Incremental and Fully Dynamic Subgraph Connectivity For Emergency Planning

Monika Henzinger
University of Vienna, Faculty of Computer Science, Vienna, Austria
monika.henzinger@univie.ac.at

Stefan Neumann
University of Vienna, Faculty of Computer Science, Vienna, Austria
stefan.neumann@univie.ac.at

July 1, 2016

Abstract

During the last 10 years it has become popular to study dynamic graph problems in a *emergency planning* or *sensitivity* setting: Instead of considering the general fully dynamic problem, we only have to process a *single* batch update of size d; after the update we have to answer queries.

In this paper, we consider the dynamic subgraph connectivity problem with sensitivity d: We are given a graph of which some vertices are activated and some are deactivated. After that we get a single update in which the states of up to d vertices are changed. Then we get a sequence of connectivity queries in the subgraph of activated vertices.

We present the first fully dynamic algorithm for this problem which has an update and query time only slightly worse than the best decremental algorithm. In addition, we present the first incremental algorithm which is tight with respect to the best known conditional lower bound; moreover, the algorithm is simple and we believe it is implementable and efficient in practice.

1 Introduction

Dynamic graph algorithms maintain a data structure to answer queries about certain properties of the graph while the underlying graph is changed, e.g., by vertex or edge deletions and additions; such properties could be, for example, the connectivity or the shortest paths between two vertices. The main goal is that after an update the algorithm does not have to recompute the data structure from scratch, but only has to make a small number of changes to it. Due to strong conditional lower bounds for various dynamic graph problems (see [1, 16, 22]), it is necessary to restrict the dynamic model in some way to improve the efficiency of the operations. One model that has become increasingly popular is to study dynamic graph problems in a *sensitivity* or *emergency planning* setting (see, e.g., [10, 23, 7, 8, 4, 3, 21]): Instead of considering the general fully dynamic problem in which we get a sequence of updates and queries, we only allow for a single batch update of size d after which we want to answer queries. Since we allow only a single update, the update and query times for such sensitivity problems are much faster than for the general fully dynamic problem.

In this paper, we consider the subgraph connectivity problem with sensitivity d: We get a graph $G = (V, E)$ of which some vertices are *activated* and some are *deactivated* and we can preprocess it. There is a single update changing the states of up to d vertices. In the subsequent queries we need to answer if two given vertices are connected by a path which traverses only activated vertices. If the update can only active previously deactivated vertices, then an algorithm for this problem is called *incremental*; if it can only deactivate activated vertices, then it is *decremental*; if it can turn vertices on and off arbitrarily, then it is called *fully dynamic*.

The problem is of high practical interest as it models a scenario which is very relevant to infrastructure problems. For example, assume you are an internet service provider and you maintain many hubs which are connected to each other. In case of a defect, some of the hubs fail but there is a small number of *backup* hubs which can be used until the defect hubs are repaired in order to provide your services to your customers. Notice that in such a scenario it is likely that the number of backup hubs is much smaller than the number of regular hubs.

1.1 Our Contributions

We present the first incremental and fully dynamic algorithms for the subgraph connectivitiy problem with sensitivity d. The update and query times of our fully dynamic algorithm are only slightly slower than those of the best decremental algorithm for this problem. In addition, the incremental algorithm is essentially tight with respect to the best known conditional lower bound for this problem. Additionally, we contribute a characterization of the paths which are added to a graph when activating some nodes.

Our result for the fully dynamic problem with sensitivity d is given in the following theorem. We state the running time with respect to a blackbox algorithm for the decremental version of the problem as subprocedure. The number of initially deactivated vertices is denoted by n_{off}.

Theorem 1. *Assume there exists an algorithm for the decremental subgraph connectivity problem with sensitivity d that has preprocessing time t_p, update time t_u, query time t_q and uses space S. Then there exists an algorithm for the fully dynamic subgraph connectivity problem with sensitivity d that uses space $O(n_{\mathit{off}}^2 \cdot S)$ and has preprocessing time $O(n_{\mathit{off}}^2 \cdot t_p)$. It can process an update of d vertices in time $O(d^2 \cdot \max\{t_u, t_q\})$ and queries in time $O(d \cdot t_q)$.*

For the decremental version of the subgraph connectivity problem with sensitivity d (which is also referred to as *d-failure connectivity*), the best known algorithm is by Duan and Pettie [11]. Their result is given in the following lemma.

