Parameterized Dynamic Data Structure for Split Completion

Konrad Majewski
Institute of Informatics, University of Warsaw, Poland

Michał Pilipczuk
Institute of Informatics, University of Warsaw, Poland

Anna Zych-Pawlewicz
Institute of Informatics, University of Warsaw, Poland

Abstract

We design a randomized data structure that, for a fully dynamic graph G updated by edge insertions and deletions and integers k, d fixed upon initialization, maintains the answer to the Split Completion problem: whether one can add k edges to G to obtain a split graph. The data structure can be initialized on an edgeless n-vertex graph in time $n \cdot (kd \cdot \log n)^{\mathcal{O}(1)}$, and the amortized time complexity of an update is $5^k \cdot (kd \cdot \log n)^{\mathcal{O}(1)}$. The answer provided by the data structure is correct with probability $1 - \mathcal{O}(n^{-d})$.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases parameterized complexity, dynamic data structures, split graphs

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.87

Related Version *Full Version*: https://arxiv.org/abs/2402.08816 [29]

Funding This work is a part of project BOBR that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 948057).

1 Introduction

In the field of *parameterized algorithms*, one measures resources used by an algorithm not only in terms of the total size of the considered instance, but also in terms of auxiliary quantitative measures associated with the instance, called *parameters*. Recently there is a growing interest in applying this principle to the area of *dynamic data structures*. In this context, we typically consider an instance I of a fixed problem of interest, with an associated parameter k. The instance is dynamic, in the sense that it is updated over time by problem-specific update operations, while for simplicity we assume that the parameter stays intact. The goal is to design a data structure that would maintain whether I is a yes-instance of the problem under the updates to I. Here we allow the update time to depend in any computable way on the parameter and sublinearly on the instance size; for instance, we are insterested in (possibly amortized) update times of the form $f(k)$, $f(k) \cdot (\log n)^{\mathcal{O}(1)}$, or $f(k) \cdot n^{o(1)}$, where f is some computable function and n is the size of I. Note that we allow f to be superpolynomial, so this framework may be applied even to NP-hard problems, as long as their static parameterized variants are fixed-parameter tractable.

Parameterized dynamic data structures were first systematically investigated by Iwata and Oka [23], followed by Alman et al. [1]. These works provided data structures with update times $f(k)$ or $f(k) \cdot (\log n)^{\mathcal{O}(1)}$ for several classic problems such as Vertex Cover, Cluster Vertex Deletion, Hitting Set, Feedback Vertex Set, or Longest Path. Other recent advances include data structures for maintaining various graph decompositions together

with runs of dynamic programming procedures [8, 13, 14, 26, 28], treatment of parameterized string problems from the dynamic perspective [33], and even an application of the framework in the context of timed automata [19].

A topic within parameterized algorithms that could be particularly productive from the point of view of dynamic data structures is that of *graph modification problems*. In this context, we fix a graph class $\mathcal{C}$ and a set of graph operations Π (e.g. vertex deletion, edge deletion, edge insertion), and consider the following parameterized problem: given a graph G and a parameter k, decide whether one can apply at most k operations from Π to G in order to obtain a graph belonging to $\mathcal{C}$. By instantiating different graph classes $\mathcal{C}$ and sets of operations Π, we obtain a wealth of parameterized problems with vastly different complexities, highly dependent on the combinatorics of the class $\mathcal{C}$ in question. The most widely studied are *vertex deletion problems* (only vertex deletions are allowed), *edge deletion* and *completion problems* (only edge deletions, respectively insertions, are allowed), and *editing problems* (both edge deletions and edge insertions are allowed).

Note that Vertex Cover, Feedback Vertex Set, and Cluster Vertex Deletion can be understood as vertex deletion problems, for the classes of edgeless, acyclic, and cluster graphs, respectively. Thus, the results of Iwata and Oka [23] and of Alman et al. [1] already give parameterized dynamic data structures for some basic graph modification problems, but the impressive volume of work on static parameterized algorithms for such problems suggests that there is much more to be explored.

