Dominating Sets and Connected Dominating Sets in Dynamic Graphs

Niklas Hjuler, Giuseppe F. Italiano, Nikos Parotsidis, David Saulpic

Dominating Sets and Connected Dominating Sets in Dynamic Graphs

Niklas Hjuler
University of Copenhagen, Denmark
hjuler@di.ku.dk

Giuseppe F. Italiano
LUISS University, Rome, Italy
gitaliano@luiss.it

Nikos Parotsidis
University of Rome Tor Vergata, Italy
nikos.parotsidis@uniroma2.it

David Saulpic
ENS Paris, France
david.saulpic@ens.fr

Abstract

In this paper we study the dynamic versions of two basic graph problems: Minimum Dominating Set and its variant Minimum Connected Dominating Set. For those two problems, we present algorithms that maintain a solution under edge insertions and edge deletions in time $O(\Delta \cdot \text{polylog } n)$ per update, where Δ is the maximum vertex degree in the graph. In both cases, we achieve an approximation ratio of $O(\log n)$, which is optimal up to a constant factor (under the assumption that $P \neq NP$). Although those two problems have been widely studied in the static and in the distributed settings, to the best of our knowledge we are the first to present efficient algorithms in the dynamic setting.

As a further application of our approach, we also present an algorithm that maintains a Minimal Dominating Set in $O(min(\Delta, \sqrt{m}))$ per update.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Dominating Set, Connected Dominating Set, Dynamic Graph Algorithms

Digital Object Identifier 10.4230/LIPIcs.STACS.2019.35

Funding This work was done while Niklas Hjuler and David Saulpic were visiting University of Rome Tor Vergata.

1 Introduction

The study of dynamic graph algorithms is a classical area in algorithmic research and has been thoroughly investigated in the past decades. Maintaining a solution of a graph problem in the case where the underlying graph changes dynamically over time is a big challenge in the design of efficient and practical algorithms. Indeed, in several applications, due to the dynamic nature of today's data, it is not sufficient to compute a solution to a graph problem only once and for all: often, it is necessary to *maintain* a solution efficiently while the input graph is undergoing a sequence of dynamic updates. More precisely, a *dynamic graph* is a sequence of graphs $G_0, ..., G_M$ on n nodes and such that G_{i+1} is obtained from G_i by adding or removing a single edge. The natural first barrier, in the study of dynamic algorithms, is to design algorithms that are able to maintain a solution for the problem at hand after each update faster than recomputing the solution from scratch. Many dynamic graph problems such as minimum spanning forests (see e.g. [22, 26]), shortest paths [12], matching [4, 27, 30] or coloring [7] have been extensively studied in the literature, and very efficient algorithms are known for those problems. Recently, a lot of attention has been devoted to the Maximal

Independent Set problem (MIS). In this problem, one wishes to find a maximal set of vertices that do not share any edge ("maximal" meaning that it is not possible to add any vertex without violating this property). Until very recently, the best known update bound on the complexity to maintain a MIS was a simple $O(\Delta)$ algorithm, where Δ is an upper bound on the degree of vertices in the graph. This bound was first broken by Assadi et al. [2] who gave a $O(m^{3/4})$ algorithm, then by Gupta and Khan [19] improved the update bound to $O(m^{2/3})$. Very recently, using randomization, Assadi et al. [3] presented an amortized fully-dynamic algorithm with an expected $\widetilde{O}(n^{1/2})$-time bound per update.

The MIS problem is closely related to the Dominating Set (DS) problem: given a graph $G = (V, E)$ the DS problems is to find a subset of vertices $\mathcal{D} \subseteq V$ such that every vertex in G is adjacent to $\mathcal{D}$ (or *dominated* by $\mathcal{D}$). Indeed, a MIS is also a Minimal DS: the fact that it is not possible to add a vertex without breaking the independence property implies that every vertex is adjacent to the MIS, so this must be also a DS; at the same time, it is not possible to remove a vertex since that vertex is no longer dominated. Thus, to find a Minimal DS one can simply find a MIS: this gives immediately a deterministic $O(m^{2/3})$ [19] bound and a randomized $\widetilde{O}(n^{1/2})$ [3] one. However, while it is known that is hard to approximate Maximum Independent Set[1] within a factor $n^{1-\epsilon}$ for every $\epsilon > 0$[21], a simple greedy approach achieves a $O(\log n)$-approximation for Minimum DS [11].

In recent years, there has been a lot of work on designing dynamic graph algorithms for maintaining approximate solutions to several problems. A notable example is matching, where for different approximations there exist different algorithms (see e.g., [4, 5, 27, 20, 8, 30]). This raises the natural question on whether there exists a dynamic algorithm capable of maintaining an approximation to Minimum DS, and even better a $O(\log n)$ approximation. In this paper, we answer this question affirmatively by presenting an algorithm that achieves a $O(\log n)$ approximation, with a complexity matching the long standing $O(\Delta)$ bound for MIS. Moreover, if one is interested in finding a DS faster, we present a very simple *deterministic* $O(m^{1/2})$ algorithm to compute a Minimal DS, improving the $O(m^{2/3})$ bound coming from MIS. We believe these are important steps towards understanding the complexity of the problem. Those two results are stated below.

▶ **Theorem 1.** *Starting from a graph with n vertices, a $O(\log n)$ approximation of* Minimum Dominating Set *can be maintained over any sequence of $\Omega(n)$ edge insertions and deletions in $O(\Delta \log n)$ amortized time per update, where Δ is the maximum degree of the graph over the sequence of updates.*

▶ **Theorem 2.** *Starting from a graph with n (fixed) vertices, a* Minimal Dominating Set *can be deterministically maintained over any sequence of edge insertions and deletions in $O(\sqrt{m})$ amortized time per update, where m is an upper bound on the number of edges in the graph.*

We also study the Minimum Connected Dominating Set problem (MCDS), which adds the constraint that the graph induced by the DS $\mathcal{D}$ must be connected. This problem was first introduced by Sampathkumar and Walikar [28] and arises in several applications. The most noteworthy is its use as a *backbone* in routing protocols: it allows to limit the number of packet transmissions, by sending packets only along the backbone rather than throughout the whole network. Du and Wan's book [13] summarizes the knowledge about

[1] It is not possible to find a polynomial-time algorithm that finds a $n^{1-\epsilon}$-approximation to Maximum Independent Set under the assumption NP $\neq$ ZPP

MCDS. A special class of graphs is geometric graphs, where vertices are points in the plane, and two vertices are adjacent if they fall within a certain range (say, their distance is at most 1). This can model wifi transmissions, and the dynamic MCDS had been studied in this setting: a polynomial-time approximation scheme is known [10], and Guibas et al. [17] show how to maintain a constant-factor approximation with polylogarithmic update time. While geometric graphs model problems linked to wifi transmissions, the general graph setting can be also seen as a model for wired networks. However, no work about dynamic MCDS is known in this setting: the static case is well studied, with a greedy algorithm developed by Guha and Keller [16] that achieves an approximation factor $O(\ln \Delta)$. They also show a lower bound matching their complexity, together with their approximation factor. MCDS had also been thoroughly studied in the distributed setting (see e.g. a heuristic to find a Minimal CDS in [9], another one that sends $O(\Delta n)$ messages and has a time complexity at each vertex $O(\Delta^2)$ [31] or a $3 \log n$ approximation that runs in $O(\gamma)$ rounds where γ is the size of the CDS found, with time complexity $O(\gamma\Delta^2 + n)$ and message complexity $O(n\Delta\gamma + m + n \log n)$ [6]). Despite all this work, no results are known in the dynamic graph setting. As another application of our approach, we contribute to filling this gap in the research line of MCDS. In particular, in this paper we show how our algorithm for Minimum DS can be adapted in a non-trivial way to maintain a $O(\log n)$ approximation of the MCDS in general dynamic graphs.

