Fully Dynamic Connectivity Oracles under General Vertex Updates

Kengo Nakamura

Graduate School of Information Science and Technology, The University of Tokyo, Tokyo, Japan
kengo_nakamura@mist.i.u-tokyo.ac.jp

Abstract

We study the following dynamic graph problem: given an undirected graph G, we maintain a connectivity oracle between any two vertices in G under any on-line sequence of vertex deletions and insertions with incident edges. We propose two algorithms for this problem: an amortized update time deterministic one and a worst case update time Monte Carlo one. Both of them allow an arbitrary number of new vertices to insert. The update time complexity of the former algorithm is no worse than the existing algorithms, which allow only limited number of vertices to insert. Moreover, for relatively dense graphs, we can expect that the update time bound of the former algorithm meets a lower bound, and that of the latter algorithm can be seen as a substantial improvement of the existing result by introducing randomization.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases Dynamic Graph, Connectivity, Depth-First Search

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2017.59

1 Introduction

In this paper, we consider the *dynamic graph connectivity problem*. Given an undirected graph G, the goal for this problem is to build a data structure which can process an on-line sequence of *graph updates* and *queries*. Here for the query $C(u, v)$, the data structure should answer whether there is a path between two vertices u and v in G. There are some variants for this problem with respect to the kinds of operations allowed as the graph updates.

- *Dynamic subgraph connectivity* (DSGC): a binary status is associated with each vertex in G, and we can switch it between "on" and "off". The query is to answer whether there is a path between two vertices in the subgraph of G induced by the "on" vertices.
- *Fully dynamic graph connectivity under edge updates* (FGCE): we can delete an existing edge e from G and insert a new edge e' to G.
- *Fully dynamic graph connectivity under general vertex updates* (FGCV): we can delete an existing vertex w from G, and insert a new vertex v and its incident edges to G.

In this paper, we study the FGCV problem.

Among these three problems, the FGCV problem is the most general framework when we focus on vertex updates. First, an FGCV data structure allows us to insert new vertices, while a DSGC data structure does not. Second, the FGCV problem can be somewhat solved by an FGCE data structure as follows, but there is a limitation on the number of new vertices. In the preprocessing, we add some isolated vertices to G. Then when a new vertex insertion occurs, we select one of the isolated vertices and regard it as the new vertex. Since single vertex update amounts to $O(n)$ edge updates (insertions or deletions) for a graph with n vertices and m edges, the FGCE data structure can process vertex updates. However, in

Table 1 Comparison of the vertex update time for the (fully) dynamic graph connectivity. "A", "M", and "D" in the first column mean that the corresponding rows show the time complexity of amortized update time deterministic, worst case update time Monte Carlo, and worst case update time deterministic algorithms, respectively. The query time is all $O(\log n)$ or $o(\log n)$.

	FGCE *		FGCV	
A	$O(n\frac{\log^2 n}{\log\log n})$	[14]	$O(\sqrt{m}\log^{1.25} n + l\frac{\log^2 n}{\log\log n} + n)$	**X**
M	$O(n\log^5 n)$	[10]	$O(\sqrt{ml}\log^{2.75} n + n)$	**Y**
D	$O(n\sqrt{\frac{n(\log\log n)^2}{\log n}})$	[11]	$O(\sqrt{mn}\log^{1.25} n)$	[12]

* They are multiplied by n since we focus on vertex updates.

this approach we cannot insert an arbitrary number of new vertices. On the other hand, an FGCV data structure can solve the DSGC and FGCE problems (in FGCE setting, one edge update can be converted to two vertex updates).

The FGCE problem is well-studied for years, and various kinds of algorithms for this problem were developed even recently, e.g. an amortized update time deterministic one [14], a worst case update time Monte Carlo one [10], and a worst case update time deterministic one [11]. There were also these kinds of algorithms for the DSGC problem [6, 8, 7]. However, there exist almost no FGCV algorithms which allow us to insert an arbitrary number of new vertices. The only exception is the algorithm of Baswana et al. [2], which maintains a depth-first search (DFS) tree of undirected graphs. Their worst case deterministic update time is recently improved by Nakamura and Sadakane [12]. The comparison of the "vertex" update time of these algorithms is shown in Table 1. Here the update time of the FGCE algorithm is multiplied by n since single vertex update amounts to $O(n)$ edge updates. Note that the update time for the DSGC algorithms [6, 8, 7] is omitted since they have $O(m^\alpha \text{polylog}(n))$ query time with $\alpha \geq 1/5$. This is much slower than $O(\log n)$, which is the upper bound for the query time of the algorithms in Table 1.

1.1 Our Results

We develop two data structures for the FCGV problem, both of which allow us to insert an arbitrary number of new vertices. One is an amortized update time deterministic algorithm (algorithm X), and the other is a worst case update time Monte Carlo algorithm (algorithm Y). Both algorithms X and Y have a query time of $O(\log n)$. Their time bounds for single vertex update are shown in the right column of Table 1.

Our time bounds in Table 1 depend on l, that is, the number of leaves of a *DFS forest* (a spanning forest generated by DFS) of G at some point. Both algorithms X and Y internally rebuild a DFS forest of G periodically, and l is in fact the number of leaves of it. Since $l \leq n$, algorithm X can solve the FGCV problem no slower than using the FGCE data structure [14]. Indeed, we can expect $l \ll n$ for relatively dense graphs as described in Sect. 7.

For algorithm X, its update time complexity becomes $O(n)$ if $l = O(n/\log^2 n)$ (unless $m = \Omega(n^2/\log^{2.5} n)$), which is a firm lower bound since the size of input incident edges around the inserted vertex may become $\Theta(n)$. In addition to this, both algorithms X and Y permit G to have some edges initially, while the amortized update time FGCE algorithm [14] assumes G has no edges initially. In summary, the advantages of using algorithm X over using amortized update time FGCE data structure [14] directly is as follows.

- Algorithm X allows us to insert an arbitrary number of new vertices.
- Algorithm X permits G to have some edges initially.

- The update time complexity of algorithm X is no slower than using [14] directly. For relatively dense graphs it is expected to become $O(n)$ which is a firm lower bound.

For algorithm Y, if $l \ll n$, the time bound $O(\sqrt{ml} \cdot \mathrm{polylog}(n))$ can be seen as a considerable improvement of that of [12] by introducing randomization. Moreover, in Sect. 7 it is shown that under ER model [9], which is a popular model of random graphs, the time bound becomes $O(n \log^{3.25} n)$ with high probability, which is faster than using the Monte Carlo FGCE data structure [10] directly. Again note that algorithm Y allows us to insert an arbitrary number of vertices while the Monte Carlo FGCE data structure [10] does not.