In this work we focus on the case of *split graphs*; recall that a graph G is a split graph if the vertices of G can be partitioned into sets C and I so that C is a clique and I is an independent set. As proved by Földes and Hammer [17], split graphs are exactly graphs that exclude $2K_2$, C_4, and C_5 as induced subgraphs. Hence, a standard branching strategy solves Split Vertex Deletion in time $5^k \cdot n^{\mathcal{O}(1)}$ and Split Completion in time $5^k \cdot n^{\mathcal{O}(1)}$ (note that Split Edge Deletion is equivalent to Split Completion in the complement of the given graph). In fact, faster algorithms are known: Split Vertex Deletion can be solved in time $1.2738^k \cdot k^{\mathcal{O}(\log k)} + n^{\mathcal{O}(1)}$ [10], while Split Completion can be solved in subexponential parameterized time, more precisely in time $k^{\mathcal{O}(\sqrt{k})} \cdot n^{\mathcal{O}(1)}$, and admits a kernel with $\mathcal{O}(k^2)$ vertices [18]. Somewhat surprisingly, Split Editing is polynomial-time solvable [20], while both Split Vertex Deletion and Split Completion remain NP-hard [27, 32]. This makes graph modification problems related to split graphs very well understood in the static setting, and of remarkably low complexity, which suggests that they may serve as a suitable testbed for considerations from the point of view of dynamic data structures.

In fact, split graphs have already been considered in the dynamic setting. Ibarra [22] gave a dynamic data structure with constant update time for the membership problem: the data structure maintains whether the graph in question is a split graph. This can be understood as the treatment of (any) modification problem for parameter k equal to 0. Further aspects of dynamic maintenance of split graphs were investigated by Heggernes and Mancini [21].

Our contribution. In this work we propose a randomized dynamic data structure for the Split Completion problem whose amortized update time depends polylogarithmically on the size of the graph. Formally, we prove the following result.

▶ **Theorem 1.** *There is a randomized data structure that for a fully dynamic graph G, updated by edge deletions and edge insertions, and a parameter k fixed upon initialization, maintains the answer to the following query: can one add at most k edges to G to obtain a split graph. The data structure can be initialized on an edgeless n-vertex graph and an accuracy parameter $d \in \mathbb{N}$ in time $k^{\mathcal{O}(1)} \cdot d^2 \cdot n \cdot (\log n)^{\mathcal{O}(1)}$, and the amortized time complexity of updates is $5^k \cdot k^{\mathcal{O}(1)} \cdot d^2 \cdot (\log n)^{\mathcal{O}(1)}$. At all times, the answer provided by the data structure is correct with probability at least $1 - \mathcal{O}(n^{-d})$.*

Note that the data structure Theorem 1 offers no query methods, because the answer to the query – whether one can add k edges to G to obtain a split graph – is a single bit that is always stored explicitly in the data structure and can be retrieved in time $\mathcal{O}(1)$. We remark that by a tiny modification of the algorithm, we can also provide an analogous data structure for the dynamic Split Edge Deletion problem, with same complexity guarantees.

Theorem 1 fits within a wider context of dynamic data structures for graph modification problems. There are multiple open questions that can be asked in this area; we discuss some of them in Section 6.

Overview. Let us now discuss the main ideas behind the proof of Theorem 1. The basic approach is to try to dynamize the standard $5^k \cdot n^{\mathcal{O}(1)}$-time branching algorithm, which works as follows: recursively find an obstruction – an induced subgraph belonging to $\{2K_2, C_4, C_5\}$ – and branch into at most 5 different ways on how to destroy this obstruction by adding an edge. To implement this strategy in the dynamic setting, we need to have an efficient way of finding an obstruction in the current graph, or concluding that it is split. Indeed, if we get such a localization procedure, then the branching algorithm can be implemented using data structure's own methods for inserting and removing edges. A detailed description is presented in the full version of the paper [29].

