Acta Cybernetica **23** (2018) 887–902.

DFS is Unsparsable and Lookahead Can Help in Maximal Matching*

Kitti Gelle[a] and Szabolcs Iván[a]

Abstract

In this paper we study two problems in the context of fully dynamic graph algorithms that is, when we have to handle updates (insertions and removals of edges), and answer queries regarding the current graph, preferably with a better time bound than that when running a classical algorithm from scratch each time a query arrives. In the first part we show that there are dense (directed) graphs having no nontrivial strong certificates for maintaining a depth-first search tree, hence the so-called sparsification technique cannot be applied effectively to this problem. In the second part, we show that a maximal matching can be maintained in an (undirected) graph with a deterministic amortized update cost of $O(\log m)$ (where m is the all-time maximum number of the edges), provided that a lookahead of length m is available, i.e. we can "take a peek" at the next m update operations in advance.

Keywords: dynamic graphs, depth-first search, sparse strong certificate, maximal matching, lookahead

1 Introduction and notation

In the past two decades, there has been a growing interest in developing a framework of algorithm design for *dynamic graphs*, that is, graphs which are subject to *updates* – in our case, additions and removals of an edge at a time. The aim of a so-called fully dynamic algorithm (here "fully" means that both addition and removal are permitted) is to maintain the result of the algorithm after each and every update of the graph, in a time bound significantly better than recomputing it from scratch each time.

While there are plenty of ad hoc algorithms for specific problems (see e.g. [1, 3, 4, 5, 7, 8]), there are also some generic methods, one of them being the *sparsificition* technique developed in [6]. This technique can speed up the computation of the query in question, achieving the same time complexity as if the query were run on

*Work of Szabolcs Iván was supported by NKFI grant no. 108448. Work of Kitti Gelle was supported by the ÚNKP-17-3 New National Excellence Program of the Ministry of Human Capacities.

[a]University of Szeged, Hungary, E-mail: {kgelle,szabivan}@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.3.2018.10

a sparse graph. In order for sparsification to be applicable, it is necessary for the problem to have *sparse strong certificates*. Essentially, such a certificate for a graph G is a sparse graph G' on which the query should produce the same output. In [11], several graph problems were shown *not* to have a sparse strong certificate, so the technique cannot be applied to these problems (we call such properties *unsparsable*). The authors of [11] left open the question whether the *depth-first search* problem (that is, given a graph G and a vertex v of G, construct a depth-first search tree of G from v as a root) has sparse strong certificates or not. One of the results of the current paper is that this is not the case: there are dense graphs having no nontrivial certificate at all for this property, thus sparsification cannot be used to speed up the computation of a depth-first search tree in a dynamic graph. Although our method is still an ad hoc construction, we do hope that it can give a better insight on the nature of problems having sparse strong certificates (like edge and vertex connectivity, bipartiteness and minimum spanning tree, to name but a few).

Also in [11], a systematic investigation of dynamic graph problems in the presence of a so-called *lookahead* was initiated: although the stream of update operations can be arbitrarily large and possibly builds up during the computation time, in actual real-time systems it is indeed possible to have some form of *lookahead* available. That is, the algorithm is provided with some prefix of the update sequence of some length (for example, in [11] an assembly planning problem is studied in which the algorithm can access the prefix of the sequence of future operations to be handled of length $\Theta(\sqrt{m/n}\log n))$, where m and n are the number of edges and nodes, respectively. Similarly to the results of [11] (where the authors devised dynamic algorithms using lookahead for the problems of strongly connectedness and transitive closure), we will execute the tasks in batches: by looking ahead at $t = O(m)$ future update operations, we treat them as a single batch, preprocess our current graph based on the information we get from the complete batch, then we run all the updates, one at a time, on the appropriately preprocessed graph. This way, we achieve an amortized update cost of $O(\log m)$ for maintaining a maximal matching.

In this paper, a graph G is viewed as a set (or list) of edges, with $|G|$ standing for its cardinality. This way notions like $G \cup H$ for two graphs G and H (sharing the common set $V(G) = V(H)$ of vertices) are well-defined.

Related work. There is an interest in computing a maximum (i.e. maximum cardinality) or maximal (i.e. non-expandable) matching in the fully dynamic setting. There is no "best-so-far" algorithm, since the settings differ: Baswana, Gupta and Sen [2] presented a randomized algorithm for maximal matching, having an $O(\log n)$ expected amortized time per update. (Note that algorithms for maximal matching automatically provide 2-approximations for maximum matching and also vertex cover.) For the deterministic variant, Ivkovič and Lloyd [9] defined an algorithm with an $O((n+m)^{0.7072})$ amortized update time, which was improved to an amortized $O(\sqrt{m})$ update cost by Neiman and Solomon [13]. For maximum matching, Onak and Rubinfeld [14] developed a randomized algorithm that achieves a c-approximation for some constant c, with an $O(\log^2 n)$ expected amortized update time. To maintain an exact maximum cardinality matching, Micali and Vazirani

[12] gave an algorithm with a worst-case update time of $O(\sqrt{n} \cdot m)$. Allowing randomization, an update cost of $O(n^{1.495})$ is achievable due to Sankowski [15].