Lemma 2 ([11])**.** *Let $G = (V, E)$ be a graph and let $n = |V|$, $m = |E|$, let $c \in \mathbb{N}$. Then there exists a data structure for the decremental subgraph connectivity problem with sensitivity d that has size $S = O(d^{1-2/c} m n^{1/c - 1/(c \log(2d))} \log^2 n)$ and preprocessing time $\tilde{O}(S)$. An update deactivating d vertices takes time $O(d^{2c+4} \log^2 n \log\log n)$ and subsequent connectivity queries in the graph after the vertex deactivations take $O(d)$ time.*

As pointed out in [11], for moderate values of d the space S used by the data structure from Lemma 2 is $o(mn^{1/c})$; further, if $m < n^2$, then we always have $S = o(mn^{2/c})$. Using the algorithm of Lemma 2 as a subprocedure for our result from Theorem 1, we obtain the following corollary. The number of initially activated vertices is given by n_{on} and the number of initially activated edges in the graph is denoted by m_{on}.

Corollary 3. *There exists an algorithm for the fully dynamic subgraph connectivity problem with sensitivity d with the following properties. For any* $c \in \mathbb{N}$, *it uses space* $S' = O(n_{\mathit{off}}^2 \cdot S)$ *and preprocessing time* $\tilde{O}(S')$, *where* $S = O(d^{1-2/c} m_{on} n_{on}^{1/c - 1/(c \log(2d))} \log^2 n_{on})$. *It can process an update of d vertices in time* $O(d^{2c+6} \log^2 n_{on} \log\log n_{on})$ *and answer queries in the updated graph in time* $O(d^2)$.

In the case that we get an update of size $d' < d$, we can make the update and query times of the data structure depend only on d': We build the data structure for all values $d' = 2^1, \dots, 2^\ell$, where ℓ is the smallest integer such that $d \leq 2^\ell$. Asymptotically this will not use more space than building the data structure once for d; for an update of size d' we use the instance of the data structure for the smallest $2^i \geq d'$.

In the incremental algorithm we only allow for initially deactivated vertices to be activated. Our result for the incremental problem is given in the following theorem.

Theorem 4. *There exists an algorithm for the incremental subgraph connectivity problem with sensitivity d which has preprocessing time* $O(n_{\mathit{off}}^2 \cdot n_{on} + m)$, *update time* $O(d^2)$ *and query time* $O(d)$. *It uses space* $O(n_{\mathit{off}} \cdot n)$.

The algorithm is simple and we believe it is implementable and efficient in practice.

For our incremental data structure the sensitivity parameter d does not have to be fixed beforehand, i.e., once initialized, the data structure can process updates of arbitrary sizes and the update and query times will only depend on the size of the given update.

We observe that the conditional lower bound given in Henzinger et al. [16] for the decremental version of the problem can easily be altered to work for the incremental problem as well. The conditional lower bound states that under the Online Matrix vector (OMv) conjecture any algorithm solving the incremental subgraph connectivity problem with sensitivity d which uses preprocessing time polynomial in n and and update time polynomial in d must have a query time of $\Omega(d^{1-\varepsilon})$ for all $\varepsilon > 0$. Examining the proof of the lower bound, we observed that the maximum of the query and update time even has to be in $\Omega(d^{2-\varepsilon})$ for all $\varepsilon > 0$. Hence, the update and query times of our incremental algorithm are essentially optimal under the OMv conjecture.

