Faster Randomized Worst-Case Update Time for Dynamic Subgraph Connectivity

Ran Duan[1,2] and Le Zhang[1,3]

[1] Institute for Interdisciplinary Information Sciences, Tsinghua University, China
[2] duanran@mail.tsinghua.edu.cn
[3] le-zhang12@mails.tsinghua.edu.cn

Abstract. Real-world networks are prone to breakdowns. Typically in the underlying graph G, besides the insertion or deletion of edges, the set of active vertices changes overtime. A vertex might work actively, or it might fail, and gets isolated temporarily. The active vertices are grouped as a set S. The set S is subjected to updates, i.e., a failed vertex restarts, or an active vertex fails, and gets deleted from S. Dynamic subgraph connectivity answers the queries on connectivity between any two active vertices in the subgraph of G induced by S. The problem is solved by a dynamic data structure, which supports the updates and answers the connectivity queries. In the general undirected graph, we propose a randomized data structure, which has $\widetilde{O}(m^{3/4})$ worst-case update time. The former best results for it include $\widetilde{O}(m^{2/3})$ deterministic *amortized* update time by Chan, Pătraşcu and Roditty [4], $\widetilde{O}(m^{4/5})$ by Duan [8] and $\widetilde{O}(\sqrt{mn})$ by Baswana, Chaudhury, Choudhary and Khan [2] deterministic *worst-case* update time.

1 Introduction

Dynamic subgraph connectivity is defined as follows: Given an undirected graph $G = (V, E)$ having m edges, n vertices with $m = \Omega(n)$, there is a subset $S \subseteq V$. The set E is subjected to edge updates of the forms $insert(e, E)$ or $delete(e, E)$, where e is an edge. There are vertex updates of the forms $insert(v, S)$ or $remove(v, S)$. Through vertex updates, S changes overtime. The query is on whether any two vertices s and t are connected in the subgraph of G induced by S.

The problem was first proposed by Frigioni and Italiano [12], and poly-logarithmic algorithms on connectivity were described for the special case of planar graphs. As to the general graphs, Chan [3] first described an algorithm of deterministic amortized update time $\widetilde{O}(m^{4\omega/(3\omega+3)})$[4], where ω is the matrix multiplication exponent. Adopting FMM (*Fast Matrix Multiplication*) algorithm of [6], the update time is $O(m^{0.94})$. Its query time and space complexity are $\widetilde{O}(m^{1/3})$ and linear, respectively. Later Chan, Pătraşcu, and Roditty [4] proposed a simpler algorithm with the improved update time of $\widetilde{O}(m^{2/3})$. The

[4] $\widetilde{O}(\cdot)$ hides poly-logarithmic factors.

F. Ellen et al. (Eds.): WADS 2017, LNCS 10389, pp. 337–348, 2017.
DOI: 10.1007/978-3-319-62127-2_29

space complexity of the new algorithm increases to $\widetilde{O}(m^{4/3})$. The new algorithm is of compact description, getting rid of the use of FMM. With the same update time, Duan [8] presented new data structures occupying linear space. Also a worst-case deterministic $\widetilde{O}(m^{4/5})$ algorithm was proposed by Duan [8]. Via an application of dynamic DFS tree [2], Baswana et al. discussed a new algorithm with $\widetilde{O}(\sqrt{mn})$ deterministic worst-case update time. Its query time is $O(1)$. An improvement of it is discussed in [5]. These results are summarized in Table 1.