▶ **Theorem 3.** *Starting from a graph with n vertices, a $O(\log n)$ approximation of* Minimum Connected Dominating Set *can be maintained over any sequence of $\Omega(n)$ edge insertions and deletions in $\widetilde{O}(\Delta)$ amortized time per update.*

We further show how to maintain independently a Dominating Set $\mathcal{D}$ and a set of vertices $\mathcal{C}$ such that the induced subgraphs on the vertices $\mathcal{C} \cup \mathcal{D}$ is connected. The set $\mathcal{C}$ has the additional property that $|\mathcal{C}| \leq 2|\mathcal{D}|$, such that $|\mathcal{C} \cup \mathcal{D}| = O(|\mathcal{D}|)$. If $\mathcal{D}$ is a α-approximation of Minimum DS, this gives a $O(\alpha)$ approximation for MCDS.

Further Related Work

It is well known that finding a Minimum DS is NP-hard [15]. It is therefore natural to look for approximation algorithms for this problem. Unfortunately, it is also NP-hard to find a $c \log n$ approximation, for any $0 < c < 1$ [14]. This bound is tight, since there is a simple greedy algorithm matching this bound [11]. Minimum DS had been studied extensively in distributed computing: an algorithm which runs in $O(\log n \log \Delta)$ rounds finds a $O(\log n)$ approximation with high probability [23] and an algorithm with constant number of rounds achieves a non-trivial approximation[25].

The DS problem is closely related to the Set Cover problem: the two problems are equivalents under L-reduction [24]. However, Set Cover was studied in the dynamic setting [18, 1], but with different kinds of updates: instead of edges being inserted or deleted (which would represent new elements in the sets according to the L-reduction), new elements are being added to the cover (which would be new vertices in DS).

Outline. The rest of the paper is organized as follows. First, we present an algorithm for Minimum DS, which will be used later on also for MCDS: we start by a $\widetilde{O}(n)$ algorithm, and then show how to overcome its bottleneck in order to achieve a $\widetilde{O}(\Delta)$ complexity. Finally, we present our $O(\sqrt{m})$ algorithm for Minimal DS.

2 A $O(\log n)$ approximation of Minimum Dominating Set in $O(\Delta \log n)$ time per update

This section aims at proving Theorem 1. Following a reduction from Set Cover, minimum DS is NP-hard to approximate within a factor $\log n$ [14]. Here we present a matching upper bound (up to a constant factor), in the dynamic setting. Our algorithm relies heavily on the clever set cover algorithm by Gupta et al. [18]. While in the static setting Set Cover is equivalent to minimum DS, in the dynamic setting these two problems are different. More precisely, in the dynamic Set Cover problem one is asked to cover a set of points S (called the universe) with a given family of sets F, while the set S is changing dynamically. To draw the parallel with DS, in the latter the set S is the set of vertices of the graph (which does not change) and for every vertex the set of its neighbors is in F. The dynamic part concerns therefore F, and not the universe S.

Gupta et al. present an $O(\log n)$-approximation for dynamic Set Cover problem: in what follows, we show how to adapt their algorithm to the DS case, with an update time of $O(\Delta \log n)$. As in [18], the approach easily adapts to the weighted case. Unfortunately, this cannot be generalized to MCDS, therefore we do not consider this property of the algorithm. The following definitions are partly adapted from [18].

2.1 Preliminaries

For a vertex v, let $N(v)$ be the set of its neighbors, including v. The algorithm maintains a solution S_t at time t such that an element of S_t is a pair composed of

- a dominant vertex v
- a set $\mathrm{Dom}(v) \subseteq N(v)$, which are the vertices that are dominated by v. We call $|\mathrm{Dom}(v)|$ the *cardinality* of the pair.

We call a *dominating pair* an element of S_t. The algorithm requires that multiple copies of a vertex can appear as the dominant vertex of a pair. However, each vertex is exactly in one $\mathrm{Dom}(v)$. The solution to the DS problem is composed of all vertices that appear as dominant vertices of a pair. Since each vertex is in exactly one $\mathrm{Dom}(v)$, each vertex is dominated and therefore the set of dominants is a valid solution to the DS problem.

The dominating pairs are placed into *levels* according to their cardinality: the level l is defined by a range $R_l := [2^{l-10}, 2^l]$, and each pair $(v, \mathrm{Dom}(v))$ is placed at an appropriate level l such that $|\mathrm{Dom}(v)| \in R_l$. In that case, elements of $\mathrm{Dom}(v)$ are said to be dominated at level l; we denote by V_l the set of all vertices dominated at level l. We say that an assignment of levels is valid if it respects the constraint $|\mathrm{Dom}(v)| \in R_l$. This allows us to define the notion of *Stability*:

- *stable solution*: A solution S_t is stable if there is no vertex v and level l such that $|N(v) \cap V_l| > 2^l$; in other words, it is not possible to introduce a new vertex in the solution to dominate some vertices at level l such that the resulting dominating pair could be at level strictly greater that l.

The algorithm will dynamically maintain a stable solution S_t, with a valid assignment of levels. Note that the ranges R_l overlap: this gives some slack to the algorithm, which allows enough flexibility to prevent too many changes while our algorithm maintains a valid solution.

2.2 The algorithm

The main part of the algorithm is the function STABILIZE, which restores the stability at the end of every update. The function is the following (see [18]):

> STABILIZE. As long as a vertex v violates the stability condition at level l, do the following: Add the pair $(v, N(v) \cap V_l)$ to the *lowest* possible level j (i.e., the lowest level such that $|N(v) \cap V_l| \in R_j$); Remove the elements of $N(v) \cap V_l$ from the set of their former covering pair: if it gets empty, remove the pair from the solution. Otherwise, if the cardinality of such a pair goes below 2^{l-10}, put it at the *highest* possible level.

Edge addition. When a new edge (u, v) is added to the graph, one just need to ensure that the solution remains stable, and thus the algorithm runs STABILIZE.

Edge deletion. When an edge (u, v) is removed from the graph, we proceed as follows. If neither u nor v dominates the other endpoint, the solution remains valid and stable, and nothing needs to be done. Otherwise, assume without loss of generality that v dominates u. Then:

- Remove u from $\mathrm{Dom}(v)$
- Add the pair $(u, \mathrm{Dom}_u = \{u\})$ to the solution with level 1
- Run STABILIZE

Correctness. All the nodes of the graph are dominated at every time. Indeed, STABILIZE does not make any node undominated and if a vertex is not dominated after an edge removal, the algorithm simply adds it to the solution. Therefore, the solution S_t maintained by the algorithm is a valid one.

2.3 Analysis

Approximation ratio. We use the following lemma by Gupta et al. [18] to bound the cost of a stable solution.

▶ **Lemma 4** (Lemma 2.1 in [18]). *The number of sets at one level in any stable solution is at most* $2^{10} \cdot OPT$.

Since for every dominating pair $(v, \mathrm{Dom}(v))$ we have that $1 \leq |\mathrm{Dom}(v)| \leq n$, there are only $\log n$ levels that can contain a set. The total cost of a stable solution is therefore $O(\log n \cdot \mathrm{OPT})$.