The next observation is that if (G, k) is a Yes-instance of the Split Completion problem, then in particular it is also a Yes-instance of the Split Editing problem: one can obtain a split graph from G by adding or removing at most k edges. This is equivalent to the following statement: there is a partition (A, B) of the vertex set of G such that the number of edges with both endpoints in B plus the number of non-edges with both endpoints in A is at most k. For a partition (A, B), this number is called the *splittance* of (A, B). As observed by Hammer and Simeone [20], a partition with optimal splittance can be computed in polynomial time using a simple greedy argument. Ibarra [22] dynamized this observation and showed that a partition (A, B) with optimum splittance can be maintained with $\widetilde{\mathcal{O}}(1)$ worst-case update time. Moreover, upon every edge update, the data structure of Ibarra moves only $\mathcal{O}(1)$ vertices between the parts A and B.

Therefore, thanks to the result of Ibarra we may assume that a partition (A, B) with optimum splittance is available to us; this gives us a basic structural understanding of G. If the splittance of (A, B) is larger than k, then we are certain that (G, k) is a No-instance of Split Completion. Hence, from now on we will work in the *promise model*, where we assume that the maintained partition (A, B) has splittance bounded by k at all times (we refer to this property as the *promise*). Lifting the result from the promise model to the general setting is handled by a wrapper data structure that works as follows: while the splittance is larger than k, the new updates are put on a queue instead of directly implemented, and when the splittance becomes at most k again, all the enqued updates are implemented in one large batch. We remark that this is the only source of amortization in our data structure: in the promise model, the claimed running time guarantees are worst-case. The wrapper data structure, presented in Section 4, is solely responsible for veryfing the promise, collecting the updates on the queue when the promise ceases to hold, and applying the updates when the promise is satisfied again. We note that together with the partition (A, B), the wrapper data structure also maintains sets `nonEdgesA` and `edgesB` consisting of non-edges with both endpoints in A and edges with both endpoints in B, with a guarantee that they are correct whenever the splittance of (A, B) is at most k. Note that then $|\texttt{nonEdgesA}| + |\texttt{edgesB}| \leqslant k$.

The wrapper data structure relies on two *promise* data structures, whose correct performance is only guaranteed when the promise is satisfied. The promise data structures are described in Section 3. Their main functionality is to implement efficient access to edges and

non-edges crossing the partition (A, B). More precisely, in $\widetilde{\mathcal{O}}(k^{\mathcal{O}(1)})$-time, given $\ell \in \mathcal{O}(k)$, they are able to list:

- for a given $a \in A$, any set of ℓ neighbors of a in B; and
- for a given $b \in B$, any set of ℓ non-neighbors of b in A.

If a has fewer than ℓ neighbors in B, then the query should list all of them; similarly for b. As the second query is symmetric to the first one, we focus on the latter. First, consider the case when a has exactly one neighbor in B. Then to quickly recover this neighbor, we can maintain in the data structure the following information:

- the sum s_A of the identifiers of all the vertices in A; and
- for every vertex u, the sum s_u of the identifiers of all the neighbors of u.

Then given a, the identifier of the sole neighbor of a in B can be obtained by taking $s_a - s_A$ and adding the identifiers of all non-neighbors of a in A, which can be listed in time $\widetilde{\mathcal{O}}(k)$ using the set `nonEdgesA`. This trick can be lifted to listing all neighbors of a in B assuming that that their number is $\mathcal{O}(k)$ using the technique of *color coding* of Alon, Yuster, and Zwick [2]. Finally, to allow listing $\ell = \mathcal{O}(k)$ neighbors from possibly much larger neighborhoods, we sample in advance a polylogarithmically-sized family $\mathcal{S}$ of vertex subsets of varying sizes so that for any possible neighborhood N, with high probability among the sampled sets there will be some $S \in \mathcal{S}$ such that $\ell \leqslant |N \cap S| \leqslant \mathcal{O}(\ell)$. Then we apply the ideas presented above to list the members of $N \cap S$, whose number is already suitably bounded. We remark that this final trick is the only element of the reasoning that we do not know how to derandomize.

