Planning for Fast Connectivity Updates

Mihai Pătraşcu
mip@mit.edu

Mikkel Thorup
mthorup@research.att.com

Abstract

Understanding how a single edge deletion can affect the connectivity of a graph amounts to finding the graph bridges. But when faced with $d > 1$ deletions, can we establish as easily how the connectivity changes? When planning for an emergency, we want to understand the structure of our network ahead of time, and respond swiftly when an emergency actually happens.

We describe a linear-space representation of graphs which enables us to determine how a batch of edge updates can impact the graph. Given a set of d edge updates, in time $O(d \operatorname{polylg} n)$ we can obtain the number of connected components, the size of each component, and a fast oracle for answering connectivity queries in the updated graph. The initial representation is polynomial-time constructible.

1. Introduction

We ask the following natural combinatorial question: can we obtain some understanding of the structure of a graph which allows us to determine how its connectivity is impacted by a few changes to the edges?

It turns out that edge insertions are easy to handle, so we concentrate on edge deletions. For the case of one deletion, this is the classic problem of identifying the bridges of a graph; see [3, Problem 22-1]. For two deletions, we can observe that they only matter if one or two bridges are deleted, or if a min-cut in a 2-edge-connected component gets deleted. To test for the later situation, we can use the cactus graph. The difficulties start when we delete $d \geq 3$ edges from a graph that is not d connected, hence when we cannot just check if a min-cut is lost. The best known solution is based on a 24-year old algorithm of Frederickson [6, 5], and handles d updates in $O(d\sqrt{n})$ time. As we discuss below, known amortized algorithms for dynamic connectivity may easily spend linear time even for $d = 1$.

In the style of union-find, let find(u) return an identifier for the connected component containing u, so that u and w are connected iff find$(u) =$ find(w). We describe a representation of an undirected graph with the following properties:

(1) the representation requires linear space.

(2) starting from the representation, a batch of d edge updates is handled in $O(d \lg^2 n \lg \lg n)$ time.

(3) subsequently, find(u) is evaluated in $O(\lg \lg n)$ time per query.

Our algorithms are deterministic. We note that the representation is simultaneously good for any d. After processing the updates in (2), the algorithm can output interesting connectivity statistics, such as the number of connected components and the size of each component. The doubly-logarithmic running time of find(u) stems from a simple binary search over levels in a tree of height $O(\lg n)$. Interestingly, we can argue a matching lower bound in the cell-probe model, even for a batch of a deletions in a tree. It follows that this "understanding of connectivity" (in super-constant time) is actually the best possible.

In its basic form, our representation is NP-hard to compute as it is strongly related to weighted sparsest cut and graph partitioning. However, if we use a γ-approximation to weighted sparsest cut, the recovery time in (2) becomes $O(\gamma \cdot d \lg^2 n \lg \lg n)$. We can, for example, use the classic approximation algorithm of Leighton and Rao [12] based on linear programming, to obtain $\gamma = O(\lg n)$. The more recent semi-definite programming algorithm of Arora, Rao and Vazirani [1] yields $\gamma = O(\sqrt{\lg n})$.

Our problem is a fundamental example in the class of emergency planning problems, and among the first examples where we can handle more than one update. Our result also highlights a programme of research for worst-case dynamic connectivity, which is perhaps the main challenge in dynamic graph algorithms today.

1.1. Emergency Planning

Our problem is motivated by emergency planning for a parallel attack on some edges in a large network. After an attack, we want to understand the connectivity of the new

graph, without a prolonged period of confusion. The basic nature of emergency planning is that we are willing to invest a significant amount of resources into preparation, in order to respond swiftly when an emergency actually happens. Formally, we translate this into polynomial time for off-line preparation, in order to respond in polylogarithmic time per actual change.

In physical networks, one can think of parallel attacks such as the impact of an earthquake on a road network. In computer networks, such as an IP backbone, the scenario is notoriously frequent. Here, a single cable cut can take down many logical links. Moreover, there are many different protocol errors that take down several routers simultaneously. Finally, we have to deal with links being flooded by malicious attacks from worms and viruses.

We note that the paradigm of emergency planning is also very relevant to nonemergency situations, where there are a lot of updates, but only a limited number of *net* changes since the last off-line preprocessing (most changes are undone). In this case, update times are linear in the number of net changes. For example, if we preprocess a copy of the US road network once a year, the picture is never too different from the last preprocessing and we do not care if the same road is opened and closed many times. Similarly in an IP backbone, the vast majority of the link updates are links going down and back up, with relatively few net changes to the network.

Previous work. There has been a lot of previous work on preparing for a single update to the network. For connectivity queries, this is precisely why the notions of bridges and articulation points are studied. Other queries include reachability in directed graphs [11], the shortest path with fixed source and destination [9, 15], and all-pairs shortest paths [4, 2]. Like our algorithm, most of these examples make essential use of the asymmetry of emergency planning: to optimize the query, the preprocessing is asymptotically slower than computing a single answer from scratch.