A token scheme to bound the number of updates. Unfortunately, the analysis of Gupta et al. cannot be applied directly to the case of DS, due to the different nature of the updates. However, we can build upon their analysis, as follows. We first bound the number of vertices that change level, and then explain how to implement a level change so that it costs $O(\Delta)$. We prove the following lemma by using a token argument.

▶ **Lemma 5.** *After k updates of the algorithm, at most $O(k \log n + n \log n)$ elements have changed levels.*

Proof. We use the following token scheme, where each vertex pays one token for each level change. In the beginning, we give $2\log n$ tokens to every vertex. If a vertex is undominated after an edge removal, we give $2\log n$ new tokens to this vertex. Since at most one vertex gets undominated for each edge deletion, the total number of tokens given after k updates is $O(k\log n + n\log n)$. To prove the lemma, we need to show that at any time each vertex has always a positive amount of tokens. We adapt the proof of Gupta et al. to show the following invariant:

▶ **Invariant 1.** *Every vertex at level l has more than $2(\log n - l)$ tokens.*

When a vertex is moved to a higher level, it pays one token for the cost of moving. It also saves one token, and gives it to an "emergency fund" of its former covering pair. Each pair has therefore a fund of tokens that can be used when the pair has to be moved to a lower level.

When the pair $(v, \mathrm{Dom}(v))$ has to be moved from level l to level $l-j$, it means that a lot of vertices have left $\mathrm{Dom}(v)$ and that the tokens they gave to the pair can be used to pay for the operation. Formally, we want to pay one token for every vertex in $\mathrm{Dom}(v)$ for its level change, but we also want to restore the invariant. We need therefore $2j+1$ tokens for each vertex of $\mathrm{Dom}(v)$. Since the pair can be moved to level $l-j$, this means that $|\mathrm{Dom}(v)| < 2^{l-j}$. Since a new pair is moved to the lowest possible level, this pair could not be at level $l-1$, which implies that $|\mathrm{Dom}_{init}(v)| > 2^{l-1}$ where $\mathrm{Dom}_{init}(v)$ is the set $\mathrm{Dom}(v)$ at the time where it was created. Moreover, each of the vertices that left gave one token: the amount of tokens usable is therefore bigger than $2^{l-1} - 2^{l-j}$. Thus we want to prove that $2^{l-1} - 2^{l-j} \geq (2j+1)\cdot|\mathrm{Dom}(v)|$. It is enough to have $2^{l-1} - 2^{l-j} \geq 3\cdot(2j+1)2^{l-j}$, i.e. to have $2^{j-1} - 1 \geq 3(2j+1)$. But since the pair was moved to level $l-j$, it means that $|\mathrm{Dom}(v)| > 2^{l-j-1}$ and $|\mathrm{Dom}(v)| < 2^{l-10}$: putting these two equations together gives $j > 9$, which ensures that $2^{j-1} - 1 \geq 3(2j+1)$ and concludes the proof. ◀

As the following corollary shows, we can bound the number of changes to $\mathcal{D}$ to $O(\log n)$ amortized. This property will be useful in Section 3.

▶ **Corollary 6.** *After k updates of the algorithm, at most $O(k\log n + n\log n)$ vertices can be added to or removed from $\mathcal{D}$.*

Proof. Whenever a vertex is added to or removed from $\mathcal{D}$, its level is changed. Lemma 5 gives the corresponding bound. ◀

We now turn to the implementation of the function Stabilize. As shown in the next lemma, we implemented so that its cost is $O(\Delta)$ for each element that changes level.

▶ **Lemma 7.** *A stable solution can be maintained in $O(\Delta\log n)$ amortized time per update.*

Proof. For all vertices v and all levels l, the algorithm maintains the set $N(v)\cap V_l$ and its cardinality. Every time a vertex changes its level, it has to inform all its neighbors: this can be done in $O(\Delta)$. When an edge (u,v) is added to or removed from the solution, the algorithm updates the sets $N(v)\cap V_{l_u}$ and $N(u)\cap V_{l_v}$, where l_u and l_v are the levels of u and v, respectively.

During a call to Stabilize, the algorithm maintains also a list of vertices that may have to be added to restore the stability: for a vertex v and level l, every time that $N(v)\cap V_l$ changes, if the new cardinality violates the stability, we add v to this list in constant time. The algorithm processes the list vertex by vertex: it checks that the current vertex still needs to be added to the solution, and add it if necessary.

Since we pay $O(\Delta)$ per level change and there are $O(\log n)$ amortized changes, the amortized complexity of each update is $O(\Delta\log n)$. ◀

Since a stable solution gives a $O(\log n)$ approximation to minimum DS, Lemmas 4 and 7 yield the proof of Theorem 1: a $O(\log n)$ approximation of Minimum Dominating Set can be maintained in $O(\Delta \log n)$ amortized time per update.

3 A $O(\log n)$ Approximation for Minimum Connected Dominating Set in $\widetilde{O}(n)$ per update

A possible way to compute a Connected DS is simply to find a DS and add a set of vertices to make it connected. Section 2 gives an algorithm to maintain an approximation of the Minimum DS: we will use it as a black box (and refer to it as the "black box"), and show how to make its solution connected without losing the approximation guarantee. If the original graph is not connected, the algorithm finds a CDS in every connected component: we focus in the following on a single of these components. Let $\mathcal{D}$ be the DS maintained, and $\mathcal{C}$ be a set of vertices such that $\mathcal{C} \cup \mathcal{D}$ is connected and $\mathcal{C}$ is minimal for that property. The minimality of $\mathcal{C}$ will ensure that $|\mathcal{C}| \leq 2|\mathcal{D}|$: since $\mathcal{D}$ is a $O(\log n)$ approximation of MDS, this leads to a $O(\log n)$ approximation for MCDS. Note that the vertices of $\mathcal{C}$ are not used for domination: $\mathcal{C} \cup \mathcal{D}$ is therefore not minimal, but still an approximation of minimum.

Overall, we will apply the following charging scheme to amortize the total running time. The main observation is that although a lot of vertices can be deleted to restore the minimality of $\mathcal{C}$, only a few can be added at every step. We thus give enough potential to a vertex whenever it is added into $\mathcal{C}$ and whenever its neighborhood changes, so that at the time of its removal from $\mathcal{C}$ it has accumulated enough potential for scanning its entire neighborhood. After an edge deletion we might have to restore the connectivity requirement. We do that by adding at most 2 new vertices in $\mathcal{C}$: this is crucial for our amortization argument.

Outline. The set $\mathcal{C}$ may have to be updated for two reasons:

- Restore the connectivity: if an edge gets deleted from the graph, or if the black box removes some vertices from $\mathcal{D}$, it may be necessary to add some vertices to $\mathcal{C}$ in order to restore the connectivity of $\mathcal{C} \cup \mathcal{D}$.
- Restore the minimality of $\mathcal{C}$: when an edge is added to the graph, or when a vertex is added to $\mathcal{C} \cup \mathcal{D}$ (either by the black box or in order to restore the connectivity), some vertices of $\mathcal{C}$ may become useless and therefore need to be removed.

We now address those two points. All our bounds are expressed in term of the total number of changes in $\mathcal{C} \cup \mathcal{D}$: let therefore k be this number of changes. We will show later that, after t updates to the graph, $k = O(t \log n)$.

The first phase of the algorithm is to restore the connectivity. We explain in the following how to decide which vertices should be added to $\mathcal{C}$ for that purpose.