1.2 Related Work

In recent years there have been several results studying data structures for problems in an emergency planning or sensitivity setting when only a single update of small size is allowed. The field was introduced by Patrascu and Thorup [23] who considered connectivity queries after d edge failures. Demetrescu et al. [8] studied distance oracles avoiding a single failed node or edge. This setting was also considered by Bernstein and Karger [3, 4]. Later, Duan and Pettie [10] studied distance and connectivity oracles in case of two vertex failures. Khanna and Baswana [21] studied approximate shortest paths for a single vertex failure. As mentioned in Section 1.1, Duan and Pettie [11] studied the decremental subgraph connectivity problem with sensitivity d. Chechik et al. [7] considered distance oracles and routing schemes in case of d edge failures.

For the decremental subgraph connectivity problem with sensitivity d there also exist conditional lower bounds by Henzinger et al. [16] from the OMv conjecture and most recently by Kopelowitz, Pettie and Porat [22] from the 3SUM conjecture. The highest conditional lower bounds is the one in [16], which states that under the OMv conjecture any algorithm using preprocessing time polynomial in n and and update time polynomial in d must have a query time of $\Omega(d^{1-\varepsilon})$ for all $\varepsilon > 0$. Hence, the query time of the decremental algorithm by Duan and Pettie [11] is essentially optimal with respect to the lower bound.

The general subgraph connectivity problem, which allows for an arbitrary number of updates, has gained an increasing interest during the last years. The problem was introduced by Frigoni and Italiano [14], who studied it for planar graphs; they achieved amortized polylogarithmic update and query times. In general graphs, Duan [9] constructed a data structure which uses almost linear space, preprocessing time $\tilde{O}(m^{6/5})$, worst-case update time $\tilde{O}(m^{4/5})$ and worst-case query time $\tilde{O}(m^{1/5})$. In an amortized setting, the data structure given by Chan, Patrascu and Roditty [6] has an update time $\tilde{O}(m^{2/3})$ and query time $\tilde{O}(m^{1/3})$; its space usage and preprocessing time is $\tilde{O}(m^{4/3})$. This improved an earlier result by Chan [5] significantly. The data structure of [6] was later improved by Duan [9] to use only $\tilde{O}(m)$ space. Baswana et al. [2] gave a deterministic worst-case algorithm with update time $\tilde{O}(\sqrt{mn})$ and query time $O(1)$. Further, conditional lower bounds were derived for the subgraph connectivity problem from multiple conjectures [1, 16]. The highest such lower bound was given in [16]; it states that under the OMv conjecture, the subgraph connectivity problem cannot be solved faster than with update time $\Omega(m^{1-\delta})$ and query time $\Omega(m^{\delta})$ for any $\delta \in (0, 1)$ when we only allow polynomial preprocessing time of the input graph. Hence, the update and query times of the aforementioned algorithms are optimal up to polylogarithmic factors and tradeoffs between update and query times.

Compared to the subgraph connectivity problem, it has a much longer tradition to study the (edge) connectivity problem in which updates delete or add edges to the graph. Henzinger and King [17] were the first to give an algorithm with expected polylogarithmic update and query times; the best algorithm using Las Vegas randomization is by Thorup [24] with an amortized update time of $O(\log n(\log\log n)^3)$. Holm, de Lichtenberg and Thorup [18] gave the first deterministic algorithm with amortized polylogarithmic update times; currently the best such algorithm is given by Wulff-Nilsen [25] which has an update time of $O(\log^2 n/\log\log n)$. Recently, Kapron, King and Mountjoy [19] were able to provide the first data structure which has expected *worst case* polylogarithmic time per update and query. The result of [19] was lately improved by Gibb et al. [15] to have update time $O(\log^4 n)$. However, the best deterministic worst case data structures still have running times polynomial in the number of nodes of the graph. For a long time the results by Frederickson [13] and Eppstein et al. [12] running in time $O(\sqrt{n})$ were the best known. Only recently this was slightly improved by Kejlberg-Rasmussen et al. [20], who were able to obtain a worst case update time of $O\left(\sqrt{\frac{n(\log\log n)^2}{\log n}}\right)$.

The rest of the paper is outlined as follows: We start with notation and preliminaries in Section 2. In Section 3 we prove the results for the incremental algorithm which will already contain the main ideas for the more complicated fully dynamic algorithm. Section 4 provides the main result of this paper.