Our work can be summarized as follows. Our algorithms use a *disjoint tree partitioning*, which is used in the dynamic DFS algorithm of Baswana et al. [2], and the FGCE data structures ([14] for algorithm X, [10] for algorithm Y). First, we develop an efficient method to maintain disjoint tree partitioning (Sect. 3.1), which reduces the update time when the number of incident edges around the new vertex is small. Second, we define some queries on the graph and show an efficient way to solve them (Sect. 4). We believe these queries are of independent interest. Third, we find a good property of the disjoint tree partitioning for the amortized time complexity analysis (Lemma 2), and develop an algorithm which fully adopts this property (Sect. 5). Lastly, we develop a method to convert the amortized update time algorithm to a worst case update time one (Sect. 6). Note that this kind of technique is also employed in various dynamic graph algorithms such as dynamic DFS [2, 12] and dynamic all-pairs shortest paths [1], but in our situation we need some additional considerations.

2 Preliminaries

Throughout this paper, n and m denote the numbers of vertices and edges of a graph, respectively. We use $\log(\cdot)$ as the base-2 logarithm; the natural logarithm is denoted by $\ln(\cdot)$. Note that they differ only by a constant factor, thus $\ln x = \Theta(\log x)$.

Given a spanning forest T of an undirected graph G each tree in which is a rooted tree, the parent vertex of a vertex v is denoted by $par(v)$. A subtree τ of T is said to be *hanging from* a path p if the root r of τ satisfies both $r \notin p$ and $par(r) \in p$. Two vertices x and y are said to have *ancestor-descendant relation* if $x = y$, x is an ancestor of y, or y is an ancestor of x. A path p in T is said to be an *ancestor-descendant path* if the endpoints of p have ancestor-descendant relation. A spanning forest T of G is a *DFS forest* of G iff each tree in T is a DFS tree of the corresponding connected component of G. The number of leaves of a DFS forest T is the sum of that of each tree in T.

The DFS tree satisfies the following property. Let G be a connected undirected graph and T be a rooted spanning tree of G. Then T is a DFS tree of G, iff every edge in G connects two vertices which have ancestor-descendant relation. We call this *DFS property*.

3 Disjoint Tree Partitioning

In this section, we refer to a *disjoint tree partitioning* [2], and develop an efficient way to maintain this partition. The disjoint tree partitioning is defined as follows.

▶ **Definition 1** ([2]). Given a DFS forest T of an undirected graph G and a set U of vertices, the forest $T - U$ obtained by deleting the vertices in U from T is considered. Then the *disjoint tree partitioning* of $T - U$ is a partition of $T - U$ into a set $\mathcal{P}$ of ancestor-descendant paths in T with $|\mathcal{P}| \leq |U|$ and a set $\mathcal{T}$ of subtrees of T.

From now we abbreviate disjoint tree partitioning as DTP.

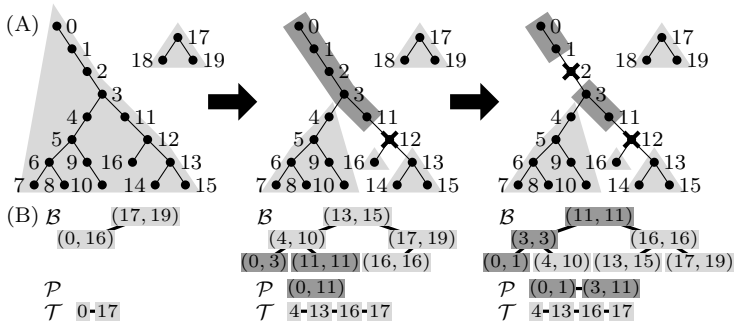

Figure 1 (A) An example of the building process of DTP. Subtrees $\tau \in \mathcal{T}$ are colored with light gray and ancestor-descendant paths $p \in \mathcal{P}$ with dark gray. (B) Corresponding binary trees $\mathcal{B}$ and lists $\mathcal{P}$ and $\mathcal{T}$. Here the pointers from $\mathcal{B}$ to $\mathcal{P}$ or $\mathcal{T}$ are omitted.

In [2] the way to construct DTP is also given. First, if $U = \emptyset$ then we set $\mathcal{P} = \emptyset$ and add all DFS trees in the DFS forest T to $\mathcal{T}$. Next, the DTP of $T - (U \cup \{v\})$ can be obtained by modifying the DTP of $T - U$ as follows. If $v \in \exists p \in \mathcal{P}$ we remove p from $\mathcal{P}$ and add (at most) two paths obtained by deleting v from p to $\mathcal{P}$. Otherwise if $v \in \exists \tau \in \mathcal{T}$, we remove τ from $\mathcal{T}$ and add a path p' from $par(v)$ to the root of τ to $\mathcal{P}$. Then we add all subtrees hanging from v or p' to $\mathcal{T}$. An example of this process is shown in Fig. 1(A). Note that since the number of paths in $\mathcal{P}$ is increased by at most one during each operation, $|\mathcal{P}| \leq |U|$ holds. This operation takes $O(n)$ time for each v, thus the DTP of $T - U$ can be calculated one by one in total $O(|U|n)$ time [2].

3.1 More Efficient Construction

Now we show a more efficient method of maintaining DTP we develop. First, if T is connected, a heavy-light (HL) decomposition [13] of T is calculated, and the order $\mathcal{L}$ of vertices is decided according to the pre-order traversal of T, such that for the first time a vertex v is visited, the next vertex to visit is one that is directly connected with a heavy edge derived from the HL decomposition. Then the vertices of T are numbered from 0 to $n-1$ according to $\mathcal{L}$; the vertex id of v is denoted by $f(v)$. If T is disconnected, we calculate the order of vertices for each DFS tree in the same way and vertices are numbered by consecutive integers from 0 to $n-1$. Note that this numbering originates in the dynamic DFS algorithm of Baswana et al. [2], but they utilize this in order to solve some other queries on G. An example of this numbering is shown in Fig. 1(A).

The important point is that the vertices of $\tau \in \mathcal{T}$ occupy single interval in $\mathcal{L}$ since $\mathcal{L}$ is a pre-order traversal of T, and those of $p \in \mathcal{P}$ occupy $O(\log n)$ intervals thanks to HL decomposition. Now we maintain these intervals by a balanced binary search tree $\mathcal{B}$. Here the key of each element is the lower endpoint of its interval. We can say $|\mathcal{B}| \leq n$ and $|\mathcal{B}| = O(|\mathcal{T}| + |\mathcal{P}| \log n)$. Besides this, $\mathcal{P}$ and $\mathcal{T}$ are retained by lists; each $p \in \mathcal{P}$ is expressed by a pair of its endpoints and each $\tau \in \mathcal{T}$ by its root. Here all vertices are stored as the vertex id $f(\cdot)$. Additionally, we add a pointer from each element in $\mathcal{B}$ to the corresponding $x \in \mathcal{P} \cup \mathcal{T}$. Examples of $\mathcal{B}$ and the lists are shown in Fig. 1(B).