Restore the connectivity after an edge deletion

To monitor the connectivity requirement, we use the following idea. The algorithm maintains a minimum spanning tree (MST) of the graph G where a weight 1 is assigned to the edges between vertices in $\mathcal{C} \cup \mathcal{D}$ (called from now on $\tilde{\mathcal{D}}$), and weight m is assigned to all other edges. These weights ensure that, as long as $\tilde{\mathcal{D}}$ is connected, the MST induces a tree on $\tilde{\mathcal{D}}$. When $G[\tilde{\mathcal{D}}]$ gets disconnected by an update, the MST uses a vertex of $V \setminus \tilde{\mathcal{D}}$ as an internal vertex: in that case, our algorithm adds this vertex to $\mathcal{C}$, to restore the connectivity. We give more details in the next section.

The edge weights are updated as the graph undergoes edge insertions and deletions and vertices enter or leave $\tilde{\mathcal{D}}$. The MST of the weighted version of the graph has the following properties.

- If $\tilde{\mathcal{D}}$ is a connected DS, then the MST has weight $(|\tilde{\mathcal{D}}| - 1) + m \cdot |V \setminus \tilde{\mathcal{D}}|$ (Kruskal's algorithm on this graph would use $|\tilde{\mathcal{D}}| - 1$ edges of weight 1 to construct a spanning tree on $\tilde{\mathcal{D}}$, then $|V \setminus \tilde{\mathcal{D}}|$ edges of weight m to span the entire graph).
- If $\tilde{\mathcal{D}}$ is a DS but $G[\tilde{\mathcal{D}}]$ is not connected, then the weight of the MST has larger value.

The two properties stem from the fact that a MST can be produced by finding a minimum spanning forest on $\tilde{\mathcal{D}}$ and extend it to a MST on V. Kruskal's algorithm ensures that this leads to a MST. In the case where $\tilde{\mathcal{D}}$ is connected, the first step yields a tree of weight $\tilde{\mathcal{D}} - 1$, and since the graph is connected the second step yields a cost $m \cdot |V \setminus \tilde{\mathcal{D}}|$. However, if $\tilde{\mathcal{D}}$ is not connected, the second step adds strictly more that $|V \setminus \tilde{\mathcal{D}}|$ edges, therefore yielding a cost bigger than $m \cdot (1 + |V \setminus \tilde{\mathcal{D}}|)$. This is more than $(|\tilde{\mathcal{D}}| - 1) + m \cdot |V \setminus \tilde{\mathcal{D}}|$, as claimed.

Furthermore, if $G[\tilde{\mathcal{D}}]$ has two connected components C_1, C_2, then the shortest of all paths between vertices $u, v,\ u \in C_1, v \in C_2$ is the minimum number of vertices whose insertion into $\mathcal{C}$ restores the connectivity requirement. Note that the shortest of all such paths must have length at most 2 (otherwise, there must be a vertex not adjacent to any vertex in $\mathcal{D}$, which contradicts the fact that $\mathcal{D}$ is a DS).

After an edge deletion, it may happen that $\tilde{\mathcal{D}}$ becomes disconnected and that the MST includes some internal vertices (at most 2, by the previous discussion) not in $\tilde{\mathcal{D}}$: in that case, we add them to $\mathcal{C}$. This turns out to be enough to ensure the connectivity.

To maintain the MST of the weighted version of the input graph we use the $O(\log^4 n)$ update time fully-dynamic MST algorithm from [22]. Since the weights of the edges incident to the vertices that enter or leave $\tilde{\mathcal{D}}$ are also updated, the algorithm runs in time $\widetilde{O}(\Delta)$ for each change in $\tilde{\mathcal{D}}$, i.e. in time $k \cdot \widetilde{O}(\Delta)$

Restore the connectivity when a vertex is deleted by the black box. When a vertex v is deleted from $\mathcal{D}$ by the black box DS algorithm, we need to be more careful: updating the edge weights and finding the new MST may add a lot of vertices to $\mathcal{C}$ (as many as Δ, one per edge of the MST incident to v). However, if the removal of v disconnects $G[\tilde{\mathcal{D}}]$, it is enough to add v to $\mathcal{C}$ to restore the connectivity. If its removal does not disconnect $G[\tilde{\mathcal{D}}]$, nothing needs to be done. It is possible to know if the graph $G[\tilde{\mathcal{D}}]$ gets disconnected using the properties of the MST, by only looking at the weight of the MST. The complexity of this step is therefore $\widetilde{O}(\Delta)$, the time needed to update the weights of the MST.

Restore the minimality. The second phase of the algorithm is to restore the minimality of $\mathcal{C}$. We explain next how to find the vertices of $\mathcal{C}$ that need to be removed to accomplish this task. This minimality condition is equivalent to the condition that all vertices in $\mathcal{C}$ are *articulation points* in the graph induced by $\mathcal{C} \cup D$. (An articulation point is a vertex such that its removal increases the number of connected components.) This turns out to be useful in order to identify which vertices need to be removed to restore the minimality of $\mathcal{C}$.

To restore the connectivity requirement, new vertices were added into $\mathcal{C}$, and the black box added some vertices to $\mathcal{D}$: this might result in some vertices in $\mathcal{C}$ not being articulation points of $G[\tilde{\mathcal{D}}]$ anymore. As observed before, these are the vertices that need to be removed. We need to identify a maximal set of such vertices that can be removed from $\mathcal{C}$ without violating the connectivity requirement. To do this, the algorithm queries in an arbitrary order one-by-one all the vertices $v \in \mathcal{C}$ to determine whether $G[\tilde{\mathcal{D}} \setminus v]$ is connected. This can be done using a data structure from Holm et al. [22] that requires $\widetilde{O}(1)$ per query. Whenever

the algorithm identifies a vertex such that $G[\tilde{\mathcal{D}} \setminus v]$ is connected, it can safely remove it from $\mathcal{C}$. The complexity of this step is therefore $\widetilde{O}(n)$ to find all articulation points, and an extra $\widetilde{O}(\Delta)$ for each of the vertices we remove from $\mathcal{C}$.

The following three lemmas conclude the proof: the first shows that the algorithm is correct, the second the $\widetilde{O}(n)$ time bound and the third the $O(\log n)$ approximation ratio.

▶ **Lemma 8.** *The algorithm that first restores the connectivity of $\mathcal{C} \cup \mathcal{D}$ and then the minimality of $\mathcal{C}$ is correct: it gives a minimal set $\mathcal{C}$ such that $\mathcal{C} \cup \mathcal{D}$ is connected.*

Proof. After restoring the connectivity requirement the algorithm maintains a spanning tree of $\tilde{\mathcal{D}}$, so $G[\tilde{\mathcal{D}}]$ is indeed connected. In the following steps, before the algorithm removes a vertex v from $\mathcal{C}$, it first verifies that $G[\tilde{\mathcal{D}} \setminus v]$ remains connected, which guarantees that $G[\tilde{\mathcal{D}}]$ is connected at the end of the update procedure. Since the black box ensures that $\mathcal{D}$ is a DS, $\tilde{\mathcal{D}}$ is a DS too: hence at the end, $\tilde{\mathcal{D}}$ satisfies both the domination and the connectivity requirements. It remains to show that $\mathcal{C}$ is minimal, i.e., that all vertices in $\mathcal{C}$ are articulation points in $G[\tilde{\mathcal{D}}]$. Since during the second step the algorithm only removes vertices from $\mathcal{C}$, a vertex that was not an articulation point cannot become one, and therefore the loop to find the articulation points is correct. The set $\mathcal{C}$ is therefore a minimal set such that $\mathcal{C} \cup \mathcal{D}$ is connected. ◀