2 Preliminaries

In this section, we formally introduce the subgraph connectivity problem with sensitivity d. At the end of the section, we show a lemma that characterizes when disconnected vertices become connected after activating additional vertices; the lemma will be essential to prove the correctness of our algorithms.

The *subgraph connectivity problem* with sensitivity d is as follows: Let $G = (V, E)$ be a graph with n vertices and m edges and a partition of the vertices into sets V_{on} and V_{off}. The vertices in V_{on} are said to be *turned on* or *activated* and those in V_{off} are said to be *turned off* or *deactivated*. We get a single batch update in which the states of up to d vertices are be changed. In a query for two vertices u and v, the algorithm has to return if there exists a path from u to v only traversing activated vertices.

As we consider the subgraph connectivity problem in a sensitivity setting, after processing a single update and a sequence of queries, we roll back to the initial input graph. Hence, the data structure does not allow to alter the graph by an arbitrary amount. This allows us to offer much faster update and query times than the best algorithms which solve the general fully dynamic problem.

We introduce more notation. By G_{on} we denote the projection of G on the vertices which are initially on, i.e., $G_{\text{on}} = G[V_{\text{on}}] = (V_{\text{on}}, E_{\text{on}})$, where $E_{\text{on}} = \{(u, v) \in E : u, v \in V_{\text{on}}\}$. We set $E_{\text{off}} = E \setminus E_{\text{on}}$ to the set of edges which have at least one endpoint in V_{off}. To distinguish between the sizes of the activated and deactivated vertices and edges, we set $n_{\text{on}} = |V_{\text{on}}|$ to the number of activated vertices and $n_{\text{off}} = |V_{\text{off}}|$ to the number of deactivated vertices. Further, we set $m_{\text{on}} = |E_{\text{on}}|$ to the number of edges in G_{on} and $m_{\text{off}} = |E_{\text{off}}|$.

With this notation we can quickly describe the main difficulties of the subgraph connectivity problem: If G_{on} is connected, then already deactivating a single vertex of V_{on} can make it fall apart into $\Theta(n_{\text{on}})$ connected components; on the other hand, in G_{on} we can have $\Theta(n_{\text{on}})$ connected components initially and activating a single vertex of V_{off} with $\Theta(n_{\text{on}})$ edges can make the resulting graph connected. Hence, when deactivating or activating vertices, the number of connected components can change arbitrarily much. However, the update and query times of our algorithms are not supposed to polynomially depend on n, but only on the size of the udpate d which will usually be much smaller.

2.1 Characterisation of Paths After Activating Vertices

In this subsection, we introduce the terminology to characterize when vertices in a graph G become connected after we activated the vertices of a set I.

We say that a deactivated vertex $v \in V_{\text{off}}$ and a connected component C of G are *adjacent*, if there exists a vertex $u \in C$ such that $(u, v) \in E$. Two vertices

$u, v \in V_{\text{off}}$ are *connected via a connected component*, if (1) there exists a connected component C to which both u and v are adjacent or (2) if $(u, v) \in E$. In other words, u and v are connected via a connected component if they can reach each other by a path that only traverses vertices from a single connected component of G or if u and v are connected by an edge. Two connected components $C_1 \neq C_2$ are *connected by the set* I if there exists a sequence of vertices $v_1, \dots, v_k \in I$ such that (1) v_1 is adjacent to C_u, (2) v_k is adjacent to C_v and (3) v_i and v_{i+1} are connected via a connected component for all $i = 1, \dots, k-1$.

We can characterize when two disconnected vertices become connected in G after the vertices of the set I are activated. This is done in the following lemma.