Thanks to $\mathcal{B}$, we can efficiently update DTP when a vertex v is deleted. First, search $f(v)$ in $\mathcal{B}$ and detect $p \in \mathcal{P}$ or $\tau \in \mathcal{T}$ which contains v, which takes $O(\log |\mathcal{B}|) = O(\log n)$ time. Then if $v \in \exists p \in \mathcal{P}$, remove p from $\mathcal{P}$ and corresponding intervals from $\mathcal{B}$, and add at most two paths obtained by deleting v from p to $\mathcal{P}$ and corresponding intervals to $\mathcal{B}$. These processes take $O(\log^2 n)$ time, since they amount to $O(\log n)$ deletions and insertions

of elements on $\mathcal{B}$. If $v \in \exists\tau \in \mathcal{T}$, first remove τ from $\mathcal{T}$ and a corresponding interval from $\mathcal{B}$. Then while traversing a path p' from $par(v)$ to the root of τ, add subtrees hanging from v or p' to $\mathcal{T}$ and corresponding intervals to $\mathcal{B}$. Finally, add p' to $\mathcal{P}$ and corresponding $O(\log n)$ intervals to $\mathcal{B}$. These take $O(\log^2 n + |p'| + \delta \log n)$ time, where δ is the number of hanging subtrees. In fact, we can limit the sum of $|p'|$ and δ as follows.

▶ **Lemma 2.** *Given a graph G, its DFS forest T and a set of vertices $U = \{v_1, \ldots, v_{|U|}\}$, suppose the DTP of $T - \{v_1, \ldots, v_i\}$ $(i = 1, \ldots, |U|)$ is calculated one by one by the above process. In calculating the DTP of $T - \{v_1, \ldots, v_i\}$ by modifying that of $T - \{v_1, \ldots, v_{i-1}\}$, if $v_i \in \exists\tau \in \mathcal{T}$, let p'_i be the traversed path (i.e. the path from $par(v_i)$ to the root of τ) and δ_i be the number of hanging subtrees from p'_i or v_i. (if $v_i \in \exists p \in \mathcal{P}$ set $p'_i = \emptyset$ and $\delta_i = 0$). Then $\sum_{i=1}^{|U|} |p'_i| \leq n$ and $\sum_{i=1}^{|U|} \delta_i \leq l + |U|$ hold, where l is the number of leaves of T.*

Proof. For any vertex v, once v is contained in some p'_i, v is always contained in one of the paths in $\mathcal{P}$ until deleted. This means that v cannot be contained in more than one of p'_i. Then $\sum_{i=1}^{|U|} |p'_i| \leq n$ holds. Next, it can be pointed out that $|\mathcal{T}|$ cannot be more than l at any time, since every $\tau \in \mathcal{T}$ has at least one distinct leaf of T. In the process of calculating DTP, $|\mathcal{T}|$ increases by 0 if $v_i \in \exists p \in \mathcal{P}$ or $\delta_i - 1$ if $v_i \in \exists\tau \in \mathcal{T}$. Therefore $\sum_{i=1}^{|U|}(\delta_i - 1) \leq l$. ◀

Note that in the preprocessing, the HL decomposition can be calculated in $O(n)$ time and the initialization of DTP can be done in $O(t)$ time, where $t \leq l$ is the number of connected components of T. Hence we can immediately obtain the following result from Lemma 2.

▶ **Lemma 3.** *By the process described above, the DTP of $T - U$ can be calculated one by one in total $O(|U| \log^2 n + n + l \log n)$ time.*

4 Queries on the Disjoint Tree Partitioning

In this section, we define some queries related to the DTP and show an efficient solution for them. We consider the following queries Q and Q'.

▶ **Definition 4.** An undirected graph G and its DFS forest T are given. Then for any subtree τ of T and ancestor-descendant path p in T, $Q(\tau, p)$ returns one of the edges in G which directly connect τ and p if exists, or $\emptyset$ otherwise. Here we assume τ and p have no common vertices. Similarly, for any two disjoint ancestor-descendant paths p_1, p_2 in T, $Q'(p_1, p_2)$ returns one of the edges in G which directly connect p_1 and p_2 if exists, or $\emptyset$ otherwise.

The motivation to consider these queries is as follows. Roughly speaking, our algorithms proposed later treat paths $p \in \mathcal{P}$ and subtrees $\tau \in \mathcal{T}$ derived from the DTP of $T - U$ as virtual vertices and maintain a data structure to answer connectivity queries among them. Therefore for any distinct $\tau_1, \tau_2 \in \mathcal{T}$ and $p_1, p_2 \in \mathcal{P}$ it is important to judge quickly whether there are some edges in $G - U$ between p_1 and p_2 or between p_1 and τ_1. Here it is noted that thanks to DFS property, there are no edges in $G - U$ between τ_1 and τ_2.

Indeed, the query very similar to $Q(\tau, p)$ is utilized in the dynamic DFS algorithm by Baswana et al. [2], and is revealed to be efficiently solved with the vertex numbering described in Sect. 3 and the *orthogonal range search problem* [2, 12].

▶ **Definition 5.** On grid points in a 2-dimensional plane, k points are given. Then for any rectangular region $R = [x_1, x_2] \times [y_1, y_2]$, the *orthogonal range one reporting query* returns one of the points within R if exists, or $\emptyset$ otherwise. We abbreviate it as ORR query.

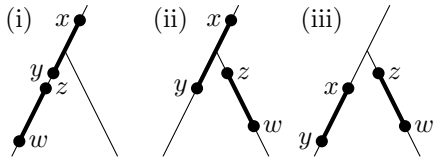

Figure 2 The possible configurations of two ancestor-descendant paths in T.

The queries $Q(\tau, p)$ and $Q'(p_1, p_2)$ can be converted to the ORR query in the following way. First, the vertices of T are numbered from 0 to $n-1$ in the same way as Sect. 3. Second, we consider a grid $\mathcal{G}$ and, for each edge (v, w) of G, put two points on the coordinates $(f(v), f(w))$ and $(f(w), f(v))$ in $\mathcal{G}$. This is equivalent to consider the adjacency matrix of G, therefore $2m$ points are placed. Now a careful case analysis shows the following results. Since Lemma 6 is almost the same as what is proved in [12], we only show the proof of Lemma 7.