We are not aware of any results on allowing lookahead for any of the matching problems, but the notion has been applied to several problems in this field: following the seminal work of Khanna, Motwani and Wilson [11], where lookahead was investigated for the problems of maintaining the transitive closure and the strongly connectedness of a directed graph, Sankowski and Mucha [16] also considered the transitive closure with lookahead via the dynamic matrix inverse problem, devising a randomized algorithm, and Kavitha [10] studied the dynamic matrix rank problem.

2 Depth-first search trees

One of the main results of the current paper is that the (general) depth-first search tree property DFS is also unsparsable. Although this is again carried out in an ad hoc way, we hope that it might give an insight into the structure of unsparsable properties.

2.1 Notation

We use the following notions introduced in [11] in this form.

A *graph property* is an arbitrary function $\mathcal{P}$ which maps graphs to *nonempty sets* of objects. For example, the depth-first search function DFS maps a given directed graph to several possible depth-first search forests.

The so-called sparsification technique (introduced originally in [11, 6] as a tool for studying properties of dynamic graphs) is based on the notion of *certificates*:

Definition 1 (Strong Certificate)**.** *For a graph property $\mathcal{P}$, a* strong $\mathcal{P}$-certificate *of a graph G is a graph G' on the same vertex set as G such that*

$$\mathcal{P}(G' \cup H) \subseteq \mathcal{P}(G \cup H)$$

holds for any graph H.

Evidently, any graph G is a certificate of itself, and the following properties of *transitivity* and *monotonicity* hold [11, 6]:

- If G' is a strong $\mathcal{P}$-certificate for G and G'' is a strong $\mathcal{P}$-certificate for G', then G'' is a strong $\mathcal{P}$-certificate for G as well.
- If G' and H' are strong $\mathcal{P}$-certificates for G and H, respectively, then $G' \cup H'$ is a strong $\mathcal{P}$-certificate for $G \cup H$.

In the state-of-the-art for dynamic graph algorithms, the sparsification technique can be used to develop a dynamic algorithm for a specific problem if there are *sparse* strong certificates for the given problem:

Definition 2 (Sparse strong certificate)**.** *A property $\mathcal{P}$ has* sparse strong certificates *if, for every graph G having n vertices, there exists a strong $\mathcal{P}$-certificate for G with $O(n)$ edges.*

If some property $\mathcal{P}$ has sparse strong certificates, having $c{\cdot}n$ edges, say, and there is a fully dynamic graph algorithm with a runtime of $T(n)$ on *sparse* graphs having n nodes and $c{\cdot}n$ edges, moreover, there is an algorithm which can compute a sparse strong certificate of a graph having n nodes and $2c{\cdot}n$ edges with a runtime of $T'(n)$, then (by arranging the graph into a form of a complete binary tree of its specific subgraphs) one can construct a fully dynamic algorithm solving $\mathcal{P}$ for arbitrary graphs with a runtime of $O(\log(n)(T(n)+T'(n))$. For example, if there were sparse strong certificates for DFS that are computable in $T'(n) = O(n)$ time, then the technique would yield a dynamic algorithm having an update cost of $O(n \log n)$ (note that for dense graphs this cost would improve over the naïve approach, which has an update cost of $O(m)$).

In particular, if one shows that for a given property $\mathcal{P}$, there exist graphs for arbitrarily large n having $\Omega(n^2)$ edges having no nontrivial certificates (we call such properties *unsparsable*), then, as a byproduct one gets that sparsification cannot be applied effectively to speed up the computation of $\mathcal{P}$ in the dynamic setting.

In [11], a number of unsparsable properties were found: the *breadth-first* search tree property, strong connectivity, the *lexicographic* depth-first search tree property, transitive closure, diameter, minimum cut and maximum matching have been shown to be unsparsable. Most of the methods developed in [11] are applicable only to *monovalued* properties, i.e., when $\mathcal{P}(G)$ is a singleton set for every graph G. Indeed, all these properties are monovalued but that of the breadth-first search tree property (in which case the result is the set of all possible breadth-first trees of the input graph) which have been shown to be unsparsable using an ad hoc reasoning. The authors of [11] explicitly state that "we are unable to extend this to the case of general depth-first search tree property, and that remains an interesting open question", after showing that the (monovalued) property of lexicographic depth-first search tree property is unsparsable.