Lemma 5. *Let $G = (V, E)$ be a graph with V_{on} and V_{off} as before. Further, let $I \subseteq V_{off}$ be a set of vertices which is activated. Let u, v be two disconnected vertices in G_{on} and let $C_u \neq C_v$ be their connected components. Then u and v are connected in G after activating the vertices in I if and only if C_u and C_v are connected by the set I.*

Proof. Assume u and v are connected in G after activating the vertices in I. Then there exists a path $u = w_0 \to w_1 \to \cdots \to w_\ell \to w_{\ell+1} = v$ in G; let $w_{j_1}, \dots, w_{j_r}$ be the vertices of the path which are from the set I with $j_i < j_{i+1}$ for all $i = 1, \dots, r$. Now observe that for all i, the vertices $w_{j_i+1}, \dots, w_{j_{i+1}-1}$ must be in the same connected component C_{j_i}. Clearly, w_{j_i} and $w_{j_{i+1}}$ are adjacent to C_{j_i} and they are connected by the connected component C_{j_i}. The same arguments can be used to show that C_u and w_{j_1} are adjacent and to show that C_v and w_{j_r} are adjacent. This implies that C_u and C_v are connected by the set I.

The other direction of the proof is symmetric. □

We will use Lemma 5 to argue about the correctness of our algorithms. In particular, when we prove the correctness of our algorithms we show that the connected components of the query vertices become connected by the set I of newly activated vertices. This is useful as we can preprocess which vertices of V_{off} are connected via connected components and which deactivated vertices are reachable from the connected components of G. With these properties, we are able to avoid having to keep track of all connected components of G after an update.

3 Incremental Algorithm

In this section, we describe an algorithm for the incremental subgraph connectivity problem that has preprocessing time $O(n_{\text{off}}^2 \cdot n_{\text{on}} + m)$, update time $O(d^2)$, query time $O(d)$ and uses space $O(n_{\text{off}} \cdot n)$. This will prove Theorem 4 stated in the introduction.

The main idea of the algorithm is to exploit Lemma 5 by preprocessing which deactivated vertices are connected by connected components of G_{on} and preprocessing the adjacency of deactivated vertices and connected components of G_{on}.

3.1 Preprocessing

We first compute the connected components $C_1, \ldots, C_k$ of G_{on} and label each vertex in V_{on} with its connected component. For each connected component C_i, we use a binary array A_{C_i} of size n_{off} to store which vertices in V_{off} are adjacent to C_i. We further equip each vertex $u \in V_{\text{off}}$ with a binary array A_u of size $k + n_{\text{off}} = O(n)$: In the first k entries of A_u, we store to which connected components u is connected; in the final n_{off} components of A_u, we store to which $v \in V_{\text{off}}$ the vertex u is connected by a connected component.

When the algorithm performs updates, it uses the arrays A_u to determine in constant time if u is connected to other deactivated vertices from V_{off} via connected components. This avoids having to check all connected components C_i of the vertex u which could take time $\Theta(n_{\text{on}})$.

The preprocessing takes time $O(m_{\text{on}})$ to compute the connected components C_i and labeling the vertices in V_{on}. Using one pass over all edges we can compute the arrays containing the connectivity information between the C_i and V_{off}, i.e., we can fill the arrays A_{C_i} and the first k components of the A_u. This takes $O(m)$ time.

Notice that if u is adjacent to C_i, then A_u must have a 1 wherever A_{C_i} has a 1. Then to finish building the arrays A_u, we can compute the last n_{off} entries of A_u as the bitwise OR of the arrays A_{C_i} for all C_i which u is adjacent to. This can be done in time $O(k \cdot n_{\text{off}}) = O(n_{\text{on}} \cdot n_{\text{off}})$ for a single vertex u. Since we have n_{off} vertices in V_{off}, computing all A_u takes time $O(n_{\text{off}}^2 \cdot n_{\text{on}})$. The computation of the A_u therefore dominates the running time of the preprocessing.

The space we require during the preprocessing is $O(n_{\text{off}})$ for each connected component of G_{on} and $O(n)$ for each vertex in V_{off}. Hence, in total we require $O(n_{\text{off}} \cdot n)$ space and preprocessing time $O(n_{\text{off}}^2 \cdot n_{\text{on}} + m)$.

3.2 Updates

During an update which activates d vertices from a set I, we build the *increment graph* S with the vertices of I as its nodes. We add an edge between a pair of vertices $u, v \in I$ if they are connected by a connected component C of G_{on}. Notice that the increment graph encodes the connectivity of the vertices in I via the connected components of G_{on}.

