An Experimental Study of Poly-Logarithmic Fully-Dynamic Connectivity Algorithms

Raj D. Iyer Jr.
Massachusetts Institute of Technology
and
David Karger
Massachusetts Institute of Technology
and
Hariharan S. Rahul
Massachusetts Institute of Technology
and
Mikkel Thorup
AT&T Research

---

conducted

Name: Raj D. Iyer Jr.
Affiliation: MIT
Address: Laboratory for Computer Science, Cambridge, MA 02138. email: rajiyer@lcs.mit.edu
URL: http://theory.lcs.mit.edu/~rajiyer Research partially supported by an NSF Graduate Research Fellowship.
Name: David R. Karger
Affiliation: MIT
Address: Laboratory for Computer Science, Cambridge, MA 02138. email: karger@lcs.mit.edu
URL: http://theory.lcs.mit.edu/~karger Research supported by NSF contract CCR-9624239 and grants from the Packard Foundation and the Alfred P. Sloane Foundation.
Name: Hariharan S. Rahul
Affiliation: MIT
Address: Laboratory for Computer Science, Cambridge, MA 02138. email: rahul@lcs.mit.edu
Name: Mikkel Thorup
Affiliation: AT&T Labs—Research
Address: 180 Park Avenue, Florham Park, NJ 07932-0971 email: mthorup@research.att.com
affiliations command

We present an experimental study of different variants of the amortized $O(\log^2 n)$-time fully-dynamic connectivity algorithm of Holm, de Lichtenberg, and Thorup (STOC'98). The experiments build upon experiments provided by Alberts, Cattaneo, and Italiano (SODA'96) on the randomized amortized $O(\log^3 n)$ fully-dynamic connectivity algorithm of Henzinger and King (STOC'95). Our experiments shed light upon similarities and differences between the two algorithms. We also present a slightly modified version of the Henzinger-King algorithm that runs in $O(\log^2 n)$ time, which resulted from our experiments.

1. INTRODUCTION

We consider data structures for fully dynamic graph connectivity. In a *fully dynamic graph problem*, we are considering a graph G over a fixed vertex set V, $|V| = n$. The graph G may be *updated* by insertions and deletions of edges. Unless otherwise stated, we assume that we start with an empty edge set. Interspersed with the updates, queries are made against the "current" graph.

For the *fully dynamic connectivity problem*, the queries are *connectivity queries*, asking whether two given vertices are connected in G. Both updates and queries are presented *on-line*, meaning that we have to respond to an update or query without knowing anything about the future. The connectivity problem reduces to the problem of maintaining a spanning forest (a spanning tree for each component) in that if we can maintain *any* spanning forest F for G at cost $O(t(n)\log n)$ per update, then, using dynamic trees [Sleator and Tarjan 1983], we can answer connectivity queries in time $O(\log n/\log t(n))$.

In this paper we study some algorithms in which $t(n)$ is polylogarithmic. We compare implement algorithms of Henzinger and King [Henzinger and King 1995] and Holm, de Lichtenberg, and Thorup [Holm et al. 1998] and compare their performance in practiced on assorted input test families. We also evaluate several heuristics that can improve the performance of these algorithms in practice.

1.1 History

The first non-trivial fully-dynamic connectivity algorithm was presented in 1985 by Fredrickson [Frederickson 1985]. It supported updates in $O(\sqrt{m})$ time and queries in constant time. In 1992, Epstein et al. [Eppstein et al. 1997] improved the update time to $O(\sqrt{n})$.

In 1995, Henzinger and King [Henzinger and King 1995] presented the first polylogarithmic algorithm for fully dynamic connectivity. The algorithm was randomized, supporting updates in $O(\log^3 n)$ expected amortized time and queries in $O(\log n/\log\log n)$ time. The (expected) update time was later improved to $O(\log^2 n)$ in 1996 by Henzinger and Thorup [Henzinger and Thorup 1997]. Finally, in 1998, Holm, de Lichtenberg, and Thorup [Holm et al. 1998] presented a deterministic fully dynamic algorithm with updates in $O(\log^2 n)$ time and queries in $O(\log n/\log\log n)$ time. The above upper bounds are complemented by a lower bound of $\Omega(\log n/\log\log n)$ which was proved independently by Fredman and Henzinger [Fredman and Henzinger 1998] and Miltersen, Subramanian, Vitter, and Tamassia [Miltersen et al. 1994]. The lower bound holds even for the restricted case where the graph is a set of disjoint paths, and it allows for both randomization

and amortization.

It is a common feature of all the above polylogarithmic fully-dynamic algorithms that they use space $\Theta(m + n \log n)$. The $n \log n$ stems from the fact that they maintain a logarithmic number of overlapping forests over the n vertices.

2. THE BASIC ALGORITHMS

In this section, we describe at a high level the two algorithms which form the basis for our experiments—a randomized one (HK) due to Henzinger and King [Henzinger and King 1995], and a deterministic one (HDT) due to Holm, de Lichtenberg, and Thorup [Holm et al. 1998]. The Henzinger and King [Henzinger and King 1995] algorithm, along with assorted heuristics, has already been implemented by Alberts, Cattaneo, and Italiano [Alberts et al. 1996]; we have continued to test their implementation. We also implemented and tested the HDT algorithm and developed some heuristics for it. For consistency in our discussion, we "invert" the level notation in the description of HK to match that of HDT.

2.1 Euler Tour Trees

Both algorithms maintain a "current spanning forest" of the current graph. Connectivity queries are answered by checking whether two vertices are in the same tree of the current spanning forest. As edges are added to and removed from the graph, the current spanning forest occasionally must be updated in order to keep it spanning. These updates involve inserting edges into and deleting edges from the spanning forest.

Euler tour trees (ET-trees) are a useful data structure for performing these operations on trees (cf. [Henzinger and King 1995]). They can be thought of as an efficient dynamic connectivity data structure for trees. They support all the operations (connect two trees with an edge, delete an edge from a tree, and check if two vertices are in the same tree) in $O(\log n)$ time per operation. A refinement can improve the query time to $O(\log n / \log \log n)$, but we did not make use of it in our experiments.

An ET-tree is implemented as a standard balanced binary tree over the Euler tour of a tree. Our implementation uses treaps for balance. Double every edge of the spanning tree and take an Euler tour of the resulting graph, outputting in order the list of vertices encountered. Store this ordered list of vertices in a treap. The point in considering Euler tours is that if trees in a forest are linked or cut, the new Euler tours can be constructed by at most 2 splits and 2 concatenations of the original Euler tour sequences. These splits and concatenations can be accomplished in $O(\log n)$ time per operation by rotating certain nodes to the roots of the treaps and then attaching or removing subtreaps of the roots.

Given the ET-trees, we can identify the tree containing a vertex in $O(\log n)$ time. This in turn lets us tell in $O(\log n)$ time whether two vertices are in the same tree. To find the tree containing a given vertex, we simply follow parent pointers from the vertex to the root of its ET-tree. Note that in fact an ET-tree generally contains many "occurrences" of a given vertex over the Euler tour; we just store with each vertex a pointer to one such occurrence from which we can walk up to its ET-tree root.

2.2 Replacement Edges

When we wish to work with graphs rather than trees, matters become more complicated. As mentioned above, both algorithms we study maintain a "current spanning forest" that makes it easy to answer connectivity queries. Given such an approach, inserting edges in the graph appears easy: we simply check (using the ET-tree structures on the current spanning forest) whether the inserted edge is in the current spanning forest; if it is we do nothing, and if not we insert it in the forest (merging the ET-tree for the two trees it connects). All of these operations can be accomplished in $O(\log n)$ time using the ET-trees.

Matters are complicated by deletions. If the edge to be deleted is not in the current spanning forest then we simply discard the edge. But if the edge is part of a tree in the current forest then its removal breaks that tree into two pieces. This might make our forest non-spanning, for some previously inserted *non-tree* edge might have one endpoint in each piece (throughout this paper, "non-tree" refers to edges not in any current spanning tree). Thus, to maintain the spanning invariant the algorithms must somehow find such a *replacement edge* if one exists and add it to the spanning tree. The naïve approach of checking all non-tree edges is too expensive, so both algorithms attempt to narrow the set of candidate edges to be examined.

One obvious observation that both algorithms exploit is that any replacement edge must be incident on (both pieces of) the tree that we separated by deleting the tree-edge. We can thus save time by examining only these edges. To do so, both algorithms *augment* the ET-tree structure. A single copy of each vertex in an ET-tree is designated *active* and contains a pointer to the adjacency list for that vertex (more precisely, the list contains only the *non-tree* edges incident on the vertex). With this modification, a traversal of the ET-tree encounters the set of vertices and through it the list of non-tree edges incident on the tree. To make this process even more efficient, the ET-tree maintains at each internal ET-tree node a bit that is set if any child of the node is an active vertex with a nonempty list of incident non-tree edges. When our traversal encounters a cleared bit, we can skip the subtree of the clear node. This trick guarantees there is at least one edge below each tree node we traverse, so the time to examine all non-tree edges incident on a tree is bounded by $O(\log n)$ per edge (assuming a balanced-tree implementation of ET-trees).

While this observation provides some benefit, it is not sufficient: if a tree has many incident non-tree edges, it can take too long to examine them all. Thus, our two algorithms use differing (but interestingly related) strategies to avoid looking at all edges incident on a component. Both strategies are based on identifying dense "clusters" in the graph, and separating the inter- and intra-cluster edges into different *levels*. Intuitively, there will be few intra-cluster edges in the "sparse" part of the graph, so checking them is fast. Checking the intra-cluster edges in the dense parts of the graph is expensive, but is paid for by the large number of insertions the user must perform to make the cluster dense in the first place.

More precisely, both algorithms break the edge set into $O(\log n)$ *levels* numbered $0, \ldots, \log n$. Let E_i denote the set of edges at level i. For each level, each algorithm maintains a spanning forest F_i of all the edges at levels i and higher, that is F_i spans

$\cup_{j \geq i} E_j$. Furthermore, $F_i \subseteq F_{i-1} (i > 0)$. Thus F_0 is the spanning forest of the entire graph, while F_i is just the set of edges of F_0 at levels i and higher. Equivalently, we see that the level-i edges in our forest unite the connected components formed by the edges at level $i + 1$.

The invariants just described for F_i ensure that any replacement edge for a given deletion is at a level no higher than the deleted edge. Thus, each algorithm can search for replacement edges only at the deletion level and below. Both algorithms employ strategies that keep "important" edges at the low levels of the data structures while pushing "unimportant" edges to higher levels where they need not be searched. This reduces the time needed to find replacement edges. Intuitively, the higher the level of an edge, the denser the component in which it is contained.

HDT starts edges at the bottom level and moves them up as they are found to be in dense regions. In an interesting contrast, HK allows edges to float up automatically but pushes them down as they are found to be in sparse regions. Both algorithms move edges among levels such that a certain parameter halves at each move—this ensures that no edge moves more than $\log n$ levels, which leads to the polylogarithmic running times.