A close related problem is dynamic graph connectivity, which cares only about the edge updates. Poly-logarithmic amortized update time was first achieved by Henzinger and King [15]. The algorithm is randomized Las Vegas. Inspired by it, Holm et al. [16] proposed a deterministic algorithm with $O(\lg^2 n)$[5] amortized update time, which is now one of the classic results in the field. A cell-probe lower bound of $\Omega(\lg n)$ was proved by Pătraşcu and Demaine [21]. The lower bound is amortized randomized. Near-optimal results were considered by Thorup [22], where a randomized Las Vegas algorithm was described with $O(\lg n(\lg\lg n)^3)$ amortized update time. The upper bound is recently improved to $O(\lg n(\lg\lg n)^2)$ by Huang et al. [17]. Besides the classic deterministic $O(\lg^2 n)$ result, a faster deterministic algorithm was proposed by Wulff-Nilsen [24], of which the update time is $O(\lg^2 n/\lg\lg n)$. Turning to the worst-case dynamic connectivity, a deterministic $O(\sqrt{n})$ update-time algorithm is Frederickson's $O(\sqrt{m})$ worst-case algorithm [11] sped up via *sparsification* technique proposed by Eppstein et al. [10]. The result holds for online updating of minimum spanning trees. With roughly the same structure, but different and simpler techniques, Kejlberg-Rasmussen et al. [19] provided the so far best deterministic worst-case bound of $O(\sqrt{n(\lg\lg n)^2/\lg n})$ for dynamic connectivity. After the discovery of $O(\sqrt{n})$ update-time algorithm, people were wondering whether any poly-logarithmic worst-case update time algorithm is possible, even randomized. The open problem stands firmly for many years. A breakthrough should be attributed to Kapron et al. [18]. Their algorithm is Monte-Carlo, with poly-logarithmic worst-case update time. It has several improvements until now, as done in [13, 23]. For subgraph connectivity, the trivial update time of $\widetilde{O}(n)$ follows from Kapron et al.'s algorithm. The query time of it for subgraph connectivity can also be improved to $O(1)$, as the explicit maintenance of connected components can be done without blowing up the $\widetilde{O}(n)$ update time. Very recently, Wulff-Nilsen [25] gave a Las Vegas data structure maintaining a minimum spanning forest in expected worst-case time polynomially faster than $\Theta(n^{1/2})$ w.h.p. per edge update. An independent work of Nanongkai and Saranurak [20] showed an algorithm with $O(n^{0.49306})$ worst-case update time w.h.p..

1.1 Our Results

The former $\widetilde{O}(m^{4/5})$ deterministic worst-case *subgraph* connectivity structure adopted as a sub-routine the $O(\sqrt{n})$ deterministic worst-case algorithm for dynamic *graph* connectivity. Now the randomized poly-logarithmic worst-case con-

[5] We use $\lg x$ to denote $\log_2 x$.

Table 1. Results on Dynamic Subgraph Connectivity

Update time	Query time	Notes
$\widetilde{O}(m^{4\omega/(3\omega+3)})$	$\widetilde{O}(m^{1/3})$	Amortized, deterministic, linear space [3]
$\widetilde{O}(\sqrt{mn})$	$O(1)$	Worst case, deterministic, space $\widetilde{O}(m)$ [2, 5]
$\widetilde{O}(m^{2/3})$	$\widetilde{O}(m^{1/3})$	Amortized, deterministic, space $\widetilde{O}(m^{4/3})$ [4]
$\widetilde{O}(m^{2/3})$	$\widetilde{O}(m^{1/3})$	Amortized, deterministic, linear space [8]
$\widetilde{O}(m^{4/5})$	$\widetilde{O}(m^{1/5})$	Worst case, deterministic, space $\widetilde{O}(m)$ [8]
$\widetilde{O}(m^{3/4})$	$\widetilde{O}(m^{1/4})$	Worst case, randomized, linear space, this paper

nectivity structures for dynamic *graph* connectivity are discovered. We consider the question of whether it brings progress in *subgraph* connectivity. The answer is affirmative. But it does not come by simple replacement. More precisely, we tried in vain to get an improvement by carefully tuning the former setting of the $\widetilde{O}(m^{4/5})$ algorithm. Intuitively, the amortized $\widetilde{O}(m^{2/3})$ update time was achieved partially because it uses the connectivity structure of poly-logarithmic amortized update time. Now poly-logarithmic worst-case algorithms are discovered, it seems that the $\widetilde{O}(m^{2/3})$ worst-case update time is in sight. Nonetheless, we found that it is still hard to get the $\widetilde{O}(m^{2/3})$ update time. Until now we obtain the update time of $\widetilde{O}(m^{3/4})$. The main contribution is a new organization of the auxiliary data structures.

