, so that we know when they are made or broken.

`ns2` imposes some limitations on us here. We could not discover any explicit notification sent when a link between two wireless nodes is broken or made. The global-registry ants implementation, `MAR` [13], has each node regularly broadcast hello packets, and builds a list of neighbors based on the replies. A timeout mechanism is employed to remove "dead" links from its list. A very simple-minded way of testing link health would be to log a message whenever an entry is added to or removed from the neighbors list through this method.

Since many operations can happen in the same timeslice, we have to consolidate these together to form a meaningful picture. Although we still perform operations on the graph one add/delete at a time, we report on the connectivity for each timeslice as a unit, as this is more useful than a report after each operation.

The algorithm could, if necessary, provide information about exactly which nodes were in exactly which subgraphs of the network. This is not usually information we

want or need. More useful is information about the number and sizes of components in the network. This tells us how tightly clustered the network is, and how many nodes we can expect to have routes to other nodes.

The algorithm, unfortunately, could not be used for a similar task on actual mobile devices. It requires global knowledge about links being made and broken that it simply would not have access to. It also cannot be used to test routing tables, as the graph of routing tables is a directed graph, and the algorithm cannot test for strong connectivity.

Chapter 8

Conclusions and Future Work

We accomplished what we had set out to do. Our implementation of the connectivity algorithm meets the expected bounds (see section 6.1) and can be applied to simulations of mobile ad-hoc networks. We can use the algorithm to get useful data about scenarios in mobile ad-hoc networks, which can aid us in implementing, debugging, and evaluating future ant-based protocols.

There are a number of directions this work could be taken in the future. The most obvious would be writing a visualization tool of some kind for the output produced when examining a mobile ad-hoc network simulation.

The implementation we have produced is also only a partial implementation of the full algorithm. What has been coded is a good base, but extensions can be added to compute minimum spanning tree, 2-edge connectivity, and biconnectivity.

`MAR` itself also has room for improvement. Currently, it does not seem to take advantage of `ns2`'s packet delivery notification. This can be used, and is used by other routing algorithms such as `AODV`, to detect broken links. Expanding the `MAR` implementation to take this into account would make broken link detection much more timely and accurate.

Satwant Sandu is working on a project for his masters thesis that will apply the algorithm to mobile ad-hoc networks in a different way. He plans to use this implementation of the algorithm to track local topology around a mobile ad-hoc

network node. Instead of attempting to map the entire network, his algorithm will only look out to the nth neighbor of the local node. He still faces the same problem, of dynamically updating information about a graph, but on a reduced scale.

References

[1] The ns-2 network simulator. http://www.isi.edu/nsnam/ns/ download in August 2004.

[2] Frederickson, G. N. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing 14*, 4 (1985), 781–798.

[3] Grundke, E. Ant colony research group homepage. http://www.cs.dal.ca/~grundke/ants/ downloaded in August 2004.

[4] Henzinger, M. R., and King, V. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing* (1995), ACM Press, pp. 519–527.

[5] Holm, J., de Lichtenberg, K., and Thorup, M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (1998), ACM Press, pp. 79–89.

[6] Holm, J., de Lichtenberg, K., and Thorup, M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM 48*, 4 (2001), 723–760.

[7] Karger, D. Advanced algorithms lecture notes. http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-854JAdvanced-AlgorithmsFall1999/5C3DBFC9-8F3B-4BC2-A87A-D9586CD3BD74/0/scribe7.pdf downloaded in July 2004, 1999.

[8] SGI, H.-P. C. Standard template library programmer's guide. http://www.sgi.com/tech/stl/ downloaded in July/August 2004.

[9] Sleator, D. D., and Tarjan, R. E. Self-adjusting binary trees. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing* (1985), ACM Press, pp. 235–245.

[10] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM J. Computing 1* (1972), 146–160.

[11] Tarjan, R. E., and Vishkin, U. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *25th Annual Symposium on Foundations of Computer Science* (1984), IEEE, pp. 12–20.

[12] Thorup, M. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing* (2000), ACM Press, pp. 343–350.

[13] Zhou, Y. Intelligent agent routing for mobile ad-hoc networks. http://www.cs.dal.ca/˜zincir/ants/mar-thesis.pdf downloaded in August 2004, 2003.