Worst-case time decremental connectivity and k-edge witness

Andrew Twigg

Computing Laboratory

Oxford University

andy.twigg@comlab.ox.ac.uk

January 24, 2014

Abstract

We give a simple algorithm for decremental graph connectivity that handles edge deletions in worst-case time $O(k \log n)$ and connectivity queries in $O(\log k)$, where k is the number of edges deleted so far, and uses worst-case space $O(m^2)$. We use this to give an algorithm for k-edge witness ("does the removal of a given set of k edges disconnect two vertices u, v?") with worst-case time $O(k^2 \log n)$ and space $O(k^2 n^2)$. For $k = o(\sqrt{n})$ these improve the worst-case $O(\sqrt{n})$ bound for deletion due to Eppstein et al. We also give a decremental connectivity algorithm using $O(n^2 \log n / \log \log n)$ space, whose time complexity depends on the toughness and independence number of the input graph. Finally, we show how to construct a distributed data structure for k-vertex witness by giving a labeling scheme. This is the first data structure for k-vertex witness that can efficiently distributed without just giving each vertex a copy of the whole structure. Its complexity depends on being able to construct a linear layout with good properties.

Keywords: k-edge witness, decremental connectivity, labeling schemes

1 Introduction

The k-edge witness problem is this: preprocess an undirected graph G so that we can decide if a given set of k edges disconnects two vertices u, v, i.e. is the set of edges a witness to the fact that u, v are not k-connected? The problem is closely related to decremental graph connectivity, where we support the following operations: $\text{DELETE}(u, v)$, which deletes edge $\{u, v\}$ from the current graph, $\text{CONNECTED}(u, v)$, which returns TRUE iff u, v are still connected, and CONNECTED, which returns TRUE iff the graph is still connected. The vertex version of the problem is similar except that we instead support $\text{DELETE}(u)$, which deletes a single vertex and all its adjacent edges.

Graph connectivity plays a crucial role in many problems and dynamic connectivity is an intensively studied problem (see [9] for references). Incremental connectivity (only handling insertions) is equivalent to the union-find problem [12]. The best bounds for the decremental case are amortized expected – Henzinger, King and Thorup [9, 10] used randomization to support updates in $O(\log^2 n)$ expected amortized time, answering queries in $O(\log n / \log \log n)$. In 1998, Holm et al. [11] obtained a deterministic algorithm with $O(\log^2 n)$ amortized time (this is amortized over adding all the edges to the graph *then* deleting them, hence it cannot give good bounds for k-edge witness). The best worst-case bound for decremental connectivity in general graphs is due to Eppstein et al. [4] who improved the result of Frederickson [6] from $O(\sqrt{m})$ to $O(\sqrt{n})$ per update, answering queries in constant time.

In this paper we improve the worst-case bound when the number of deletions is small ($o(\sqrt{n})$), and moreso when the graph has small cutwidth and path covering number, or has a spanning tree with small maximum degree. Our algorithms are simple and reduce the problem of maintaining decremental connectivity to maintaining fully dynamic connectivity (supporting both insertions and deletions) on some auxiliary graph H, which usually has size linear in the number of deletions

already performed. We can then use known algorithms to maintain connectivity on H, and answer queries on G by quickly translating them to queries on H. An artefact of our approach is that the time for a deletion depends on the number of vertices or edges already deleted from the original graph, which explains why it only works for small numbers of deletions.

The following is a summary of our results.

- An algorithm for decremental connectivity using worst-case space $O(m^2)$, handling the kth edge deletion in worst-case time $O(k \log n)$ and connectivity queries in $O(k^2)$, where k is the number of edges deleted from G so far. For $k = o(\sqrt{n})$, this improves the deletion time due to Eppstein et al. [4]. This gives an an $O(k^2 \log n)$-time algorithm for k-edge witness using space $O(k^2 n^2)$.

- An alternative algorithm handles deletions in time $O(k^{3/2} + k \log n)$ and connectivity queries in time $O(\log k)$, which improves on Eppstein et al. for $k = o(n^{1/3})$ but has the advantage of small query time.

For a graph $G = (V, E)$ let $d_G(v)$ be the degree of vertex $v \in V$, and $\Delta(G) = \max_{v \in V} d_G(v)$. Let T be a spanning tree with minimal $\Delta(T)$ over all spanning trees of G.

- We give a decremental connectivity algorithm using space $O(n^2 \log n / \log \log n)$, handling the kth deletion in time $O(\Delta(T)^2 \log n + k \log n)$ and connectivity queries in $O(k^2)$. For Hamiltonian graphs, graphs with bounded independence number, $1/O(1)$-tough graphs and almost all r-regular graphs (for fixed $r \geq 3$), this gives $O(k \log n)$ time for deletions.

We also construct a *distributed* data structure for k-vertex witness using a *labeling scheme*: given a graph, assign short labels L to the vertices so that given only $L(u), L(v)$ and $L(S) = L(s_1), \ldots, L(s_k)$, we can efficiently decide if S is a uv-cut of size at most k in G. We are not aware of any other decremental connectivity algorithm that can be efficiently distributed without giving each vertex a copy of the whole data structure.

In the full version of the paper, we give algorithms for decremental connectivity under vertex deletions with similar times.