Once the promise data structures and the wrapper data structure are at hand, the last step is to implement the procedure for finding an obstruction. Here, the main idea is that every obstruction present in G must contain a non-edge with both endpoints in A or an edge with both endpoints in B; for otherwise it would be a split graph. For these (non-)edges, we have at most k candidates contained in the lists `nonEdgesA` and `edgesB`. Therefore, while searching for an obstruction, we can immediately "anchor" two of its vertices by guessing a member of `nonEdgesA` $\cup$ `edgesB` that is contained in the obstruction. Localizing the other two or three vertices of the obstruction requires a skillful juggle of the functionality specified above, in particular the methods for enumeration of (non-)neighbors across the partition (A, B). This part of the argument can be found in Section 5.

2 Preliminaries

For a positive real number x, we define $[x] = \{1, 2, \ldots, \lceil x \rceil\}$. For two sets A and B, we denote their symmetric difference by $A \triangle B = (A \setminus B) \cup (B \setminus A)$. Given any sets A, B, C we write $C = A \uplus B$ if $C = A \cup B$, and $A \cap B = \emptyset$. For a boolean condition b, we use its indicator function $\mathbb{1}_b$, defined as $\mathbb{1}_b = 1$ if b is true, and $\mathbb{1}_b = 0$ otherwise. We use the $\widetilde{\mathcal{O}}(\cdot)$ notation to hide polylogarithmic factors; that is, $\widetilde{\mathcal{O}}(f(n))$ is the class of all functions $g(n)$ that are upper-bounded by $c \cdot f(n) \cdot (\log n)^c$ for some constant c.

All graphs considered in this paper are undirected and simple. For a graph G, we denote the sets of vertices and of edges of G by $V(G)$ and $E(G)$, respectively. For convenience our data structures assume that $V(G) = [n]$ for some natural number n. We refer to an edge $\{u, v\} \in E(G)$ as uv, for every pair of vertices u and v. For a vertex $v \in V(G)$, we denote the degree of v by $d_G(v)$ and the set of all neighbors of v by $N_G(v)$ (we may skip index G if it is clear from the context). Additionally, we define $\overline{N}_G(v)$ as the set of non-neighbors of v, that is, $\overline{N}_G(v) = V(G) \setminus (N_G(v) \cup \{v\})$. For a subset $A \subseteq V(G)$, we denote by $G[A]$ the subgraph of G induced by A. For a graph F, we say that G is F-free if it does not contain an induced subgraph isomorphic to F. Similarly, for a family of graphs $\mathcal{F}$, a graph G is $\mathcal{F}$-free if it is F-free for every $F \in \mathcal{F}$.

We say that G is *split* if its vertices can be partitioned into two sets A and B such that $G[A]$ is a clique, and $G[B]$ is an independent set. For any partition (A, B) of $V(G)$, we denote

$$\mathsf{splittance}_G(A,B) \coloneqq \binom{|A|}{2} - |E(G[A])| + |E(G[B])|,$$

that is, $\mathsf{splittance}_G(A, B)$ counts the number of non-edges of G within A and edges of G within B. For the whole graph G, we write

$$\mathsf{splittance}(G) \coloneqq \min_{(A,B)\colon V(G)=A\uplus B} \mathsf{splittance}_G(A,B).$$

One can observe that $\mathsf{splittance}(G)$ counts the minimal number of edge updates (insertions or deletions) that need to be done on G in order to make G split. In particular, G is split if and only if $\mathsf{splittance}(G) = 0$. A partition with optimum splittance can be computed greedily by taking a prefix of the vertex set ordered by decreasing degrees, as stated by Hammer and Simeone [20].