Computationally, this can be done in time $O(d^2)$: For each pair of vertices $u, v \in I$, we check in A_u if u is connected to v via a connected component in time $O(1)$. As we have to consider $O(d^2)$ pairs of vertices, the total time to construct the increment graph is $O(d^2)$.

Finally, we compute the connected components $S_1, \ldots, S_\ell$ of S and label each vertex in S with its connected component. This can be done in time $O(|S|) = O(d^2)$. Hence, the total update time is $O(d^2)$.

3.3 Queries

Consider a query if two activated vertices u and v are connected.

We find the connected components C_u and C_v of u and v, respectively. If $C_u = C_v$, then we return that u and v are connected and we are done.

Otherwise, let S_i be a connected component of S. We consider each vertex w of S_i and check if it is connected to C_u or C_v using A_{C_u} and A_{C_v}. After considering all vertices of S_i, we check if both C_u and C_v are connected to S_i. If this is the case, we return that u and v are connected, otherwise, we proceed to the next connected component of S.

During the query we considered each vertex in S exactly once and spent time $O(1)$ processing it. Hence, the total query time is $O(d)$.

It is left to prove the correctness of the result of the queries. This is done in the following lemma.

Lemma 6. *Consider an update which activates the vertices from a set $I \subseteq V_{off}$. Then a query if two vertices u and v are connected in G after the update delivers the correct result.*

Proof. If in the query procedure we encountered that $C_u = C_v$, then the result of the algorithm is clearly correct.

If $C_u \neq C_v$, then observe that the algorithm returns true if and only if C_u and C_v are connected by the set I: Let S_i be the connected component of S for which the query returns true. Then there must exist vertices $w_1, \ldots, w_t$ in the increment graph such that (1) w_1 is adjacent to C_u, (2) w_t is adjacent to C_v and (3) (w_i, w_{i+1}) is an edge in S for all $i = 1, \ldots, t-1$. The first two claims are true because the query procedure checks this in the arrays A_{C_u} and A_{C_v}. By construction of the increment graph, the increment graph has an edge (w_i, w_{i+1}) if and only if those vertices are connected by a connected component (this follows from what we preprocessed in the arrays A_{w_i}). This implies that a query returns true iff C_u and C_v are connected by the set I.

By Lemma 5 the algorithm returns the correct answer. □

4 Fully Dynamic Algorithm

In this section, we present the main result of the paper. We provide a data structure for the fully dynamic subgraph connectivity problem with sensitivity d, i.e., we process a batch update which changes the states of at most d vertices. Our algorithm uses a data structure for the decremental problem as a subprocedure. Assume the decremental algorithm uses space S, preprocessing time t_p, update time t_u and query time t_q. Then the fully dynamic algorithm uses space $O(n_{\text{off}}^2 \cdot S)$, preprocessing time $O(n_{\text{off}}^2 \cdot t_p)$, update time $O(d^2 \cdot \max\{t_u, t_q\})$ and query time $O(d \cdot t_q)$.

We reuse the increment graphs which we used in the incremental algorithm. For the construction of the increment graphs we replace the vectors A_u and A_{C_i} of the previous section by slightly augmented versions of G_{on} which are equipped with a decremental subgraph connectivity data structure, e.g., the one of Lemma 2 by Duan and Pettie [11]. The purpose of the augmented graphs is to check if a pair of initially deactivated vertices is connected via a connected component after deactivating some vertices of V_{on}.

We sketch the main steps of our algorithm. In the preprocessing we build an augmented graph for each pair of vertices of V_{off}; each augmented graph is equipped with a decremental subgraph connectivity data structure. In an update, we first process the vertex deactivations in the augmented graphs. Then we build the increment graph of vertices that were activated. Queries are handled similarly to the incremental algorithm by using the increment graph, but we have to check if the vertices of the increment graph can still reach the query vertices (this connectivity may have been destroyed by the vertex deactivations).