▶ **Lemma 6** ([12]). *For any subtree τ of T and ancestor-descendant path p in T, the query $Q(\tau, p)$ can be answered by solving single ORR query on $\mathcal{G}$.*

▶ **Lemma 7.** *For any ancestor-descendant paths p_1, p_2 in T, the query $Q'(p_1, p_2)$ can be answered by solving $O(\log n)$ ORR queries on $\mathcal{G}$.*

Proof. Let x, y be the endpoints of p_1 with $f(x) \leq f(y)$ and z, w be those of p_2 with $f(z) \leq f(w)$. W.l.o.g. we can assume $f(x) < f(z)$. Due to the HL decomposition, the vertices of p_2 occupy $O(\log n)$ intervals $[a_1, b_1], \ldots, [a_k, b_k]$ in the vertex id. Now we assume that p_1 and p_2 are in the same connected component in G. Then there are three patterns on the configuration of p_1 and p_2 as drawn in Fig. 2, and two patterns on the vertex id: (a) $f(x) \leq f(y) < f(z) \leq f(w)$ and (b) $f(x) < f(z) \leq f(w) < f(y)$.

When (a) holds, the answer for $Q'(p_1, p_2)$ can be obtained by solving ORR queries on $\mathcal{G}$ with $R = [f(x), f(y)] \times [a_i, b_i]$ for $i = 1, \ldots, k$ and combining these results. The inequality (a) can appear in all configurations in Fig. 2. In (i), it may be that $[f(x), f(y)]$ contains some branches forking from p_1, but it makes no problem since there are no edges between these branches and p_2 thanks to DFS property. The same argument can be applied to (ii). In (iii), the answer for $Q'(p_1, p_2)$ is $\emptyset$ due to DFS property, and each ORR query also returns $\emptyset$. Note that even if p_1 and p_2 are in different connected components in G, (a) holds and the above procedure returns $\emptyset$ correctly. When (b) holds, the answer can be obtained in a similar way except that the rectangles are $R = [f(x), f(\mathrm{LCA}(y, z))] \times [a_i, b_i]$, where $\mathrm{LCA}(y, z)$ is the lowest common ancestor of y and z in T. The inequality (b) can appear in only (ii). In (ii), all edges between p_1 and p_2 are indeed between the path from x to $\mathrm{LCA}(y, z)$ and p_2, and again it does not matter $[f(x), f(\mathrm{LCA}(y, z))]$ contains some branches forking from p_1. Note that the LCA query can be solved in $O(1)$ time with a data structure constructed in $O(n)$ time [5]. This construction time is absorbed in the cost of HL decomposition. ◀

Recently, Belazzougui and Puglisi [4] proved that an ORR query with k points in a rank space can be solved in $O(\log^{\varepsilon} k)$ time with a data structure of $O(k)$ space constructed in $O(k\sqrt{\log k})$ time. With a standard conversion between a rank space and a general grid via bit vectors (see e.g. [12]), we can apply it to Lemma 6 and 7, and obtain the following lemma.

▶ **Lemma 8.** *The queries $Q(\tau, p)$ and $Q'(p_1, p_2)$ can be solved in $O(\log^{\varepsilon} n)$ time and $O(\log^{1+\varepsilon} n)$ time for arbitrary $0 < \varepsilon < 1$, respectively, with a data structure of $O(m)$ space constructed in $O(m\sqrt{\log n})$ time.*

5 Amortized Update Time Algorithm

In this section, we show an amortized update time FGCV algorithm. First we give an overview of our algorithm. As described in Sect. 4, paths $p \in \mathcal{P}$ and subtrees $\tau \in \mathcal{T}$ derived from the DTP of $T - U$ are treated as virtual vertices. If we deal with only deletion of vertices, all we have to do is maintain a connectivity oracle (i.e. a data structure to answer connectivity queries) among them. However, in the fully dynamic setting we deal with insertion of vertices and their incident edges. Thus we also treat each inserted vertex as a virtual vertex, i.e. we maintain a connectivity oracle among $\mathcal{P} \cup \mathcal{T} \cup \mathcal{V}$, where $\mathcal{V}$ is a set of inserted vertices. Then our amortized update time algorithm can be described as follows. First, perform DFS on G to build a DFS forest T and initialize the DTP of T, an FGCE data structure (a connectivity oracle) $\mathcal{C}$, and the data structure of Lemma 8 to solve Q and Q'. Second, for the first $\Delta(\leq n)$ vertex updates, if vertex insertion occurs then insert the new vertex to $\mathcal{V}$ and update $\mathcal{C}$, otherwise if vertex deletion occurs then update $\mathcal{V}$ (if the deleted vertex is in $\mathcal{V}$) or the DTP (otherwise) and also update $\mathcal{C}$. Third, when Δ vertex updates are processed, again perform DFS on G to rebuild the DFS forest T and reinitialize the DTP, the connectivity oracle, and the data structure in Lemma 8, which is used for the next Δ updates. In summary, we initialize data structures periodically after every Δ updates.

We proceed to the detailed description of the initialization. In the initialization, we have to construct the FGCE data structure $\mathcal{C}$. From now we call the vertex in $\mathcal{C}$ *node* to avoid confusion. Since in the edge update setting we cannot change the number of nodes, we must decide at first the number of nodes $\mathcal{C}$ has. Because $\mathcal{C}$ maintains connectivity among $\mathcal{P} \cup \mathcal{T} \cup \mathcal{V}$, it suffices to prepare M nodes for $\mathcal{C}$ where M is an upper bound of $|\mathcal{P} \cup \mathcal{T} \cup \mathcal{V}|$ while Δ updates are being processed. The following lemma ensures us that $l + \Delta$ nodes are sufficient. Note that $\mathcal{C}$ is initialized to have no edges between any nodes since at first $\mathcal{P} = \mathcal{V} = \emptyset$ and each $\tau \in \mathcal{T}$ is indeed a connected component of G.

▶ **Lemma 9.** *While Δ updates are being processed from the initialization, $|\mathcal{P} \cup \mathcal{T} \cup \mathcal{V}|$ is not more than $l + \Delta$ at any time, where l is the number of leaves of T.*

Proof. It suffices to show that $|\mathcal{P} \cup \mathcal{T} \cup \mathcal{V}|$ is not more than $l + \Delta$ "when" Δ updates are processed. Let Δ_D and Δ_I be the numbers of deleted and inserted vertices during Δ updates, respectively. First, it is already shown that $|\mathcal{T}| \leq l$ at any time. Second, from the definition of DTP, $|\mathcal{P}| \leq \Delta_D$. Finally, $|\mathcal{V}| \leq \Delta_I$ holds trivially (the case $|\mathcal{V}| < \Delta_I$ occurs when some inserted vertices are deleted). Then $|\mathcal{P} \cup \mathcal{T} \cup \mathcal{V}| \leq l + \Delta_D + \Delta_I = l + \Delta$. ◀

Since each element in $\mathcal{P} \cup \mathcal{T} \cup \mathcal{V}$ can be deleted, we may have to reuse the nodes of deleted elements when new elements are created. This can be addressed by numbering the nodes in $\mathcal{C}$, storing the corresponding node id for each $p \in \mathcal{P}$, $\tau \in \mathcal{T}$ and $v \in \mathcal{V}$, and maintaining the unused nodes by list. We assume this node recycling runs in background, and for simplicity, the node in $\mathcal{C}$ representing $x \in \mathcal{P} \cup \mathcal{T} \cup \mathcal{V}$ is denoted by $C(x)$.