▶ **Fact 2** ([20]). *Let G be a graph on n vertices, and let $d_1 \geqslant d_2 \geqslant \ldots \geqslant d_n$ be a sorted sequence of the vertex degrees of G. Define $m \coloneqq \max\{i \in [n] \mid d_i \geqslant i-1\}$. Then,*

$$\mathsf{splittance}(G) = m(m-1) - \sum_{i=1}^{m} d_i + \sum_{i=m+1}^{n} d_i,$$

and a certifying partition (A, B) of $V(G)$ can be obtained by taking the vertices of G corresponding to the degrees $(d_1, \ldots, d_m)$ and $(d_{m+1}, \ldots, d_n)$, respectively.

Ibarra [22] gives a dynamic data structure that maintains whether a dynamic graph G is split. Lemma 3 describes a slight adaptation of the data structure of Ibarra.

▶ **Lemma 3.** *There exists a data structure $\mathsf{DSplit}[n]$ that runs on a dynamic graph $G = (V, E)$ on n vertices, and maintains a partition (A, B) of its vertices satisfying $\mathsf{splittance}_G(A, B) = \mathsf{splittance}(G)$. The data structure supports the following operations:*

- `initialize`(n)*: fixes the set of vertices $V \coloneqq [n]$ for the entire run, sets the initial graph G to an edgeless graph $G \coloneqq (V, \emptyset)$ and sets the initial partition (A, B) to $(A, B) \coloneqq (\emptyset, V)$. Runs in time $\widetilde{\mathcal{O}}(n)$.*
- `update`(uv)*: inserts edge uv if $uv \notin E(G)$ or removes edge uv if $uv \in E(G)$. Let $G' \coloneqq (V, E(G) \triangle \{uv\})$ be the updated graph and (A', B') be the updated partition for G'. The method returns a set $V^{moved} \subseteq V$ of size $|V^{moved}| \leqslant \mathcal{O}(1)$ such that*

  $$(A', B') = (A \triangle V^{moved}, B \triangle V^{moved}).$$

  The running time is $\widetilde{\mathcal{O}}(1)$.
- `splittance`$()$*: returns the current value of $\mathsf{splittance}_G(A, B) = \mathsf{splittance}(G)$ in time $\mathcal{O}(1)$.*

Other than by splittance, we can also characterize split graphs via forbidden subgraphs.

▶ **Fact 4** ([17]). *A graph G is split if and only if G is $\mathcal{F}$-free, where $\mathcal{F} = \{2K_2, C_4, C_5\}$.*

Computation. We assume the standard word RAM model of computation with machine words of length $\mathcal{O}(\log n)$. We use *sets* and *dictionaries* assuming standard implementations using self-balancing binary search trees, which offer all operations in worst-case time $\mathcal{O}(\log n) \subseteq \widetilde{\mathcal{O}}(1)$. In particular, graphs are represented as dictionaries mapping vertices to the sets of their neighbors. Most of our data structures are randomized. We stress that the random choices happen only during the initialization, and therefore the events of incorrectness of further queries are *not* independent – they all depend on the initial randomness.

3 Listing (non-)edges across the partition

Let $G = (A \uplus B, E)$ be a dynamic graph whose vertices are partitioned into two sets A and B which might change over time. In this section, we devise two auxiliary data structures that are responsible for listing all neighbors of a given $a \in A$ in the set B, provided this neighborhood is small, or sampling a sufficient number of neighbors of a in B. Symmetric queries can be also given for non-neighborhoods in A of vertices $b \in B$.

Both data structures work under the promise that $\mathsf{splittance}_G(A, B) \leqslant k$. This promise will be lifted in the next section, where we design a wrapper data structure that works also when $\mathsf{splittance}_G(A, B) > k$. To facilitate the design of this wrapper data structure, both data structures presented in this section need to be able to process updates in batches: instead of being given only single edge updates, they will be given sets of edge modifications such that after applying all of them the condition $\mathsf{splittance}_G(A, B) \leqslant k$ is preserved.