Before getting into the differing details of these clustering strategies, we remark on some consequences of the algorithms' designs:

(1) Both algorithms answer connectivity queries by checking the ET-tree for the current spanning forest F_0. Thus, there is unlikely to be a noticeable difference in behavior for connectivity queries, which in any case will be much faster than expensive updates (see Item 4).

(2) Insertions are trivial for both algorithms, involving a connectivity check and possibly a single ET-tree update. (HK additionally rebuilds its data structure).

(3) Deletions of non-spanning-tree edges are also trivial for both algorithms.

(4) The major difference arises in how the algorithms deal with deletions of tree-edges.

2.3 The HDT algorithm

In this section, we outline the $O(\log^2 n)$ time deterministic fully dynamic algorithm of Holm, de Lichtenberg, and Thorup for graph connectivity. First we give a high level description, ignoring all problems concerning data structures. Second, we describe the algorithm with concrete data structures and analyze its operations' running times.

2.3.1 *High level description.* The dynamic algorithm maintains a spanning forest F of a graph G. The edges in F will be referred to as *tree-edges*. Internally, the algorithm associates with each edge e a level $\ell(e) \leq L = \log_2 n$. For each i, F_i denotes the sub-forest of F induced by edges of level at least i. Thus, $F = F_0 \supseteq F_1 \supseteq \cdots \supseteq F_L$. The following invariants are maintained.

(i) F is a maximum (w.r.t. ℓ) spanning forest of G, that is, if (v, w) is a non-tree edge, v and w are connected in $F_{\ell(v,w)}$.

(ii) The maximal number of nodes in a tree (component) of F_i is $\lfloor n/2^i \rfloor$. Thus, the highest nonempty level is $\log n$.

Initially, all edges have level 0, and hence both invariants are satisfied. We are going to present an amortization argument based on increasing the levels of edges. The levels of edges are never decreased, so we can have at most L increases per edge. Intuitively speaking, when the level of a non-tree edge is increased, it is because we have discovered that its end points are close enough in F to fit in a smaller tree on a higher level. Concerning tree edges, note that increasing their level cannot violate (i), but it may violate (ii).

We are now ready for a high-level description of insert and delete.

Insert(e)**:** The new edge is given level 0. If the end-points were not connected in $F = F_0$, e is added to F_0.

Delete(e)**:** If e is not a tree-edge, it is simply deleted. If e is a tree-edge, it is deleted and a *replacement edge*, reconnecting F at the highest possible level, is searched for. Since F was a maximum spanning forest (invariant (i)), we know that the replacement edge has to be of level at most $\ell(e)$. So call Replace$(e, \ell(e))$.

Replace$((v, w), i)$**:** Assuming that there is no replacement edge on level greater than i, finds a replacement edge of the highest level less than or equal to i, if any.

Let T_v and T_w be the trees in F_i containing v and w, respectively after the deletion. Assume, without loss of generality, that $|T_v| \leq |T_w|$. Before deleting (v, w), $T = T_v \cup \{(v, w)\} \cup T_w$ was a tree on level i with at least twice as many nodes as T_v. By (ii), T had at most $\lfloor n/2^i \rfloor$ nodes, so now T_v has at most $\lfloor n/2^{i+1} \rfloor$ nodes. Hence, preserving our invariants, we can take all edges of T_v of level i and increase their level to $i + 1$, so as to make T_v a tree in F_{i+1}.

Now visit level-i non-tree edges incident to T_v one by one until either a replacement edge is found, or all edges have been considered. Let f be an edge visited during the search.

If f does not connect T_v and T_w, then both its ends are in T_v. So increase its level to $i + 1$. This increase pays for considering f, but maintains the invariant since T_v is now at level $i + 1$. If f does connect T_v and T_w, insert it as a replacement edge and stop the search.

If there are no level i edges left, call Replace$((v, w), i - 1)$ unless $i = 0$, in which case we conclude that there is no replacement edge for (v, w).

2.3.2 *Implementation.* For each i, we wish to maintain the forest F_i together with all non-tree edges on level i. For any vertex v, we wish to be able to:

—Identify the tree T_v in F_i containing v.

—Compute the size of T_v (to choose the smaller half).

—Find an edge of T_v on level i, if one exists (so we can promote such edges in time proportional to their number).

—Find a level i non-tree edge incident to T_v, if any (to test as a replacement edge).

The trees in F_i may be cut (when an edge is deleted) and linked (when a replacement edge is found, an edge is inserted or the level of a tree edge is increased). Moreover, non-tree edges may be introduced and any edge may disappear on level i (when the level of an edge is increased or when non-tree edges are inserted or deleted).

As was discussed in Section 2.2, all of the operations above can be supported in $O(\log n)$ time (total or per edge checked, as appropriate) using ET-trees augmented with some counts and bits. For our application we need an ET-tree over each tree in F_i for each i (taking a total of $O(n \log n)$ space).

It is now straightforward to analyze the amortized cost of the different operations. When an edge e is inserted on level 0, the direct cost is $O(\log n)$. However, it may be promoted $O(\log n)$ times at cost $O(\log n)$ per promotion, so the amortized cost is $O(\log^2 n)$.

Deleting a non-tree edge e takes time $O(\log n)$. When a tree edge e is deleted, we have to cut all forests F_j, $j \leq \ell(e)$, giving an immediate cost of $O(\log^2 n)$. We then have $O(\log n)$ recursive calls to Replace, each of cost $O(\log n)$ plus the cost amortized over increases of edge levels. Finally, if a replacement edge is found, we have to link $O(\log n)$ forests in $O(\log^2 n)$ total time.

Thus, the cost of inserting and deleting edges from G is $O(\log^2 n)$. The balanced binary tree over $F_0 = F$ immediately allows us to answer connectivity queries between arbitrary nodes in time $O(\log n)$. Thus, we conclude:

THEOREM 1. *Given a graph G with m edges and n vertices, the HDT data structure answers connectivity queries in $O(\log n)$ time worst-case, and uses $O(\log^2 n)$ amortized time per insert or delete.*

One can reduce the query time to $O(\log n / \log \log n)$ [Henzinger and King 1995]. The hidden constants are quite large however, so we did not explore this theoretical improvement in our experiments.

2.4 The HK algorithm

Inverting the HDT approach of promoting dense regions, the HK algorithm [Henzinger and King 1995] works mainly by *demoting* edges in sparse regions. When a tree edge is deleted from a tree in forest F_i, breaking the tree into two pieces T_1 and T_2, HK looks for a replacement edge at level i by inspecting the smaller of T_1 and T_2, say T_1. This is done by *sampling*: we randomly select non-tree edges incident on T_1, and check if they connect to T_2 (each such test is done in $O(\log n)$ time using ET-tree operations). If within $O(\log^2 n)$ samples a replacement edge is found, we are done. Otherwise, we can deduce that a large majority of the non-tree edges incident on T_1 actually have both endpoints in T_1. We therefore traverse the entire tree T_1 (finding a replacement edge if one exists) and demote all the edges that have only one endpoint in T_1, decreasing their level by one. If necessary, we then continue our search for a replacement edge one level lower.

The point of the above algorithm is that we only demote edges if the cut they cross is sparse, meaning that we are demoting edges that it was difficult to find to level $i-1$. Thus, the next time we delete a edge (now at level $i-1$) that connects T_1 and T_2, we will begin our search at level $i-1$, where we will not be distracted by the many edges internal to T_1. The fact that we demote only a small fraction

of a component's non-tree edges ensures that many edges must be inserted before we could demote anything by over $\log n$ levels.

In order to make sure that the cut is really sparse enough to demote, we must sample $O(\log^2 n)$ edges. Since each sample costs $O(\log n)$, we get a sample cost of $O(\log^3 n)$. Henzinger and Thorup [Henzinger and Thorup 1997] have suggested an iterative sampling scheme, making only $O(\log n)$ samples yet getting the same guarantees. However, the constants involved (60^4) are too big for the approach to be relevant to our experimental analysis.

Of course over a sufficiently long sequence of operations, demotions could carry edges down a large number of levels; to prevent this from happening, HK also *rebuilds* its data structure periodically, raising the low-level edges back up. This ensures that all edges remain within $\log n$ levels while still aiming to keep more important edges at lower levels.

3. PLAN OF ATTACK

Given these two algorithms, some natural questions arose. The $O(\log n)$ asymptotic improvement in the HDT running time versus HK suggested that HDT was likely the better choice in practice (it is also somewhat simpler to implement). However, we wanted to carry out experiments to verify that this was indeed the case. In particular, since the HK analysis might not be tight for many inputs, we aimed to develop some indication that the worse running time of HK versus HDT will actually be achieved at times. We also set out to develop heuristics to improve the running time of the HDT algorithm.

An important rule in our work was that we refused to implement any heuristics that invalidated the proofs of worst-case running time bounds for our algorithms. Thus, our algorithms with heuristics still obey the polylogarithmic time bounds proven for the original algorithms, while being faster in many cases.

We made use of the HK implementation built in the LEDA platform in the previous experimental study of Alberts, Cattaneo, and Italiano [Alberts et al. 1996]. LEDA [Mehlhorn and Naher 1995] is a general-purpose platform for implementing graph algorithms. Developing on LEDA also meant that our implementation of HDT could use the same ET-tree structures as were used in the HK implementation [Alberts et al. 1996]. This gives us some hope of avoiding differences in runtime caused by unimportant implementation hacks (use of bit fields, etc.) and lets us trust that faster times for a particular algorithm in our implementation will imply faster times for any other implementation as well.

Besides trying to tune the algorithms, we also attempted to develop a better understanding of how both algorithms "learn" the input graph structure in order to avoid doing work.

4. HEURISTICS FOR HDT

As we discussed earlier, the most expensive operation for HDT is deletion of tree edges. This is expensive for several reasons:

(1) Deleting a tree edge and inserting its replacement edge causes a split and a join in an ET-tree. Although splits and joins are $O(\log n)$ time operations, they involve rotations (numerous reads and writes) rather than the simple path

traversal (a small number of reads) that is used to test connectivity.

(2) Searches involve promotions of many edges—we promote the entire smaller half of our split tree (many more splits and joins) plus numerous non-tree edges.

(3) We may have to do all of the above on many different levels of the spanning forest.

These costs pressed us to find ways to make deletions less expensive. We developed two heuristics.

4.1 Sampling

We first took the sampling idea from the HK algorithm. Our *sampling* heuristic says that before promoting, we should randomly choose a number of incident tree edges and test if any of them are replacements. If we find a replacement edge, we can use it without performing any promotions.

Of course, such sample-and-test operations are not amortized away by promotions, so there is a limit on the number we can try. We can afford $O(\log n)$ samples without affecting the worst-case running time: each sample takes $O(\log n)$ time for a total of $O(\log^2 n)$ time that we can charge to the operation. However, if we examine *more* than $O(\log n)$ edges then the actual time spent exceeds $O(\log^2 n)$, so we must promote edges to amortize away the additional cost and maintain our $O(\log^2 n)$ amortized time bound.