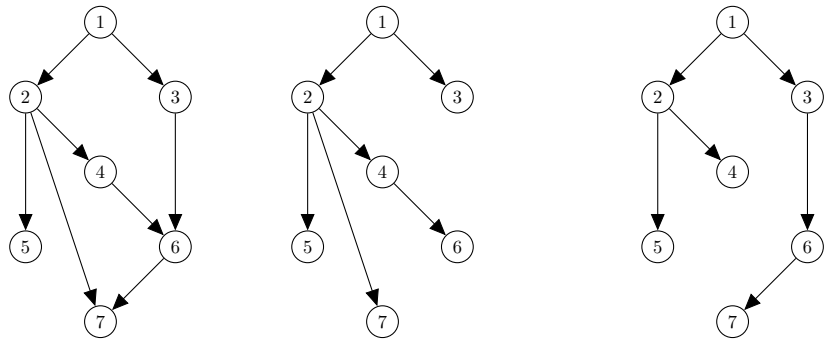
2.2 The property DFS is unsparsable

The property DFS assigns to a graph the set of all of its subgraphs that may be the result of a depth-first search according to *some* arbitrary ordering of the vertices, called on a specific vertex 1. For the sake of completeness, the algorithm is explicitly presented below.

```
dfs( v : Node ) {
  for each neighbour u of v do
    if( parent[u]==NULL){
      parent[u] := v
      dfs( u )
    }
}
```

```
for each node v do
  parent[v] := NULL
dfs( 1 )
```

The source of ambiguity in this algorithm is that the *order* of the neighbours of a node in which they are traversed is not specified. As an example, the reader is referred to Figure 1 where a graph G and two of its possible DFS trees are depicted. Of course if there is a total ordering defined on the nodes, and the neighbours of each node are traversed according to this ordering, then the DFS tree is unique and the property becomes a *monovalued* property called *lexicographic depth-first search tree*, which, due to the results of [11], is unsparsable. Clearly, every possible depth-first search tree corresponds to a lexicographic one, e.g. the search trees depicted in Figure 1 correspond to the orderings $1 \prec 2 \prec 5 \prec 7 \prec 4 \prec 6 \prec 3$ and $1 \prec 3 \prec 6 \prec 7 \prec 2 \prec 4 \prec 5$, respectively.



(a) Original graph (b) A DFS tree from node 1 (c) Another DFS tree from node 1

Figure 1: Possible depth-first search trees

Next we show that the DFS property is unsparsable as well by means of giving an explicit family of graphs having $\Omega(n^2)$ edges such that the smallest strong certificate of any member G of the family is G itself. By "smallest", we mean minimal:

Definition 3. *A strong $\mathcal{P}$-certificate G' of a graph G is a* minimal *strong $\mathcal{P}$-certificate of G if no proper subgraph of G' is a strong $\mathcal{P}$-certificate of G.*

Observe that a minimal strong DFS-certificate has no edges of the form $(i, 1)$. Indeed, since 1 is always the root of any depth-first search tree, such edges cannot be tree edges and thus can be removed from a graph without changing the set of its depth-first search trees. Similarly, a minimal strong DFS-certificate does not have loop edges.

In order to show that DFS is unsparsable, we first need to prove the following three lemmas.

Lemma 1. *Assume G is a graph such that all of its vertices are reachable from the vertex* 1*. Then the same holds in any strong* DFS*-certificate G' of G.*

Proof. Clearly, any vertex v is a descendant of the root node 1 in any depth-first search tree in a graph G' if and only if v is reachable from 1 in G'. Hence, if some vertex v is not reachable from 1 in G', then $\textsc{dfs}(G')$ consists of trees in which v does not occur while $\textsc{dfs}(G)$ consists of trees in which all the nodes occur, so $\textsc{dfs}(G') \cap \textsc{dfs}(G) = \emptyset$. In particular, $\textsc{dfs}(G') \not\subseteq \textsc{dfs}(G)$ and G' is not a strong DFS-certificate of G. □

The next lemma allows us to consider only subgraphs as possible certificates.

Lemma 2. *Assume G' is a minimal strong* DFS*-certificate of G. Then $G' \subseteq G$.*

Proof. Assume $V(G') = V(G)$ and $G' \not\subseteq G$ is a minimal strong DFS-certificate of G. Then $(i, j) \in G' - G$ for some nodes $i, j \in V$. By minimality, $j \neq 1$ and $i \neq j$. There are two cases: either $i = 1$ or $i \neq 1$.

- If $i = 1$, then there exists a depth-first search tree of G' in which j is a depth-one node. (That is, any tree we get if we uncover j first.) Since $(1, j)$ is not an edge in G, there is no such tree in $\textsc{dfs}(G)$ and thus $\textsc{dfs}(G') \not\subseteq \textsc{dfs}(G)$, which is a contradiction.
- If $i \neq 1$, then let H be the graph consisting of the single edge $(1, i)$. It suffices to check that $\textsc{dfs}(G' \cup H) \not\subseteq \textsc{dfs}(G \cup H)$. If in $G' \cup H$ we uncover the neighbour i of 1 first; then we uncover the neighbour j of i; then we get a depth-first search tree of $G' \cup H$ in which (i, j) is a tree edge. This is clearly not possible in $G \cup H$ as (i, j) is not an edge in that graph. Hence $\textsc{dfs}(G' \cup H) \not\subseteq \textsc{dfs}(G \cup H)$ and G' is not a strong DFS-certificate of G, which is a contradiction.