2 Trees

As an introduction, we show how to solve k-edge witness and k-vertex witness on n-vertex trees in worst-case time $O(k)$. A vertex w is a uv-cut in a tree iff the least common ancestor of u, v is an ancestor of w and w is an ancestor of u or v. Bender et al. [1] give a least common ancestor algorithm that answers queries in constant time and uses $O(n)$ space. Hence we can solve k-vertex witness in time $O(k)$ by testing each vertex s_i of the query set $s_1, \ldots, s_k$ and answering 'yes' iff any s_i is a uv-cut. For k-edge witness, see that an edge (x, y) (where y is the parent of x) is a uv-cut iff y is a uv-cut. To transform this into a decremental connectivity algorithm for trees, store the set of edges deleted so far, and answer each connectivity query in time $O(k)$ by solving k-edge witness.

Interestingly, Bender et al. [1] use a reduction to the RMQ (range minimum query) problem to solve LCA, which in turn uses the Euler tour data structures. This is quite similar to our connectivity algorithms, which we now describe.

3 Euler tour

In this section we present an algorithm for decremental graph connectivity based on an Euler tour representation of the graph. An Euler tour of a graph is a tourpath that traverses every edge exactly once. A well-known theorem of Euler states that a graph has an Euler tour iff every vertex has even degree. A simple trick to ensure that G has an Euler tour is to 'double up' each edge, so it gets traversed once in each direction.

In an Euler tour $ET(G)$ of G, each edge is visited twice (traversed once in each direction) and every degree-d vertex d times. Each time any vertex u is encountered in the tour, we call this an occurence of u and denote the set of occurences of u by $O(u)$, and we shall refer to a particular occurence by its unique position in the tour. If the sequence $ET(G)$ is stored in a balanced binary search tree, then one may insert an interval or splice out an interval (delete an edge of the tour) in time $O(\log n)$, while maintaining the balance of the tree.

Henzinger and King [9] use an Euler tour data structure to represent an arbitrary spanning tree of G in their randomized dynamic connectivity algorithm. In our algorithm we construct an Euler tour $ET(G)$ of the whole graph. We maintain an auxiliary graph H where each vertex of H represents a connected interval of $ET(G)$ (i.e. a connected subpath of the Euler tour), and the vertices of H form a disjoint partition of the tour. There is an edge between two vertices of H iff there is some vertex u with an occurence in both intervals. Let $h(u)$ be an arbitrary vertex of H whose interval in $ET(G)$ contains an occurence of u. We also maintain a binary search tree on the intervals of H so that in worst-case time $O(\log|H|)$ we can find $h(u)$ for any $u \in G$.

Lemma 1 *Vertices $u, v \in G$ are connected in G iff $h(u), h(v)$ are connected in H.*

Proof. This follows by the definition of H – every path from u to v in G corresponds to a set of paths from $h(u)$ to $h(v)$ in H, and every path from u to v in H corresponds to a set of paths from u to v in G. $\qquad\square$

Now we must maintain connectivity on H under vertex insertions, and both edge deletions and insertions. A simple method is to just store its adjacency list representation, then each edge insertion and deletion takes $O(1)$ time, but connectivity queries are answered by using any algorithm for USTCONN on H (e.g. a depth-first search) and take $O(|H|^2)$ time. An alternative is to use the fully-dynamic algorithm of Eppstein which handles edge insertions and deletions in time $O(\sqrt{|H|})$ and answers connectivity queries in time $O(1)$.

The crucial part of our algorithm is the ability to efficiently test for an edge in the graph H. We do this by using two-dimensional orthogonal range queries as follows. Given some graph G and a unique identifier $I(u) \in [1 \mathinner{.\,.} n]$ for each vertex u, an edge $\{u, v\}$ of G is a point $(I(u), I(v))$ on an $n \times n$ grid. Then there is an edge in G with endpoints in the intervals $[a \mathinner{.\,.} b], [c \mathinner{.\,.} d]$ iff the box $(a, c) \times (b, d)$ is nonempty, as illustrated in Figure 3.
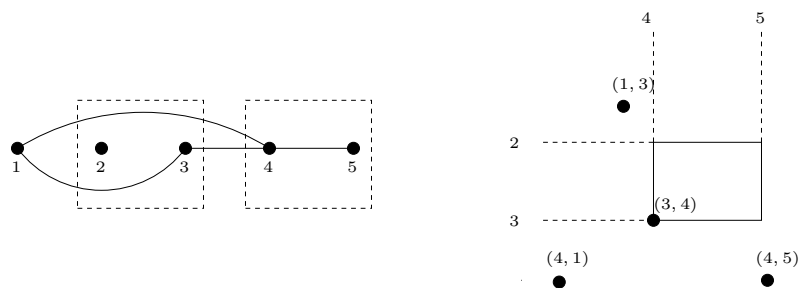
Figure 1: The box $(2, 4) \times (3, 5)$ is empty iff there are no edges between the intervals $[2 \mathinner{.\,.} 3]$ and $[4 \mathinner{.\,.} 5]$ on the graph

This applies to our algorithm as follows. For each pair of occurences u_i, u_j of vertex u, add a point (u_i, u_j) to the range tree. Now, each vertex of H represents a connected interval $[i \mathinner{.\,.} j]$ on the Euler tour. Then two vertices $[a \mathinner{.\,.} b], [c \mathinner{.\,.} d] \in H$ are adjacent iff there exists some vertex u with occurences in $[a \mathinner{.\,.} b]$ and $[c \mathinner{.\,.} d]$ iff the box $(a, c) \times (b, d)$ is nonempty. Orthogonal range searching has been extensively studied during the last thirty years with applications to databases and computational geometry. This is the first time we know they have been used for graph connectivity. Chazelle [2] has given an algorithm for the static case that stores r points with space $O(r)$ and answers emptiness queries in worst-case time $O(\log r)$ (the algorithm of this section does not remove points).