▶ **Lemma 9.** *The amortized complexity of the algorithm is $\widetilde{O}(n)$ per update.*

Proof. The amortized cost of the black box to compute $\mathcal{D}$ is $\widetilde{O}(\Delta)$. We analyze now the additional cost of maintaining $\tilde{\mathcal{D}}$. As shown in this section, the cost to add or delete a vertex from $\tilde{\mathcal{D}}$ is $\widetilde{O}(\Delta)$. To prove the lemma, we bound the number of changes in $\tilde{\mathcal{D}}$. For that, we count the number of vertices *added* to $\tilde{\mathcal{D}}$: in an amortized sense this bounds the number of changes too. Formally, we pay a budget $deg(v)$ when v is added to $\tilde{\mathcal{D}}$. Following insertions and deletions of edges adjacent to v, we update this budget (with a constant cost), so that when v gets deleted from $\tilde{\mathcal{D}}$ a budget equal to its degree is available to spend.

From Corollary 6, the black box makes at most $\widetilde{O}(1)$ changes to $\mathcal{D}$ per update (in an amortized sense). If it removes a vertex from $\mathcal{D}$, we showed previously that no new vertex is added to $\tilde{\mathcal{D}}$. The number of additions to $\tilde{\mathcal{D}}$ is therefore $\widetilde{O}(1)$. Moreover, in the case of an edge deletion, at most two vertices are added to $\tilde{\mathcal{D}}$ to maintain the connectivity. Since restoring the minimality requires only to delete vertices, the total number of additions into $\tilde{\mathcal{D}}$ is $\widetilde{O}(1)$. As the cost for any of these additions is $\widetilde{O}(\Delta)$, the total cost of this algorithm is upper bounded by the loop to find the articulation points, which is $\widetilde{O}(n)$. ◀

▶ **Lemma 10.** *The algorithm maintains a $O(\log n)$ approximation for MCDS, i.e. $|\mathcal{C} \cup \mathcal{D}| = O(\log n) \cdot OPT$*

Proof. We first prove that $|\mathcal{C}| \leq 2|\mathcal{D}|$, using the minimality of $\mathcal{C}$. Each vertex of $\mathcal{C}$ is there to connect some components of $\mathcal{D}$. Consider the graph (W, F) where vertices W are either connected components of $\mathcal{D}$ or vertices of $\mathcal{C}$, and the set F of edges is constructed as follows. Start with a graph containing one vertex for each connected component of $\mathcal{D}$, and add vertices of $\mathcal{C}$ one by one. When the vertex v is added, identify a node u in $\mathcal{D}$ adjacent to v such that adding the edge (u, v) to F does not create a cycle: add to F an edge between v and the node corresponding to the connected component containing u. It is always possible to find such a vertex u, otherwise v would not be necessary for the connectivity, which would contradict the minimality of $\mathcal{C}$. This process gives a forest such that every node of $\mathcal{C}$ is adjacent to a connected component of $\mathcal{D}$. Since $\mathcal{C} \cup \mathcal{D}$ is connected, it is possible to complete F to make it a tree, adding some other edges. This tree has the two following properties.

1. The leaves are vertices that correspond to connected component of $\mathcal{D}$: indeed, if a vertex of $\mathcal{C}$ was a leaf in this tree, it could be removed without losing the connecting of $\mathcal{C} \cup \mathcal{D}$, which would contradict the minimality of $\mathcal{C}$.
2. Any vertex of $\mathcal{C}$ is adjacent to a connected component of $\mathcal{D}$, by construction of the forest.

These properties ensure that for every subtree rooted at a vertex of $\mathcal{C}$, there is a $\mathcal{D}$ vertex at distance at most 2 from the root: otherwise, the vertices at distance 1 from it would be from $\mathcal{C}$ and adjacent only to $\mathcal{C}$ vertices. Moreover, since a $\mathcal{C}$ vertex is not a leaf, it has necessarily some descendant and the reasoning applies. Therefore, by rooting the tree at an arbitrary vertex of $\mathcal{C}$, we can charge every $\mathcal{C}$ vertex to a $\mathcal{D}$ descendant at distance at most 2. As a $\mathcal{D}$ vertex can be charged only by an ancestor at most two levels above it, it is charged at most twice. This ensures that $|\mathcal{C}| \leq 2|\mathcal{D}|$.

Moreover, since $\mathcal{D}$ is a $O(\log n)$ approximation of MDS, $|\mathcal{D}| = O(\log n) \cdot \mathrm{OPT}$. Putting things together, we have $|\mathcal{C} \cup \mathcal{D}| = |\mathcal{C}| + |\mathcal{D}| = O(\log n) \cdot \mathrm{OPT}$. ◀

Combining Lemmas 8, 9 and 10 proves our claim: there is a $\widetilde{O}(n)$ algorithm to maintain a $O(\log n)$ approximation of the Minimum Connected Dominating Set. The main bottleneck of this approach is the time spent by the algorithm in the second phase to query all vertices in $\mathcal{C}$ in order to identify the vertices that are no longer articulation points. In the next section we present an algorithm that overcomes this limitation and is able to identify the necessary vertices more efficiently.

4 A more intricate $\widetilde{O}(\Delta)$ algorithm to restore the minimality of $\mathcal{C}$

In this section we present a more sophisticated algorithm for implementing the phase that guarantees the minimality of the maintained connected dominating set. This gives a proof of Theorem 3. We focus on a single edge update: indeed, when a vertex is added to (or removed from) $\tilde{\mathcal{D}}$, one can simply add (or remove) all its edges one by one. As in the analysis of the complexity in Lemma 9, the amortized number of changes in $\tilde{\mathcal{D}}$ is $\widetilde{O}(1)$. We aim now at proving that the time required for handling a single change is $\widetilde{O}(\Delta)$: for that, we treat edge insertions and deletions to $\tilde{\mathcal{D}}$ one by one, and prove that any edge update can be done in $\widetilde{O}(1)$, which would prove the claimed bound. Our algorithm maintains another spanning forest F of $G[\tilde{\mathcal{D}}]$ (unweighted) using the algorithm from [22].