□

The last technical lemma of the section provides a sufficient condition for some edges being unremovable when looking for a certificate subgraph:

Lemma 3. *Assume $G' \subseteq G$, and $(i, j) \in G$ is an edge such that j is not reachable from i in G'; moreover, both i and j are reachable in G from 1, and $(1, j)$ is* not *an edge in G.*

Then G' is not *a strong* DFS*-certificate of G.*

Proof. Assume G' is a strong DFS-certificate of G. By Lemma 1 we get that both i and j are reachable from 1 in G' as well. Let $k \notin \{1, j\}$ be a node of a path from 1 to j in G'. Notice that $k \neq i$ in this case, since by assumption j is not reachable from i in G'. Also, k is thus reachable from 1 in G'.

Consider the graph H on the same set of nodes consisting of the edges (k, i), (k, j), $(1, k)$ and $(1, i)$.

Then there exists a depth-first search tree of $G' \cup H$ in which i and k are depth-one nodes, and j is a child of k.

To see this, first observe that neither j nor k is reachable from i in $G' \cup H$:

- By assumption, j is not reachable from i in G'.

- Since the edges (k, i) and $(1, i)$ are never used in a shortest $i \rightsquigarrow j$ path, j is not reachable in $G' \cup \{(k, i), (1, i)\}$.
- Adding $(1, k)$ and (k, j) does not change the transitive closure of the graph since k is already reachable from 1 in G', and j is also reachable from k in G'. Hence, j is not reachable from i in $G' \cup H$.
- Since $(k, j) \in H$, j is reachable from k in $G' \cup H$.
- Thus k is not reachable from i in $G' \cup H$.

So, if during a depth-first search on $G' \cup H$ one uncovers the node i first via the edge $(1, i) \in H$, then traverses the node in an arbitrary manner, neither j nor k appears as the descendant of i since these nodes are not reachable from i. Then, after finishing traversal of i, one uncovers k via the edge $(1, k) \in H$, and then j via $(k, j) \in H$. Finishing the procedure in an arbitrary way we get a depth-first search tree in which i and k are both depth-one nodes and j is a child of k.

We claim that there is no such depth-first search tree of $G \cup H$, proving the above lemma. To see this, we list the possible orders in which the nodes i, j and k are uncovered during a depth-first search of $G \cup H$:

- Assume i is uncovered first. Then, as $(i, j) \in G$, j becomes a descendant of i in the tree.
- Assume j is uncovered first. Then j cannot be a child of k.
- Assume k is uncovered first. Then, since $(k, i) \in H$, we get that i also becomes a descendant of k.

Hence, during any depth-first search of $G \cup H$ it cannot happen that i and k are both depth-one nodes and j is a child of k, hence $\text{DFS}(G' \cup H) \not\subseteq \text{DFS}(G \cup H)$ and G' is not a strong DFS-certificate of G. □

Now we are ready to show the main result of this section.

Theorem 1. *The property* DFS *is unsparsable: there exists an infinite family of graphs having $\Omega(n^2)$ edges such that for each member G of the family, the only minimal strong* DFS*-certificate is G itself.*

Proof. Let $n = 2k + 1$ be an odd number for some integer $k > 1$ and G_n be the graph on n vertices consisting of the following edges:

- The edges $(1, i)$ for each $2 \leq i \leq k + 1$.
- The edges (i, j) for each $2 \leq i \leq k + 1$ and $k + 2 \leq j \leq 2k + 1$.

That is, a three-layered graph such that the first layer consists of the node 1, the other two layers contain k nodes each, and from each node of each layer there is an edge to every node of the next layer. (See Figure 2).

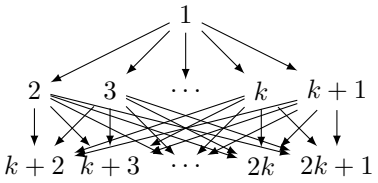


Figure 2: The graph G_{2k+1}

It is clear that all the nodes of G_n are reachable from 1. By Lemma 2, any minimal strong DFS-certificate of G_n is a subgraph G' of G_n. Now G' has to contain all the edges of the form $(1, i)$ since removing such an edge would make node i unreachable from 1, contradicting Lemma 1. We claim that none of the edges (i, j) with $2 \leq i \leq k+1$ and $k+2 \leq j \leq 2k+1$ can be removed. Indeed, as the removal of (i, j) would make j unreachable from i, and there is no direct edge $1 \to j$ in G, we see from Lemma 3 that G' has to contain all the edges from the middle towards the bottom layer. Hence $G' = G_n$ is the only minimal strong DFS-certificate of G and this graph has $k^2 + k = \Theta(n^2)$ edges. □

3 Maximal matching with lookahead

In this section we present an algorithm that maintains a maximal matching in a dynamic graph G with constant query and $O(\log m)$ update time (note that $O(\log m)$ is also $O(\log n)$ as $m = O(n^2)$), provided that a *lookahead* of length m is available in the sequence of (update and query) operations. This is an improvement over the currently best-known deterministic algorithm [13] that has an update cost of $O(\sqrt{m})$ without lookahead, and has the same amortized update cost as the best-known randomized algorithm [2].