4.1 Preprocessing

For each pair of nodes $u, v \in V_{\text{off}}$, we build the augmented graph $G_{u,v} = G[V_{\text{on}} \cup \{u, v\}]$, i.e., $G_{u,v}$ consists of G_{on} after adding u and v. Observe that u and v cannot introduce more than $O(n_{\text{on}})$ edges and hence $G_{u,v}$ still has $O(n_{\text{on}})$ vertices and $O(m_{\text{on}})$ edges. We equip $G_{u,v}$ with a decremental subgraph connectivity data structure with sensitivity d. Later, we use the graph $G_{u,v}$ to check if u and v are connected via a connected component after deleting vertices from G_{on}; intuitively, the graphs $G_{u,v}$ replace the vectors A_u and A_v of the incremental algorithm. We need space $O(n_{\text{off}}^2 \cdot S)$ to store the $G_{u,v}$ where S is the space to store G_{on} with the decremental data structure.

For each $u \in V_{\text{off}}$, we build the graph $G_u = G[V_{\text{on}} \cup \{u\}]$ and equip it with the decremental data structure; we further equip G_{on} with the decremental data structure. We use the graphs G_u to replace the arrays A_{C_i} of the incremental algorithm; we cannot use the arrays anymore because the connected components of G_{on} can fall apart due to vertex deactivations. The space we need to store the graphs G_u and G is $O(n_{\text{off}} \cdot S)$.

In total, the preprocessing takes space $O(n_{\text{off}}^2 \cdot S)$ and time $O(n_{\text{off}}^2 \cdot t_p)$.

4.2 Updates

Assume that we get an update U which deactivates the vertices of a set $D \subseteq V_{\text{on}}$ and activates the vertices of a set $I \subseteq V_{\text{off}}$ with $|D| + |I| \leq d$. Our update procedure has two steps: We first remove the vertices in D from $G_{u,v}$ for all newly activated vertices $u, v \in I$. After that we build the increment graph consisting of the vertices of I as we did in the incremental algorithm.

We describe the sketched steps of the update procedure in more detail. Firstly, we process the deletions of the set D. For each pair $u, v \in I$, we delete the vertices of D in $G_{u,v}$ in time t_u. Since we have $O(d^2)$ pairs of vertices of I to consider, this takes time $O(d^2 \cdot t_u)$.

We update G_{on} and all G_u by deleting the vertices the vertices from D. This does not take longer than updating the graphs $G_{u,v}$.

Secondly, we build the increment graph consisting of the vertices in I. For each pair of vertices $u, v \in I$, we add an edge $e = (u, v)$ to the increment graph if a query in $G_{u,v}$ returns that u and v are connected. Such a query takes time t_q. The time we spend to build the increment graph is $O(d^2 \cdot t_u)$. Finally, we compute the connected components of the increment graph in time $O(d^2)$.

Altogether, the total update time of the update procedure is $O(d^2 \cdot \max\{t_u, t_q\})$.

4.3 Queries

We handle the query if two vertices u and v of G are connected similarly as in the incremental algorithm by using the increment graph.

Before we use the increment graph, we query if u and v are connected in the instance of G_{on} in which we deactivated the vertices of the set D. If the query returns true, then u and v are connected, otherwise, we proceed by using the increment graph.

For each connected component B of the increment graph, we consider each vertex $w \in B$ and we query in G_w if w is connected to u or v. If B had vertices w, w' which are connected to u and v, respectively, then we return that u and v are connected. Otherwise, we proceed to the next connected component of the increment graph.

The total query time of our algorithm is $O(d \cdot t_q)$ as in the worst case we have to perform a query in G_w for each of the $O(d)$ vertices $w \in I$.

Notice that due to the vertex deactivations we cannot precompute the connected components C_i of G_{on} and their connectivity with vertices in V_{off} as we did in the incremental algorithm: Each C_i may consist of $\Theta(n_{\text{on}})$ vertices and might as well fall apart into $\Theta(n)$ connected components after the vertex deactivations. Hence, in the update procedure we cannot keep the information about the connectivity of the vertices C_i and the added vertices up to date, as this may take time $\Theta(n)$. For our construction this also rules out obtaining a better query time.

We conclude the section by proving that the query returns the correct results in the following lemma.