We next describe the update procedure. Let n be the number of vertices of G at the initialization. Since the initial vertices are numbered from 0 to $n - 1$, the newly inserted vertices, i.e. the vertices in $\mathcal{V}$, are numbered one by one from n.

First we describe how to update the data structures when vertex insertion occurs. At this time the list $\mathcal{A}$ of vertices the newly inserted vertex v is incident with is given. Let $k(\geq n)$ be the vertex id of v. First, convert each element of $\mathcal{A}$ to the vertex id $f(\cdot)$ and store in an array $A[k][\cdot]$ sorted in ascending order. A is an adjacency list for the newly inserted vertices. If $\mathcal{A}$ has a vertex $u \in \mathcal{V}$, append k to $A[f(u)]$, and connect $C(v)$ with $C(u)$. Next, for each $x \in \mathcal{P} \cup \mathcal{T}$ judge whether v is incident with x and connect $C(v)$ with $C(x)$ if so.

These judgments can be performed by traversing $\mathcal{B}$ while scanning $A[k]$ from left to right, since $A[k]$ is sorted. For each interval $[a, b]$ in $\mathcal{B}$, if $[a, b]$ contains some elements in $A[k]$ then we mark the corresponding $x \in \mathcal{P} \cup \mathcal{T}$ incident with v. If $y \in \mathcal{P} \cup \mathcal{T}$ is not marked when the traversal of $\mathcal{B}$ finishes, y is not incident with v.

Next we describe what to do when the deletion of a vertex w occurs. Let $k = f(w)$. The vertex u with $f(u) = x$ is denoted by $f^{-1}(x)$. First, if $w \in \mathcal{V}$, i.e. $k \geq n$, then undo what is performed when inserting w. More specifically, first disconnect $C(f^{-1}(A[k][i]))$ from $C(w)$ for each i such that $A[k][i] \geq n$, then judge whether w is incident with each $x \in \mathcal{T} \cup \mathcal{P}$ and disconnect $C(x)$ from $C(w)$ if so. This judgment can be done in the same way as described above. Next, if $w \notin \mathcal{V}$, detect $y \in \mathcal{P} \cup \mathcal{T}$ which contains w by a search on $\mathcal{B}$, and then update DTP and $\mathcal{C}$ simultaneously as described later. When the DTP is updated as in Sect. 3, some of the following four operations may occur: adding a path p to $\mathcal{P}$, removing p from $\mathcal{P}$, adding a subtree τ to $\mathcal{T}$, and removing τ from $\mathcal{T}$.

Now we focus on a path p, and show how to judge whether $x \in \mathcal{P} \cup \mathcal{T} \cup \mathcal{V} \setminus \{p\}$ is incident with p, i.e. there are some edges between x and p. For each $u \in \mathcal{V}$ we can judge it by the following way. Let $[a_1, b_1], \ldots, [a_k, b_k]$ be the intervals the vertices of p occupy in the vertex id, and $\mathrm{lb}(A[i], j)$ be the smallest element in $A[i]$ which is not less than j, which can be obtained by a binary search on $A[i]$. Then p is incident with u iff there exists i such that $\mathrm{lb}(A[f(u)], a_i) \leq b_i$. For each $p' \in \mathcal{P}$ and $\tau' \in \mathcal{T}$ we can judge the incidence by the queries $Q'(p', p)$ and $Q(\tau', p)$, respectively. Using these judging frameworks, we can update $\mathcal{C}$ when the addition or removal of p occurs: for each $x \in \mathcal{P} \cup \mathcal{T} \cup \mathcal{V}$ incident with p, disconnect $C(x)$ from $C(p)$ when p is removed from $\mathcal{P}$, or connect $C(x)$ with $C(p)$ when p is added to $\mathcal{P}$.

We can cope with the addition or removal of a subtree τ in a similar way. Let $[a, b]$ be the interval the vertices of τ occupy in the vertex id. Then τ is incident with $u \in \mathcal{V}$ iff $\mathrm{lb}(A[f(u)], a) \leq b$. For each $p' \in \mathcal{P}$, we can judge whether p' is incident with τ by the query $Q(\tau, p')$. Again note that τ is not incident with any $\tau' \in \mathcal{T} \setminus \{\tau\}$ due to DFS property. The update procedure for $\mathcal{C}$ is the same: for each $x \in \mathcal{P} \cup \mathcal{V}$ incident with τ, disconnect $C(x)$ from $C(\tau)$ when τ is removed from $\mathcal{T}$, or connect $C(x)$ with $C(\tau)$ when τ is added to $\mathcal{T}$.

Finally we show how to answer the connectivity query between v and w. First we detect $x \in \mathcal{P} \cup \mathcal{T} \cup \mathcal{V}$ containing v. Even if $v \notin \mathcal{V}$ we can determine $x \in \mathcal{P} \cup \mathcal{T}$ by searching the interval which contains $f(v)$ on $\mathcal{B}$. The same argument can be applied to the conversion from w to $y \in \mathcal{P} \cup \mathcal{T} \cup \mathcal{V}$. If $x = y$ then v and w are obviously connected, otherwise the answer can be obtained by querying on $\mathcal{C}$ whether $C(x)$ and $C(y)$ are connected.

5.1 Time Complexity Analysis

We proceed to the time complexity analysis of this algorithm. In our analysis, we use the FGCE data structure proposed by Wulff-Nilsen [14] as $\mathcal{C}$, which has $O(\log^2 k / \log\log k)$ amortized update time and $O(\log k / \log\log k)$ query time for a graph with k nodes. Since $k = l + \Delta \leq n + n$ as in Lemma 9, these are bounded by $O(T_u)$ amortized time for update and $O(T_q)$ time for query, with $T_u = \log^2 n / \log\log n$ and $T_q = \log n / \log\log n$. First of all, the query time of our algorithm is $O(\log n)$, since a search on $\mathcal{B}$ takes $O(\log |\mathcal{B}|) = O(\log n)$ time and a query on $\mathcal{C}$ takes $O(T_q) = o(\log n)$ time. From now the update time is considered.