In this problem, a *matching* of a(n undirected) graph G is a subset $M \subseteq G$ of edges having pairwise disjoint sets of endpoints. A matching M is *maximal* if there is no matching $M' \supsetneq M$ of G. Given a matching M, for each vertex v of G let MATE(v) denote the unique vertex u such that $(u, v) \in M$ if such a vertex exists, otherwise MATE(v) = NULL.

In the fully dynamic version of the maximal matching problem, the update operations are edge additions $+(u, v)$, edge deletions $-(u, v)$ and the queries have the form MATE(u).

The following is clear:

Proposition 1. *Suppose G is a graph in which M is a maximal matching. Then a maximal matching in the graph $G + (u, v)$ is*

- $M \cup \{(u, v)\}$*, if* MATE(u) = MATE(v) = NULL*,*
- M*, otherwise.*

This proposition gives the base algorithm GREEDY for computing a maximal matching in a graph:

```
Let M be an empty list of edges;
for( (u,v) ∈ G ) {
  if( MATE(u) == NULL and MATE(v) == NULL){
    MATE(u) := v; MATE(v) := u;
    insert (u,v) to M;
  }
}
return M;
```

Note that if one initializes the MATE array in the above code so that it contains some non-NULL entries, then the result of the algorithm represents a maximal matching within the subgraph of G spanned by the vertices having NULL MATEs initially. Also, with M represented by a linked list, the above algorithm runs in $O(m)$ total time using no lookahead. Hence, by calling this algorithm on each update operation (after inserting or removing the edge in question), we get a dynamic graph algorithm with no lookahead (hence it uses a lookahead of at most m operations), a constant query cost (as it stores the MATE array explicitly) and an $O(m)$ update cost. Using this algorithm A_1, we build up a sequence A_k of algorithms, each having a smaller update cost than the previous ones. (In a practical implementation there would be a single algorithm A taking k as a parameter along with the graph G and the update sequence, but for proving the time complexity it is more convenient to denote the algorithms in question by A_1, A_2, and so on.)

In our algorithm descriptions the input is the current graph G (which is $\emptyset$ the first time we start running the program) and a sequence $(q_1, \ldots, q_t)$ of operations. Of course as the sequence can be arbitrarily long, we do not require an explicit representation, just the access of the first m elements (that is, we have a lookahead of length m).

Lemma 4. *Assume A_k is a fully dynamic algorithm for maintaining a maximal matching with an $f(k) \cdot m^{1/k}$ amortized update cost, constant query cost using a lookahead of length m.*

Then there is a universal constant c such that there exists a fully dynamic algorithm A_{k+1} that also maintains a maximal matching with $(f(k)+c(1+\log m))m^{1/k+1}$ amortized update cost, and a constant query cost using a lookahead of length m.

Before proving the above lemma, we derive the main result of the section. As A_1 is an algorithm satisfying the conditions of this lemma with $k = 1$ and $f(k) = c_0$ for some constant c_0, it implies that for each $k > 1$ that there is a fully dynamic algorithm that maintains a maximal matching with an amortized update cost of $(c_0 + kc(1 + \log m))m^{1/k} = O(k \log m \cdot m^{1/k})$. Setting $k = \log m$, we get that $A_{\log m}$ has an amortized update cost of $O(\log^2 m \cdot m^{1/\log m}) = O\left(\log^2 m \cdot \left(2^{\log m}\right)^{1/\log m}\right) = O(\log^2 m \cdot 2) = O(\log^2 m)$.

Hence we get:

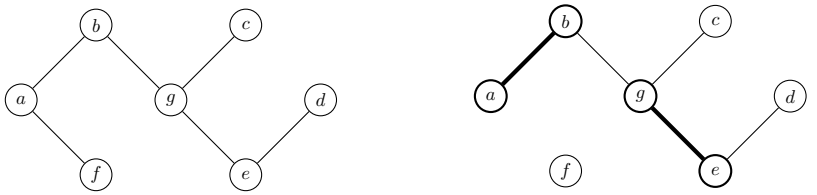
Theorem 2. *There exists a fully dynamic algorithm for maintaining a maximal matching with an $O(\log^2 m)$ amortized update cost and constant query cost, using a lookahead of length m.*

Now we prove Lemma 4 by defining the algorithm A_{k+1} below.