The $\widetilde{O}(\sqrt{mn})$ result comes from dynamic DFS tree [2, 5], which is a periodic rebuilding technique with fault tolerant DFS trees. Our result is always no worse than $\widetilde{O}(\sqrt{mn})$ as $n = \Omega(m^{1/2})$. Faster query time can be traded with slower update time for the bottom four results in Table 1. As to our result, $\widetilde{O}(m^{3/4+\epsilon})$ update time and $\widetilde{O}(m^{1/4-\epsilon})$ query time can be implemented. Note that the trade-offs are in *one direction*, i.e. better query time with worse update time, but not vice-versa. Consequently, the former $\widetilde{O}(m^{4/5})$ algorithm never gives update time of $\widetilde{O}(m^{3/4})$. The trade-off phenomenon is definitely hard to break, as indicated by the OMv (*Online Boolean Matrix-Vector Multiplication*) conjecture proposed by Henzinger et al. [14], which rules out *polynomial* pre-processing time algorithms with the product of amortized update and query time being $o(m)$. Based on the conjecture of no truly subcubic combinatorial boolean matrix multiplication, Abboud and Williams [1] showed that any combinatorial dynamic algorithm with truly sublinear in m query time and truly subcubic in n preprocessing time must have $\Omega(m^{1/2-\delta})$ update time for all $\delta > 0$ unless the conjecture is false. Our result is grouped as the following theorem.

Theorem 1.1 (Main Theorem) *Given a graph $G = (V, E)$, there is a data structure for the dynamic subgraph connectivity, which has the worst-case vertex (edge) update time $\widetilde{O}(m^{3/4})$, query time $\widetilde{O}(m^{1/4})$, where m is the number of edges in G, rather than in the subgraph of G induced by S. The answer to each query is correct if the answer is "yes", and is correct w.h.p. if the answer is "no." The pre-processing time is $\widetilde{O}(m^{5/4})$, and the space usage is linear.*

2 Preliminaries

Theorem 2.1 ([19]) *A spanning forest F of G can be maintained by a deterministic data structure of linear space, with $O(\sqrt{m(\lg\lg n)^2/\lg n})$ worst-case update time for an edge update in G, and constant query time to determine whether two vertices are connected in G.*

Theorem 2.2 ([18, 23]) *There is a randomized data structure on dynamic graph connectivity, which supports the worst-cast time $O(\lg^4 n)$ per edge insertion, $O(\lg^5 n)$ per edge deletion, and $O(\lg n/\lg\lg n)$ per query. For any constant c the answer to each query is correct if the answer is "yes" and is correct with probability $\geq 1 - 1/n^c$ if the answer is "no." The pre-processing time of it is $O(m\lg^3 n + n\lg^4 n)$.*

Moving to subgraph connectivity, here we consider only the case of vertex updates, with the extension to edge updates deferred to the full paper [9]. Hence temporarily G is assumed to be static, as E does not change if there are no edge updates. The vertex updates change S. Initially, G is slightly modified to keep $m = \Omega(n)$ during its lifetime, i.e., for every $v \in V$, insert a new vertex v' and a new edge (v, v'). The variant graph has $m = \Omega(n)$, which facilitates the presentation of time and space complexity as functions of m in the case of degenerate graphs.

3 The Data Structure

We give some high-level ideas, which originate from [4]. Main difficulties are the update of S (recall that S is the set of active vertices) incurred by the high-degree vertices, as their degrees are too high to explicitly delete their incident edges one by one. Nonetheless, if the low-degree vertices had been removed, the graph became smaller, and consequently former high-degree vertices were not high-degree anymore. Hence our aim is to remove the low-degree vertices. After that, some artificial edges are added to restore the loss of connectivity due to the removal of the low-degree vertices. Next a dynamic connectivity data structure is maintained on the modified graph, i.e., the graph with the low-degree vertices removed, and the artificial edges added. Besides, as S evolves dynamically, we need to update the artificial edges accordingly. Hence the *point* is how to maintain these artificial edges consistently and efficiently. We now move to the details. We partition V according to their degrees in G. Use $\deg_G(v)$ to denote the degree of v in G.