Also, both data structures are assumed to have access to sets `nonEdgesA`, `edgesB` consisting of non-edges with both endpoints in A and edges with both endpoints in B, respectively. This access will be provided by the wrapper data structure. We start by providing the first data structure, called PromiseNL (standing for neighbors listing).

▶ **Lemma 5.** *There is a data structure* $\mathsf{PromiseNL}[n, k, \ell, d, S]$*, parametrized by integers* $k, \ell, d \in \mathbb{N}$ *and a subset* $S \subseteq V(G)$*, that runs on a dynamic graph* $G = (V, E)$ *with* $V = [n]$*, and maintains a partition* (A, B) *of vertices of* G *by supporting the following operations:*

- `initialize`(n, k, ℓ, d, S)*: sets* $V := [n]$*, fixes the values of* k*,* ℓ*,* d *and* S *for the entire run, initializes* $G := (V, \emptyset)$ *and* $(A, B) := (\emptyset, V)$*. Runs in time* $\widetilde{\mathcal{O}}(\ell d \cdot n)$*.*
- `update`$(V^{moved}, E^{mod}, \texttt{nonEdgesA}, \texttt{edgesB})$*: modifies the edge set of the underlying graph to* $E' := E \triangle E^{mod}$ *and modifes the partition to* $(A', B') := (A \triangle V^{moved}, B \triangle V^{moved})$*, provided that after the update* $\mathsf{splittance}_{G'}(A', B') \leqslant k$ *for the modified graph* $G' = (V, E')$*. Requires two lists of edges* `nonEdgesA`, `edgesB` $\subseteq \binom{V}{2}$ *storing the sets of all non-edges of* $G'[A']$ *and all edges of* $G'[B']$*, respectively. Note that both sets are of size at most* k *since* $\mathsf{splittance}_{G'}(A', B') \leqslant k$*. The running time is* $\widetilde{\mathcal{O}}(k + \ell d \cdot (|V^{moved}| + |E^{mod}|))$*.*

Moreover, the data structure can answer the following queries:

- `listNeighborsBS`(a)*: given a vertex* $a \in A$*, either returns the set*

$$N := N_G(a) \cap B \cap S$$

provided $|N| \leqslant \ell$*, or reports* TooMany *if* $|N| > \ell$*.*

- `listNonNeighborsAS`(b)*: given a vertex* $b \in B$*, either returns the set*

$$N := \overline{N}_G(b) \cap A \cap S$$

provided $|N| \leqslant \ell$*, or reports* TooMany *if* $|N| > \ell$*.*

Both queries run in time $\widetilde{\mathcal{O}}(k\ell d)$ *and return correct answers with probability* $1 - \mathcal{O}(n^{-d})$*.*

Proof. Let γ be some large enough constant whose value will be set later. We assume that $\ell \leqslant n$, for otherwise we might set $\ell := n$, and this would not affect answers to the queries. We also denote the sum of the vertex indices in a set $X \subseteq V$ as $\texttt{sum}(X) = \sum_{v \in X} v$. The data structure $\mathsf{PromiseNL}[n, k, \ell, d, S]$ stores:

- the adjacency list representation of G and representation of A and B as sets;
- the integer values of k, ℓ, d and $S \subseteq V(G)$ represented as a set;
- the sets of edges $\texttt{nonEdgesA}, \texttt{edgesB} \subseteq \binom{V(G)}{2}$ provided in the updates, as lists;
- for every $i \in [\gamma d \log n]$, a vertex ℓ-coloring $\chi_i \colon V(G) \to [\ell]$. Define $S_{i,c} := S \cap \chi_i^{-1}(c)$, for every $i \in [\gamma d \log n]$ and $c \in [\ell]$; these sets are also stored in the data structure;
- the following integer values:
  $\texttt{countS}(v) = |N_G(v) \cap S|$ for every $v \in V(G)$;
  $\texttt{countAS} = |A \cap S|$; $\texttt{countBS} = |B \cap S|$;
  $\texttt{idSumS}_{i,c}(v) = \texttt{sum}(N_G(v) \cap S_{i,c})$ for every $i \in [\gamma d \log n], c \in [\ell]$ and $v \in V(G)$;
  $\texttt{idSumAS}_{i,c} = \texttt{sum}(A \cap S_{i,c})$ for every $i \in [\gamma d \log n]$ and $c \in [\ell]$;
  $\texttt{idSumBS}_{i,c} = \texttt{sum}(B \cap S_{i,c})$ for every $i \in [\gamma d \log n]$ and $c \in [\ell]$.