In general, for the case of a larger batch of updates, there has been little progress in obtaining very good update time based on preprocessing. Besides our algorithm, we are aware of one other example in this direction: the algorithm of [19] for all-pairs shortest paths, which is the key component in a worst-case fully-dynamic algorithm.

An obvious alternative to designing new algorithms is to use existing work under other paradigms, such as fully-dynamic or decremental problems. However, due to its unique constraints, emergency planning is not easily replaced by these paradigms. For example, we have been vastly more successful at designing amortized algorithms than worst-case algorithms for dynamic graph problems. But emergency planning has little use for amortization, and the perspective of spending a lot of time for a few updates is unacceptable in real-time systems. The cases of shortest paths [19] or connectivity (Section 1.2) are telling: the amortized algorithms degenerate into a full reconstruction in the worst possible moments, when one key edge gets deleted, significantly disrupting the connectivity of the network.

On the other hand, even if we had good worst-case algorithms, it is somewhat unlikely that they will ever be as efficient as algorithms that benefit from careful preprocessing. Even in our own example, we remark that the $O(\lg \lg n)$ upper bound for queries is exponentially better than the query *lower bound* for the fully-dynamic case [13].

Finally, we find that the challenge of preprocessing provides significant appeal for studying such problems. The planning aspect forces us to understand the structure of graphs in new, algorithmically interesting ways, which inevitably leads to appealing combinatorial problems.

1.2. Relation to Fully-Dynamic Connectivity

Worst-case algorithms. A solution to dynamic connectivity with worst-case running times immediately solves our problem. In STOC'83, Frederickson presented such an algorithm with update time $O(\sqrt{m})$. A general sparsification technique of Eppstein et al. [5] from FOCS'92 transformed this bound into $O(\sqrt{n})$. Since then, however, this problem has seen no improvement, and worst-case dynamic connectivity stands as perhaps the most fundamental challenge in dynamic graph algorithms today.

Our results provide some hope for this problem, showing how to be worst-case efficient after some preprocessing: if the number of net changes since the preprocessing is $O(n^{1/2-\varepsilon})$, we obtain a better worst-case running time. In fact, our approach highlights some explicit challenges, and possibly a path for future research towards dynamic connectivity with polylogarithmic time per update. See Section 1.3.

Amortized algorithms. If we are satisfied with amortized bounds, significantly better bounds are known. Henzinger and King [8] were the first to obtain a randomized algorithm with polylogarithmic update time. Holm, de Lichtenberg and Thorup [10] provide the best deterministic solution, requiring $O(\lg^2 n)$ amortized time per update. Thorup [18] gives the currently best randomized solution, supporting updates in $O(\lg n(\lg \lg n)^3)$ time.

Unfortunately, the polylogarithmic amortization in these results only works if we start from an empty graph. When starting from an arbitrary graph, these algorithms can happily spend linear time even on the first edge deletion.

In particular, a central amortization technique originating in [8] and present in all subsequent algorithms is based on the following observation: If a component is split in two,

we can afford to investigate all vertices in the smaller half, since this guarantees that each vertex is investigated only a logarithmic number of times overall. Amortized over an arbitrary long sequence of deletions, this leads to a total number of $O(n \lg n)$ investigations. However, if we just have one edge deletion splitting a component in two, this deletion alone may take linear time. Such accounting is useless for getting bounds limited in terms of d, the number of edges actually deleted.

To make matters worse, note that this amortization scheme tends to be slow precisely when the state of affairs is already bad enough. If a central hub in the network goes down, affecting the graph structure significantly, the algorithm degenerates into a linear scan.

Amortization starting with an arbitrary graph. An interesting question is whether we can in fact obtain an amortized algorithm starting from an arbitrary graph, not just an empty graph. We would like to handle an arbitrary number of updates, spending polylogarithmic time per update on the average. Our algorithm does not quite achieve that, but gives some noteworthy bounds. After d updates, we can recover connectivity information in $O(d \cdot \text{polylg}(n))$ time, but this does not mean that a single new change will take less than $O(d \cdot \text{polylg}(n))$ time. Nevertheless, if the total number of changes is $O(m^{1/2-\varepsilon})$, our total update time is sublinear, beating previous approaches. Obviously, our improvement is bigger if the number of net changes is small, or the changes come in batches, like parallel attacks.

1.3. Technical Contributions

The starting point for our algorithm is the amortization of Henzinger and King [8]. The main idea of that algorithm is to classify edges into $\lg m$ levels, with at most $m/2^i$ edges on level i. The algorithm maintains a spanning tree of the graph, which is minimal with respect to level numbers. If an edge of the spanning tree is deleted, we search for a replacement at the edge's level. That is done by sampling edges, and testing if they cross the cut that was just induced on the tree. If we fail to find a replacement edge after a number of trials, we can conclude with high probability that the cut is sparse. Then, we can promote all edges of the cut to the next level, without violating the constraint on level sizes. Paying for the promotion is done by the amortization technique of promoting things in the smaller half. After this promotion, the search for a replacement edge continues on the next level.