Due to limited time, rather than implementing true random sampling, we simply perform a deterministic enumeration of a small number of non-tree edges from the ET-tree to look for a replacement. Of course the time bounds are still valid. As will be seen below, even this weaker heuristic was quite effective.

4.2 Truncating Levels

A second way to spend less time on a deletion is to ensure that we have fewer levels in the graph. At a high level, where the trees of the forest are guaranteed to be small, it is no longer worth doing anything sophisticated. The actual time spent looking at such small trees (without doing any promotions with their insertions and deletions) is less than the time spent doing promotions to amortize search in the more sophisticated scheme.

5. INPUT FAMILIES

In this section we discuss the input families we chose to test the algorithms. Our work stands in relation to the previous implementation work of Alberts, Cattaneo, and Italiano [Alberts et al. 1996]. Thus, for comparison purposes, we implemented several of the tests they carried out. However, we also identified some limitations in these tests and developed some tests of our own to tackle them.

The input to a dynamic connectivity algorithm consists of an initial graph and a sequence of updates (edge insertions and deletions) and connectivity queries. Thus the performance of the algorithm depends on both the connectivity structure of the initial graph and how the updates change that structure. We considered three different sets of inputs, categorized as random, structured, and worst-case, described in the following sections.

We begin each of our tests with a fixed graph of *candidate edges*. These are the only edges we will ever place in our data structure. We initially add some or all of the candidate edges to the data structure, depending on the test. We then perform updates and queries. The number of operations of each type is specified in advance. At each step in the test, if the operations not yet performed consist of I insertions, D deletions, and Q queries, we choose which operation to do next by choosing a random number r between 1 and $I+D+Q$. If $r \leq I$, we insert a random candidate edge not in the current graph. If $I < r \leq I+D$, we delete an edge from the current graph, and if $r > I+D$, we choose a random pair of nodes and ask if they are connected. We then decrement the count of the operation we performed and continue. In all our experiments, we always performed an equal number of insertions, deletions, and queries.

The random sequence of insertions and deletions can cause the number of currently present edges to fluctuate. We were concerned that this might affect the behavior of our algorithms, since some our input families are intended to operate at certain key graph sizes. Thus, we tested an operation sequence in which the insertions and deletions are grouped into "blocks" of k insertions and k deletions, randomly ordered within the block. Thus, at the end of a block, we have the same number of edges as we started with. This reduced the fluctuation in edge count. However, it did not noticeably affect the results, so we do not discuss this model further.

5.1 Random Graphs

We first tested the random graph input family. For these inputs, the set of candidate edges is all $n(n-1)/2$ edges. We initially insert m random edges and, during the course of the updates, we maintain the invariant that no more than m edges are present in the graph. To generate our operation sequence, we randomly choose whether to insert an edge (randomly chosen from the candidate edges not in the current graph), delete an edge (randomly chosen from the edges in the current graph), or ask if a random pair of vertices is connected.

This family of inputs was explored by Alberts et al. They tested both sparse random graphs ($m = n/2, n$) and dense random graphs ($m = n \log n, n^{2/3}, n^2/4$). Since the number of tree edges in an n vertex graph is bounded by n, if the total number of edges in the graph dramatically exceeds n, randomly chosen edges are likely to be non-tree edges. We have already shown that both HK and HDT can delete non-tree edges quite easily, making denser graphs trivial. Thus, for this model, we kept the initial number of edges close to n.[1] We report specifically on random graphs with $m = n/2$ and $2n$ edges.

This family of inputs is a natural starting place and provides a point of reference for other input graphs and sequences. However, it fails to challenge the algorithms. Intuitively, any time we cut a tree, if there are any non-tree edges in the tree's component, half of them will cross the induced cut and serve as replacement edges.

[1] An alternative for future work would be to bias deletions towards tree edges—this would make denser graphs interesting. However, it is not clear that a non-oblivious adversary model (one that knows which edges are in the algorithm's spanning forest) makes sense in this random-graph context.

Thus, both HK and HDT with sampling will almost always be successful without doing any demotion/promotion. This was confirmed by our experiments (see Section 7).

Our particular generator for input graphs required that the entire candidate edge set be maintained in memory (to keep track of which edges are present and which are absent). In a true random graph model, the candidate edge set is the complete graph consisting of a quadratic number of edges. Using this candidate edge set prevented us from testing large graphs ($n > 2000$).

To get around this problem, we instead chose our candidate set to be a random subset of the whole edge set. If the intent was for m edges to be actually present, we generated a candidate edge set of $50m$ edges. For example, for a graph with $n = 20,000$ vertices and an intended present edge count of $2n = 100,000$, our candidate edge set contained $5,000,000$ edges. This is substantially more than the number of edges actually inserted and deleted over the course of our experiments, meaning a typical edge is inserted into the graph only once. Thus, we expect the candidate set to behave pretty much like a truly random graph. A special case generator for this problem could be used to test truly random graphs, but we do not expect a substantial difference.

5.2 Semirandom Graphs

In an attempt to creep closer to a realistic model, we considered a *semirandom* graph model. Instead of the candidate edge set consisting of all the edges (as in the random model), in this model the candidate edge set consists of a fixed random set of edges, and we begin by inserting all candidate edges in the graph. Put another way, we first choose a random graph and then randomly delete and randomly reinsert its edges. This model seems *slightly* more realistic than true random graphs for the application of maintaining a network as links fail and recover, since presumably the network is fixed and it is just the fixed links that vanish and return.

Unlike the random graphs of the previous model, semirandom graphs have a structure which persists over the course of updates. At any given moment, the existing graph is random (a random subset of a random graph) but the different instances of the graph are strongly correlated over time. We thought that HK and HDT might detect this structure and perform better on these inputs than on random graphs. However, our experiments revealed no significant difference between the two input families. We report on inputs with sparse candidate edge sets, either $n/2$ or $2n$ edges, as discussed in the previous section.

5.3 Two-level Graphs

Our next input family explored more structured inputs to see the how algorithms were able to exploit this structure. We considered a *two-level* graph family constructed as follows. We generate k cliques of c vertices each (for a total of $n = kc$ vertices). Then we connect these cliques to one another using $2k$ randomly chosen *interclique* edges. Any spanning tree of this graph consists of in-clique trees connected by the interclique edges. We begin with all edges present in the graph and then randomly delete and insert only the interclique edges. In other words, we are taking a semirandom graph and replacing each vertex with a clique. The edges inside each clique are a distraction for the algorithms—how does their presence

affect the algorithms?

When an interclique tree-edge is deleted, the algorithms consider both the interclique and clique non-tree edges, but only the interclique edges can hope to be replacements (as was discussed with random graphs, we expect that fully half of the interclique edges will work as replacements). The fraction of interclique edges is the number of such edges ($2k$) divided by the total number of edges (roughly $kc^2/2$), which is $4/c^2$. Thus, by changing the size of the cliques, we can control the relative sparsity of the interclique edges.

We found this family of graphs interesting for several reasons. First and most obvious, it has natural structure not found in truly random graphs. This hierarchical structure is found, for example, in city roads versus highways and local area networks versus Internet backbones. It is worth studying; indeed, tools for modeling Internet-like topologies [Zegura et al. 1997; Calvert et al. 1997; Zegura et al. 1996; Doar 1996] expressly reject random graphs in favor of such hierarchical models.

Second, this two-level structure exhibits the clustering behavior that motivated the development of the HK and HDT algorithms. We wanted to observe whether the algorithms actually discovered the clustered structures in order to run faster.

A third motivation is that these inputs appear to be a challenge to the algorithms, in contrast to random graphs. Since we are constantly inserting and deleting (interclique) tree edges, the algorithms must constantly find a replacement edge. Since there are relatively few replacement edges (compared to all the non-tree clique edges), we expect that the algorithms will have a hard time finding them. This same consideration motivated Alberts et al. to study a similar input family, which they called "non-random inputs." They considered a connected graph of k cliques with c nodes and $k-1$ interclique edges. Said differently, their input is a path of cliques, whereas ours is a sparse random graph of cliques.

Our experiments revealed that both algorithms were able to learn the structure of these graphs, quite quickly in some cases.

5.4 Worst-case inputs

Our next goal was to develop worst-case inputs to stress our algorithms. Natural questions arise here. What are worst-case inputs for HDT and HK? Are these inputs pathological and easily fixed with heuristics? How does HK do on HDT's worst case, and vice versa? Do they both have a common worst-case input, which could point us to an inherently difficult dynamic connectivity instance family? Even were the answers to these questions to remain obscure, a provably worst-case input tightens the algorithm's running time bound.

We were not able to construct an input which forced HK to run in its analyzed time of $O(\log^3 n)$ per update. This ultimately led us to a slight modification of the algorithm which runs in $O(\log^2 n)$ time (see Section 8). For HDT we were able to construct a worst-case input, described in this section. HDT achieves its worst-case running time of $O(\log^2 n)$ per operation when, during $O(n)$ operations, it promotes $\Omega(n)$ edges $\Omega(\log n)$ times through the levels of the data structure.

Our worst-case graph is simply a line on $n = 2^k$ vertices. We begin by inserting the $n-1$ edges that make up the line. We number the edges from 1 to $2^k - 1$. We delete the middle edge which has index 2^{k-1}. This splits the line into two identical copies of a line on 2^{k-1} vertices. HDT responds by promoting the $2^{k-1} - 1$ edges of

the "smaller" side to level 1 (in this instance both sides are the same size, so either may be promoted). It then searches for a replacement edge which of course it does not find. The two sides are now independent (disconnected).

Now we delete the two remaining edges whose indices are multiples of 2^{k-2}, namely 2^{k-2} and $3 \cdot 2^{k-2}$. Half of each half-line gets promoted as a result—to level 1 on the side which remained at level 0 in the first phase; to level 2 on the side which was promoted to level 1 in the first phase. We end up with four independent lines on 2^{k-2} vertices, from which we proceed to delete the four middle edges (whose indices are odd multiples of 2^{k-3}). Continuing, in the j^{th} phase, we have 2^j independent lines of $2^{k-j} - 1$ edges, for $j = 0, \ldots, k-1$. We delete the middle edge of each line, which leads to the promotion of $2^{k-j-1} - 1$ edges in each line for a total of $2^{k-1} - 2^j$ promotions.