Now we can describe the algorithm. To delete a edge $\{u, v\}$, the Euler tour is spliced at this edge in time $O(\log n)$, which corresponds to splitting some vertex h of H into two vertices h_1, h_2. The edges of h_1 union the edges of h_2 are equal to the edges of h, so we need only test each of the neighbours of h with h_1 and h_2 to see if there should be an edge. By definition of H, there is an edge between two vertices of H iff their corresponding intervals on the Euler tour contain an occurence of the same vertex of G. This test is done in worst-case time $O(\log n)$ for each edge by asking the range tree if some box is empty. A connectivity query for u, v is handled by finding $h(u), h(v)$ and then calling $\text{CONNECTED}_H(h(u), h(v))$. To handle the query CONNECTED, we simply call CONNECTED_H since G is connected iff H is connected.

INITIALISE(G)

1 Double up each edge of G
2 $ET(G) \leftarrow$ an Euler tour of G
3 **for** each vertex u
4   **do** $O(u) \leftarrow$ the set of occurences of u in $ET(G)$
5   **for** each pair $u_i, u_j \in O(u)$
6   **do** RANGE-TREE-INSERT(u_i, u_j)
7 Let H be a graph with a single vertex $[1 \mathinner{.\,.} 2m]$.

DELETE(u, v)

 1 **for** each occurence i of edge $\{u, v\}$ in $ET(G)$
 2   **do** EULER-TOUR-DELETE(i) $\triangleright\ O(\log n)$
 3         $[a \mathinner{.\,.} b] \leftarrow h(i)$ $\triangleright\ O(\log |H|)$
 4   INSERT-EDGE$_H([a \mathinner{.\,.} i])$ $\triangleright\ O(1)$
 5   INSERT-EDGE$_H([i + 1 \mathinner{.\,.} b])$
 6   **for** each neighbour $[c \mathinner{.\,.} d]$ of $[a \mathinner{.\,.} b]$ in H
 7   **do if** BOX-NOT-EMPTY$((a, c), (i, d))$ $\triangleright\ O(\log n)$
 8   **then** INSERT-EDGE$_H([a \mathinner{.\,.} i], [c \mathinner{.\,.} d])$ $\triangleright\ O(1)$
 9   **if** BOX-NOT-EMPTY$((i + 1, c), (b, d))$
10   **then** INSERT-EDGE$_H([i + 1 \mathinner{.\,.} b], [c \mathinner{.\,.} d])$
11   DELETE-VERTEX$_H([a \mathinner{.\,.} i])$ $\triangleright\ O(\Delta(H))$

CONNECTED(u, v)

    $\triangleright$ Returns true iff u, v are connected in G
1 $[a \mathinner{.\,.} b] \leftarrow h(u)$ $\triangleright\ O(\log |H|)$
2 $[c \mathinner{.\,.} d] \leftarrow h(v)$
3 **return** CONNECTED$_H([a \mathinner{.\,.} b], [c \mathinner{.\,.} d])$ $\triangleright\ O(|H|^2)$

CONNECTED

    $\triangleright$ Returns true iff G is connected
1 **return** CONNECTED$_H$ $\triangleright\ O(|H|^2)$

k-EDGE-WITNESS$(u, v, \{x_1, y_1\}, \ldots, \{x_k, y_k\})$

1 $G' \leftarrow$ a sparse k-connectivity certificate of G
2 INITIALISE(G')
3 Let H contain a single vertex $[1 \mathinner{.\,.} 2m]$
4 **for** each $i \in \{1, \ldots, k\}$
5   **do** DELETE$(\{x_i, y_i\})$
6 **return** CONNECTED(u, v)

3.1 Complexity

After k edges have been deleted from G, H has at most $k + 1$ vertices. Each edge occurs twice in the euler tour, so the loop at line 1 of DELETE only incurs a constant factor. It takes time $O(\log n)$

to splice out an interval of the euler tour, and each range tree query takes time $O(\log n)$. If we use the adjacency list to represent H, the times are as shown in the code. The loop at line 6 is repeated $O(\Delta(H))$ times. Hence procedure k-EDGE-WITNESS takes worst-case time $O(k^2 \log n)$. Additionally, in time $O(k^2)$ we can construct the reachability matrix for all vertices of H to subsequently test if the same set of edges is a witness for the non-k-connectedness of any pair of vertices in G, in time $O(\log k)$.

Alternatively, using the fully-dynamic connectivity algorithm of Eppstein et al. makes lines 4,5,8,9 take time $O(\sqrt{|H|})$ and line 11 takes time $O(\Delta(H)\sqrt{|H|})$, so DELETE takes worst-case time $O(k^{3/2} + k \log n)$, but CONNECTED(u, v) and CONNECTED take $O(\log k)$ time.