We view the levels of this analysis as a hierarchical decomposition of a graph. As discussed previously, the obvious problem with the algorithm of [8] is the amortization technique: promoting all edges of a sparse cut can be triggered by one delete, and it can be very expensive if the cut is close to linear size. To avoid this possibility one would intuitively want to "preload" the hierarchy such that no promotion is necessary. This is roughly equivalent to guaranteeing that every level of the hierarchy is an expander, i.e. contains no sparse cut to promote.

It is probably not a surprise that such a preloading scheme fails. Roughly speaking, deletion of some d edges on level 1 can "split", say, $2d$ edges on level 2: these edges are no longer in the same connected component. Once these edges are disconnected, they can induce a split of $4d$ edges on level 3, and so on. The point is that locally in the hierarchy, the effects of a delete may outnumber slightly the deleted edges, and things can spiral out of control. This example again emphasizes the contrast to amortization, where we would be happy that a lot of edges get promoted to high levels at once, since the total number of promotions over time is bounded anyway.

Our suggestion is to preload everything, so that on every level i, even *deep* cuts (cuts including edges from levels $\leq i$) are not sparse. This enforces a global constraint, and thus guarantees that the magnitude of side effects does not cascade. To put things in a different light, we view the levels as a reweighting of the graph, where weight is concentrated on increasingly fewer, but more critical edges. We then exclude sparse cuts with respect to this weighting. This deep-cut approach induces many important changes, including unfortunate ones like reliance on weighted sparsest cut.

Finally, we need to warn the reader that our actual implementation of this combinatorial idea is totally divorced from [8]. Our algorithm is deterministic, and faster by a logarithmic factor. Our representation uses linear-space, in the spirit of [18].

Somewhat surprisingly, our algorithm manages to answer connectivity queries without maintaining a certificate for connectivity, i.e. without even implicitly maintaining a spanning forest. If an explicit spanning forest is requested, we can still maintain one using a randomized algorithm with a slightly larger running time of $O(\gamma d \lg^3 n)$. To our knowledge, ours is the first dynamic connectivity algorithm which can benefit from nonexplicit understanding of connectivity.

Challenges for Future Research. Depending on the reader's mood, our solution brings renewed hope for dynamic connectivity with worst-case polylogarithmic bounds, or highlights some major roadblocks which keep us away from such a result. Here are some of these challenges:

1. Obtain an amortized algorithm starting from any graph. Without insisting that our hierarchy be made fully dynamic, we can ask at least that it provide a smooth transition into maintaining a hierarchy with an amortization technique. One reason this is difficult is

that our recovery algorithm needs to examine deleted edges in a strict order through the hierarchy.

2. Reduce the preprocessing time to $O(n \operatorname{polylg} n)$. This is clearly required before any dynamization is attempted. One way to do this would be to first give a $\operatorname{polylg} n$ approximation for weighted sparsest cut in almost linear time. This was recently achieved by Spielman and Teng [16] for the unweighted version. However, extending combinatorial approaches similar to theirs to the weighted case appears to be a difficult problem.

3. Show that (something similar to) the hierarchical decomposition of this paper can be maintained dynamically with few edges moving. This is unclear even if in each step we can take linear time to examine the whole graph.

2. Edge-Expanding Graphs

As a warm-up before the general algorithm, we show how to solve our problem in the case of an edge expander, defined as follows:

Definition 1 *For $\Phi \in [0,1]$, a graph $G = (V, E)$ is an Φ-edge-expander if for any vertex subset $U \subset V, |U| \leq \frac{1}{2}|V|$, the number of edges leaving U is more than a fraction Φ of the number of edges incident to U, where an edge internal to U is counted twice. Formally $|E \cap (U \times \overline{U})| > \Phi \sum_{v \in U} \deg(v)$. The edge expansion of G is the maximal such Φ.*

We note that random regular graphs of degree > 2 are $\Omega(1)$-edge-expanders with high probability. Finding the exact edge expansion is NP-hard, but we can use approximation algorithms to get a lower-bound Φ on the edge expansion. For the recovery below, we assume that such a Φ is known for G.

2.1. Handling Insertions

First we once and for all show that insertions are trivial. Suppose we are given a set D of deletions and a set I of insertions, and suppose we first recover from the D deletions with a find-routine find^D. Next, we take each inserted edge (u, w) and view it as an edge $(\text{find}^D(u), \text{find}^D(w))$ over $\text{find}^D(V)$. We identify the connected components of this graph in time proportional to $|I|$, and name each non-trivial component. Now $\text{find}(u) = \text{find}^D(u)$ if $\text{find}^D(u)$ did not get connected by I; and otherwise, $\text{find}(u)$ is the name of the component of $\text{find}^D(u)$. Thus we recover from the inserts using $|I|$ calls to find^D, and then this slows down the final find by only an additive constant.