- C: Vertices with $\deg_G(v) > m^{1/2}$
- B: Vertices with $m^{1/4} < \deg_G(v) \leq m^{1/2}$
- A: Vertices with $\deg_G(v) \leq m^{1/4}$

Denote $C \cap S$, $B \cap S$, and $A \cap S$ as V_C, V_B, and V_A respectively. Consider the subgraph G_A of G induced by V_A. Define the *degree* of a component as the sum of $\deg_G(v)$'s for v's in it. According to the degrees of the components, partition the components of G_A into two types: *high* component, with its degree $> m^{1/4}$; *low* component, with its degree $\leq m^{1/4}$. A spanning forest F_A of G_A is maintained by the deterministic connectivity structure of Theorem 2.1.

3.1 Path Graph

A path graph inserts some artificial edges to reflect the "are connected" relation of the vertices within V_B via directly linking with a component of G_A. We give a more elaborate analysis based on [8]. W.l.o.g. assume $V = \{0, \ldots, n-1\}$. Consider a spanning tree T of F_A.

- *subpath tree*: For $v \in T$, identify the set of vertices in V_B that are adjacent to v. Store the set of vertices in a balanced search tree, which has the worst-case $O(\lg n)$ update time for the well-known search-tree operations [7]. Name the search tree as the subpath tree of v. Given the subpath tree of v, a sequence of artificial edges is added to link the vertices stored in the subpath tree of v. The sequence of artificial edges constitutes a subpath.
- *path tree*: Given $T \in F_A$, group all $v \in T$ with the *non-empty* subpath tree as a balanced search tree, ordered by the Euler-tour order of T. Name it as the path tree of T. As each vertex stored in the path tree of T has an associated subpath, these subgraphs are also concatenated one by one via the artificial edges, generating a path. To emphasize its difference from an ordinary path, it is referred to as the *path graph* of V_B w.r.t. T. An example is shown in Fig. 1.

Lemma 3.1 *The path graphs can be updated in $\widetilde{O}(m^{1/2})$ time for a vertex update in V_B, and in $\widetilde{O}(1)$ time for a link or cut on F_A.*

Proof. We categorize the analysis into two cases.

- Reflect a vertex update in V_B: Suppose $v \in V_B$ is removed from S. The case of insertion is similar. v has $\leq m^{1/2}$ edges adjacent to F_A. Consider (v, w) with $w \in T$. We locate w in the path tree of T. Now the subpath associated with w is known. Update the subpath of w by removing v from the subpath. If v happens to be the first or the last vertex on the subpath, the path graph of T is also updated. As the subpaths and the path graph are concerned with the nodes stored in the subpath trees and the path tree respectively, which are all balanced search trees, the removal of (v, w) needs $\widetilde{O}(1)$ time. The removal of all such (v, w)'s requires $\widetilde{O}(m^{1/2})$ time.

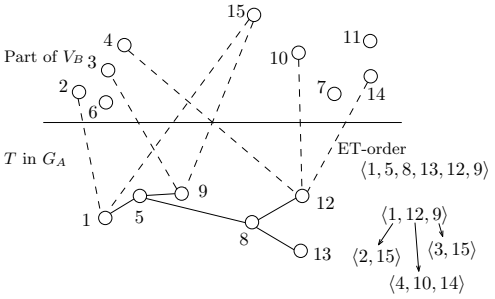

Fig. 1. The path graph of V_B w.r.t. a spanning tree T in F_A. The *dashed* edges represent edges between V_B and V_A. The path tree is on sequence $\langle 1, 12, 9\rangle$, and three subpath trees are on sequences $\langle 2, 15\rangle$, $\langle 4, 10, 14\rangle$, and $\langle 3, 15\rangle$ respectively. The resulted path graph is a path $\langle 2, 15, 4, 10, 14, 3, 15\rangle$.

– Reflect a link or cut on F_A: We only discuss the edge cut on F_A. The edge link is similar. Assume the edge cut is $(v, w) \in T$, and the Euler tour of T is $\langle L_1, (v, w), L_2, (w, v), L_3\rangle$ (The details can be found in the full paper [9].). After the cut of (v, w), the Euler tours for the two resulted trees are $\langle L_1, L_3\rangle$ and $\langle L_2\rangle$. We can determine the first vertex a and the last vertex b of $\langle L_2\rangle$. With the order tree of T (discussed in the full paper [9]), the predecessor of a and the successor of b in the path tree of T can be found in $O(\lg^2 n)$ time. With the predecessor and the successor, the path tree of T is split. After the split, $O(1)$ edges in the path graph are removed to reflect the split of the path tree of T. As a conclusion, the path graph can be updated in $\widetilde{O}(1)$ time to reflect a link or cut on F_A.