The space requirement is dominated by the cost of storing the vertex occurences in the range tree, using $O(\sum_{v \in G} d_G(v)^2) = O(m^2)$ bits. If we are only interested in cuts of size at most k (as for k-edge witness), we can use less space. A *sparse k-connectivity certificate* for a graph G is a subgraph G' containing at most kn edges, such that any cut of value at most k in G has the same value in the certificate. Nagamochi and Ibaraki [14] give a linear-time algorithm for computing sparse k-connectivity certificates.

The modified algorithm is the same except that we replace G with its sparse k-connectivity certificate in INITIALISE. The correctness follows from the properties of the connectivity certificate. The space complexity is reduced to that needed to store the vertex occurences in the range tree for the Euler tour of the certificate, i.e. $O(\sum_{v \in V} d_{G'}(v)^2) = O(k^2 n^2)$ bits in the worst case that some vertex has almost all the edges. If the certificate has degree at most k then the space requirement is reduced to $O(k^2 n)$ bits. The query time is unchanged, as it only depended on the set of edges being deleted.

4 An algorithm using less space

In this section we give an decremental algorithm that uses space $O(n^2 \log n / \log \log n)$ and handles the kth deletion in time $O(\Delta(T)^2 \log n + k \log n)$. It answers connectivity queries in time $O(k^2)$, and therefore gives an $O(k\Delta(T)^2 \log n + k^2 \log n)$-time algorithm for k-edge witness, which is $O(k^2 \log n)$ for Hamiltonian graphs, graphs of bounded independence number, $1/O(1)$-tough graphs and almost all r-regular graphs (for fixed $r \geq 3$). As in the previous section, we can also use a different algorithm to maintain connectivity on the auxiliary graph and this gives $O(\Delta(T)^2 \log n + k^{3/2} \log n)$ time for deletions but with $O(\log k)$ query time.

The algorithm is most similar to that of Henzinger and King [9] with the main exception that instead of *maintaining* a spanning forest of G, we do not bother to patch up tree edges when they are removed. Instead, we keep track of the fragmented parts of the forest as edges are removed (using a dynamic connectivity algorithm) and use this information to efficiently answer the connectivity queries. Consider a graph $G = (V, E)$ and a spanning tree $T = (V, F)$. We construct the Euler tour $ET(T)$ on T with its edges doubled up by calling the following procedure with the root vertex. The sequence produced can be stored in a binary search tree so that tree edge deletions (interval splicing) take time $O(\log n)$.

ET(v)

```
1   visit v
2   for each child u of v
3        do ET(u)
4            visit u
```

We maintain a graph H whose vertices are intervals on the Euler tour $ET(T)$ of the spanning tree T, and build the range tree with the points representing pairs of vertex occurences in $ET(T)$. The number of points is at most $\sum_{v \in T} d_T(v)^2 = O(n \min(n, \Delta(T)^2))$ since T is a tree and so $\sum_{v \in T} d_T(v) = n - 1$. However, we must also handle the nontree edges of G. We do this by adding to the range tree, for each *nontree* edge $\{u, v\} \in G$, a point (u_i, v_j) for each pair of occurences

5

$u_i \in O(u), v_j \in O(v)$. Since each vertex appears in the Euler tour at most $\Delta(T)$ times, this is

$$\sum_{\{u,v\} \in E \setminus F} d_T(u)d_T(v) \leq \sum_{\{u,v\} \in E} d_T(u)d_T(v) = \left(\sum_u d_T(u)\right)^2 = (2n-2)^2$$

for any spanning tree $T = (V, F)$ of $G = (V, E)$. This dominates the space used to store the vertex occurences and so we use at most $O(n^2)$ points in total.

Assume that we represent the auxiliary graph H with its adjacency list (so that edge operations are $O(1)$). Deleting a tree edge is handled as before: we delete the two appearences of the edge from the Euler tour of T, which splits some vertex of H into two vertices, and then test the new edges of the new vertices of H using emptiness queries. This takes total time $O(k \log n)$, after k edges have been deleted. Deleting a nontree edge $\{u, v\}$ has no effect on the Euler tour but now we must delete all the points $(u_i, v_j) \in O(u) \times O(v)$ from the range tree, corresponding to the edge $\{u, v\}$. Mortensen [13] has given a dynamic range query data structure which handles emptiness queries and deletions in worst-case time $O(\log r)$ and uses space $O(r \log r / \log \log r)$ for r points. Therefore our algorithm handles the kth deletion in worst-case time $O(\Delta(T)^2 \log n + k \log n)$ and uses space $O(n^2 \log n / \log \log n)$.

4.1 Complexity

The algorithm relies on being able to construct a spanning tree of G with small maximum degree, or more accurately, if $\{u, v\}$ is a *nontree* edge of G then the product $d_T(u)d_T(v)$ should be small, i.e. we want all the high degree vertices of T to be neighbours in T also.

Let $\Delta^*(G, T)$ be the smallest integer d such that G has a spanning tree of maximum degree d. It is NP-hard to determine exactly $\Delta^*(G, T)$ since $\Delta^*(G, T) = 2$ iff G has a Hamiltonian path, which is NP-complete [8]. Furer and Raghavachari [7] give a polynomial-time algorithm that outputs a spanning tree T with degree at most $\Delta^*(G, T) + 1$. A theorem of Dirac [3] says that if each vertex of G has degree at least $n/2$ then G contains a Hamiltonian cycle. A graph $G = (V, E)$ is t-tough if the number of connected components of $G \setminus S$ is at most $|S|/t$ for every cutset $S \subseteq V$. In 1989, Win proved the following theorem.