2.2. Recovery from Edge Deletions

We now get a set D of edges to be deleted. We are going to grow components in $G \setminus D = (V, E \setminus D)$, starting with vertices that have lost incident edges. At some stage we will stop growing. At that point we will have isolated some of the components as (maximal) connected components of $G \setminus D$, and we will conclude that everything else belongs to the same "giant component" of $G \setminus D$.

To grow components, we will *explore edges* from $G \setminus D$. The components spanned by the explored edges are referred to as *explored components*. An *unexplored edge* is understood not to be deleted.

If U is a set of vertices, we define $\deg(U)$ as the number of all incident edges in G, including deleted ones, and $\text{del}(U)$ as the number of incident deleted edges from D. The recovery algorithm classifies an explored component C as one of the following:

isolated if it has no unexplored incident edges.

active if the following three conditions are satisfied: **(1)** C has unexplored incident edges; **(2)** C contains at most half the vertices of G; **(3)** $\text{del}(C) > \Phi \deg(C)$.

passive otherwise.

As long as there is some active component C, the recovery algorithm proceeds by considering any unexplored edge (u, w) incident to C. If one of the end-points, say w, is not in C, it unites C with the component of w. In either case (u, w) is now considered explored, and it is contained in an explored component.

Since an active component requires an unexplored edge, and since each of the above iterations explores an edge, the recovery algorithm must terminate with no active components left. Trivially each isolated component forms a maximal connected component of $G \setminus D$. When no active components remain, we declare that each vertex that is not in an isolated component belongs to the same giant component of $G \setminus D$.

2.3. Analysis

The following lemma established that the our recovery algorithm is correct.

Lemma 1 *If no active components are left, the vertices in the passive components all belong to the same component of $G \setminus D$.*

Proof: Let H be the subgraph of $G \setminus D$ induced by the vertices in the passive components. If H is not connected in $G \setminus D$, there is a component B of $G \setminus D$ containing at most

half the vertices. The vertices in the same passive component are connected, so B must be the disjoint union of some passive components. For each of these passive components C, we know that $\text{del}(C) \leq \Phi \deg(C)$, but then $\text{del}(B) \leq \Phi \deg(B)$. However, G was a Φ-edge-expander, and $|B| \leq n/2$, so G had more than $\Phi \deg(B)$ edges leaving B. With $\text{del}(B) \leq \Phi \deg(B)$, we conclude that $G \setminus D$ has some edge leaving B, contradicting that B is a maximal connected component of $G \setminus D$. ∎

The next lemma will be used to bound the running time needed by the recovery.

Lemma 2 *The number of edges explored is at most $2/\Phi$ times the number of edges deleted.*

Proof: For the amortization, we consider *final active components*, meaning active components that never again becomes part of an active component. The final active components are thus disjoint. If an edge (u, w) is explored, it is because an endpoint u is in an active component, and then there must be a last active component C containing u. We allow here that the edge (u, w) was explored before C was created by the union of other components. Each explored edge is thus counted in the degree of at least one last active component C, and since C is active, $\deg(C) < \text{del}(C)/\Phi$. The factor 2 comes from that fact that each deleted edge makes a contribution in two end-points. ∎

Theorem 3 *Given a Φ-edge-expander G, after linear time preprocessing, we can recover from the deletion of d edges in $O(\frac{d}{\Phi} \cdot \alpha(\frac{d}{\Phi}, d))$ time, and subsequently answer* find$(\cdot)$ *queries in constant time.*

Proof: The component number of each vertex, returned by find$(\cdot)$, is stored in an array, which is initialized to 0 at preprocessing (all vertices are in the giant component). Since each isolated component is spanned by explored edges, the recovery algorithm only need to change $O(d/\Phi)$ entries in this table. In the preprocessing phase, we compute an array with the degrees $\deg(u)$. At recovery time, we can compute all nonzero $\text{del}(u)$ in $O(d)$ time. During the recovery exploration, we maintain a queue of active components, and a concatenation of the incidence lists for each component. To know whether an edge goes outside an active component, we maintain a union-find data structure [3, 17] over vertices in the explored components.