- The algorithm A_{k+1} works in *phases* and returns a graph G (as an edge set) and a matching M (as an edge list).
- The algorithm accesses the *global* MATE array in which the current maximal matching of the whole graph is stored. (A_{k+1} might get only a subgraph of the whole actual graph as input.)
- In one phase, A_{k+1} either handles a block $\vec{q} = (q_1, \ldots, q_t)$ of $t = m^{\frac{k}{k+1}}$ operations or a single operation.
- Let G and M respectively denote the current graph and matching we have at the beginning of a phase.
- If $|G|$ is smaller than our favorite constant 42, then the phase handles only the next operation by explicitly modifying G, afterwards recomputing a maximal matching from scratch, in $O(42)$ (constant) time. That is,
  1. We iterate through all the edges $(u, v) \in M$, and set MATE$[u]$ and MATE$[v]$ to NULL (in effect, we remove the "local part" M of the global matching);
  2. We apply the next update operation on G;
  3. We set $M :=$ GREEDY$(G,$ MATE$)$.

  Otherwise the phase handles t operations as follows:
  1. Using lookahead (observe that $t < m$) we collect all the edges involved in $\vec{q}$ (either by a $+(u, v)$ or a $-(u, v)$ update operation) into a graph G'.
  2. We construct the graph $G'' = G - G'$.
  3. We iterate through all the edges $(u, v) \in M$, and set MATE$[u] :=$ NULL, MATE$[v] :=$ NULL.
  4. We run $M :=$ GREEDY$(G'',$ MATE$)$.
  5. We call $A_k(G \cap G', (q_1, \ldots, q_t))$. Let G^* and M^* be the graph and matching returned by A_k.
  6. We set $G := G'' \cup G^*$ and $M := M \cup M^*$.

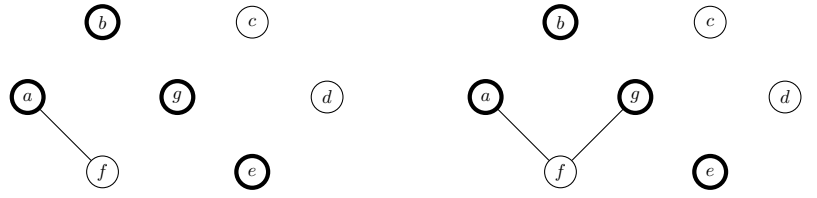
In order to give the reader a better insight, we give an example before analyzing the time complexity. To make the example more manageable, we adjust the constants as follows: we shall use the constant 1 instead of 42 (that is, if G contains at most one edge, we do not make a recursive call but recompute the matching) and also, the block size A_2 handles in one phase will be set to 1 while A_3, which we call at the topmost level, will handle 3 operations in one phase.



(a) The original graph G. (b) $G - G'$ with a maximal matching.

Figure 3: Executing Steps $1-3$ of A_3 on G, looking ahead the operations $+(f,g)$, $-(a,f)$, $+(d,c)$

Example 1. Let us assume that we call the algorithm A_3 on the graph G of Figure 3 (a). As the graph contains more edges than our threshold 1, a block of update operations of length 3 will be handled in a phase, using lookahead. (Note that if we used our actual algorithm to compute the length of the phase, we would get $6^{2/3} \approx 3.3$ as the number of edges in G is $m = 6$ and in A_3, k is 2.) Now assume the next three update operations are $+(f,g)$, $-(a,f)$ and $+(d,c)$. Thus $G' = \{(f,g),(a,f),(d,c)\}$ is the set of edges involved, that's for Step 1. In Steps 2 and 3, we construct the graph $G'' = G - G'$ and run the greedy matching algorithm on it, the (possible) result is shown in Figure 3 (b). (Note that since GREEDY does not specify the order of the edges during the traversal, the actual results can vary.) In the Figure, thick circles denote those vertices having a non-NULL mate at this point (that is, MATE$[a] = b$, MATE$[b] = a$, and so on, c, d and f having a NULL mate). Now, A_2 is called on $G \cap G'$ (depicted in Figure 4 (a)), and the whole block of three updates is passed to A_2.



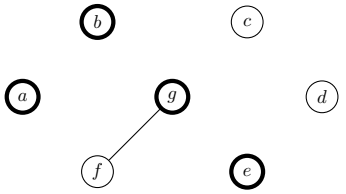
(a) The graph $G \cap G'$ (b) A_2 adds (f,g) directly

Figure 4: Handling the first recursive call.

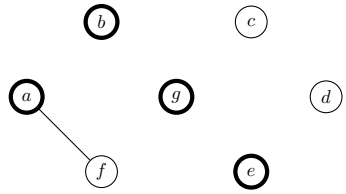
Now as the input graph of A_2 has only one edge, A_2 just handles the next update $+(f,g)$; that is, it inserts the edge (f,g) into its input of Figure 4 (a) and runs GREEDY on this, resulting in the graph of Figure 4 (b). Observe that at this point MATE$[a] = b$ and MATE$[g] = e$, so neither of these two edges is added to the maximal matching managed by A_2. (That is, the MATE array is a global variable. This is vital: this way one can ensure that the union of the matchings of different

recursion levels is still a matching, and also ensures a constant-time query cost.)