After $k = \log n$ phases we will have deleted all $n - 1$ edges and performed a number of promotions equal to

$$\begin{aligned}\sum_{j=0}^{k-1} 2^{k-1} - 2^j &= k \cdot 2^{k-1} - (2^k - 1) \\ &= \frac{1}{2} n \log n - n + 1 \\ &= \Omega(n \log n) \ .\end{aligned}$$

This gives a tight bound on the total number of edge promotions. We must still take care though, as promotions in an ET-tree on r vertices take $\Theta(\log r)$ time, not $\log n$ time. In our instance, there are $2^{k-1} - 2^j$ promotions involving lines of size $2^{k-j} - 1$, each taking $\Theta(k - j)$ time. Thus, the overall time spent in our sequence of n deletions (after n insertions) is proportional to

$$\sum_{j=0}^{k-1} (2^{k-1} - 2^j)(k - j) = \Omega(n \log^2 n) \ ,$$

as desired.

Simple heuristics in the implementation of HDT may do away with this specific worst-case example. For example, one might keep a count of the number of non-tree edges and, if there are none, skip searching for a replacement edge. This is implemented in our sampling variants of HDT (HDT$(s, \cdot)$, $s \geq 1$), which do fine on the line. However, the line can be easily modified to foil this heuristic: for example, we can replace each vertex with a clique on 3 vertices (a triangle), which will contain a non-tree edge that is not a replacement edge.

6. TESTING FRAMEWORK

In this section we discuss the details of our testing procedure.

6.1 Test Environment and Input Size

We carried out our experiments on a Sun UltraSPARC 2 with two 296 MHz processors, 512MB of RAM, a 16K instruction cache and 16K data cache on chip, and a 2MB external cache. Given the fact that we are looking at logarithmic (in n) running times, we have to be cautious about tests where n is too small and constants

dominate asymptotic running times. With respect to our goal of large problem instances, the limiting factor was available RAM. The size of our data structure is essentially determined by the number of edges in the graph, and were it too large to fit in memory, disk I/O would dominate all timing results. We were able to run on large random inputs ($n = 20,000$) and worst-case inputs ($n = 65,536$), but our two-level graphs are small ($n = 2,000$) because when we have a few large cliques, the (quadratic) number of edges becomes too much large to fit in memory.

6.2 Measures

When measuring the performance of the algorithms, two important issues arise. First, since the algorithms amortize, the instantaneous time per operation is likely to fluctuate wildly. This motivates averaging over some number of operations. Second, both HK and HDT are adaptive, so their performance may improve over the course of the operations. Put another way, the algorithms may learn how to exploit the structure of the dynamic graph in order to reduce their time per operation. Thus it is not meaningful to report performance after some number of operations. Rather, we study how soon and to what value the algorithms' performance converges as they adapt to the input.

To present this information, we measure the total time spent by the algorithm after certain numbers of operations: time t_1 after k_1 operations, time t_2 after k_2 operations, and so on. We then compute the total time spent between adjacent points, $t_{i+1} - t_i$ and divide by the number of operations $k_{i+1} - k_i$ in this interval to determine the average time per operation $a_{i+1} = (t_{i+1} - t_i)/(k_{i+1} - k_i)$ in this interval. We plot the point (k_{i+1}, a_{i+1}) in the graph. Averaging over an interval lets us measure the amortized performance without distortion from fluctuating individual operation times. Plotting the averages over multiple windows lets us see the evolving behavior of the algorithms over time.

Qualitatively, the observed running times were on the order of seconds or minutes.

6.3 Specific Tests

Below we report results for several combinations of the heuristics discussed in Section 4. For both HK and HDT, we explore variation in the number of samples taken before switching to the more expensive approach of exhaustive search with promotion (or demotion). We also explore reducing the number of levels in each algorithm's data structure, essentially implementing a base case. We name each algorithm $\{\text{HK}, \text{HDT}\}(s, b)$. The first parameter s denotes the number of samples taken before we begin promoting (HDT) or demoting (HK). For HDT, b denotes the size of trees which we don't bother to promote in favor of exhaustive search. Note that this means we have only $\log n - \log b$ levels in HDT's data structure. We use the same notation for HK—parameter b means we do not perform "rebuilds" above level $\log b$. Although HK maintains no explicit relationship between levels and tree sizes, the limit on rebuilds keeps edges restricted to a range of only $\log n - \log b$ levels, thus reducing the time spent on demotions, but at the cost of time spent on exhaustive searches. (We abuse this notation by setting b to 0 to indicate the absence of the base case heuristic; properly, b should be 1.)

We explored numerous parameter settings, and we report on the following versions as representative examples in all of our tests:

HK$(16\log^2 n, 0)$ is the vanilla implementation of the Henzinger-King algorithm.

HK$(20, n)$ (aka *hk_var*) from Alberts, et al. [Alberts et al. 1996], maintains only one level; on that level, it samples a few edges, then gives up and exhaustively searches. This algorithm is designed specifically for random graphs and does not have a provable polylogarithmic running time (indeed, there are easy ways to make it perform badly).

HDT$(0, 0)$ is the vanilla implementation of Holm, de Lichtenberg and Thorup's algorithm.

HDT$(256, 64)$ is an implementation that does some sampling and also goes to exhaustive search on trees of size 64 or less. We found that these parameter settings work well over all tested inputs, but it is hard to tune them better without the ability to try very large values of n. In particular, while it seems clear that some fixed constant is optimal for the base case size b, it is not clear whether the sample size should be a constant or should grow, e.g. logarithmically with n.

We also compared these algorithms to the naïve dynamic connectivity heuristics *fast_update* and *fast_query* as implemented by Alberts et al. These algorithms use no sophisticated data structures, working only with the naïve representation of the graph. Algorithm *fast_update* handles updates simply by inserting or deleting the given edge ($O(1)$ time) and answers queries via breadth-first search ($O(n+m)$ time). Algorithm *fast_query* maintains a spanning forest and labels each vertex with its component. It answers queries by comparing labels ($O(1)$ time) and updates the labels during insertions or deletions via breadth-first search ($O(n_0 + m_0)$ time, where n_0 and m_0 specify the size of the affected component).

7. TEST RESULTS

In this section we report on the results of our tests. Our experiments indicate that (i) the heuristics provide significant benefit, (ii) HK is dominated by HDT with heuristics, although we were unable to make HK exhibit its analyzed worst-case $O(\log^3 n)$ performance, (iii) the heuristic algorithms *fast_query* and *fast_update* that performed well for Alberts et al. on 700-vertex graphs are not competitive on larger graphs.

7.1 Sparse Random Inputs

As previously discussed, our first input family begins with a random graph of m edges and performed a sequence of random updates involving deletion of current edges and insertion of new random edges. Figures 1 and 2 show the amortized time per operation of various algorithms on two random graphs with $n = 20,000$ vertices and $m = n/2$ and $2n$ edges, respectively.

First consider the plot for $m = n/2$ (Figure 1). The theory of random graphs tells us that this graph consists almost entirely of trees, with a few ($O(n^{2/3})$) edges in components of size $O(n^{2/3})$ (here and below we use well-known facts about random graphs [Bollobás 1985]). In particular, any deleted edge is overwhelmingly likely to come from a tree, meaning that no replacement edges exist—indeed, there aren't even any non-tree edges to examine! Thus, the vanilla version of HDT, which immediately promotes the smaller tree side, is wasting effort. Algorithms that avoid

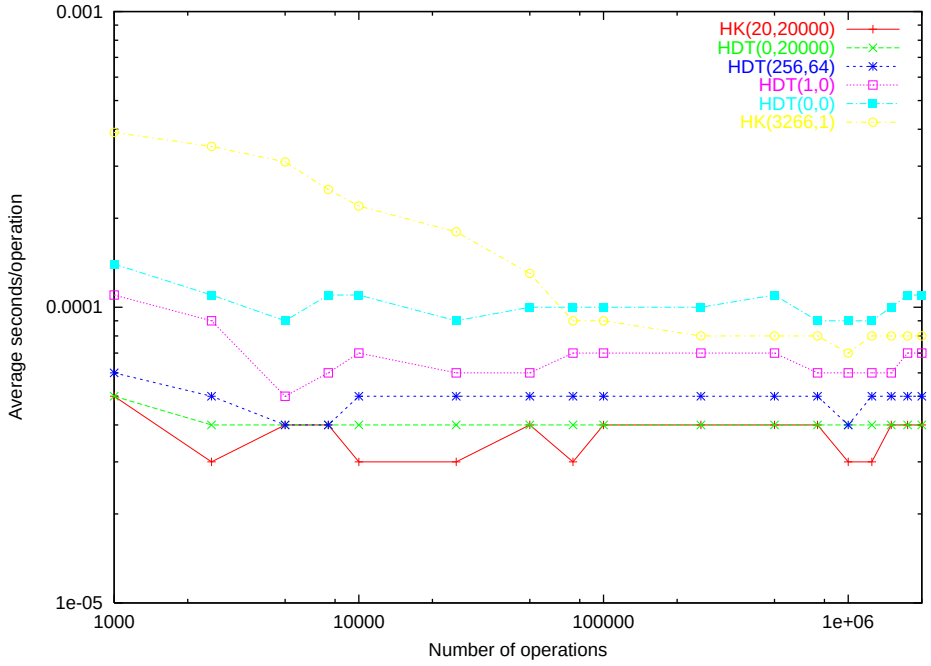

Fig. 1. Performance on random graphs: $n = 20,000$ and $m = n/2$. Average time per operation over a sequence of measurement intervals.

promotion do best. In particular, the variants HK$(20, 20000)$ and HDT$(0, 20000)$, which *never* promote, do best of all. These algorithms simply scan all non-tree edges in the smaller half of the broken component (which in this case is usually no edges).

Particularly noteworthy is the performance of vanilla HK. While this algorithm eventually converges to a runtime close to that of the other algorithms, it takes an extremely long time to do so. This behavior is explained as follows. When started on a nonempty graph, HK initially places all its edges at the top level of the data structure (where they would end up after an infinite number of rebuilds). Edges descend from this level in two ways: they may be demoted as the result of a failed sampling phase, or they may vanish in a delete operation (and reappear at the bottom level if reinserted).

High level tree-edges are problematic. The algorithm keeps a spanning tree at *every* level of the data structure. When a tree edge is deleted, we have to cut the spanning tree at *every* level below that of the deleted edge. Thus, if the deleted tree edge is at a high level, we perform many expensive ET-tree splitting operations. Initially, the entire tree is at the top level, so all deletions are very expensive. Eventually, the tree edges end up at lower levels so that deletions become cheaper. This shows up in the convergence of the HK running time to the other algorithms.

This analysis conforms to what we see in the plot. Even ignoring demotions, edges are inserted at level 0. After 90000 operations (which means 30000 deletions), every top level edge survives with probability $(1-1/m)^{30000} \approx e^{-3} \approx 0.05$, so only a small

fraction of the top level edges will not have been deleted by then. More generally, the *log* of the expected number of high level edges decays linearly with the number of operations, which might explain the near-linear improvement in (log of) running time we see plotted for HK.