▶ **Lemma 11.** *The vertices of $\mathcal{C}$ that are not articulation points after the insertion of the edge (v, w) all lie on the tree path $v...w$ of F. Moreover, the removal of any of these vertices results in the other vertices being articulation points again.*

Proof. Let G_b be the graph before the insertion of (v, w), and G_a be the one after. Let u be a vertex that is an articulation point in $G_b[\tilde{\mathcal{D}}]$ but not in $G_a[\tilde{\mathcal{D}}]$. Suppose by contradiction that u is not on the tree path $v...w$: that means that v and w are connected in $G_b[\tilde{\mathcal{D}}] \setminus \{u\}$. Since u is an articulation point in $G_b[\tilde{\mathcal{D}}]$, v is not connected to some vertex x in $G_b[\tilde{\mathcal{D}}] \setminus \{u\}$. But as v and w are connected in $G_b[\tilde{\mathcal{D}}] \setminus \{u\}$, adding the edge (v, w) does not connect v and x and therefore u is still an articulation point after the insertion of the edge. Therefore, all the articulation points that can be removed are in the cycle $v...w, v$. Since they are not articulation points in $G_a[\tilde{\mathcal{D}}]$, they separate $G_b[\tilde{\mathcal{D}}]$ in only two components: one with v, the other with w. Therefore, $v...w, v$ is the only cycle containing v and w, and removing any vertex from it make the articulation points of $G_b[\tilde{\mathcal{D}}]$ be articulations point in $G_a[\tilde{\mathcal{D}}]$, because they disconnect v and w again. ◀

Lemma 11 allows us to focus on the following problem: find a vertex in $\mathcal{C}$ that is no longer an articulation point in $G[\tilde{\mathcal{D}}]$ after the insertion of the edge (v, w). To achieve this, the algorithm maintains for each vertex $v \in \mathcal{C}$ the number $nc(v)$ of connected component of $G[\mathcal{D} \setminus v]$. For $v \notin \mathcal{C}$ we set for convenience $nc(v)$ to be the number of connected component in $G[\mathcal{D} \setminus v]$ plus n. This information can be used as follows: when an edge (v, w) is added, if for one vertex $u \in \mathcal{C}$ it holds $nc(u) = 1$ then u is removed from $\mathcal{C}$ (because it is no longer an articulation point). To identify such a vertex, the algorithm queries for the minimal value along the path $v...w$ in T: if the minimum value is 1, the corresponding vertex is removed from $\mathcal{C}$. This removal makes all the other vertices of the set $\mathcal{C}$ articulation points again: by Lemma 11, the cycle created by the insertion of (v, w) is broken by the deletion of u from $G[\tilde{\mathcal{D}}]$.

Notice that we are only interested in the $nc(v)$ values of the vertices in $\mathcal{C}$, as $nc(v) > n$ for $v \notin \mathcal{C}$. Since we compute a minimum and the values relevant are smaller than n, this is equivalent to ignoring v. The advantage of this offset is that when v becomes part of $\mathcal{C}$, it is sufficient to decrease its value by n to make it consistent. We now show how to keep this value up to date after adding or removing an edge.

Maintaining the $nc(v)$ values in a top-tree. For this purpose, we use the biconnectivity data structure from [22] (called *top-tree*) on the subgraph $G[\tilde{\mathcal{D}}]$. To avoid cumbersome notation, we pretend that we execute the algorithm on G, although the underlying graph on which we execute the algorithm is $G[\tilde{\mathcal{D}}]$. We also assume that the number of vertices remains n throughout the execution, which is simply implemented by removing from G all incident edges from the vertices with no incident edges in $G[\tilde{\mathcal{D}}]$.

We now briefly describe the approach of [22]. The algorithm maintains a spanning forest F of G and assigns a level $\ell(e)$ to each edge e of the graph. Let G_i be the graph composed of F and all edges of level at least i. The levels are attributed such that the following invariant is maintained:

▶ **Invariant 2.** *The maximal number of vertices in a biconnected component of G_i is $\lceil n/2^i \rceil$.*

Therefore the algorithm needs only to consider $\lceil \log_2 n \rceil$ levels. Whenever an edge (v, w) is deleted, one needs to find which vertices in the path $v...w$ in F are still biconnected. We use the following notion to describe the algorithm.

▶ **Definition 12.** *A vertex u is* covered *by a nontree edge (x, y) if it is contained in a tree cycle induced by (x, y). We say that a path $v...w$ is covered at level i if every of its node is in a tree cycle induced by an edge at level greater than i.*

Mark that all the vertices that are covered by a given edge are in the same biconnected component.

When a non-tree edge (v, w) is removed, it may affect the 2-edge connected components along the tree-path $v...w$ in T. To find which vertices are affected, the following algorithm is used in [22]. It first marks the vertices in $v...w$ as no longer covered at level $\ell(v, w)$. Then, it iterates over edges (x, y) that could cover $v...w$, i.e., the ones such that the intersection between $x...y$ and $v...w$ is not empty, and marks the vertices in this intersection as covered. This step is explained in the following function, which is called for all level i from $\ell(v, w)$ down to 0. $meet(v, w, x)$ is the intersection of the tree paths $v...w$, $v...x$ and $x...w$.

Recover(v, w, i). Set $u := v$, and iterate over the vertices of $v...w$ towards w. For each value of u, consider each nontree edge (q, r) with $meet(q, v, w) = u$ and such that $u...q$ is covered at level i. If it is possible without breaking Invariant 2, increase the level of (q, r)

to $i+1$ and mark the edges of $q...r$ covered at level $i+1$. Otherwise, mark them covered at level i and stop. If the phase stopped, start a second symmetric phase with $u = w$ and iterating on $w...v$ towards v.

As shown in [22], this is correct and runs in $O(\log^4)$ amortized time.

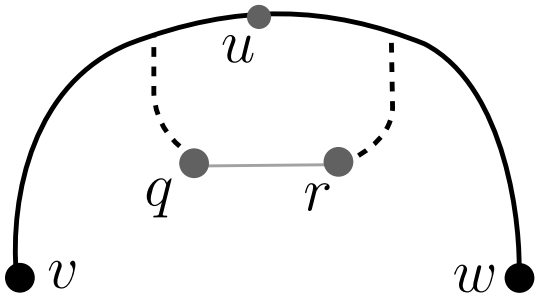

Figure 1 The edge (q, r) covers some node u on the path $v...w$.

In our case, we are interested in the vertices u whose value $nc(u)$ changes. They are exactly those that are still marked as not covered at the end of the process. Indeed, if an edge (q, r) covers a vertex u (see Figure 1), then v and w are still connected in $G[D \setminus u]$, hence the connected component of $G[D \setminus u]$ do not change. However, if u is not covered by any edge, then v and w gets disconnected in $G[D \setminus u]$, thus $nc(u)$ must be updated.

We maintain the $nc(\cdot)$ values in a top-tree, as follows. We call a *segment* a subpath of $v...w$. The idea is to maintain the non-covered segments and decrease the nc values along these at the end of the process. The top-trees allow us to alter the value of a segment of a path in $O(\text{polylog}n)$ time.

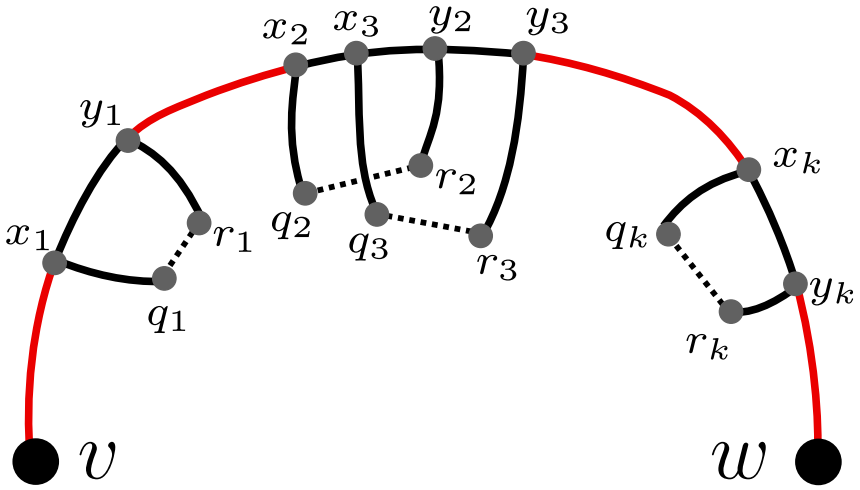

Figure 2 The black segments are covered by edges (q_i, r_i). The red segments are uncovered.