First, we consider the time consumed by the (periodic) initialization, which is amortized over Δ updates. The data structure in Lemma 8 can be built in $O(m\sqrt{\log n})$ time and $\mathcal{C}$ in $O(k \log k) = O((l+\Delta) \log n)$ time (though the initialization cost of $\mathcal{C}$ is not explicitly described in [14], we prove that for a graph with k nodes and no edges $\mathcal{C}$ can be initialized in $O(k \log k)$ time). From Lemma 3, the total time of maintaining DTP is $O(\Delta \log^2 n + n + l \log n)$.

Next, we focus on the vertex insertion. Let m_v be the number of incident vertices of the newly inserted vertex v, i.e. the number of newly inserted edges. Sorting these incident vertices takes $O(m_v \log m_v) = O(m_v \log n)$ time, or $O(n)$ time by bucket sort. Traversing $\mathcal{B}$ while scanning $A[k]$ takes $O(m_v + |\mathcal{B}|) = O(m_v + |\mathcal{T}| + |\mathcal{P}| \log n)$ time. Connecting $C(v)$ with some $C(x)$ occurs at most $|\mathcal{P} \cup \mathcal{T} \cup \mathcal{V}|$ times, thus updating $\mathcal{C}$ takes at most $O((|\mathcal{P}| + |\mathcal{T}| + |\mathcal{V}|)T_u)$ time.

Finally we consider the deletion of a vertex w. There are three cases, namely, $w \in \mathcal{V}$, $w \in \exists p \in \mathcal{P}$ and $w \in \exists \tau \in \mathcal{T}$. If $w \in \mathcal{V}$, the time complexity is almost the same as that of vertex insertion, except that sorting incident vertices is not needed. If not, we can detect $y \in \mathcal{P} \cup \mathcal{T}$ which contains v in $O(\log |\mathcal{B}|) = O(\log n)$ time. Then if $y = p \in \mathcal{P}$, one path is removed from $\mathcal{P}$ and at most two paths are added to $\mathcal{P}$. For each removal or addition of a path, judging the incidence takes $O(\log^2 n)$ time for each $u \in \mathcal{V}$ (since this amounts to at most $O(\log n)$ binary searches on $A[f(u)]$), $O(\log^{1+\varepsilon} n)$ time for each $p' \in \mathcal{P}$, and $O(\log^{\varepsilon} n)$ time for each $\tau' \in \mathcal{T}$ (Lemma 8). Updating $\mathcal{C}$ takes at most $O((|\mathcal{P}| + |\mathcal{T}| + |\mathcal{V}|)T_u)$ time. Then the total cost is bounded by $O((|\mathcal{P}| + |\mathcal{T}|)T_u + |\mathcal{V}| \log^2 n)$ time. The most complicated case is $y = \tau \in \mathcal{T}$. In this case, one subtree is removed from $\mathcal{T}$, one path is added to $\mathcal{P}$, and δ_w subtrees are added to $\mathcal{T}$, where δ_w is the number of hanging subtrees as described in Sect. 3. Here judging the incidence between $\tau \in \mathcal{T}$ and each $u \in \mathcal{V}$ takes $O(\log n)$ time, since this amounts to one binary search on $A[f(u)]$. Then a similar analysis shows that the total cost is bounded by $O((|\mathcal{P}| + |\mathcal{T}|)T_u + |\mathcal{V}| \log^2 n + \delta_w(|\mathcal{P}| + |\mathcal{V}|)T_u)$.

Now we sum up all of the costs described above. The most crucial point is that we can amortize the sum of δ_w over Δ updates by Lemma 2: $\sum_w \delta_w \leq l + \Delta$. Since $|\mathcal{P}| \leq \Delta$, $|\mathcal{T}| \leq l$ and $|\mathcal{V}| \leq \Delta$, the amortized cost for the update procedure (other than initialization) over Δ updates is bounded by $O(lT_u + \Delta \log^2 n + \min\{\overline{m} \log n, n\})$ per update, where $\overline{m} = \sum_v m_v / \Delta$ is the average number of newly inserted edges per update. The overall amortized update time is obtained by adding the initialization cost divided by Δ to it. Some terms are absorbed in lT_u and $\Delta \log^2 n$ terms and we obtain the following bound: $O((n + m\sqrt{\log n})/\Delta + \Delta \log^2 n + lT_u + \min\{\overline{m} \log n, n\})$. By taking $\Delta = \lceil \sqrt{m} / \log^{0.75} n \rceil$, this bound becomes $O(\sqrt{m} \log^{1.25} n + l \log^2 n / \log \log n + n)$. If $m = \Omega(n/\sqrt{\log n})$, the n/Δ term is absorbed in $m\sqrt{\log n}/\Delta$ term and the last n term becomes $\min\{\overline{m} \log n, n\}$.

▶ **Theorem 10.** *There exists a deterministic fully dynamic connectivity algorithm under general vertex updates such that each update can be processed in amortized* $O(\sqrt{m} \log^{1.25} n + l \log^2 n / \log \log n + n)$ *time and each query in* $O(\log n)$ *time, where* l *is the number of leaves of a DFS forest of* G *at some point. If* $m = \Omega(n/\sqrt{\log n})$*, the amortized update time complexity is reduced to* $O(\sqrt{m} \log^{1.25} n + l \log^2 n / \log \log n + \min\{\overline{m} \log n, n\})$*, where* $\overline{m}$ *is the average number of newly inserted edges per update.*

6 Worst Case Update Time Algorithm

In this section, we show a worst case update time FGCV algorithm. In our worst case update time algorithm, the procedure for processing graph updates and queries is kept same as the amortized update time algorithm in Sect. 5. We alter the periodic initialization. The principles to achieve "worst case" update time are as follows: (i) to perform simultaneously the processing of graph updates and queries and the initialization of data structures, and (ii) to utilize the data structures built from a DFS forest with "less" number of leaves. The idea (i) is used for various worst case update time dynamic graph algorithms such as dynamic DFS [2, 12] and dynamic all-pairs shortest paths [1]. The idea (ii) is due to the observation that in the amortized update time algorithm, smaller l leads to a better update time bound.