The plot raises another question: why does HK converge to a performance better than HDT$(0,0)$ but worse than HDT$(1,0)$? The first answer is clear: HDT$(0,0)$ is outperformed by HK because HK samples for a replacement rather than aggressively promoting. This is demonstrated by the improved performance of HDT$(1,0)$, which samples one edge before promoting. The difference in performance between HDT$(1,0)$ and HK arises in the situation, frequent when $m = n/2$, where they are sampling for a replacement edge and there are no non-tree edges. It is a side effect of their implementation. Before sampling, HDT$(1,0)$ checks to see if there are no incident non-tree edges; if not, it does no more work. HK checks to see if the number of non-tree edges is less than some fraction. If so (it is so), then HK acts as if sampling had failed: it searches for edges that cross the cut. If there are no such edges (there are none), HK continues searching fruitlessly on the lower levels. Of course, HK could easily be modified to give up immediately just like HDT$(1,0)$.

We also tested *fast_update* and *fast_query* on the above family. Alberts et al. found that on their 700-vertex test graphs, these naïve heuristics were competitive, matching and sometimes beating the more sophisticated algorithms. However, on our 20,000-vertex inputs, this was no longer the case. *fast_update* and *fast_query* were so slow that plotting them would have compressed the axes in our figure, making the other algorithms indistinguishable.

This fits in with analysis. In a random graph of this density, the expected size of a random component (which determines the expected insert/delete time for *fast_query* and the expected query time for *fast_update*) is $n^{1/3}$ [Luczak et al. 1994]; this is therefore the asymptotic running time of the heuristics. At our input size of $n = 20,000$, this has already crossed the polylogarithmic running times of the more sophisticated algorithms. We therefore skip further consideration of these heuristics.

7.2 Less Sparse Random Inputs

We next considered the plot for $m = 2n$ (Figure 2). We see that the vanilla versions of HK and HDT perform worst of all, while the best performers are the ones which avoid promotions. Indeed, the top two performers never promote at all. This is explained as follows.

The theory of random graphs tells us that this graph contains one giant component of $\Omega(n)$ (roughly $3n/4$ in our experiments) vertices while almost all other components are trees of size $O(\log n)$. This description also holds true for the graph *after* an edge deletion. So we therefore consider two different types of tree-edge deletions. If the deleted edge has at least one endpoint outside the giant component, then we know that the smaller side of the tree after deletion is almost surely a tree of size $O(\log n)$. Thus there are no replacement edges, and this fact can be verified extremely quickly. The argument of the previous input graph applies here as well, telling us that promotions are a waste of time.

The other possibility is that both endpoints of the deleted tree edge are in the giant component. To understand this case, consider *any* spanning tree of the giant

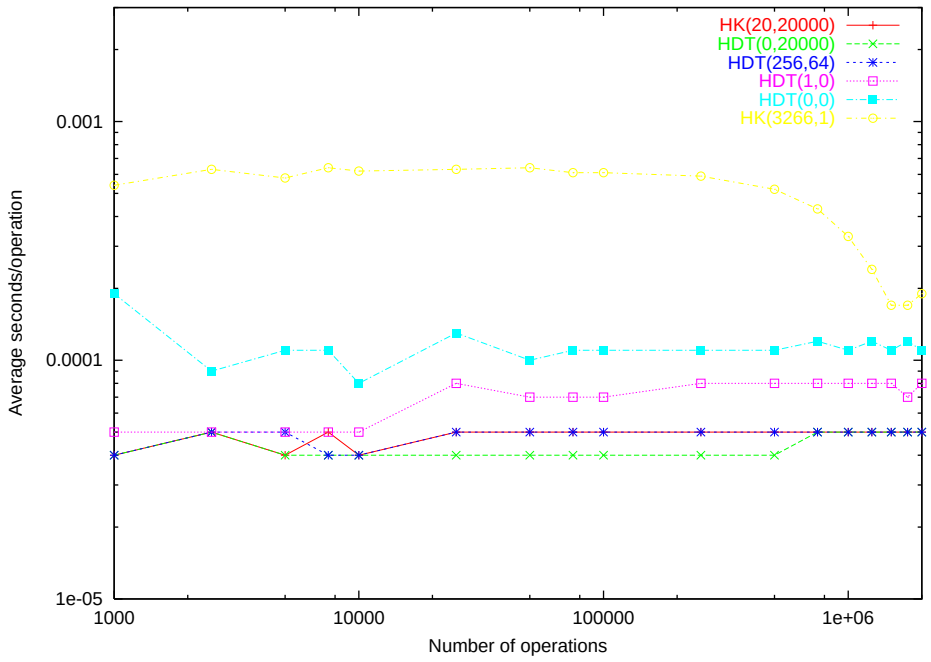

Fig. 2. Performance on random graphs: $n = 20,000$ and $m = 2n$. Time per operation averaged over a varying window.

component (we don't know which one is actually chosen by our data structure), and consider any non-tree edge with an endpoint in the smaller half of the spanning tree after our deletion. The other endpoint is a random vertex of the giant component, so is at least 50% likely to have its other endpoint in the other half of the spanning tree, and thus to serve as a replacement edge. If any given non-tree edge is 50% likely to serve as a replacement edge, then sampling will work extremely well for this case.

This analysis explains why the non-promoting algorithms work best. Either there are no replacement edges to test or, if there are replacement edges, we will test only a constant number (in expectation) before finding a replacement (note that the "exhaustive search" algorithms do not actually test random edges, but since the graph is random, their deterministic enumeration of edges fits the same analysis). The analysis also explains why HDT$(1,0)$ exhibits middle-of-the road performance. The algorithm sometimes wastes time promoting edges. However, its one random sample gives a reasonable chance of finding a replacement without resorting to promotions.

It is noteworthy that the standard HK implementation takes *substantially* longer to converge on this instance then in the previous instance. As we discussed before, convergence happens when all the tree edges have been brought down to lower levels of the graph. In this problem family, however, that takes a lot longer to happen (we directly measured this by instrumenting the source code). This delay is apparently due to a heuristic of Alberts et al. They propose, when inserting

a new edge, to put it in as high a level as possible subject to the spanning tree invariant. This makes intuitive sense: putting the new edge in at a high level gets it out of sight of low-level edges, which eases the search for replacement edges at the low levels. In this problem family, however, the heuristic works against us. Since initially all edges are at the top level, the giant component (and its spanning tree) are at the top level. A newly inserted edge is quite likely to be spanned by the giant component, thus inserted at the top level. This makes the large presence of top-level edges self-perpetuating, which in turn perpetuates the period in which a lot of ET-tree splits take place on tree edge deletions.

7.3 Semirandom Inputs

Next we considered semirandom graphs. We do not consider the $m = n/2$ case, since it degenerates to a number of independent components that (because we never change our edge set) never connect to each other, so we are basically managing a collection of disjoint ET-trees. In the case of $m = 2n$ edges, we know the graph will have some structure, as described in the section on random graphs. And this structure will persist over a long sequence of operations. How do the algorithms deal with this structure?

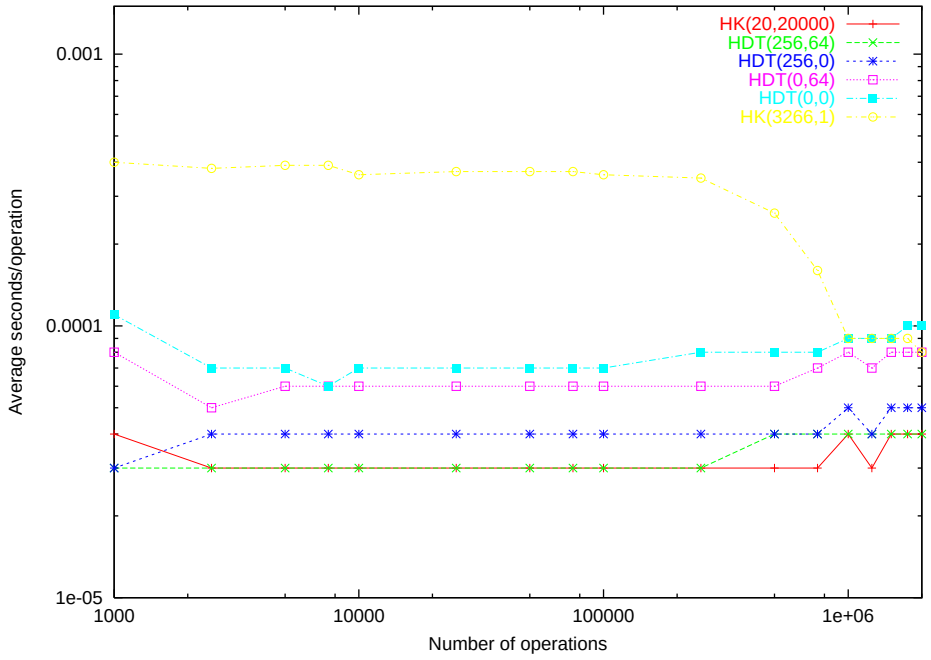

Fig. 3. Performance on a semirandom graph with $n = 2000$. Average time per operation over a sequence of measurement windows.

Figure 3 shows the performance of the algorithms on such a graph. Somewhat surprisingly, none of the algorithms "learn"—their amortized runtime is essentially the same as for truly random graphs. To see this more explicitly, consider Figure 4, which shows the performance of a few representative algorithms on both random and semirandom inputs. Every algorithm but one performs the same on both

inputs.

The one exception is vanilla HK, which starts out slower than the others and then converges to a faster speed than HDT$(0, 0)$. That HK samples is not an explanation because the other sampling algorithms outperform HK. Instead, the problem again is due to the Alberts heuristic. As above, this heuristic works against our desire to drop edges to low levels of the data structure. Random graph insertions are better at thwarting this heuristic than semirandom graph insertions.

More precisely, on a truly random graph at this density, since the giant component has size at most a constant fraction of n, it is reasonably likely that a new random edge insertion will connect two distinct components. Since such an edge is not spanned by the current graph, the Alberts heuristic cannot raise the edge. So the edge is placed at level 0 (and in the current spanning tree). This means that the graph's edges drop quickly to level 0—the key fact for the speedup.

Semirandom graphs take longer to speed up. The semirandom graph's candidate edges consist of a giant component of size $\Omega(n)$ plus a number of small trees. Let us consider only the giant component (a constant fraction of the operations happen there). At any given time, a few candidate edges are missing (deleted). But the giant component is still quite likely to be connected. So any giant component edge that we insert will be spanned by the current tree, so will be raised to the (high) level of the spanning tree by the Alberts heuristic. Once again, the high level of edges is self-perpetuating and takes substantial time to fade away.