Before proceeding, let us give a rough idea on how these variables are used. Consider a vertex $a \in A$. Using the variables `count-` and the set `nonEdgesA`, one can obtain the number of neighbors of a in the set $B \cap S$ by applying a simple inclusion-exclusion formula. If this neighborhood N is of size at most ℓ, we need to list all the vertices of N. We use the vertex colorings χ_i and the corresponding variables `idSum-`. Each coloring χ_i is initialized randomly. Therefore, with high probability, for every vertex $v \in N$, there is a coloring χ_i such that under χ_i, v has a different color than all the other vertices of $N \setminus \{v\}$. By applying a similar inclusion-exclusion principle on the variables `idSum-` we can retrieve an identifier of v.

Initialization.

- We initialize the adjacency representation of G to store an edgless graph on vertices $[n]$. We initialize $A = \emptyset$ and $B = V$. Also, we set the lists $\texttt{nonEdgesA}, \texttt{edgesB}$ to be empty.
- We store the values of k, ℓ, d and S as a set.
- All colorings χ_i are initialized randomly, that is, for every $i \in [\gamma d \log n]$ and $v \in V(G)$, we choose a color for $\chi_i(v)$ uniformly at random from the set $[\ell]$. The colorings are fixed for the entire run as well, and so are sets $S_{i,c}$.
- Since the initial graph is empty, all the values `count-` and `idSum-` can be computed according to their definitions in total time $\widetilde{\mathcal{O}}(\ell d \cdot n)$.

Update. Consider an update $\texttt{update}(V^{\text{moved}}, E^{\text{mod}}, \texttt{nonEdgesA}, \texttt{edgesB})$. First, we save the new values of $\texttt{nonEdgesA}$ and $\texttt{edgesB}$. Observe that the functions $\texttt{countS}(\cdot)$ and $\texttt{idSumS}_{i,c}(\cdot)$ need to be changed only for the endpoints of the edges of E^{mod}, and the remaining $\widetilde{\mathcal{O}}(\ell d)$ stored variables are affected only by vertices moved between A and B. Therefore, the entire update takes $\widetilde{\mathcal{O}}(k + \ell d \cdot (|V^{\text{moved}}| + |E^{\text{mod}}|))$ time.

Queries. We show how to answer the query $\texttt{listNeighborsBS}(a, \texttt{nonEdgesA})$ for a vertex $a \in A$. Recall that since $\mathsf{splittance}_G(A, B) \leqslant k$, there are at most k non-edges within A, i.e., $|\texttt{nonEdgesA}| \leqslant k$. Let $N := N_G(a) \cap B \cap S$. We start with the following observation.

▷ Claim 6. One can compute the value of $|N|$ in time $\widetilde{\mathcal{O}}(k)$.