Computing the list of uncovered segments. To find the uncovered segments (in red on Figure 2), we sort the covered ones and take the complementary. Let $(q_1, r_1), ..., (q_k, r_k)$ be the nontree edges considered in the execution of Recover, and let $x_i = LCA(v, q_i)$ and $y_i = LCA(v, r_i)$ (where $LCA(u, v)$ is the lowest common ancestor of u and v in the tree). The covered segments are exactly the (x_i, y_i). Using lowest common ancestor queries, it is

possible to sort those segments according to the position of x_i along the path $v...w$. Given the segments in order, it is then possible to determine the uncovered segments in linear time: they correspond to the complementary of those segments. Answering a lowest common ancestor query on a dynamic tree can be done in $O(\log n)$ (see [29]), hence it is possible to sort the covered segments in time $O(k \log^2 n)$ and to find the uncovered segments with the same complexity.

Since k is the number of edges that move to a higher level during a call to Recover, and the maximum level is $\log n$, the total complexity of computing the uncovered segments is at most $\log^3 n$ per edges. Hence the overall complexity is $O(\log^4 n)$, which is the cost of the function Recover.

Adding an edge. To add an edge, two things are required: first decrease some nc value, and then query if a vertex has a nc value 1. We have to decrease the nc value of a vertex y if and only if its predecessor and its successor along the tree path $v...w$ were not connected in $\mathcal{D} \setminus \{y\}$ before the insertion of (v, w). This turns out to be equivalent to saying that y is not covered: thus, the algorithm needs to compute the list of segments along $v...w$ that were uncovered before the insertion of (v, w). It then must decrease the nc values along these segments, because they become connected. This is analogous to the case of an edge deletion: the latter can be used the following way. First add the edge (v, w) (and make updates to the data structure according to [22]), then delete it using the algorithm from the previous section, with the only difference that, instead of increasing the nc values along the uncovered segments, the algorithm decrease them.

It is then easy to find the minimum nc value along the path $v...w$, using the top-tree. If this value is 1, we can remove the corresponding vertex from $\mathcal{C}$. To remove it, we remove its incident edges one by one, each time updating the nc values of the remaining vertices.

The results of this section are summarized in the following lemma.

▶ **Lemma 13.** *After these updates, $\mathcal{C}$ is minimal. Moreover, the algorithm runs in amortized time $\widetilde{O}(1)$ for a single edge update.*

A direct corollary of this lemma and Lemma 9 is Theorem 3.

▶ **Corollary 14** (Theorem 3)**.** *The whole algorithm to maintain the Connected DS is correct and runs in time $\widetilde{O}(\Delta)$*

Proof. The correctness follows from Lemma 13 and from the correctness of the $\widetilde{O}(n)$ algorithm. As for the running time, the only difference from Lemma 9 is the search for articulation points: this takes $\widetilde{O}(1)$ for each edge added or removed from $\tilde{\mathcal{D}}$, and consequently $\widetilde{O}(\Delta)$ for each node added to or removed from $\tilde{\mathcal{D}}$. This yields that the algorithm takes $\widetilde{O}(\Delta)$ amortized time per update. ◀

5 A $O(\min(\Delta, \sqrt{m}))$ amortized algorithm for Minimal Dominating Set

This section presents a faster algorithm if one is only interested in finding a *Minimal* DS. This is a DS in which it is not possible to remove a vertex, but it can be arbitrarily big. For instance, in a star, the Minimum DS is only one vertex (the center), but its complementary is another minimal DS and has size $n - 1$. This result highlights the difference between MIS and Minimal DS: the best known *deterministic* complexity for MIS is $O(m^{2/3})$, whereas we present here a $O(\sqrt{m})$ algorithm for Minimal DS.

Key idea. When one needs to add a new vertex to the dominating set in order to dominate a vertex v, he can choose a vertex with degree $O(\sqrt{m})$, either v or one of its neighbors (a similar idea appears in Neiman et al. [27]). We present an algorithm with complexity proportional to the degree of the vertex added to the DS: this will give a $O(\min(\Delta, \sqrt{m}))$ algorithm. To analyze the complexity, we follow an argument similar to the one for CDS. At most one vertex is added to the DS at every step, even though several can be removed. Therefore we can pay for the (future) deletion of a vertex at the time it enters the DS.

For a vertex v, $N(v)$ is the set of its neighbors, including v. Let $\mathcal{D}$ be the dominating set maintained by the algorithm. If $v \in \mathcal{D}$ and $u \in N(v)$, we say that v *dominates* u.

For each vertex v, the algorithm keeps this sets up-to-date:

- let $N_{\mathcal{D}}(v)$ be the set of neighbors of v that are in the dominating set $\mathcal{D}$, i.e., $N_{\mathcal{D}}(v) = \mathcal{D} \cap N(v)$
- if $v \in \mathcal{D}$, let OnlyBy(v) be the set of neighbors of v that are dominated only by v, i.e., OnlyBy(v) $= \{u \in N(v) \mid |N_{\mathcal{D}}(u)| = 1\}$

Note that $N_{\mathcal{D}}(v)$ and OnlyBy(v) are useful to check, throughout any sequence of updates, whether a vertex v must be added to or removed from the current dominating set. In particular, if $N_{\mathcal{D}}(v) = \emptyset$ then v is not dominated by any other vertex, and thus it must be included in the dominating set. On the other hand, if OnlyBy(v) $= \emptyset$, all the neighbors of v (v included) are already dominated by some other vertex, and thus v could be removed from the dominating set.

5.1 The algorithm

We now show how to maintain a minimal dominating set $\mathcal{D}$ and the sets $N_{\mathcal{D}}(v)$ and OnlyBy(v), for each vertex v, under arbitrary sequences of edge insertions and deletions. We first describe two basic primitives, which will be used by our insertion and deletion algorithms: adding a vertex to and deleting a vertex from a dominating set $\mathcal{D}$.

Adding a vertex v to $\mathcal{D}$. Following some edge insertion or deletion, it may be necessary to add a vertex v to the current dominating set $\mathcal{D}$. In this case, we scan all its neighbors u and add v to the sets $N_{\mathcal{D}}(u)$. If before the update $N_{\mathcal{D}}(u)$ consisted of a single vertex, say w, we also have to remove u from the set OnlyBy(w), since now u is dominated by both v and w. If OnlyBy(w) becomes empty after this update, we remove w from $\mathcal{D}$ since it is no longer necessary in the dominating set.

Removing a vertex v from $\mathcal{D}$. When a vertex v is removed from the dominating set, we have to remove v from all the sets $N_{\mathcal{D}}(u)$ such that $u \in N(v)$. If after this update $N_{\mathcal{D}}(u)$ consists of a single vertex, say w, we add u to OnlyBy(w).

Edge insertion. Let (u, v) be an edge to be inserted in the graph. We distinguish three cases depending on whether u and v are in the dominating set $\mathcal{D}$ before the insertion. If neither of them is in the dominating set (i.e., $u \notin \mathcal{D}$ and $v \notin \mathcal{D}$), then nothing needs to be done. If both are in the dominating set (i.e., $u \in \mathcal{D}$ and $v \in \mathcal{D}$), then we start by adding v to the set $N_{\mathcal{D}}(u)$. If u was only necessary to dominate itself, we remove u from $\mathcal{D}$. Otherwise, we add u to $N_{\mathcal{D}}(v)$ and perform the same check on v.