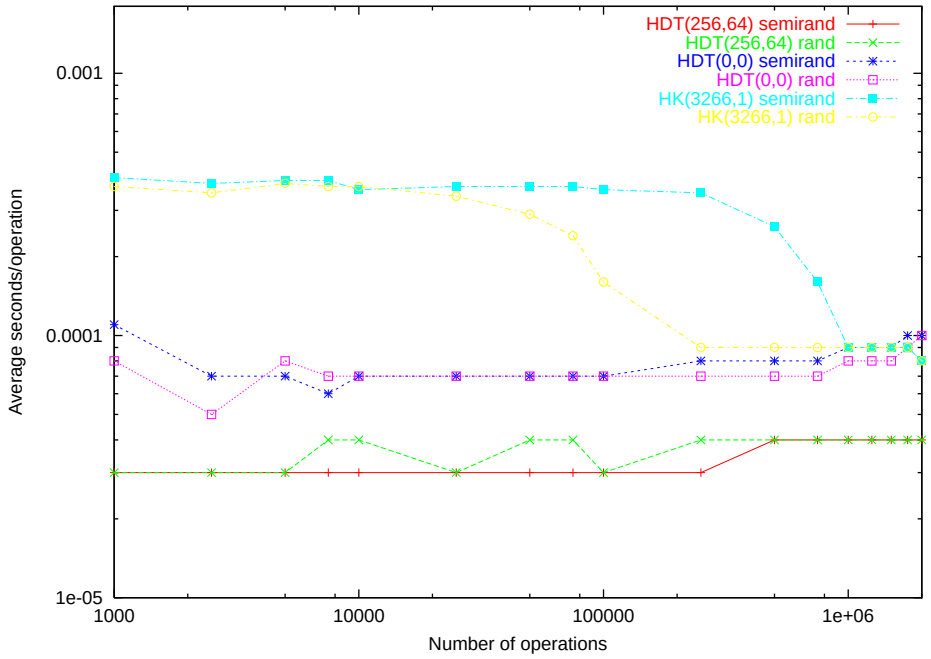

Fig. 4. Comparison of performance a random graph versus a semirandom graph: $n = 2000$, $m = 2n$. Time per operation averaged over a varying window of previous operations.

7.4 Two-Level Inputs

We now report results on two-level inputs: k cliques of size c, connected by $2k$ random interclique edges (analogous to the $m = 2n$ case of the previous section, with each vertex replaced by a clique). We start with all (clique and interclique) edges present and begin randomly inserting and deleting only the interclique edges. As we will see, the algorithms adapted to the structure of these inputs in order to run faster.

We used graphs of size $n = 2,000$, which is small. The limiting factor is the number of edges since these graphs can contain big cliques. The example we describe next, which consists of two cliques connected by an edge, has $m = 999,001$ edges.

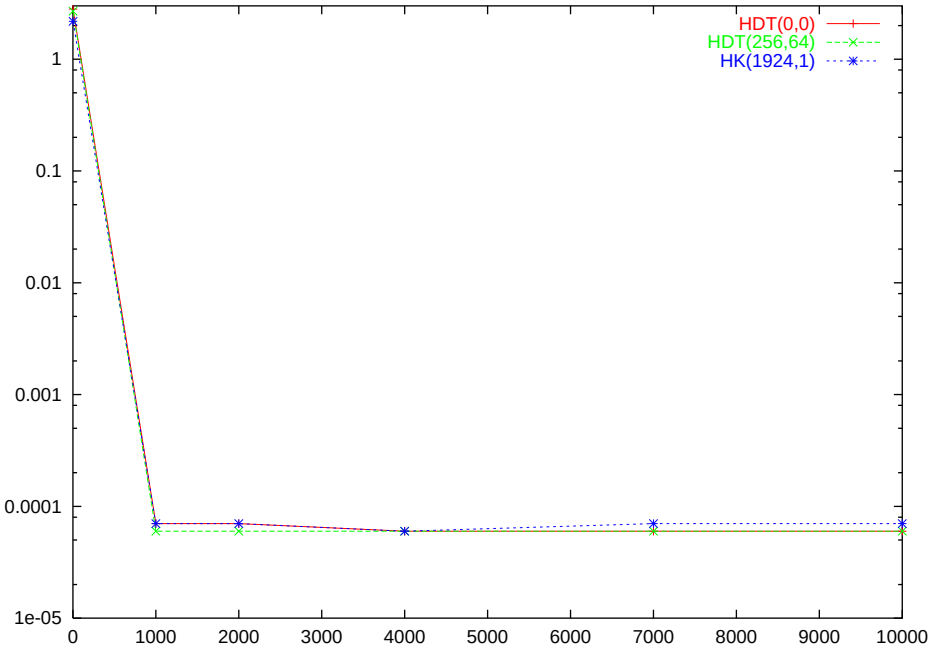

Fig. 5. Performance on two cliques connected by an edge.

The update sequence for this two clique input consists of repeatedly deleting and inserting the interclique edge. Although this input, considered also by Alberts et al., doesn't fit strictly into our two-level model (there should be four interclique edges), we discuss it because it highlights a general behavior of this input family. Figure 5 shows a plot of time per operation for this input, averaged over a varying number of previous operations. This plot immediately reveals an interesting story: the first operation, a delete of the interclique edge, is incredibly expensive, and all future updates are incredibly cheap.

When the edge is deleted, both HK and HDT exhaustively search all of the edges of one clique, finding no replacement. HDT promotes the clique edges up to level 1. Since the interclique edge is reinserted at level 0, when it is again deleted, HDT considers the smaller side of its tree at level 0, which consists of the single vertex from the promoted clique. Thus all future updates are trivial. HK begins with all

the initial edges of the graph at the top level of the data structure. Since the clique edges are never deleted, they remain at the top level forever. So the only way HK could consider the clique edges again is if its rebuilds floated the interclique edge back to the top level. But this doesn't happen since the edge is repeatedly deleted and dragged back down to level 0. So this input challenges both algorithms for one operation and then never again!

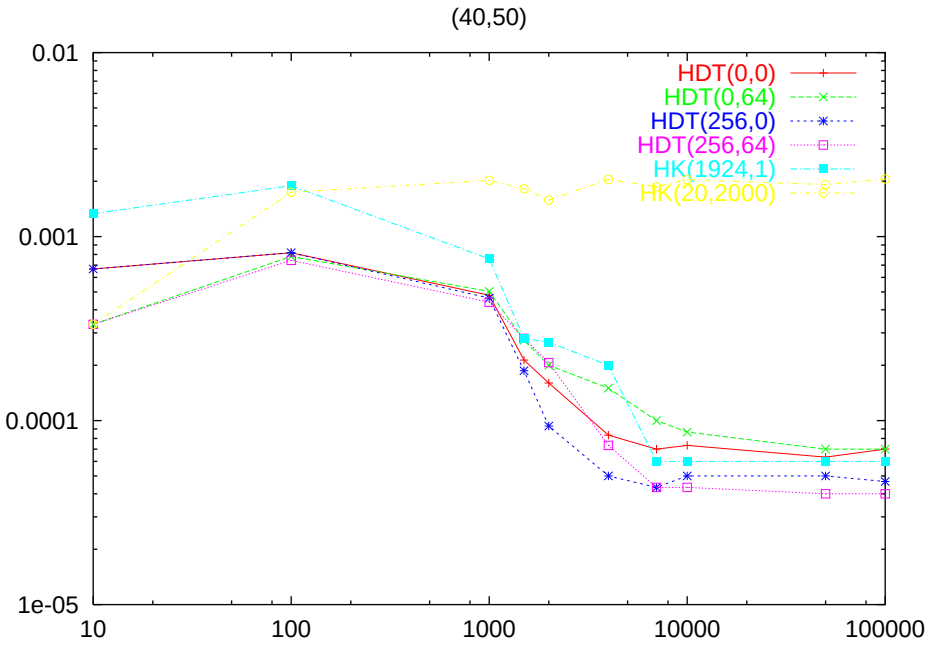

Fig. 6. Performance on 40 cliques of 50 vertices. Time per operation averaged over a varying window of previous operations.

This kind of behavior is replicated in our two-level model when $k < c$. In Figure 6 we present a representative example, $(k, c) = (40, 50)$. The plot tells the story. The performance of HK$(20, n)$ is poor and does not change much over time. This is because it maintains only one level of the data HK structure and therefore searches one of the cliques every time an interclique edge is deleted. These cliques are large and slow down the algorithm.

The adaptive algorithms are slow for the first 1000 operations and then quickly speed up. At this point many of the clique edges are higher in the data structure than the interclique edges and so few expensive non-replacement edge tests are performed. The algorithms ultimately reach a steady state where the clique edges are out of the way and this input behaves like a sparse semirandom graph on k nodes ($m = 2k$). We have observed behavior similar to this for values of k between 2 and 40; as k decreases, the performance of the algorithms becomes indistinguishable, aside from HK$(20, n)$.

Vanilla HK is initially slower than the HDT variants but eventually catches up. This is because, in the beginning, the clique edges are at the same levels as the interclique edges. Thus HK does not succeed in its sampling step (so takes $O(\log^3 n)$

time per operation). At this early stage the interclique edges (which work as replacement edges) make up only a tiny fraction of the edges sampled, so with high probability (roughly $1 - 4/c^2 = 0.9984$), sampling will fail. So the algorithm will sample $16 \lg^2 n \approx 1924$ edges with each sample costing $O(\lg n)$ time. After enough sampling failures (with consequent demotion of clique edges), sampling starts working so replacement edges are found much more quickly.

As for the HDT variants, notice that the sampling variants (HDT$(256, \cdot)$) ultimately perform better than the non-sampling variants (HDT$(0, \cdot)$) that always promote tree edges on a deletion. This makes sense: after sufficient time, most of the clique edges are at level 1 or higher while most of the interclique edges are at level 0. Thus, when an interclique edge is deleted (at level 0) we seek a replacement at level 0. Since this level is dominated by interclique edges, sampling finds one quickly. Just as in random graphs, such an interclique edge is likely to serve as a replacement edge, thus avoiding an expensive promotion of tree edges. This exactly matches the observations for the one-level semirandom graph model: there too, sampling is a beneficial heuristic for HDT. We expect that, as n increases, so too does the performance gap between the sampling and non-sampling variants on these two-level families (as is the case for large one-level graphs).

Once again, the use of a base case is beneficial, as the fastest algorithm is HDT$(256, 64)$.