Theorem 1 ([16]) *Let t be a positive integer. Every $1/t$-tough graph G has a spanning tree of degree $t + 2$, i.e. $\Delta^*(G, T) = O(t)$.*

Combining the above theorem with the algorithm we get the following result.

Theorem 2 *Let G be $1/t$-tough. There is a decremental connectivity algorithm with polynomial preprocessing time for G where the kth deletion takes time $O(t^2 \log n + k \log n)$ and connectivity queries $O(k^2)$, using space $O(n^2 \log n / \log \log n)$. Also, k-edge witness can be solved in time $O(kt^2 \log n + k^2 \log n)$.*

5 A distributed data structure for k-vertex witness

In this section we show how to construct a distributed data structure for k-vertex witness by giving a labeling scheme for the problem. This consists of a marker algorithm that takes the graph and outputs a binary string called a *label* $L(u)$ for each vertex u. There is also a decoder algorithm that, given only $L(u), L(v)$ and $L(S) = L(s_1), \ldots, L(s_k)$, decides if the set S of vertices is a uv-cut in G. We give only a brief outline here, and defer a more detailed exposition to the full version of the paper.

We consider two variants of the problem: in the first, each vertex u has a set $S(u)$, which is given to the marker algorithm, and the decoder takes $L(u), L(v)$ and decides if $S(u)$ is a uv-cut in G. In the second variant, the marker algorithm only has knowledge of the graph G, and the decoder takes $L(u), L(v)$ and $\{L(s_i)\}$ for each $s_i \in S$, and decides if S is a uv-cut in G.

5.1 The sets are given to the marker algorithm

When the sets are given to the marker algorithm, the problem becomes how to efficiently represent the sets, and the partitions they induce in G (if any). For the case where S is a set of edges, we can use the tree layout algorithm of Section 4 to compute an auxiliary graph H_u corresponding to each set $S(u)$. The label $L(u)$ then stores the following: u's its identifier in the tree (from the postorder traversal), a mapping $g_u : \{1, \ldots, n\} \to \{0, 1\}$ from vertex ids in the tree to binary values, where $g_u(v) = 0$ iff the vertex of H_u containing v can be reached from the vertex of H_u containing u, i.e. iff the set $S(u)$ is a uv-cut in G. This gives a labeling scheme where the label $L(u)$ is of size $O(|H_u|) = O(|S(u)|)$ bits, since the mapping g_u can be stored using a binary search tree on the intervals of H_u (where the key is the smallest identifier contained in the interval). Given two labels $L(u), L(v)$, the decoder algorithm looks up the identifier of v in g_u and returns the binary value stored there, in time $O(\log |S(u)|) = O(\log n)$. If S is a set of vertices, we can use the same method but with $H(u)$ having $O(\mathrm{degree}(G)|S|)$ vertices.

5.2 The sets are not given to the marker algorithm

The algorithm in Section 3 used an Euler path to efficiently represent the edge cuts of a graph. Here we use a path cover of G to efficiently represent the vertex cuts of a graph, and because we shall be able to efficiently distribute it. A family of paths $P_1, \ldots, P_k$ in a graph G is a *path cover* of G if every vertex of G is contained in exactly one of these paths. The *path covering number* $\vartheta(G)$ of G is the minimum number of paths in a path cover of G. Computing $\vartheta(G)$ exactly is NP-hard since $\vartheta(G) = 1$ iff G contains an Hamiltonian path.

A linear layout of a graph with n vertices is a bijection $\mathcal{L} : V \to \{1, \ldots, n\}$. For a vertex v, let cutwidth$(\mathcal{L}, u)$ be the number of edges crossing the point $\mathcal{L}(u)$ in the layout. The cutwidth of a layout cutwidth$(\mathcal{L})$ is $\max_u$ cutwidth$(\mathcal{L}, u)$, i.e. the maximum number of edges crossing any point in the layout. The cutwidth of a graph G is the minimum cutwidth of a layout of G. Deciding whether cutwidth$(G) \leq k$ (for an integer k) is NP-complete [8], but is approximable to within a factor of $O(\log n \log \log n)$ in polynomial time [5]. For a linear layout $\mathcal{L}$ of G, we say that there is a *hole* at position i in the layout if there is no edge $\{\mathcal{L}^{-1}(i), \mathcal{L}^{-1}(i+1)\}$ in G, and let holes$(\mathcal{L})$ be the set of holes in the layout. We have the following observation.

Lemma 2 *Any graph G has a linear layout with $\vartheta(G) - 1$ holes.*

Proof. Given a graph G, we can construct a linear layout with $\vartheta(G) - 1$ holes: for each path $P = v_1, \ldots, v_k$ in the path cover, add the vertices of the path to the layout in the order they appear on the path. The layout clearly has the claimed number of holes.