The running time is dominated by the calls to the union-find data structure. By Lemma 2 we have at most $2d/\Phi$ such calls, hence a recovery time of $O(d/\Phi \cdot \alpha(d/\Phi))$. We can improve this to $O(d/\Phi \cdot \alpha(d/\Phi, d))$, where $\alpha(d/\Phi, d)$ is $O(1)$ if e.g. $\Phi = \Omega(\lg^* n)$. At preprocessing, we choose an arbitrary spanning tree T, and explore non-deleted edges of this tree before exploring any other edges. In the tree, the union-find data structure spends only constant time per operations [7]. Afterward, we can have d/Φ finds, but only d unions in the regular union-find structure. ∎

3. General Graphs

3.1. Edge-Expanding Hierarchies

Our strategy is to decompose a general graph into a "hierarchy of expanders", and apply the above recovery algorithm. During preprocessing of the input graph $G = (V, E)$, we assign each edge $e \in E$ to a level $\ell(e) > 0$. The graph $G|_i$ is induced by the edges of level i. We define $G|_{\leq i}, G|_{<i}$ etc. in the intuitive fashion, and extend the notation freely to other objects (e.g. $\deg_{>i}$ counts degrees in $G|_{>i}$). Note that $G|_{\leq 0}$ consists of singleton vertices.

The *hierarchy* is a rooted tree whose height h is the maximal level of an edge. On level $i = 0, \ldots, h$ of the hierarchy (tree), we have the components of $G|_{\leq i}$, and the parent of such a component C is the component of $G|_{\leq i+1}$ subsuming it.

The goal of preprocessing is to assign levels which induce a Φ-edge-expander hierarchy:

Definition 2 *A* hierarchical level-i cut *is defined by a subset U with at most half of the vertices of a component C of $G|_{\leq i}$. The* density *of the cut is the ratio of the number of cut edges leaving U, over the number of all level i edges incident to U, i.e. $|E_{\leq i} \cup (U \times \overline{U})| \,/\, \sum_{v \in U} \deg_i(v)$. We have a* Φ-edge-expander hierarchy *if, for each level i, there is no hierarchical level-i cut with density $\leq \Phi$.*

Note that for the above density, we compare *all* cut edges from $G|_{\leq i}$, i.e. edges from all levels up to i, against the incident edges on level i alone. This accounting is crucial to our efficient recovery algorithm. In [8], the level-i cuts were well-behaved in that they could not cut components of $G|_{<i}$. They only counted cut edges on level i against incident edges on level i, and that sufficed for their amortized and randomized algorithm. It our case, reasoning about the cuts is more tricky, but we can still adapt the accounting scheme from [8] to show the existence of a $1/(2 \lg n)$-edge-expander hierarchy with at most $\lg m + 1$ levels.

Below, we will sometimes refer to the Φ-edge-expanders considered in the last section as *flat* Φ-edge-expanders, so as to more clearly distinguish them from the expander hierarchies defined over arbitrary graphs. If a graph G is itself a flat Φ-edge-expander, we get a Φ-edge-expander hierarchy of height 1, if we simply assign all edges to level 1.

3.2. Constructing the Hierarchy

Given an arbitrary graph G, we start with all edges on level 1. We say we promote a hierarchical level-i cut if for all cut edges, we increase their level to $i+1$. The following lemma states that the number of levels remain logarithmic as long as we only promote cuts of density at most $1/(2\lg n)$.

Lemma 4 *If all edges start at level* 1*, and we only promote hierarchical cuts of density at most* $1/(2\lg n)$*, then the height is at most* $1+\lg m$*.*

Proof: Let m_i be the number of edges that end up on level i *or higher*. Since the level of an edge can only go up, this is a bound for the number of edges at level i or higher at any moment of time before the end. We have $m_1 = m$ by the starting conditions. Inductively, we will prove that $m_{i+1} \leq m_i/2$. In particular, this will imply that we can get at most $\lg m + 1$ nonempty levels.

Consider a hierarchical cut on level i. When the cut edges are moved up to level $i+1$, we want to pay \$1 for each edge by charging the current level-i edges within the smaller side of the cut. By the sparsity guarantee, each level-i edge on the smaller side has to pay at most $1/(2\lg n)$. Now, how many times can one level-i edge be charged? Each time this happens, the smaller side of the cut becomes a new connected component in $G|_{\leq i}$ because the cuts are hierarchical. This component is half the size of the old component of $G|_{\leq i}$ which contained the charged edge. Furthermore, such a component size cannot increase, because edges only move up. Then, a fixed edge can only be charged $\lg n$ times while it is at level i. This means that edges at level i pay at most $1/2$ for edge promotions over time, so $m_{i+1} \leq m_i/2$. ∎

Lemma 4 implies that promoting cuts of density below $1/(2\lg n)$ is a bounded process, leading to at most $O(m\lg m)$ increases to edge levels. Hence, we are free to promote any hierarchical level-i cut of density $1/(2\lg n)$ that we identify, eventually leading to a $1/(2\lg n)$-expander hierarchy. This shows that such a hierarchy exists.