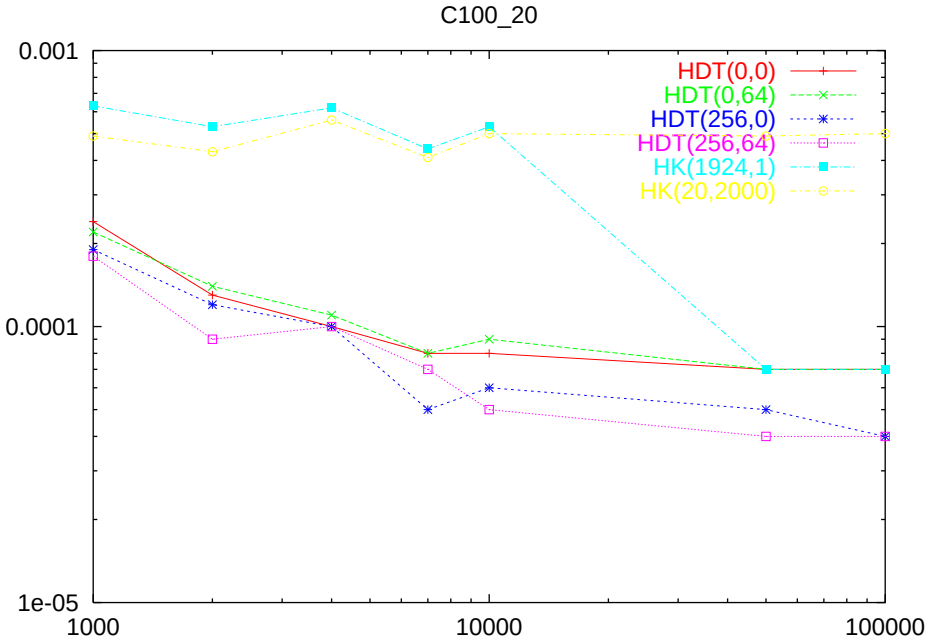

Fig. 7. Performance on 100 cliques of 20 vertices. Time per operation averaged over a varying window of previous operations.

Now let's look at an input where $k > c$, specifically $(k, c) = (100, 20)$, shown in Figure 7. First we notice that the algorithms take slightly longer to converge to a steady state (in both cases in the region of 10,000 operations). We also see a clear

separation between the HK and HDT variants. We explain this as follows. When $\mathrm{HK}(16\lg^2 n, 0)$ deletes an interclique edge for the first time, the edge is at the top level along with the cliques (recall the heuristic of Alberts et al. that places initial edges at the top of the data structure). This being the case, HK samples a large number of clique edges on every (first) deletion. By comparison, $\mathrm{HK}(20, n)$ samples clique edges on *every* deletion, so in the beginning of the run the algorithms behave similarly. After most of the interclique edges have been deleted once, $\mathrm{HK}(16\lg^2 n, 0)$ performs inserts and deletes mostly at the lower levels, which are free from clique edges.

Why do the HDT variants behave different from HK? Because HDT only searches through a clique when deleting an edge incident to two cliques that are *both* at same level as the edge. After that edge is deleted, one of the cliques is promoted out of the way. So HDT doesn't have to search through a clique as often.

7.5 Worst-case for HDT

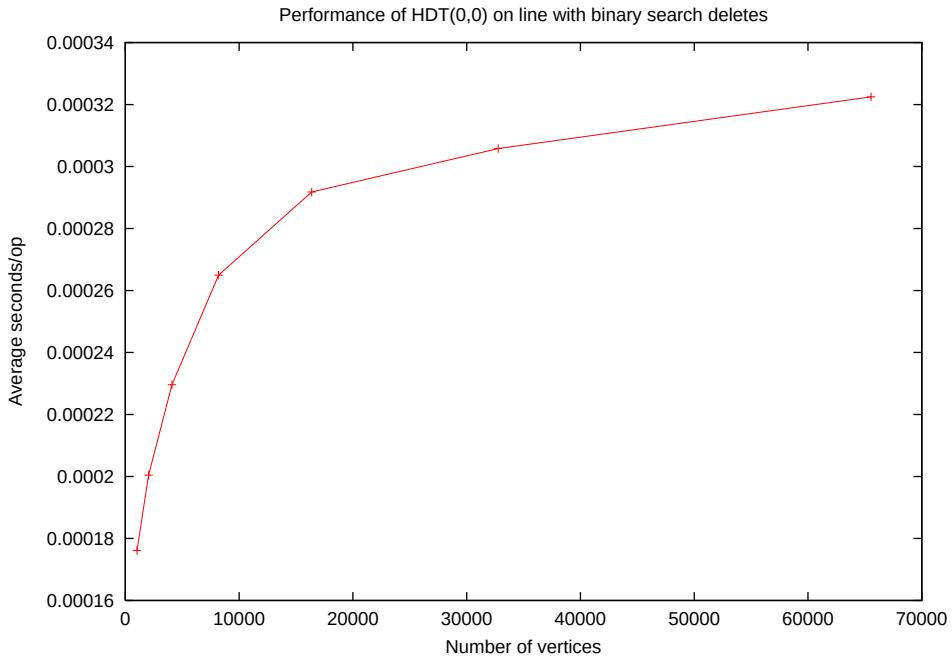

Fig. 8. Performance on worst-case input for HDT: amortized time per operation (averaged over a large number of operations) throughput as a function of n

The above plot shows the result of running vanilla HDT ($\mathrm{HDT}(0,0)$) on the worst-case input described in Section 5.4. The numbers show that at $n = 20000$, this HDT runs roughly 3 times slower on this instance than on the random graph instances. This plot does not exactly fit an $O(\log^2 n)$ curve; we suspect that lower order terms are still influencing the overall running time at these instance sizes which we were able to test. We verified this by plotting the performance divided by $\log^2 n$ and the performance divided by $\log n$. The performance dominates $\log n$ and is dominated by $\log^2 n$. We don't yet have a better explanation.

8. AN IMPROVEMENT TO HK

We tried and failed to construct an example that would force HK to run in its analyzed expected worst-case time of $O(\log^3 n)$ per update. Instead, we found two small twists to the HK algorithm that improve its running time to $O(\log^2 n)$ per update. This matches the bound of Henzinger and Thorup [Henzinger and Thorup 1997] which was achieved using quite complicated techniques. The two twists essentially close some slack in the HK analysis by drawing a careful distinction between the use of $\log n$ and $\log w(T_1)$, where $w(T_1)$ is the number of non-tree edges on the smaller tree side. It follows that only in cases where this difference is big will the original HK algorithm possibly use $\omega(\log^2 n)$ time.

To see that the above twists work requires a subtle change in the analysis. First we recall the relevant part of the HK algorithm, using the notation from Henzinger and King [Henzinger and King 1995].

8.1 The Henzinger-King approach

Henzinger and King sample $c \log^2 n$ edges in $O(\log^3 n)$ time. Edges with both endpoints in the smaller half-tree T_1 are twice as likely to be selected as edges with one endpoint in T_1 (replacement edges), but this constant-factor difference is irrelevant and we will ignore it. If a replacement edge is found during sampling, they immediately achieve an $O(\log^3 n)$ time bound for the operation. If no replacement edge is found, they identify the set S of replacement edges by checking all incident non-tree edges in $O(w(T_1) \log n)$ time.

If $|S| \leq \frac{w(T_1)}{c' \log m}$, where $m = O(n^2)$ is the maximum number of edges ever appearing in the graph, they test all $w(T_1)$ edges and demote all edges in S in $O(w(T_1) \log n)$ time. They use an elegant amortization argument to show that, for an appropriately chosen c', the sum of $w(T_1)$ over all the operations in which they demote edges is $O(\log^2 n)$ per update. Hence, the total cost of this case per update is $O(\log^3 n)$.

On the other hand, if $|S| > \frac{w(T_1)}{c' \log m}$, they return one edge from S as a replacement edge but *do not* demote. This is an expensive operation (since it is not amortized away by demotions), but rare. The probability that we fail to find a replacement edge for such large S during sampling is at most $(1 - 1/(c' \log m))^{c \log^2 n} = O(1/n^2)$ for $c = 2c'$. Hence, the expected cost of the operation when S is large is

$$(1 - O(1/n^2)) \cdot O(\log^3 n) + O(1/n^2) \cdot O(w(T_1) \log n) = O(\log^3 n).$$

By considering these cases, we have shown that *regardless* of the relative sizes of $|S|$ and $w(T_1)$, the expected time for exhaustive searches and demotions is $O(\log^3 n)$ per operation.

8.2 The variant

To simplify notation in the following discussion, we let s be the size of the set S of replacment edges and w be the total number of non-tree edges on the smaller side of the cut tree ($s = |S|$ and $w = w(T_1)$ from the previous discussion).

We make two small changes to the Henzinger-King algorithm. First, we make a slight modification to the ET-tree data structure which lets us sample edges in $O(\log w)$ time per sample (instead of $O(\log n)$ and perform an exhaustive search

in $O(w)$ time (instead of $O(w \log n)$. Second, we reduce the number of samples performed before reverting to an exhaustive search from $O(\log^2 n)$ to $O(\log n \log w)$. We also make sure to stop sampling as soon as we find a replacement—this is an obvious optimization, but Henzinger and King do not mention it since it does not improve their time bound. For our analysis it is crucial to stop.

These small changes in the algorithm are quite straightforward. More subtle is our analysis that shows the variant's improved running time. We closely follow the analysis of Henzinger and King. As we search the smaller side of a tree split by an edge deletion, we compare s, the number of replacement edges, to w, the total number of non-tree edges on the smaller side of the tree. We show that whether we succeed in sampling or perform a demotion, the expected amortized cost is $O(\log^2 n)$. To do so, we perform an *expected amortization* analysis, bounding the time spent sampling in an operation by the *expected* work spent demoting edges, which in turn is bounded by $O(\log^2 n)$ per operation.

8.3 A modified data structure

As a first step in improving the algorithm, we modify the data structure to reduce the time for each individual sampling operation or exhaustive search. Our ET-trees are based on treaps, meaning that each node gets a random priority and the tree is heap-ordered so that nodes with higher priority are closer to the root. We now interpret all nodes with incident non-tree edges as having higher priority than those without. Put another way, we represent the Euler tour using an "upper" balanced binary tree on the nodes with incident non-tree edges, each of whose "leaves" is the root of a balanced binary tree of nodes without incident non-tree edges. This only doubles the expected height of our binary trees, preserving all the original time bounds.

The importance of this change is that operations involving non-tree edges now appear to be running on trees with fewer nodes. If there are w non-tree edges incident on the tree being tested, there are $O(w)$ nodes in the upper tree. Thus, we can select a random non-tree edge in $O(\log w)$ time. If the opposite endpoint is in the same ET-tree, then, since it has an incident edge, it is also in the upper tree, so discovering this fact takes $O(\log w)$ time. If the opposite end is in a different ET-tree (which may have many more incident non-tree edges) finding the root may take more time. But even in this case the time to find the root is $O(\log m)$, and in this case we have found a replacement edge—this happens at most once per operation (since we stop as soon as we succeed in finding a replacement). So we can charge the $O(\log m)$ time of this single successful sample to the operation, and ignore it for the remainder of the analysis. This lets us reduce the sampling time from $O(\log n)$ per sample to $O(\log w)$ per sample.

Similarly, we can use the modified data structure in a faster exhaustive search. We provide each vertex of each ET tree with a "mark" bit, intially unset. To perform an exhaustive search, we begin by traversing the upper tree, setting all mark bits on the vertices we encounter in $O(w)$ time. Then, we traverse the upper tree a second time, again in $O(w)$ time. For each non-tree edge we encounter, we check whether the other endpoint of that edge is an unmarked vertex. Since all vertices in the upper tree have been marked, this occurs if and only if the other endpoint is not in the same tree, meaning the edge is a replacement edge. We test

$O(w)$ edges in $O(w)$ time. Finally, we traverse the upper tree again to unmark all the vertices in $O(w)$ time.