□

3.2 Adjacency Structure

Given $T \in F_A$ and $v \in C$, we want a data structure that provides the fast query of whether T and v are adjacent, i.e., whether an edge (u, v) exists with $u \in T$. We give a more elaborate analysis based on [8]. Assuming $v \in C$, the adjacency structure of v contains the following search trees.

– *sub-adjacency tree*: Given $T \in F_A$, identify the set of vertices in T that are adjacent to v. Store the set of vertices as a balanced search tree, ordered by the Euler-tour order of T. Name the balanced search tree as the sub-adjacency tree of v w.r.t. T.
– *adjacency tree*: Identify $T \in F_A$ by the smallest vertex in T. Group all $T \in F_A$, w.r.t. which v has non-empty sub-adjacency trees, as a balanced search tree. Name the balanced search tree as the adjacency tree of v.

The sub-adjacency trees and the adjacency tree of v constitute the adjacency structure of v w.r.t. F_A. The query aforementioned is answered by checking

whether T is in the adjacency tree of v. Note $v \in C$, rather than $\in V_C$. The adjacency structure of $v \in C$ w.r.t. F_A is maintained even if $v \notin S$.

Lemma 3.2 *The adjacency structures of C w.r.t. F_A can be renewed in $\widetilde{O}(m^{1/2})$ time for a link or cut on F_A. Given a query of whether $v \in C$ is adjacent to $T \in F_A$, it can be answered in $\widetilde{O}(1)$ time.*

Proof. We only discuss the edge cut on F_A. The edge link is similar. The adjacency structures of the vertices in C are renewed one by one. Consider $v \in C$. Suppose the edge cut occurs on T, splitting T into T_1 and T_2. We check whether T is in the adjacency tree of v. If "no", the update is done; if "yes", remove T from it, and update the sub-adjacent tree of v w.r.t. T to reflect the edge cut on T. For T_j $(j = 1, 2)$, add T_j into the adjacent tree of v if it is adjacent to v (determined by whether a sub-adjacent tree of v exists w.r.t. T_j). For every vertex in C, we need to check and update when necessary. Hence the total update time is $\widetilde{O}(m^{1/2})$, since $|C|$ is $O(m^{1/2})$. The query is answered by checking whether T is in the adjacency tree of v. □

3.3 The Whole Structure

Now we turn to the discussion of the whole structure of our result. First, V_A is removed. After that some artificial vertices and edges are added to the subgraph of G induced by $V_B \cup C$, resulting in a graph H. (Note that we include the vertices in $C \setminus S$, rather than just V_C, which is $C \cap S$.) The artificial vertices and edges are used to restore the loss of connectivity due to the removal of V_A. Recall that the components of G_A are either low or high. We describe how the artificial edges or vertices are added as follows.

- Added by the path graphs: For $T \in F_A$, construct the path graph of V_B w.r.t. T.
- Added by the high components: For a high component $P \in G_A$, add a *meta-vertex*. For $v \in C$ adjacent to P, add an artificial edge between v and the meta-vertex. Identify the first vertex of the path graph of V_B w.r.t. T, where T is the spanning tree of P. Add an artificial edge between *the* first vertex and the meta-vertex.
- Added by the low components: For a low component $Q \in G_A$, construct a complete graph within the vertices in C that are adjacent to Q. Similarly as above, identify the first vertex of the path graph of V_B w.r.t. T, where T is the spanning tree of Q. Add the artificial edges between *the* first vertex and the vertices in C that are adjacent to Q.

After these, H can be defined as follows.

- The vertex set $V(H)$ of H: $V_B \cup C \cup M$, where M is the set of meta-vertices. Since the degree of a high component is $> m^{1/4}$, and the vertices in $V_B \cup C$ are of degree $> m^{1/4}$, H has $O(m^{3/4})$ vertices.