To see that every linear layout with h holes gives a path cover of size $h + 1$, simply remove all the crossing edges (those not on the line). The spine of the layout then consists of $h + 1$ vertex-disjoint paths where each vertex of G appears in exactly one path, and the paths cover G. □

The algorithm works as follows. Construct a linear layout having both small cutwidth *and* a small number of holes (we note that we do not currently know of any linear layout algorithm that simultaneously minimizes both these quantities, and moreso, they appear to be opposing so the best we might hope for is some sort of tradeoff – we discuss this in the full paper). We maintain incremental connectivity (edge insertions only) on an auxiliary graph H where each vertex of H represents an interval with no holes on the layout, with an edge between two intervals if there is an edge in G with endpoints in both intervals. However, we shall not use the range tree.

We must also deal with the fact that intervals of the layout do not necessarily represent connected subgraphs of G, due to the holes. We will handle the holes in the following simple way: for a hole at position i, we pretend that there is a dummy vertex v connected to both i and $i + 1$, then proceed as before except that the query set is now $S \cup \{v\}$. So we add the dummy vertex v and then use our decremental connectivity algorithm to delete it at query time. The correctness of the algorithm is a consequence of the following simple lemma.

Lemma 3 *Vertices u, v are connected in G iff $h(u), h(v)$ are connected in H.*

Proof. This follows by the definition of H: it is a disjoint partition of the vertices of G into connected subgraphs with an edge between two subgraphs in H iff there is an edge between two vertices in the subgraphs in G. Therefore, every path from u to v in G corresponds to exactly one path from $h(u)$ to $h(v)$ in H, and every path from u to v in H corresponds to a set of paths from u to v in G. ◻

Instead of storing the set of crossing edges centrally, the label for vertex u stores a list of its crossing edges in the layout. To handle the holes, each vertex can simply store the *labels* for all the holes. However, the following lemma shows that we need only consider a subset of these holes. For a graph G, a set S and two vertices u, v with S being a uv-cut in G, a hole is *important* if replacing it with an edge would result in u, v being connected in $G \setminus S$. We can prove the following lemma.

Lemma 4 *For a linear layout $\mathcal{L}$, let $\mathrm{holes}(\mathcal{L}, u)$ be the set of holes spanned by the crossing edges of vertex u in the layout. Then all the important holes for S, u, v are covered by $\mathrm{holes}(\mathcal{L}, u) \cup \mathrm{holes}(\mathcal{L}, v) \cup \bigcup_{i=1}^{k} \mathrm{holes}(\mathcal{L}, s_i)$.*

The decoder algorithm is given below. We create all the vertices of H and store their intervals in a binary tree (so that we can find the interval $h(u)$ containing vertex u in time $O(\log |H|)$), then use the Union-Find data structure [12] to maintain incremental connectivity as we add the edges. This allows us to implement Find in worst-case time $O(\log |H|)$ and Union in worst-case time $O(1)$ (these worst-case bounds are known to be optimal). The inner loop at line 7 is executed at most $O(|H|\mathrm{cutwidth}(\mathcal{L})) = O((k + |\mathrm{holes}(\mathcal{L})|)\mathrm{cutwidth}(\mathcal{L}))$ times, so the decoder takes worst-case time $O((k + |\mathrm{holes}(\mathcal{L})|)\mathrm{cutwidth}(\mathcal{L}) \log n)$.

k-Vertex-Witness-Decoder$(L(u), L(v), L(s_1), ..., L(s_k))$

```
1   T ← holes(L)
2   sort the elements of S ∪ T by their position in the layout
3   H ← set of intervals of G \ (S ∪ T)
4   Construct the binary search tree on intervals of H
5   for each vertex k ∈ H
6       do Insert k into the Union-Find structure          ▷ O(1)
7   for each vertex k ∈ (S ∪ T)
8       do for each crossing edge (i, j) of k
9           do x ← Find(h(i))                               ▷ O(log |S ∪ T|)
10             y ← Find(h(j))
11             Union(x, y)                                  ▷ O(1)
12  return Find(h(u)) ≠ Find(h(v))                          ▷ O(log |S ∪ T|)
```

The label assigned to a vertex u is has worst-case size $O(|\mathrm{holes}(\mathcal{L}, u)|\mathrm{cutwidth}(\mathcal{L}) \log n)$. We note that a consequence of using the Union-Find data structure is that after the decoder algorithm has finished, we can repeatedly answer connectivity queries in worst-case time $O(\log n)$ on $G \setminus S$.

It is worth noting that any labeling scheme with labels of size $O(s)$ bits gives a nondistributed data structure of size $O(ns)$ bits with the same time complexities. Using the above labeling scheme, and storing the crossing edges of each hole exactly once gives a nondistributed data structure using $O(\sum_{u \in G} \mathrm{cutwidth}(\mathcal{L}, u) \log n) = O(\mathrm{wirelength}(\mathcal{L}) \log n)$ bits total space, handling queries in the same time as above.

We can prove the following simple lower bound on the label size.