In the case where the number of replacement edges $s \leq w/c' \log m$, the algorithm prescribes demotion of all replacement edges. Since each demotion takes $O(\log n)$ time, the overall time spent on demotions is $O((w/\log m)\log n) = O(w)$.

In summary, we are able to perform an exhaustive search for a replacement edge, followed by demotions if needed, in $O(w)$ time.

8.4 Fewer samples

As we saw earlier, Henzinger and King proved that the total size of edge sets involved in demotions is $O(\log^2 n)$ per operation. Their argument only hinges on $s \leq \frac{w}{c' \log m}$ being the criterion for demotion. We use the same criterion, so the total cost of demotions remains $O(\log^2 n)$ per update. Thus, our improved data structure reduces the time for exhaustive searches followed by demotions to $O(\log^2 n)$ per operation. As with the original algorithm, exhaustive searches *not* followed by demotions will be so rare as to not matter. Thus, the key remaining step is to reduce the time spent on sampling. We do this by reducing the number of samples we perform before reverting to an exhaustive search from $c \log^2 n$ to $c \log n \ln w$.

We need to bound the work spent sampling and testing edges by $O(\log^2 n)$ per operation. At first glance this is tricky, since we perform $O(\log m \log w)$ samples at a cost of $O(\log w)$ each, for a cost of $O(\log m \log^2 w)$ time which might not be $O(\log^2 n)$. But we can amortize this bound away with a more careful analysis.

The Henzinger-King analysis showed that the total size of edge sets subject to demotions is $O(\log^2 n)$ per operation. Put another way, if we think of a demotion on w edges as providing w "credits," then the total number of credits issued to the algorithm as it runs is $O(\log^2 n)$ per operation. Therefore, so long as we prove that the total time spent sampling is less than the total number of credits issued, we will have our desired result. We show this by considering three distinct cases on the relative size of s versus w.

Case 1: $s \geq w/c' \log m$**.** In this case, each time we choose a random edge, it is in S with probability at least $1/c' \log m$. Thus, the expected number of samples before we find a replacement edge is $c' \log m = O(\log n)$. We terminate the search after $O(\log m \log w)$ samples, but this only reduces the expected number of samples taken. Thus, assuming we find a replacement edge, the expected work is $O(\log n)$ samples times $O(\log w)$ work per sample which is $O(\log^2 n)$. Since we give up after some number of samples, there is a chance we do not find a replacement edge and instead perform an $O(w)$-time exhaustive search for a replacement. However, the probability that we find no replacement is

$$\begin{aligned}(1 - s/w)^{c' \log m \log n} &\leq (1 - 1/c' \log m)^{c' \log m \ln w} \\ &\leq e^{-\ln w} \\ &= 1/w.\end{aligned}$$

It follows that we do $O(\log^2 n)$ expected work with probability at most 1 and $O(w)$ work with probability at most $1/w$, for an overall expected work bound

of $O(\log^2 n)$. Note that we made no reference to the credit scheme for this case, for no demotions take place.

Case 2: $1/2c' \log m \leq s \leq 1/c' \log m$**.** In this case, as in the prior one, the expected number of samples taken if we do find a replacement edge is $O(\log m)$, so the expected work is $O(\log^2 n)$. On the other hand, if we do not find a replacement, we perform $O(\log m \log w)$ samples at a total cost of $O(\log m \log^2 w)$ time. However, we also perform an exhaustive search and demotion which yields w credits. If $w \leq \log^3 n$, then our sampling time of $O(\log m \log^2 \log n) = O(\log^2 n)$. On the other hand, if $w \geq \log^3 n$, then our sampling time is still $O(\log^3 n) = O(w)$, meaning the time spent sampling is asymptotically dominated by the credits we receive by demoting . Thus, over the entire execution of the algorithm, we spend $O(\log^2 n)$ time per operation handling trees that fit Case 2.

Case 3: $s \leq w/c' \log m$**.** This is the complicated case. If we knew we were in it, we could perform an exhaustive search and demotion, which is already paid for in an amortized sense. As in the previous case, the w credits received from this demotion would cover the $O(\log m \log^2 w)$ time spent sampling for a replacement edge.

However, sometimes we do not notice that a demotion is possible: we perform many samples (exceeding the desired $O(\log^2 n)$ time bound) but discover a replacement edge before we give up. This prevents us from performing the demotions which would amortize away the excess time spent sampling. We show, however, that the demotions still amortize the sampling in an expected sense—if we expect to perform many samples, we also expect to receive enough credit to cover the sampling cost.

Let us determine the *expected* number of credits received during this operation. Recall that we are considering the case $s/w \leq 1/2c' \log m$. We demote if we find no replacement edge during sampling. This happens with probability

$$\begin{aligned}(1 - s/w)^{c' \log m \ln w} &\geq (1 - 1/2c' \log m)^{c' \log m \ln w} \\ &= \Omega(1/\sqrt{w}).\end{aligned}$$

If we do demote, we get a credit of w. Therefore, the expected number of credits received is $O(\sqrt{w})$. This exceeds the sampling work of $O(\log n \log^2 w)$ whenever $w > \log^6 n$. If on the other hand $w \leq \log^6 n$, the sampling time is $O(\log n \log\log n) = O(\log^2 n)$ without any credit accounting.

We have shown that in each operation, the sampling work is less than the expected credits received. The Henzinger-King analysis shows that over the entire execution of the algorithm, at most $O(\log^2 n)$ credits are received per operation. Thus the total *expected* number of credits over all operations is also bounded by $O(\log^2 n)$ per operation. It follows that the total time spent sampling edges in Case 3 is also upper bounded by $O(\log^2 n)$ per operation.

9. CONCLUSION

We have completed a study of the relative behaviors of the two theoretically fastest dynamic connectivity algorithms. Our experiments demonstrates the ways the algorithms learn the structure of their inputs over time. We have explored some simple heuristics that substantially improve the performance of the HDT algorithm. With

these heuristics, that the better theoretical performance of HDT is borne out by better practical performance.

The chief observations of this work are:

—Over a large number of operations, HDT and HK perform similarly on the instances we tried. In the early part of the run, HDT outperforms HK.

—Sampling to try for a quick edge replacement before working harder is an important heuristic that improves the performance of HDT beyond HK. Using a base case also helps.

—Random graphs are easy and uninteresting. Special purpose algorithms can do incredibly well on them.

—The algorithms can effectively learn structure. They did so on two-level graphs.

—WThere is a simple, provable worst-case input family for HDT.

—The original HK algorithm's expected running time of $O(\log^3 n)$ can be improved to $O(\log^2 n)$ via a simple modification.

There are several directions for future work. One is to build a more realistic structured network models and study the algorithms' behavior on them. One natural choice is the Waxman [Waxman 1988] model, of which our semirandom graphs are a degenerate case. The Waxman model puts points in the plane and creates random links with probability biased in favor of linking nearby points. These graphs tend to have natural hierarchical structures so we expect the algorithms will extremely well on them.

Another interesting direction is the construction of additional worst-case inputs. Our worst-case input for HDT is quite specialized, and we have no worst case for HK. Is there a family of graphs and updates that is generically "hard" for dynamic connectivity algorithms?

Another direction involves the study of graphs which have structure that changes over time. Are there inputs that continually stress the algorithms, whose structure changes after a while and forces the algorithms to adapt? Our initial experiments in this directions involved starting with a k,c graph and after a large number of operations, evolving into another k,c graph. This fails to stress the algorithms because the huge number of inserts of new k,c edges pays for the expensive deletions.

Another direction is to attempt to develop an experimental measure of the asymptotic run times of these algorithms. The graph sizes we were able to work with were (due to memory limitations) too small to let us clearly distinguish the rates of constant, $\log n$, and $\log^2 n$. With substantially larger graphs these distinctions might be drawn.

ACKNOWLEDGMENTS

We are deeply grateful to David Alberts, Giuseppe Cattaneo, and Giuseppe Italiano for making available their implementation of the Henzinger-King algorithm. If we have been able to implement anything, it is because we have stood on the shoulders of giants.

REFERENCES

Alberts, D., Cattaneo, G., and Italiano, G. F. 1996. An experimental study of dynamic

graph algorithms. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)* (1996), pp. 192–201.

Bollobás, B. 1985. *Random Graphs*. Harcourt Brace Janovich.

Calvert, K., Doar, M., and Zegura, E. W. June 1997. Modeling internet topology. *IEEE Communications Magazine, June 1997 35*, 6, 160–163.

Doar, M. B. 1996. A better model for generating test networks. In *Proceedings of Gobecom '96* (1996). Also at `ftp://ftp/nexen.com/pub/papers`.

Eppstein, D., Galil, Z., Italiano, G. F., and Nissenzweig, A. 1997. Sparsification — a technique for speeding up dynamic graph algorithms. *Journal of the ACM 44*, 5 (Sept.), 669–696. See also FOCS'92.

Frederickson, G. N. 1985. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Computing 14*, 4, 781–798. See also STOC'83.

Fredman, M. and Henzinger, M. 1998. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica 22*, 3, 351–362.

Henzinger, M. R. and King, V. 1995. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proc. 27th Symp. on Theory of Computing* (1995), pp. 519–527.

Henzinger, M. R. and Thorup, M. 1997. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Structures and Algorithms 11*, 369–379. See also ICALP'96.

Holm, J., de Lichtenberg, K., and Thorup, M. 1998. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. 30th Symp. on Theory of Computing* (1998), pp. 79–89.

Luczak, Pittel, B., and Wierman. 1994. The structure of a random graph at the point of the phase transition. *TAMS: Transactions of the American Mathematical Society 341*.

Mehlhorn and Naher. 1995. LEDA: A platform for combinatorial and geometric computing. *CACM: Communications of the ACM 38*.

Miltersen, P. B., Subramanian, S., Vitter, J. S., and Tamassia, R. 1994. Complexity models for incremental computation. *Theoretical Computer Science 130*, 1, 203–236.

Sleator, D. and Tarjan, R. 1983. A data structure for dynamic trees. *Journal of Computer and System Sciences 26*, 3, 362–391. See also STOC'81.

Waxman, B. M. 1988. Routing of multipoint connections. *IEEE Jour. Selected Areas in Communications (Special Issue: Broadband Packet Communications) 6*, 9 (Dec.), 1617–1622.

Zegura, E. W., Calvert, K. L., and Bhattacharjee, S. 1996. How to model an internetwork. In *Proc. 15th IEEE Conf. on Computer Communications (INFOCOM)* (1996), pp. 594–602.

Zegura, E. W., Calvert, K. L., and Donahoo, M. J. 1997. A quantitative comparison of graph-based models for internet topology. *IEEE/ACM Transactions on Networking 5*, 6 (December), 770–783.