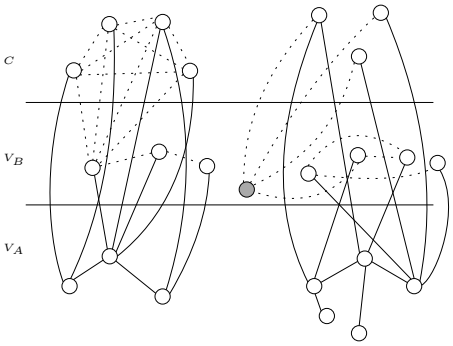

Fig. 2. An example of the whole structure. The irrelevant edges within V_A, V_B, and C are omitted for clarity. The *solid* edges are the edges in G, while the *dotted* edges denote the artificial edges. The *grey* vertex in the V_B layer indicates a meta-vertex. The left component of V_A is low; whereas the right one is high. We construct a complete graph within the vertices in C w.r.t. the low component.

– The edge set $E(H)$ of H: The original edges of G within $V_B \cup C$, and the artificial edges.

Figure 2 gives an example for the construction. H is a multigraph. Use $D[u, v] > 0$ of edge multiplicity to represent the edge $(u, v) \in E(H)$. The maintenance of $D[u, v]$'s is discussed later. Now we construct a graph G^*, based on H.

– The vertex set $V(G^*)$ of G^*: $V_B \cup V_C \cup M$.
– The edge set $E(G^*)$ of G^*: The edges (u, v)'s with $D[u, v] > 0$, where $u, v \in V(G^*), u \neq v$.

G^* is a variant of the subgraph of H induced by $V_B \cup V_C \cup M$. It excludes the vertices in $C \setminus S$, i.e., only the vertices in V_C of C are contained. Besides, the multiple edges are substituted by the single ones. G^* is a simple graph. The randomized connectivity structure of Theorem 2.2 is maintained on G^*.

About the $D[u, v]$'s aforementioned, a balanced search tree is used to store them, with $D[u, v]$ indexed by $u + nv$ (assuming $u \leq v$). Only $D[u, v] > 0$ is stored in the search tree. Along the process of the updates, we might increment or decrement $D[u, v]$'s. When $D[u, v]$ decrements to 0, we remove it from the search tree. If both u and v are the vertices in G^* and $u \neq v$, the edge (u, v) is deleted from G^*. Similar updates works for incrementing. G^* captures the property of connectivity, which is stated in the following lemma.

Lemma 3.3 *For any two vertices $u, v \in V_B \cup V_C$, they are connected in the subgraph of G induced by S if and only if they are connected in G^*.*

Proof. G^* is a variant of the subgraph of G induced by S. G^* removes V_A from *the* subgraph. Connectivity within V_B via V_A is restored by the path graphs. Connectivity within V_C via V_A is restored *either* by linking with the same meta-vertex,

or by the complete graph constructed. Lastly, for the connectivity between V_C and V_B via V_A, it is restored by the first vertex of the path graph linking with the meta-vertex, or with all the relevant vertices in V_C. Consider a path between u and v in the subgraph induced by S, the segments of the path consisting only of the vertices in V_A can be eliminated, as the "via V_A" connectivity is restored as discussed. The lemma follows. □

3.4 Update and Query

The difficulty of the vertex updates is to keep $D[u, v]$'s being consistent with S. As $E(G^*)$ is a subset of the (u, v)'s with $D[u, v] > 0$, it might also need to be updated.

Lemma 3.4 *The whole structure constructed has the worst-case vertex update time* $\widetilde{O}(m^{3/4})$.

Proof. We discuss the various cases of vertex updates, categorized according to whether $v \in A$, or $\in B$, or $\in C$.