Theorem 3 *Any labeling scheme for the k-vertex witness problem on n-vertex graphs must assign to some vertex a label of $\Omega(n)$ bits in the worst-case.*

Proof. By reduction from adjacency on bipartite graphs. Given a bipartite graph $G = ((U, V), E)$, construct G' by adding a vertex t, connected to each vertex of V, as in Figure 3. Now consider

Figure 2: Illustrating Theorem 3
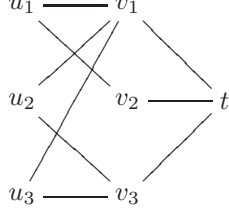
two vertices $u \in U$ and $v \in V$. It is clear that u is not adjacent to v in G iff $V \setminus \{v\}$ is a cut between u and t in G'. Since any labeling scheme for deciding adjacency on bipartite graphs must assign some vertex a label of size $\Omega(n)$ bits, the theorem follows. $\square$

6 Decremental connectivity on small cutwidth graphs

Assume that we can construct a linear layout of G with bounded cutwidth and a bounded number of holes. In this section we show that decremental connectivity can be done in time $O(\sqrt{k})$ for the kth deletion, with connectivity queries answered in time $O(\log k)$. This lets us answer k-vertex witness and k-edge witness in worst-case time $O(k)$. The algorithm is essentially the same as in the previous section, but we make use of the following observation.

Lemma 5 *The degree of any vertex of H is at most twice the cutwidth of the layout.*

Proof. Consider some vertex $h \in H$ representing an interval with vertices u, v being the endpoints of this interval. Every edge of h corresponds to a crossing edge of either u or v in the linear layout of G, i.e. leaving the interval to the left or to the right. $\square$

The algorithm is given below. On the linear layout, we call an interval $[i \mathinner{\ldotp\ldotp} j]$ a *maximal connected interval* if there are no holes in $[i \mathinner{\ldotp\ldotp} j]$ and every interval containing $[i \mathinner{\ldotp\ldotp} j]$ also contains a hole. It is easy to see that the set of maximal connected intervals gives a unique partition of the layout into disjoint intervals.

Initially, we add all the maximal intervals to the auxiliary graph H and the crossing edges of the layout to the static range tree (for example, using the algorithm of Chazelle [2]). We then use the fully dynamic algorithm of Eppstein et al. [4] to maintain connectivity on H, at a cost of $O(\sqrt{|H|})$ per edge insertion or deletion.

INITIALISE

1 $\mathcal{L} \leftarrow$ a linear layout of G
2 **for** each edge $\{u, v\} \in G$
3   **do** RANGE-TREE-INSERT$(L(u), L(v))$
4 **for** each maximal connected interval $[i \mathinner{\ldotp\ldotp} j]$ on $\mathcal{L}$
5   **do** Add the vertex $[i \mathinner{\ldotp\ldotp} j]$ to H
6 **for** each pair of vertices $[a \mathinner{\ldotp\ldotp} b], [c \mathinner{\ldotp\ldotp} d]$ in H
7   **do if** BOX-NOT-EMPTY$((a, c), (b, d))$
8   **then** Add edge $([a \mathinner{\ldotp\ldotp} b], [c \mathinner{\ldotp\ldotp} d])$ to H

DELETE(u)

1	$[a \mathinner{.\,.} b] \leftarrow h(u)$	$\triangleright O(\log|H|)$
2	**if** $(a = b)$	
3	**then return**	
4	$\triangleright$ Otherwise, split $[a \mathinner{.\,.} b]$ at $\mathcal{L}(u)$	
5	INSERT-VERTEX$_H([a \mathinner{.\,.} \mathcal{L}(u) - 1])$	
6	INSERT-VERTEX$_H([\mathcal{L}(u) + 1 \mathinner{.\,.} b])$	$\triangleright O(\sqrt{|H|})$
7	**for** each neighbour $[c \mathinner{.\,.} d]$ of $h(u) \in H$	
8	**do if** BOX-NOT-EMPTY$((a, c), (\mathcal{L}(u) - 1, d))$	$\triangleright O(\log n)$
9	**then** INSERT-EDGE$_H(([a \mathinner{.\,.} I(u) - 1], [c \mathinner{.\,.} d])$	$\triangleright O(\sqrt{|H|})$
10	**if** BOX-NOT-EMPTY$((\mathcal{L} + 1, a), (b, d))$	
11	**then** INSERT-EDGE$_H([\mathcal{L} + 1 \mathinner{.\,.} b], [c \mathinner{.\,.} d])$	
12	DELETE-VERTEX$_H([a \mathinner{.\,.} b])$	$\triangleright O(\sqrt{|H|}\Delta(H))$

CONNECTED(u, v)

1	**return** CONNECTED-IN-H$(h(u), h(v))$	$\triangleright O(\log|H|) + O(1)$

6.1 Analysis

Let the layout have j holes. Since $|H| \le k + j$ and $\Delta(H) \le 2\text{cutwidth}(\mathcal{L})$, the kth deletion takes worst-case time $O(\sqrt{k + j} \min(k + j, \text{cutwidth}(\mathcal{L})))$ and connectivity queries are answered in $O(\log k)$. Alternatively, we could just store H using adjacency lists, then the kth update takes worst-case time $O(\min(k + j, \text{cutwidth}(\mathcal{L})))$ but queries take time $O((k + j)^2) = O(k^2 + j^2)$ by performing a depth-first search of the auxiliary graph H.

A modification of the above algorithm is to only add the holes as they are required, rather than adding them all in the initialisation step. If the queries do not involve vertices whose crossing edges cover many holes, then this may be more efficient than the previous algorithm.

Remark. We could use the linear layout to handle edge deletions by deleting the edges from the range tree, but we would still incur the same worst-case cost as a single vertex deletion.