Finding a sparse hierarchical level-i cut of a component from $G|_{\leq i}$ can be done using a weighted sparsest cut algorithm. We assign to each vertex a weight corresponding to number of incident level-i edges. Using γ-approximation algorithm for this weighted sparsest cut problem, we are guaranteed to find a cut of sparsity at most $1/(2\lg n)$ if there is one with sparsity at most $1/(\gamma\cdot 2\lg n)$. Repeating this as long as possible, we exclude all hierarchical level-i cuts of density $1/(\gamma 2\lg n)$. Then, we have a $1/(\gamma\cdot 2\lg n)$-expander hierarchy, which by Lemma 4 has at most logarithmic height. We have shown:

Lemma 5 *Using an* γ *approximation algorithm for vertex-weighted sparsest cut, we can construct a* $1/(\gamma\cdot 2\lg n)$ *edge-expanding hierarchy of height most* $\lg m+1$*. If the approximation algorithm runs in polynomial time, then so does the preprocessing algorithm.*

The sparsest cut algorithm of [1] allows vertex weights and has an approximation factor of $\gamma = O(\sqrt{\lg n})$. This leads to a $1/O(\lg^{3/2} n)$-expander hierarchy with $\lg m+1$ levels in polynomial time.

3.3. Recovery from Edge Deletions

Starting with a Φ-edge-expanding hierarchy of height h, we will show how to recover from the deletion of a set D of d edges, relating the number of operations to Φ, h, and d. We now describe the algorithm at an abstract graph theoretic level, leaving implementation details to later. The recovery is going to be done bottom–up in the hierarchy, on levels $i = 1,\ldots,h$. When we start on level i, we assume that we know the new connected components of $(G\setminus D)|_{<i}$. The components on level i are still the old components of $G|_{\leq i}$. The parent of a new level-$(i-1)$ component is the old level-i component subsuming it. If no edges of level $\leq i$ were deleted from an old component H of $G|_{\leq i}$, then H remains a level-i component. Thus, it suffices to consider components H which lose some edges.

To find the connected (sub)components of $(H\setminus D)|_{\leq i}$, we essentially use the algorithm for flat edge expanders from Section 2.2, treating the components of $(H\setminus D)|_{<i}$ as vertices. These vertices form the initial components to be grown by exploration of live level-i edges. To grow components, we will explore level-i edges from $H\setminus D$. As before, explored components represent components spanned by the explored edges. The recovery algorithm classifies an explored component C as one of:

isolated if it has no unexplored incident edges at level $\leq i$.

active if the following three conditions are satisfied: **(1)** C has unexplored incident level i edges; **(2)** C contains at most half the vertices from H; **(3)** $\mathrm{del}_{\leq i}(C) > \Phi\cdot\deg_i(C)$.

passive otherwise.

The recovery algorithm runs as long as there is some active component C. It considers any unexplored level-i edge incident to C, and unites C with another component if the edge leaves C. Since each iteration explores an edge, the recovery algorithm must terminate with no active components left. Each isolated component forms a component of $H\setminus D$. When no active components remain, we declare that each vertex that is not in an isolated component belongs to the same giant component of $H\setminus D$.

Returning to the hierarchy, we create a new node in the tree at level i, for each isolated component C of $H \setminus D$. The parent of each node is the old parent of H. The old level-i node that represented H will now represent the giant component of $H \setminus D$. For each isolated component C, we have explored edges connecting the level-$(i-1)$ components it subsumes. We make the new node for C the new parent of these nodes at level $i-1$.

On the highest level h, we end up with a root node for each component of $G \setminus D$. If the preprocessing algorithm stores the size of each component in its node of the hierarchy, the recovery algorithm can sum sizes along the way, and output the size of each connected component of $G \setminus D$. To answer a query find(u), we (conceptually) follow parent pointers from the hierarchy leaf corresponding to u to its root.

3.4. Analysis

Our analysis is similar to that for the flat hierarchy in Section 2.3. The following lemma shows correctness:

Lemma 6 *If no active components are left, the vertices in the passive components all belong to the same component of $H \setminus D$.*

Proof: Let I be the subgraph of $H \setminus D$ induced by the vertices in the passive components. If I is not connected in $H \setminus D$, there is a component B of $H \setminus D$ containing at most half the vertices. The vertices in the same passive component are connected, so B must be the disjoint union of some passive components. For each of these passive components C, we know that $\text{del}_{\leq i}(C) \leq \Phi \deg_i(C)$, so $\text{del}_{\leq i}(B) \leq \Phi \deg_i(B)$. However, H was a level-i component in a Φ-edge-expander hierarchy, and B contained at most half its vertices. Therefore B had more than $\Phi \deg_i(B)$ edges at level $\leq i$, leaving it. With $\text{del}_{\leq i}(B) \leq \Phi \deg_i(B)$, we conclude that $H \setminus D$ has some edge level $\leq i$ leaving B, contradicting that B is a maximal connected component of $H \setminus D$. ∎

Lemma 7 *On every level i, the number of level-i edges explored is at most $2d/\Phi$.*