Then, as the current graph has two edges (which is larger than the threshold), A_2 handles a complete block of operations in a phase. (Now the length of the block happens to be 1 so this does not make that much of a difference. Actually, as $m = 2$ and $k = 1$, the length of the block should indeed be $2^{1/2} \approx 1.4$.) Thus, using a lookahead of length 1, the only operation to be handled is $-(a, f)$. So we compute the difference graph and run GREEDY on it (Figure 5 (a)), compute the intersection graph and call A_1 on this along with the update sequence consisting of the single operation $-(a, f)$ (Figure 5 (b)). As the input of A_1 is now a graph



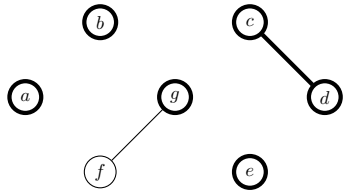
(a) The result of GREEDY run on the difference graph



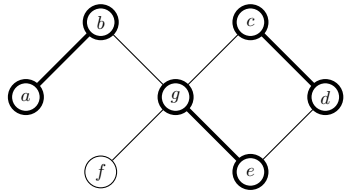
(b) The graph passed to A_1 along with the single update $-(a, f)$

Figure 5: Handling the second update

consisting of a single edge, it gets removed (as the edge in question is not involved in the matching, which can be seen e.g. from the MATE array, the global matching is not changed), resulting in an empty graph on which GREEDY gives an empty matching as well. Then, A_1 returns, as it handled the only operation it received. Now A_2 takes control. Concluding the second phase, it constructs the union of its intersection graph and the empty graph returned by A_1, so its current graph G becomes the graph on Figure 5 (a). As now the graph has only one edge, the next update $+(d, c)$ is handled directly: the edge (c, d) is inserted and GREEDY is run (Figure 6 (a)). Now as A_2 has handled its whole input block, it returns its current



(a) The edge (c, d) is added to the matching by A_2.



(b) The current graph and matching after handling all the updates.

Figure 6: Handling the last update

graph: A_3 takes control and glues together its difference graph from Figure 3 (b) and the returned graph 6 (a), resulting in the graph in Figure 6 (b) which would be the starting graph of further updates. Note that in the actual algorithm, as

$2 < \log m < 3$, we would call A_2 at the top level and handle a block of $6^{1/2} \approx 2.44$; that is, two updates, but we deliberately chose to call A_3 for the sake of covering almost all the possibilities the algorithm can have (the exception being the case where a matched edge gets removed, which can be simply handled by setting the endpoints' MATE values to NULL and removing the edge from the local matching of the given recursion level).

Having completed this example, we will now show its correctness. That is, we claim that each A_k maintains a maximal matching among those vertices having a NULL MATE when the algorithm is called. This is true for the greedy algorithm A_1. Now assuming A_k satisfies our claim, let us check A_{k+1}. When the graph is small, the algorithm throws away its locally stored matching M, resetting the MATE array to its original value in the process (in fact, this is the only reason why we store the local matching at each recursion level: the global matching state can be queried by accessing the MATE array alone). Then we handle the update and run GREEDY, which is known to compute a maximal matching on the subgraph of G spanned by the vertices having a NULL mate. So this case is clear.

For the second case, if a block of t operations is handled, then we split the graph into two, namely a difference graph G' and an intersection graph G''. By construction, when handling the block, the edges belonging to G' do not get touched. Hence, at any time point, a maximal matching of G can be computed by starting from a maximal matching of G' and then extending the matching by a maximal matching in the subgraph of G'' not covered by the matching of G'. Thus, if we compute a maximal matching M' in the subgraph of G' spanned by the vertices having a NULL MATE, updating the MATE array accordingly (that is, calling GREEDY on G'), and maintaining a maximal matching M'' over the vertices of G'' having a NULL MATE after that point (which is done by A_k, by the induction hypothesis), we get that at any time $M' \cup M''$ is a maximal matching of G. Hence, the algorithm is correct.

Now we analyse the time complexity of A_{k+1}. When a phase handles t operations, then Step 1 can be executed in $O(t \log t) = O(m \log m)$ time (if we use a self-balancing tree representation for storing our graphs, say an AVL tree). Then in Step 2, we construct the difference of the two sets of size $O(m)$ in $O(m \log m)$ time. Step 3 requires an additional time of $O(m)$, since the matching is of size $O(m)$ and it is stored as a list of edges. For Step 4, as $|G''| \leq |G| = m$, also an $O(m)$ time is required, and for Step 5, computing the intersection $G \cap G'$ requires a time of $O(m \log m)$, and A_k, being run on a dynamic graph having at most $t = m^{\frac{k}{k+1}}$ edges during its whole lifecycle of t operations needs $t \cdot f(k) \cdot t^{1/k} = m^{\frac{k}{k+1}} \cdot f(k) \cdot m^{\frac{1}{k+1}} = f(k) \cdot m$ computation steps. Gluing together the graphs and the matchings in Step 6 needs a time of $O(m \log m) + O(m)$. Hence the total cost of Steps 1-6 handling a whole phase is $O(m) + O(m \log m) + O(m) + f(k) \cdot m + O(m \log m) + O(m) = (f(k) + c(1 + \log m))m$ for some universal constant c, and since a phase consists of $m^{\frac{k}{k+1}}$ operations, the amortized cost of a single operation becomes $(f(k) + c(1 + \log m))m^{\frac{1}{k+1}}$ and Lemma 4 is proved.