7 Further work

There are many interesting open problems in dynamic connectivity. Probably the most important is still to break the $O(\sqrt{n})$ worst-case bound for a single deletion, over an *arbitrarily long* sequence of deletions. Very recently, Thorup [15] gave a fully-dynamic all-pairs shortest paths algorithm with good worst-case time by constructing a data structure that supports any sequence (called an *epoch*) of Δ deletions in $\tilde{O}(n^{2.5}\sqrt{\Delta})$ total time, then partially rebuilding the structure in the background to avoid incurring a single expensive update. We have tried to apply a similar technique to our algorithms but with no luck – although we can support an epoch of $O(\sqrt{n})$ deletions in total time $\tilde{O}(n)$ with worst-case $\tilde{O}(\sqrt{n})$ per deletion, which allows us to spend at most an extra $\tilde{O}(\sqrt{n})$ time per deletion on rebuilding the data structure, this only amounts to $\tilde{O}(n)$ time over the epoch. To break the $o(\sqrt{n})$ bound for deletions, we would have to have $o(\sqrt{n})$-length epochs, but then we would only be able to spend $o(n)$ rebuilding time during each epoch.

It would also be interesting to investigate other ways of constructing distributed data structures for important graph connectivity problems such as k-vertex witness, as these would appear to be very useful in large distributed networks such as the Internet.

References

[1] BENDER, M. A., AND FARACH-COLTON, M. The lca problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics* (London, UK, 2000), Springer-Verlag, pp. 88–94.

[2] CHAZELLE, B. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput. 17*, 3 (1988), 427–462.

[3] DIRAC, G. A. Some theorems on abstract graphs. *Proceedings of the London Mathematical Society 2* (1952), 69–81.

[4] EPPSTEIN, D., GALIL, Z., ITALIANO, G. F., AND NISSENZWEIG, A. Sparsification — A technique for speeding up dynamic graph algorithms. *J. ACM 44*, 5 (September 1997), 669–696.

[5] EVEN, G., NAOR, J. S., RAO, S., AND SCHIEBER, B. Divide-and-conquer approximation algorithms via spreading metrics. *J. ACM 47*, 4 (2000), 585–616.

[6] FREDERICKSON, G. N. Data structures for on-line updating of minimum spanning trees. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1983), ACM Press, pp. 252–257.

[7] FURER, M., AND RAGHAVACHARI, B. Approximating the minimum-degree steiner tree to within one of optimal. *Journal of Algorithms 17* (1994), 409–423.

[8] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[9] HENZINGER, M. R., AND KING, V. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM 46*, 4 (1999), 502–516.

[10] HENZINGER, M. R., AND THORUP, M. Sampling to provide or to bound: with applications to fully dynamic graph algorithms. In *Proceedings of the workshop on Randomized algorithms and computation* (New York, NY, USA, 1997), John Wiley & Sons, Inc., pp. 369–379.

[11] 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.

[12] KAPLAN, H., SHAFRIR, N., AND TARJAN, R. E. Union-find with deletions. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2002), Society for Industrial and Applied Mathematics, pp. 19–28.

[13] MORTENSEN, C. W. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2003), Society for Industrial and Applied Mathematics, pp. 618–627.

[14] NAGAMOCHI, H., AND IBARAKI, T. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica 7*, 5&6 (1992), 583–596.

[15] THORUP, M. Worst-case update times for fully-dynamic all-pairs shortest paths. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing* (New York, NY, USA, 2005), ACM Press, pp. 112–119.

[16] WIN, S. On a connection between the existence of the k-trees and the toughness of a graph. *Graphs and Combinatorics 7* (1989), 201–205.

Figure 3: Illustrating Theorem 3
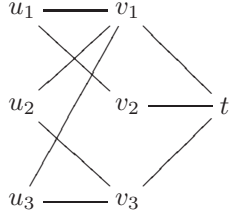
A Proof of Lemma 2

Proof. Given a graph G, we can construct a linear layout with $\vartheta(G)-1$ holes: for each path $P = v_1, \ldots, v_k$ in the path cover, add the vertices of the path to the layout in the order they appear on the path. The layout clearly has the claimed number of holes.

To see that every linear layout with h holes gives a path cover of size $h + 1$, simply remove all the crossing edges (those not on the line). The spine of the layout then consists of $h + 1$ vertex-disjoint paths where each vertex of G appears in exactly one path, and the paths cover G. $\square$

B Proof of Lemma 3

Proof. This follows by the definition of H: it is a disjoint partition of the vertices of G into connected subgraphs with an edge between two subgraphs in H iff there is an edge between two vertices in the subgraphs in G. Therefore, every path from u to v in G corresponds to exactly one path from $h(u)$ to $h(v)$ in H, and every path from u to v in H corresponds to a set of paths from u to v in G. $\square$

C Proof of Theorem 3

Proof. By reduction from adjacency on bipartite graphs. Given a bipartite graph $G = ((U, V), E)$, construct G' by adding a vertex t, connected to each vertex of V, as in Figure 3. Now consider two vertices $u \in U$ and $v \in V$. It is clear that u is not adjacent to v in G iff $V \setminus \{v\}$ is a cut between u and t in G'. Since any labeling scheme for deciding adjacency on G must assign some vertex a label of size $\Omega(n)$ bits, the theorem follows. $\square$