Lemma 7. *Consider an update U deactivating the vertices from a set D and activating the ones from a set I. Then a query if two vertices u and v are connected in G after the update delivers the correct result.*

Proof. In the query procedure, we first check if u and v are connected in G_{on} after deleting the vertices from D. Clearly, if the algorithm returns true, then u and v are connected.

We move on to argue about the correctness in the case that u and v are not connected in the graph $H = G_{\text{on}} \setminus D$. Let $C_1, \dots, C_k$ be the connected components of H (not those of G_{on}) and let C_u and C_v be the connected components of u and v. We show that a query returns that u and v are connected if and only if C_u and C_v are connected by the set I. Then Lemma 5 implies the correctness of the algorithm.

Observe that a query returns that u and v are connected if and only if there exists a connected component B in the increment graph which contains vertices $w_1, \dots, w_\ell \in B \subseteq I$, such that (1) w_1 is connected to u, (2) w_ℓ is connected to v and (3) there is an edge between w_i and w_{i+1} in the increment graph for all $i = 1, \dots, \ell - 1$: We obtain the first two properties from the queries in G_{w_1} and G_{w_ℓ}; the third property is true due to the queries in the augmented graphs $G_{w_i, w_{i+1}}$ and implies that the w_i are connected via connected components.

Hence, we conclude that a query returns that u and v are connected if and only if C_u and C_v are connected by the set I. Lemma 5 implies that the algorithm is correct. □

Acknowledgements. We would like to thank the reviewers for their helpful comments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

References

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *IEEE 55th Annual*

Symposium on Foundations of Computer Science (FOCS), pages 434–443, Philadelphia, PA, USA, 2014.

[2] Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. Dynamic DFS tree in undirected graphs: breaking the O(m) barrier. *CoRR*, abs/1502.02481, 2015.

[3] Aaron Bernstein and David Karger. Improved distance sensitivity oracles via random sampling. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 34–43, Philadelphia, PA, USA, 2008.

[4] Aaron Bernstein and David Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (STOC)*, pages 101–110, New York, NY, USA, 2009.

[5] Timothy M. Chan. Dynamic subgraph connectivity with geometric applications. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 7–13, New York, NY, USA, 2002.

[6] Timothy M Chan, Mihai Patrascu, and Liam Roditty. Dynamic connectivity: Connecting to networks and geometry. *SIAM Journal on Computing*, 40(2):333–349, 2011.

[7] Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. f-sensitivity distance oracles and routing schemes. *Algorithmica*, 63(4):861–882, 2011.

[8] Camil Demetrescu, Mikkel Thorup, Rezaul Alam Chowdhury, and Vijaya Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM Journal on Computing*, 37(5):1299–1318, 2008.

[9] Ran Duan. New data structures for subgraph connectivity. In *37th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 201–212, Bordeaux, France, 2010.

[10] Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 506–515, Philadelphia, PA, USA, 2009.

[11] Ran Duan and Seth Pettie. Connectivity oracles for failure prone graphs. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing (STOC)*, pages 465–474, New York, NY, USA, 2010.

[12] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997.

[13] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC)*, pages 252–257, New York, NY, USA, 1983.

[14] D. Frigioni and F. G. Italiano. Dynamically switching vertices in planar graphs. *Algorithmica*, 28(1):76–103, 2000.

[15] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015.

[16] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC)*, pages 21–30, New York, NY, USA, 2015.

[17] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[18] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

[19] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1131–1142, 2013.

[20] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Faster worst case deterministic dynamic connectivity. *CoRR*, abs/1507.05944, 2015.

[21] Neelesh Khanna and Surender Baswana. Approximate Shortest Paths Avoiding a Failed Vertex: Optimal Size Data Structures for Unweighted Graphs. In *27th International Symposium on Theoretical Aspects of Computer Science (STACS)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 513–524, 2010.

[22] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1272–1287, 2016.

[23] Mihai Patrascu and Mikkel Thorup. Planning for fast connectivity updates. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 263–271, 2007.

[24] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing (STOC)*, pages 343–350, 2000.

[25] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1757–1769, 2013.