Proof: When an edge is explored, it belongs to an active component. Consider the final active component it belonged to, even if it was explored before this component was created. These final active components are disjoint. By definition, an active component C has $\deg_i(C) < \text{del}_{\leq i}(C)/\Phi$. Finally, the sum of $\text{del}_{\leq i}(C)$ over the final active components is twice the number of deleted edges on levels $\leq i$. ∎

3.5. Implementation

We first describe an implementation using $O(nh^2 + m)$ space, $O(d(h^2 + h\alpha(d))/\Phi)$ update time, and $O(h)$ query time. Remember that this ultimately means $O(m + n \lg^2 n)$ space, $O(d \lg^3 n)$ update time, and $O(\lg n)$ query time. In the next section, we sketch how to improve all of these bounds.

Before the deletions arrive, we have the rooted tree representing the Φ-edge-expander hierarchy, with the nodes on level i identifying components of $G|_{\leq i}$. With each level i node C, we store the number of vertices in C, as well as the number $\deg_i(C)$. For each level $j \geq i$, the node C also stores pointers to the head and tail of a doubly linked list with the edges on level j incident to the component.

The representation of the lists of level-j incident edges is in fact more subtle. For each level j, we in fact have single global list with level j edges appearing twice (once per end-point). This global list respects the hierarchy in the sense that the level-j edges incident to a level-i component C form a connected sublist. Then the node for C simply has pointers to the head and tail of this sublist in the global list. The most important effect of a single global list is that when a node reorganizes its sublist, this immediately impacts all other nodes whose sublists contain or are contained in the modified sublist.

When a set D of d deletions arrives, we remove D from the global incidence lists in $O(d)$ time, and compute all the $\text{del}_{\leq i}$ counters in $O(dh)$ time. All such counters can be initialized to zero during preprocessing. We have now identified the nodes whose components lose any edges.

We are now going to implement the recovery on some level i, assuming this has already been done for levels up to $i-1$. For each component C of $(G \setminus D)|_{<i}$, and each $j \geq i$, we assume that we have correct heads and tails for the list of level-j edges incident to C. We no longer care about edges on level less than i, so we do not worry about these incidence lists.

We now have all the information for the recovery algorithm from Section 3.3. As in Theorem 3, we use union-find to maintain the level $i-1$ components making up an explored component. When checking if the end-points of an edge (u, v) are in the same explored component, we first use the parent pointers of the hierarchy to find their components on level $i-1$ in $O(h)$ time, and then we find the explored component in $\alpha(d)$ time, spending $O(h + \alpha(d))$ total time exploring an edge.

When an active component C is united with another component D, we add their counters and concatenate their lists of incident edges on levels $j > i$. That is, we make these sublists adjacent in the appropriate larger list of level-j edges. This takes constant time per level, so it does not affect the $O(h + \alpha(d))$ time used to explore an edge.

When no active components remain, exploration is over, and we want to collect all the passive components in a giant component. As in Section 3.3, we let the giant component take over the node of the original level-i component. This means that no parent pointer needs to be updated for the components on level $i-1$ which are subsumed by the giant component. For each isolated component, we create a new hierarchy node with the same parent as the giant component. Since we know the composition of each component, we can add parent pointers from the appropriate level-$(i-1)$ components to the isolated components on level i. Finally, we must obtain sublists of level-j edges adjacent to the giant component. This is done by "extracting" each level-j sublist for an isolated component, and reinserting is immediately after the sublist of the giant component.

3.6. Sketch of Further Refinements

Query time. We first improve the running time of find(u) from $O(h)$ to $O(\lg h)$. The idea is to binary search for the smallest i, such that the level-i ancestor C of u's leaf has $\text{del}_{\leq i}(C) > 0$. Querying the level-$i$ ancestor can be done in constant time. Knowing the lowest level affected by a delete is enough, because the update algorithm can go back down in the subtree of the hierarchy that it explored, and add a pointer to the final root of each node.

Recovery time. The update time can also be improved to $O(d \lg^2 n \lg\lg n)$, but details are more complicated. First, we need to find the level-i component for an end-point of an edge. Again, we binary search for the lowest explored ancestor. To go from there, we maintain a global union-find structure for all leaves of the explored subtree, which can find the current level-i root for each leaf.

The real challenge is how to handle maintenance of level-j sublists in $o(h)$ time. First observe that we can afford to pay for all level-j edges incident to a final active component on level i. This can cover the cost of concatenations of nonempty sublists. Empty sublists can be avoided by storing a bitmap in every node marking such lists. After reaching the final active step, we may do a last union yielding a passive component. The $O(h)$ time spent there can be amortized over the deletion creating the initial active component. Each deletion creates one active component per level (regardless of Φ), so we obtain an additive $O(h^2) = O(\lg^2 n)$ cost.