The careful reader may observe that a major part of the time bound comes from

the set operations. If an initialization cost of $O(n^2 \log n)$ is affordable (i.e. if there are $\Omega(n^2 \log n)$ operations in total), then we can do better:

- Each algorithm A_k has an adjacency matrix as well, initialized to an all-zero matrix in the very beginning (this initialization takes the aforementioned $O(n^2 \log n)$ setup cost).
- In Step 1, edges of G' are stored into this matrix (taking still $O(m)$ time).
- Now the graphs $G - G'$ and $G \cap G'$, as lists of edges, can be constructed in $O(m)$ time (since lookup in G' now takes constant time instead of the previous $O(\log m)$).
- Since G'' is represented as an edge list, GREEDY still takes $O(m)$ time.
- After performing Step 5, we have to set the auxiliary matrix to an all-zero matrix by looking ahead once again the very same sequence and setting each accessed edge to 0. This takes $O(m)$ time.
- Also, taking the unions of the graphs and matchings upon returning can be destructive to the original lists, thus it can be done in constant time.

Hence in this case the total cost spent for a phase becomes $O((f(k) + c)m)$ for some universal constant c, yielding an amortized update cost of $O((k+1) \cdot m^{1/k+1})$ for A_k, which boils down to an amortized update cost of $O(\log m)$ by choosing $k = \log m$ and we showed:

Theorem 3. *There exists a fully dynamic algorithm for maintaining a maximal matching with an $O(n^2 \log n)$ initialization cost, $O(\log m)$ amortized update cost and constant query cost using a lookahead of length m.*

4 Conclusion

In this study we dealt with two problems arising in the context of fully dynamic graph algorithms. First, we showed via an ad hoc method that the depth-first search tree (or, forest) property is unsparsable; that is, there are dense graphs for this property having no nontrivial strong certificates. Thus, the technique of sparsification cannot be applied to this problem effectively – if it could be, it would result in an algorithm having in an update cost of $O(n \log n)$, but it's not. This solves an open problem mentioned in [11].

In the second, more detailed part of the study we showed that by using a *lookahead* of linear length, there is a *deterministic* algorithm achieving an $O(\log m)$ amortized update cost (after a somewhat costly initialization which, if cannot be afforded for some matter, then the update cost becomes $O(\log^2 m)$). This result shows that lookahead can help in the dynamic setting for problems other than the transitive closure (and the SCC) properties, studied in [11]: indeed, the best known

deterministic algorithm for the problem using no lookahad has an update cost of $O(\sqrt{m})$.

It is an interesting question to study further the possibilities of using lookahead for different problems, and maybe factor in also randomization as well.

References

[1] Alberts, D. and Henzinger, M. R. Average-case analysis of dynamic graph algorithms. *Algorithmica*, 20(1):31–60, Jan 1998.

[2] Baswana, Surender, Gupta, Manoj, and Sen, Sandeep. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM Journal on Computing*, 44(1):88–113, 2015.

[3] Chan, Timothy M. Dynamic subgraph connectivity with geometric applications. *SIAM J. Comput.*, 36(3):681–694, September 2006.

[4] Demetrescu, Camil and Italiano, Giuseppe F. Trade-offs for fully dynamic transitive closure on dags: Breaking through the o(n2 barrier. *J. ACM*, 52(2):147–156, March 2005.

[5] Demetrescu, Camil and Italiano, Giuseppe F. Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *J. Discrete Algorithms*, 4(3):353–383, 2006.

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

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

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

[9] Ivković, Zoran and Lloyd, Errol L. *Fully dynamic maintenance of vertex cover*, pages 99–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[10] Kavitha, Telikepalli. Dynamic matrix rank with partial lookahead. *Theor. Comp. Sys.*, 55(1):229–249, July 2014.

[11] Khanna, S., Motwani, R., and Wilson, R. H. On certificates and lookahead in dynamic graph problems. *Algorithmica*, 21(4):377–394, Aug 1998.

[12] Micali, S. and Vazirani, V. V. An $o(\sqrt{|V|} \cdot |e|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 17–27, Oct 1980.

[13] Neiman, Ofer and Solomon, Shay. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, November 2015.

[14] Onak, Krzysztof and Rubinfeld, Ronitt. Maintaining a large matching and a small vertex cover. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, pages 457–464, New York, NY, USA, 2010. ACM.

[15] Sankowski, Piotr. Faster dynamic matchings and vertex connectivity. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 118–126, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[16] Sankowski, Piotr and Mucha, Marcin. Fast dynamic transitive closure with lookahead. *Algorithmica*, 56(2):180–197, February 2010.

Received 3th May 2018