- $v \in A$: Consider the case of inserting v into S. v is first inserted as a singleton component containing only v in G_A. Next the edges incident on v are restored in the following order: First, the edges between v and C; second, the edges between v and V_B; third, the edges between v and V_A.
  Restore the edges between v and C: For every u adjacent to v where $u \in C$, construct a sub-adjacency tree (containing only v) of u, and insert v into the adjacency tree of u. Next the complete graph within these u's in C is constructed. Because $\deg_G(v) \leq m^{1/4}$, i.e. a low component, the update time is $\widetilde{O}(m^{1/2})$, dominated by constructing the complete graph.
  Restore the edges between v and V_B: Construct the subpath tree and the path tree of v. Add the path-graph edges associated with v (*Add* means incrementing the corresponding entry $D[u, v]$), and the edges *between* the first vertex of the path graph *and* the vertices in C that are adjacent to v. The update time is $\widetilde{O}(m^{1/4})$.
  Restore the edges between v and V_A: $\widetilde{O}(\sqrt{m})$ deterministic data structure maintaining F_A is updated in $\widetilde{O}(m^{3/4})$ time. As $\deg_G(v) \leq m^{1/4}$, the link or cut on F_A happens $O(m^{1/4})$ times. Consequently, according to Lemma 3.1, the path graphs are updated in $\widetilde{O}(m^{1/4})$ time. According to Lemma 3.2, the adjacency structures are updated in $\widetilde{O}(m^{3/4})$ time.
  $O(m^{1/4})$ components of G_A are affected. For every high component, using the adjacency structures, the edges between C and the meta-vertex (corresponding to the high component) can be determined in $\widetilde{O}(m^{1/2})$ time according to Lemma 3.2, since $|C| = O(m^{1/2})$; for every low component, as the degree of a low component is $\leq m^{1/4}$, $\widetilde{O}(m^{1/2})$ time suffices to construct the complete graph within the vertices in C that are adjacent to the low component, and $\widetilde{O}(m^{1/4})$ time suffices to construct the edges *between* the first vertex of the

path graph w.r.t. the low component *and* the vertices in C that are adjacent to the low component. Hence no matter whether the component is low or high, the update time is $\widetilde{O}(m^{1/2})$. The time needed to update all these components is $\widetilde{O}(m^{3/4})$. Deleting of $v \in S$ from S is a reverse process. In summary, a vertex update of $v \in A$ requires $\widetilde{O}(m^{3/4})$ time.

- $v \in B$: Consider the case when $v \in S$ is removed. The case of insertion is the reverse. First destroy the edges between v and V_A. According to Lemma 3.1, the path graphs can be updated in $\widetilde{O}(m^{1/2})$ time. Besides, v might be the first vertex of some path graphs. We see how it is updated. v can be adjacent to $\leq m^{1/2}$ components of G_A, as $\deg_G(v) \leq m^{1/2}$. For a high component, as only one edge linking v with the meta-vertex, the update is easy; for a low component, since only $\leq m^{1/4}$ edges can be outward for a low component, $\widetilde{O}(m^{1/4})$ time suffices for updating the edges between v and the vertices in C that are adjacent to the low component. Hence the update time for v being the first vertex of some path graphs is $\widetilde{O}(m^{3/4})$. Until now the artificial edges concerning v are removed. Other edges concerning v are the original edges in G. Hence we can remove these original edges one by one in $\widetilde{O}(m^{1/2})$ time as $\deg_G(v) \leq m^{1/2}$. In summary, the total update time of $v \in B$ is $\widetilde{O}(m^{3/4})$.
- $v \in C$: As there are only $O(m^{3/4})$ vertices in G^*, the update time is $\widetilde{O}(m^{3/4})$. The relevant $D[u, v]$'s are left intact, and the adjacency structure of v is not destroyed (if v is removed from S). The total update time is $\widetilde{O}(m^{3/4})$.

□

The query algorithm is as follows: Given $u, v \in S$, the goal is to substitute them with the *equivalent* vertices in G^*, where an *equivalent* vertex of u (or v) is a vertex in G^* that is connected with u (or v). As $V(G^*) = V_B \cup V_C \cup M$, if $u, v \in V_B \cup V_C$, the search for the equivalent vertices is done. Otherwise, if u (or v) is in a high component, replace u (or v) with the meta-vertex corresponding to the high component; if u (or v) is in a low component, exhaustively search the outward edges of the low component for a vertex of G^*. When the equivalent vertex of u (or v) cannot be found, it indicates that u (or v) is in a low component of G_A, and the low component is not connected with any vertex in $V_B \cup V_C$. Intuitively u (or v) is on an "island" of G_A.