If only one of them is in the dominating set (say, $u \notin \mathcal{D}$ and $v \in \mathcal{D}$), we have to add v to the set $N_{\mathcal{D}}(u)$. As in the case of adding a vertex to $\mathcal{D}$, this may cause the removal of another vertex from the dominating set. This can happen only if before the insertion, $N_{\mathcal{D}}(u) = \{w\}$

for some vertex w and $\text{OnlyBy}(w) = \{u\}$: in other terms, u was dominated only by w, and w was in the dominating set only to dominate u. Since after the addition of the edge (u, v) u is also dominated by v, w can be removed from the dominating set.

Edge deletion. Let (u, v) be the edge being deleted from the graph. We distinguish again the same three cases as before. If $u \notin \mathcal{D}$ and $v \notin \mathcal{D}$, nothing needs to be done. If both $u \in \mathcal{D}$ and $v \in \mathcal{D}$, we just have to remove u (resp. v) from the sets $N_{\mathcal{D}}(u)$ and $\text{OnlyBy}(u)$ (resp. $N_{\mathcal{D}}(v)$ and $\text{OnlyBy}(v)$).

If only one of them is in the dominating set, say $u \notin \mathcal{D}$ and $v \in \mathcal{D}$, then we have to remove v from $N_{\mathcal{D}}(u)$. Now, there are two different subcases:

- If $N_{\mathcal{D}}(u) \neq \{v\}$ before the deletion, then nothing needs to be done.
- Otherwise, we have to remove u from $\text{OnlyBy}(v)$: if $\text{OnlyBy}(v) = \emptyset$ after this operation, then we can safely remove v from $\mathcal{D}$. The algorithm must find a new vertex to dominate u: we simply add u to the dominating set.

5.2 Running time

Adding or removing a vertex v from the dominating set can be done in time $O(deg(v))$, where $deg(v)$ is the degree of v in the current graph. While several vertices can be removed from $\mathcal{D}$ at every step, only one can be added (following an edge deletion): the amortized complexity of the algorithm is therefore $O(\Delta)$, where Δ is an upper bound on the degree of the nodes.

Nevertheless, it is possible to chose the vertex to be added to the dominating set more carefully. When the algorithm must find a new vertex to dominate vertex u, it does the following:

- If $deg(u) \leq 2\sqrt{m} + 1$, the algorithm simply adds u to $\mathcal{D}$.
- Otherwise, $deg(u) > 2\sqrt{m} + 1$. The algorithms finds a vertex $w \in N(u)$ with $deg(w) \leq \sqrt{m}$ and adds w to $\mathcal{D}$. Note that such a vertex w can be found by simply scanning only $2\sqrt{m} + 1$ neighbors of u, since (by averaging) at least one of them must have degree smaller than $\sqrt{m}$.

In both cases, the insertion takes time $O(\min(\Delta, \sqrt{m}))$.

When a vertex v is deleted from the dominating set, its degree can be potentially larger than $2\sqrt{m}$. However, when v was added to the dominating set its degree must have been $O(\sqrt{m})$: this implies that many edges were added to v, and we can amortize the work over those edges. More precisely, when a vertex v enters the dominating set, we put a budget $deg(v)$ on it. Every time an edge incident to v is added to the graph, we increase by one this budget, so that when v has to be removed from $\mathcal{D}$, v has a budget larger than $deg(v)$ that can be used for the operation.

References

1 Raghavendra Addanki and Barna Saha. Fully Dynamic Set Cover–Improved and Simple. *arXiv preprint*, 2018. arXiv:1804.03197.

2 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, 2018.

3 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. *Fully Dynamic Maximal Independent Set with Sublinear in n Update Time*, pages 1919–1936. SIAM, 2019. doi:10.1137/1.9781611975482.116.

4 Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A Deamortization Approach for Dynamic Spanner and Dynamic Maximal Matching. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1899–1918. SIAM, 2019. doi:10.1137/1.9781611975482.115.

5 Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 692–711. Society for Industrial and Applied Mathematics, 2016.

6 Sivakumar R Bevan Das and V Bharghavan. Routing in ad-hoc networks using a virtual backbone. In *Proceedings of the 6th International Conference on Computer Communications and Networks (IC3N'97)*, pages 1–20, 1997.

7 Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–20. Society for Industrial and Applied Mathematics, 2018.

8 Sayan Bhattacharya, Monika Henzinger, and Giuseppe F Italiano. Deterministic fully dynamic data structures for vertex cover and matching. *SIAM Journal on Computing*, 47(3):859–887, 2018.

9 Sergiy Butenko, Xiuzhen Cheng, Carlos A Oliveira, and Panos M Pardalos. A new heuristic for the minimum connected dominating set problem on ad hoc wireless networks. In *Recent developments in cooperative control and optimization*, pages 61–73. Springer, 2004.

10 Xiuzhen Cheng, Xiao Huang, Deying Li, Weili Wu, and Ding-Zhu Du. A polynomial-time approximation scheme for the minimum-connected dominating set in ad hoc wireless networks. *Networks: An International Journal*, 42(4):202–208, 2003.

11 V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979. URL: http://www.jstor.org/stable/3689577.

12 Camil Demetrescu and Giuseppe F. Italiano. A New Approach to Dynamic All Pairs Shortest Paths. *J. ACM*, 51(6):968–992, 2004.

13 Ding-Zhu Du and Peng-Jun Wan. *Connected dominating set: theory and applications*, volume 77. Springer Science & Business Media, 2012.

14 Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

15 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

16 Sudipto Guha and Samir Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20(4):374–387, 1998.

17 Leonidas Guibas, Nikola Milosavljević, and Arik Motskin. Connected dominating sets on dynamic geometric graphs. *Computational Geometry*, 46(2):160–172, 2013.

18 Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 537–550. ACM, 2017.

19 Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for Maximal Independent Set and other problems. *arXiv preprint*, 2018. arXiv:1804.01823.

20 Manoj Gupta and Richard Peng. Fully dynamic (1+ e)-approximate matchings. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 548–557. IEEE, 2013.

21 Johan Håstad. Clique is hard to approximate withinn 1- ε. *Acta Mathematica*, 182(1):105–142, 1999.

22 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic Deterministic Fully-dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-edge, and Biconnectivity. *J. ACM*, 48(4):723–760, 2001.

23 Lujun Jia, Rajmohan Rajaraman, and Torsten Suel. An efficient distributed algorithm for constructing small dominating sets. *Distributed Computing*, 15(4):193–205, 2002.

24 Viggo Kann. *On the approximability of NP-complete optimization problems*. PhD thesis, Royal Institute of Technology Stockholm, 1992.

25 Fabian Kuhn and Roger Wattenhofer. Constant-time distributed dominating set approximation. *Distributed Computing*, 17(4):303–310, 2005.

26 D. Nanongkai, T. Saranurak, and C. Wulff-Nilsen. Dynamic Minimum Spanning Forest with Subpolynomial Worst-Case Update Time. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 950–961, October 2017.

27 Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Transactions on Algorithms (TALG)*, 12(1):7, 2016.

28 E Sampathkumar and HB Walikar. The connected domination number of a graph. *J. Math. Phys*, 1979.

29 Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

30 S. Solomon. Fully Dynamic Maximal Matching in Constant Update Time. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 325–334, October 2016. doi:10.1109/FOCS.2016.43.

31 Jie Wu and Hailan Li. On calculating connected dominating set for efficient routing in ad hoc wireless networks. In *Proceedings of the 3rd international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 7–14. ACM, 1999.