The overview of our algorithm is described as follows. Let $a > 1$ be a positive constant and $\Delta_0, \Delta_1, \ldots$ be positive integers (their values are decided later). We virtually divide the sequence of graph updates into *phases*; the first Δ_0 updates are called phase 0, the next Δ_1 updates are called phase 1, and similarly from $(\Delta_0 + \cdots + \Delta_{j-1} + 1)$-st to $(\Delta_0 + \cdots + \Delta_j)$-th updates are called phase j. Let G_j be the graph at the end of phase $(j-1)$. First, given an original undirected graph G, calculate a DFS forest T_0 of G to get the number of leaves l_0 of T_0, initialize the data structure $\mathcal{D}_0$ for G and T_0, and use $\mathcal{D}_0$ for processing updates and queries in phase 0 and 1. Here $\mathcal{D}_0$ is indeed a collection of the DTP of T_0, the FGCE data structure $\mathcal{C}$, and the data structure to solve Q and Q'. Besides this, in phase $j \geq 1$, in the first half (i.e. first $\Delta_j/2$ updates) the followings are performed gradually: calculate a DFS forest T_j of G_j to get the number of leaves l_j of T_j and initialize the data structure $\mathcal{D}_j$ for G_j and T_j. Then if $l_j > al_{j-1}$ (i.e. the number of leaves of T_j is too much), do nothing other than processing updates and queries in the second half, and in the next phase $(j+1)$ the data structure used in phase j is consecutively utilized for processing updates and queries. If $l_j \leq al_{j-1}$, in the second half apply the Δ_j vertex updates (from $(\Delta_0 + \cdots + \Delta_{j-1} + 1)$-st to $(\Delta_0 + \cdots + \Delta_j)$-th) on $\mathcal{D}_j$ gradually. This can be done by applying two updates on $\mathcal{D}_j$ during each graph update. In this way $\mathcal{D}_j$ is ready to use for processing updates and queries in the end of phase j, and $\mathcal{D}_j$ is utilized in the next phase $(j+1)$.

6.1 Probability and Time Complexity Analysis

Now we consider the probability of correctness and the update time complexity. Here we use the Monte Carlo FGCE data structure proposed by Kapron et al. [10] as $\mathcal{C}$, which has $O(\log^5 k)$ worst case update time and $O(\log k / \log\log k)$ query time for a graph with k nodes. Their algorithm has only one-sided error: if their algorithm answers "yes" for the query, the answer is always correct, otherwise the answer is correct with probability at least $1 - k^{-c}$ for any fixed constant c. If the data structures are used for processing updates and queries in the same way as Sect. 5, the most time consuming case occurs when a vertex w with $w \in \exists\tau \in \mathcal{T}$ is deleted, which causes $\delta_w \leq l$ subtrees to be added to $\mathcal{T}$. When Δ updates are already processed, $|\mathcal{P}| \leq \Delta$ and $|\mathcal{V}| \leq \Delta$. Therefore the worst case cost for single update in phase j is bounded by $O(l\Delta \log^5 n + n)$ when Δ updates are processed, where l is the number of leaves of a DFS forest the data structures used in phase j are built from. Note that the $O(n)$ term derived from the vertex insertion is also not negligible.

First we consider the probability of correctness. The connectivity oracle by Kapron et al. [10] maintains a spanning forest of the graph internally. Indeed, their oracle guarantees that this spanning forest is maintained correctly with probability at least $1 - k^{-c}$. This means that in our algorithm the spanning forest of a graph with vertex set $\mathcal{P} \cup \mathcal{T} \cup \mathcal{V}$ is maintained correctly in $\mathcal{C}$ with probability at least $1 - k^{-c}$. Later we set $k \geq \sqrt{n}$, then our algorithm answers the query correctly with probability at least $1 - n^{-c/2}$.

Next we consider the time complexity. In the analysis, we assume the number of edges of G is not drastically changed during each phase for simplicity. In other words, let m_j be the number of edges of G_j, then we assume $c_l m_{j-1} \leq m_j \leq c_u m_{j-1}$ for all $j \geq 1$ with some fixed constants c_l and c_u (note that this assumption is also implicitly imposed on the analysis of the dynamic DFS algorithms [2, 12]). We set $a = 4$, $\Delta_0 = \lfloor \sqrt{m/l_0} / \log^{2.25} n \rfloor$ and $\Delta_j = \lfloor \sqrt{m/l_{j-1}} / \log^{2.25} n \rfloor$ $(j \geq 1)$.

In phase $j \geq 1$, performing DFS and initializing the data structure to solve Q and Q' takes $O((m\sqrt{\log n} + n)/(\Delta_j/2)) = O(\sqrt{ml_{j-1}} \log^{2.75} n + n)$ time per update (similar argument can be applied to phase 0). If $l_j \leq 4l_{j-1}$, in the second half of phase j applying Δ_j updates on $\mathcal{D}_j$ takes $O(l_j \Delta_j \log^5 n + n) = O(\sqrt{ml_{j-1}} \log^{2.75} n + n)$ time per update.

The most important point is how many updates each data structure $\mathcal{D}_t$ is processed. It seems to be difficult to analyze it because even if m does not change drastically during each phase, l may change drastically. However, we can obtain the following lemma.

▶ **Lemma 11.** *During our algorithm, $\mathcal{D}_t$ processes at most $O(\sqrt{m/l_t}/\log^{2.25} n)$ updates.*

Proof. Suppose $\mathcal{D}_t$ $(t \geq 1)$ is used for processing updates and queries from phase $(t+1)$ to $(t+k)$. Then $\mathcal{D}_t$ processes $\Delta_t + \cdots + \Delta_{t+k}$ updates overall. Due to the assumption of the periodic initialization described above, we can say $al_{t-1} \geq l_t < l_{t+1}/a < \cdots < l_{t+k-1}/a^{k-1}$. Therefore $\Delta_{t+i} \leq (\sqrt{m/l_t}/\log^{2.25} n)/2^{i-1}$ $(i = 0, \ldots, k)$ (with $a = 4$). Since the sum of geometric series converges to a constant, $\Delta_t + \cdots + \Delta_{t+k} \leq (\sqrt{m/l_t}/\log^{2.25} n) \cdot (2 + 1 + \cdots + 1/2^{k-1}) = O(\sqrt{m/l_t}/\log^{2.25} n)$. Similar arguments can be applied to $\mathcal{D}_0$. ◀

Then the worst case cost of processing single update with $\mathcal{D}_t$ is bounded by $O(l_t \log^5 n \cdot \sqrt{m/l_t}/\log^{2.25} n + n) = O(\sqrt{ml_t}\log^{2.75} n + n)$. If $\mathcal{D}_t$ $(t \geq 1)$ is used for processing updates and queries in phase $j > t$, we can say $l_t < l_{j-1}/a^{j-1-t}$, so the bound can be written as $O(\sqrt{ml_{j-1}}\log^{2.75} n + n)$. Similar arguments hold for the cases $t = 0$ and $t = j$.