Linear space. A first problem is that the tree of the hierarchy itself may use hn nodes, since it has n leaves and height h. Moreover, from each node, we maintain pointers to incidence sublists on up to h different levels. The main idea is to compress this information to linear, and then uncompress it on the fly during recovery.

For the hierarchy, we compress paths of degree-1 nodes, by only storing leaves and nodes whose original parents had degree at least 2. This means that if a node is not stored explicitly, it represents the same subgraph as its parent. The compressed hierarchy has less than $2n$ nodes.

A similar approach is taken for the incidence lists. For each node C and each level j, the information we wanted to store was the heads and tails of the sublist in the global level-j edge list, as well as the counter $\deg_j(C)$. We will only store this information if the level-j incidence list is non-empty and strictly smaller than that of the parent. Note that if the original parent has a strictly larger incidence list, then it must also have degree bigger than one and hence be an explicit node. Finally, for each explicit node, we store a bitmap occupying one word, which specifies on which levels we have incident edges. If we have incident edges on level j, but no explicit information, we inherit the list from our nearest ancestor with this information.

Putting together all these refinements we will obtain:

Theorem 8 *Let G be a graph on n vertices and $\Phi \leq 1/\lg n$. Given a Φ-edge-expanding hierarchy for G, after linear time preprocessing, we can recover from the deletion of d edges in $O(d \lg n \lg\lg n/\Phi)$ time and subsequently answer* find$(\cdot)$ *queries in $O(\lg\lg n)$ time.*

An edge expander hierarchy with $\Phi = 1/(\gamma \cdot 2\lg n)$ can be constructed by a polynomial number of calls to a γ-approximation algorithm for vertex-weighted sparsest cut.

Optimality of query time. Interestingly, we can argue a tight lower bound on the query time, based on our optimal lower bound for the predecessor problem [14]. Specifically, [14] implies that for a set of N integers in the universe $\{1, \ldots, U\}$, with say $U^\varepsilon \leq N \leq U^{1-\varepsilon}$, a data structure of size $O(N \operatorname{polylg} U)$ cannot answer predecessor queries in less than $\Omega(\lg\lg U)$ time.

Our hard instance will be a perfect binary tree over U leaves (thus, $n, m = O(U)$). Given N numbers, let $\mathcal{P}$ be the union of the paths from the root to each one of the N leaves. We then perform $d = O(N \lg U)$ deletes to the tree, removing all edges outside but incident to $\mathcal{P}$. If these deletes run in $O(d \operatorname{polylg} n) = O(N \operatorname{polylg} U)$ time, they effectively construct a data structure of $O(N \operatorname{polylg} U)$ space, because they cannot write more memory cells. Thereafter, finding the component of a leaf is equivalent to finding the lowest ancestor in $\mathcal{P}$. This is the longest prefix problem, which is equivalent to predecessor search.

References

[1] S. Arora, S. Rao, and U. V. Vazirani. Expander flows, geometric embeddings and graph partitioning. In *Proc. 36th STOC*, pages 222–231, 2004.

[2] R. Chowdhury and V. Ramachandran. Improved distance oracles for avoiding link-failure. In *Proc. 13th ISAAC, LNCS 2518*, 2002.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, McGraw-Hill, 2nd edition, 2001. ISBN 0-262-03293-7, 0-07-013151-1.

[4] C. Demetrescu and M. Thorup. Oracles for distances avoiding a link-failure. In *Proc. 13th SODA*, pages 838–843, 2002.

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

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

[7] H. Gabow and R. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proc. 15th STOC*, pages 246–251, 1983.

[8] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(502–536), 1999. Announced at STOC'95.

[9] J. Hershberger and S. Suri. Vickrey prices and shortest paths: What is an edge worth? In *Proc. 42nd FOCS*, pages 252–259, 2001. Erratum in FOCS'02.

[10] J. Holm, K. Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. *J. ACM*, 48(4):723–760, 2001. Announced at STOC'98.

[11] V. King and G. Sagert. Fully dynamic algorithms for maintaining the transitive closure. In *Proc. 31st ACM Symp. on Theory of Computing*, pages 492–498, 1999.

[12] F. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999. Announced at FOCS'88.

[13] M. Pătraşcu and E. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006. Announced at SODA'04 and STOC'04.

[14] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th STOC*, pages 232–240, 2006.

[15] L. Roditty and U. Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. In *Proc. 32nd ICALP*, volume 3580 of *LNCS*, pages 249–l260, 2005.

[16] D. Spielman and S.-H. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proc. 36th STOC*, pages 81–90, 2004.

[17] R. E. Tarjan. Efficiency of a good but not linear set union algorithms. *J. ACM*, 22:215–225, 1975.

[18] M. Thorup. Near-optimal fully-dynamic connectivity. In *Proc. 32nd ACM Symp. on Theory of Computing*, 2000.

[19] M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proc. 37th STOC*, pages 112–119, 2005.