Lemma 3.5 *The time complexity of the query algorithm is $\widetilde{O}(m^{1/4})$. The answer to every query is correct if the answer is "yes", and is correct w.h.p. if the answer is "no".*

Proof. Connectivity within G^* is answered by the randomized connectivity structure on G^*; whereas for the other cases, u and v are connected if and only if they are in the same component of G_A, of which the queries can be answered by the deterministic connectivity structure on G_A. The time complexity is dominated by the exhaustive search if u (or v) is in a low component, and thus is $\widetilde{O}(m^{1/4})$.

The correctness can be analyzed as follows. If $u, v \in V_B \cup V_C$, it follows from Lemma 3.3; otherwise, for any one not in, we only replace it with an equivalent vertex of G^*. If such an equivalent vertex cannot be found, the queried vertex is

on an island aforementioned of G_A. Then u and v are connected if and only if they are on the same island. We analyze the error probability. A deterministic connectivity structure is adopted for G_A. F_A is always a spanning forest of G_A. The queries are answered *either* by the deterministic connectivity structure if at least one queried vertex is on an island aforementioned of G_A, *or* by the randomized connectivity structure if both queried vertices are (replaced with) the vertices in G^*. The deterministic connectivity structure always gives the right answer; whereas the randomized one might answer erroneously. The randomized algorithm of [18] maintains a private *witness* of a spanning forest of G^*. The algorithm has the property that after every update, the witness is a spanning forest of G^* with probability $\geq 1 - 1/n^c$. It is the property which ensures the answers are correct w.h.p.. Here, after every vertex update (which is transformed into a sequence of edge updates in G^*), the witness for G^* is also a spanning forest of G^* w.h.p. after the vertex update. We can just focus on the correctness of the witness at the point after the last transformed edge update. Consequently, the error probability is negligible, i.e., $\leq 1/n^c$ for any constant c. □

The proofs of the pre-processing time being $\widetilde{O}(m^{5/4})$, and the space usage being linear can be found in the full paper [9]. Hence Theorem 1.1 follows.

Acknowledgments. This work was supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61361136003. R. Duan is supported by a China Youth 1000-Talent grant.

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*, pages 434–443. IEEE, 2014.

[2] Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. Dynamic DFS in undirected graphs: breaking the $O(m)$ barrier. In *Proceedings of the twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 730–739. SIAM, 2016.

[3] Timothy M. Chan. Dynamic subgraph connectivity with geometric applications. In *Proceedings of the thiry-fourth annual ACM Symposium on Theory of Computing*, pages 7–13. ACM, 2002.

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

[5] Lijie Chen, Ran Duan, Ruosong Wang, and Hanrui Zhang. Improved algorithms for maintaining DFS tree in undirected graphs. *CoRR*, abs/1607.04913, 2016.

[6] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280, 1990. Computational algebraic complexity editorial.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[8] Ran Duan. New data structures for subgraph connectivity. In *Automata, Languages and Programming*, pages 201–212. Springer, 2010.
[9] Ran Duan and Le Zhang. Faster worst-case update time for dynamic subgraph connectivity. *CoRR*, abs/1611.09072, 2016.
[10] 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.
[11] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.
[12] Daniele Frigioni and Giuseppe F. Italiano. Dynamically switching vertices in planar graphs. *Algorithmica*, 28(1):76–103, 2000.
[13] 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.
[14] 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*, pages 21–30. ACM, 2015.
[15] 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.
[16] 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.
[17] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time. In *Proceedings of the twenty-eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2017.
[18] 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*, pages 1131–1142. SIAM, 2013.
[19] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Faster worst case deterministic dynamic connectivity. In *Proceedings of the twenty-fourth Annual European Symposium on Algorithms*, 2016.
[20] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: Adaptive, Las Vegas, and $O(n^{1/2-\epsilon})$-time. In *Proceedings of the forty-ninth Annual ACM on Symposium on Theory of Computing*, 2017.
[21] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.
[22] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM Symposium on Theory of Computing*, pages 343–350, 2000.
[23] Zhengyu Wang. An improved randomized data structure for dynamic graph connectivity. *CoRR*, abs/1510.04590, 2015.
[24] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1757–1769, 2013.
[25] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the forty-ninth Annual ACM on Symposium on Theory of Computing*, 2017.