We do not care the cost of $\mathcal{C}$'s initialization, but this does not cause trouble. From the choice of Δ_j, $k = \lceil \max\{l_j + 2\sqrt{m}/\log^{2.25} n, \sqrt{n}\} \rceil$ is enough for $\mathcal{D}_j$. The initialization cost of $\mathcal{C}$ is $O(k \log^4 n)$ [10] since $\mathcal{C}$ is initialized to have no edges as in Sect. 5. Here $l_j \log^4 n/\Delta_j$, $\sqrt{m}\log^{1.75} n/\Delta_j$ and $\sqrt{n}\log^4 n/\Delta_j$ are absorbed in $l_j\Delta_j \log^5 n$, $m\sqrt{\log n}/\Delta_j$ and n/Δ_j, respectively. Similarly, the initialization cost of DTP is also negligible. Overall, it can be said that the worst case update time complexity is $O(\sqrt{ml_0}\log^{2.75} n + n)$ in phase 0 and $O(\sqrt{ml_{j-1}}\log^{2.75} n + n)$ in phase j. Now we obtain the following theorem.

▶ **Theorem 12.** *There exists a Monte Carlo fully dynamic connectivity algorithm under general vertex updates such that each update can be processed in worst case $O(\sqrt{ml}\log^{2.75} n + n)$ time and each query in $O(\log n)$ time, where l is the number of leaves of a DFS forest of G at some point. If this algorithm answers "yes" for the query, the answer is always correct, otherwise correct with probability at least $1 - n^{-c}$ for any fixed constant c.*

7 The Number of Leaves of DFS Forest

In this section, we focus on the value of l, that is, the number of leaves of the DFS forest. Now we state that for relatively dense random graphs $l = o(n)$ holds with high probability.

Here we consider the ER model [9] $G(n, p)$. In a random graph $G(n, p)$ which is an undirected graph with n vertices, for every pair (v, w) of vertices an edge between v and w is added to the graph with probability p independent of other pairs. The average number of edges is $\overline{M} = Np$ with $N = \binom{n}{2} \leq n^2/2$. Recently, Baswana et al. [3] proved the following result, which is about the property of DFS on $G(n, p)$.

▶ **Lemma 13** ([3]). *Given a random graph $G(n, p)$ with $p = (\ln n_0 + c)/n_0$ for any integers $n_0 \leq n$ and $c \geq 1$, the DFS on $G(n, p)$ proceeds without moving backward for the first $n - n_0$ vertices with probability at least $1 - 2/e^c$.*

The DFS on a graph can be seen as a sequence of *moving forward* and *moving backward*; if there exist unvisited adjacent vertices then it moves forward, otherwise it moves backward. This lemma implies that with high probability the number of leaves of the DFS forest of $G(n, p)$ is less than n_0 since the first $n - n_0$ vertices are all non-leaf vertices in the DFS forest. The proof of Lemma 13 in [3] is very simple: the probability that the DFS on $G(n, p)$ proceeds without moving backward for the first $n - n_0$ vertices is $\prod_{j=1}^{n-n_0}\{1 - (1-p)^{n-j}\}$, and this probability is lower bounded by $1 - 2/e^c$ using some elementary inequalities.

In Lemma 13, if we set $\overline{M} = n \ln^{1+\alpha} n/2$ then $n_0 \leq n/\ln^{\alpha} n$ and $\overline{M} = n^{1+\varepsilon}/2$ then $n_0 \leq n^{1-\varepsilon} \ln n$. If $l = O(n/\log^2 n)$, the amortized update time complexity in Theorem 10 becomes $O(n)$ (unless $m = \Omega(n^2/\log^{2.5} n)$), which is a firm lower bound. Simple calculations show that $l \leq n_0 = O(n/\log^2 n)$ is achieved with high probability when $\overline{M} = \Omega(n \log^3 n)$ or $\Omega(n^{1+\varepsilon})$. Moreover, if we set $\overline{M} = gn$ with $g \geq 1$ then $n_0 \leq n \ln n/2g$ in Lemma 13. This means that under ER model, $\overline{M}l \leq \overline{M}n_0 \leq n^2 \ln n/2$ holds with high probability. Therefore the worst case update time complexity in Theorem 12 becomes $O(n \log^{3.25} n)$ (which is faster than $O(n \log^5 n)$ [10]) also with high probability.

It is regrettable that maintaining a connectivity oracle of dense $G(n, p)$ is often useless since $G(n, p)$ with $\overline{M} > n \ln n/2$ is almost surely connected. However, these observations suggest that for a graph with a few parts each of which is dense (e.g. a graph with a few isolated and dense connected components), algorithms X and Y work fast. We think this kind of graph may appear in a social graph with a few isolated or almost isolated communities.

Acknowledgements. The author would like to thank Kunihiko Sadakane for helpful comments and discussion on this work.

References

1 I. Abraham, S. Chechik, and S. Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proc. SODA*, pages 440–452, 2017. doi:10.1137/1.9781611974782.28.

2 S. Baswana, S. R. Chaudhury, K. Choudhary, and S. Khan. Dynamic DFS in undirected graphs: breaking the $O(m)$ barrier. In *Proc. SODA*, pages 730–739, 2016. doi:10.1137/1.9781611974331.ch52.

3 S. Baswana, A. Goel, and S. Khan. Incremental DFS algorithms: a theoretical and experimental study. arXiv:1705.02613, 2017. arXiv:1705.02613.

4 D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. SODA*, pages 2053–2071, 2016. doi:10.1137/1.9781611974331.ch143.

5 M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. LATIN*, pages 88–94, 2000. doi:10.1007/10719839_9.

6 T. M. Chan, M. Pătraşcu, and L. Roditty. Dynamic connectivity: connecting to networks and geometry. *SIAM J. Comput.*, 40:333–349, 2011. doi:10.1137/090751670.

7 R. Duan. New data structures for subgraph connectivity. In *Proc. ICALP, Part I*, pages 201–212, 2010. doi:10.1007/978-3-642-14165-2_18.

8 R. Duan and L. Zhang. Faster randomized worst-case update time for dynamic subgraph connectivity. In *Proc. WADS*, pages 337–348, 2017. doi:10.1007/978-3-319-62127-2_29.

9 P. Erdős and A. Rényi. On random graphs I. *Publ. Math.*, 6:290–297, 1959.

10 B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proc. SODA*, pages 1131–1142, 2013. doi:10.1137/1.9781611973105.81.

11 C. Kejlberg-Rasmussen, T. Kopelowitz, S. Pettie, and M. Thorup. Faster worst case deterministic dynamic connectivity. In *Proc. ESA*, pages 53:1–53:15, 2016. doi:10.4230/LIPIcs.ESA.2016.53.

12 K. Nakamura and K. Sadakane. A space-efficient algorithm for the dynamic DFS problem in undirected graphs. In *Proc. WALCOM*, pages 295–307, 2017. doi:10.1007/978-3-319-53925-6_23.

13 D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26:362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

14 C. Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proc. SODA*, pages 1757–1769, 2013. doi:10.1137/1.9781611973105.126.