Proof. The following equalities hold:

$$\begin{aligned}|N| = |N_G(a) \cap B \cap S| &= |N_G(a) \cap S| - |N_G(a) \cap A \cap S| = \texttt{countS}(a) - |N_G(a) \cap A \cap S| \\ &= \texttt{countS}(a) - (|A \cap S| - \mathbb{1}_{a \in S} - |\overline{N}_G(a) \cap A \cap S|) \\ &= \texttt{countS}(a) - (\texttt{countAS} - \mathbb{1}_{a \in S} - |\overline{N}_G(a) \cap A \cap S|).\end{aligned}$$

* Observe that $\overline{N}_G(a) \cap A \cap S \subseteq \overline{N}_G(a) \cap A$, and we can obtain all the non-neighborhood of a in A by iterating over the given set `nonEdgesA` of size at most k. Thus, we can compute the value of $|\overline{N}_G(a) \cap A \cap S|$ in time $\widetilde{\mathcal{O}}(k)$. ◁

Now, as a first step to answer `listNeighborsBS`(a, `nonEdgesA`) query, we compute $|N|$. If $|N| > \ell$, we report TooMany. So assume that $|N| \leqslant \ell$. For a vertex $b \in N$ and a coloring χ_i, let us say that b is *exposed* by the coloring χ_i if $\chi_i(b) \neq \chi_i(b')$ for every $b' \in N \setminus \{b\}$. First, we note that every member of N is exposed with high probability. (Proofs of statements marked with (♠) can be found in the full version of the paper [29].)

▷ Claim 7 (♠). With probability at least $1 - n^{-d}$, the following event holds: For every $b \in N$, there is an index $i \in [\gamma d \log n]$ such that b is exposed by χ_i.

Next, we note that once a vertex of N is exposed, its identifier can be easily computed using a similar inclusion-exclusion principle as in the proof of Claim 6.

▷ Claim 8 (♠). Let $i \in [\gamma d \log n]$ be an index and $c \in [\ell]$ be a color such that there is a vertex b exposed by χ_i with color $\chi_i(b) = c$. Then, given c and i, one can retrieve the identifier of b in time $\widetilde{\mathcal{O}}(k)$.

We already have all the tools to finish the implementation of the query `listNeighborsBS`(a). To list all the elements of N, it is enough to iterate over all indices $i \in [\gamma d \log n]$ and colors $c \in [\ell]$, and for each pair (i, c) retrieve the value that is described in Claim 8 and check whether the returned value is a vertex that belongs to N. By Claim 7, this procedure will retrieve the entire set N with probability at least $1 - n^{-d}$ and the total running time is bounded by $\widetilde{\mathcal{O}}(k\ell d)$.

To answer the queries `listNonNeighborsAS`(b, `edgesB`) for $b \in B$ we proceed analogously. To this end, we need additionally the values of:

- $|\overline{N}_G(b) \cap S|$ which can be obtained from the values of $|N_G(b) \cap S|$ and $|S|$; and
- `sum`$(\overline{N}_G(b) \cap S_{i,c})$ which we can obtain from `sum`$(N_G(b) \cap S_{i,c})$ and `sum`$(S_{i,c})$, where the latter sums can be precomputed during the initialization. ◀

The application of color coding presented in Claim 7 and Claim 8 can be derandomized in a standard way, see e.g. [9, Section 5.6]. So far, we can list ℓ neighbors of a vertex $a \in A$ in the set $B \cap S$ only under an assumption that this neighborhood N is small. However, this condition does not need to hold in general, and in the next sections we will also need to find several neighbors of a in B (more precisely, a subset of N of size $\Theta(k)$) in the case when the set N is large. This is the responsibility of our next data structure, PromiseNS (standing for neighbors sampling). Roughly speaking, this data structure will store a set of layers of the vertex set, where the ith layer contains roughly $n/2^i$ random vertices of G. Then, provided $|N| > \ell$, with high probability one of the layers will contain $\Theta(\ell)$ vertices of N. By running an instance of PromiseNL on each layer, we will be able to retrieve those vertices.

▶ **Lemma 9.** *There exists a data structure* PromiseNS$[n, k, \ell, d]$*, parametrized by integers $k, \ell, d \in \mathbb{N}$, that runs on a dynamic graph $G = (V, E)$ with $V = [n]$, and maintains a dynamic partition (A, B) of vertices of G by supporting the following operations:*

- `initialize`(n, k, ℓ, d)*: sets $V := [n]$, fixes the values of k, ℓ and d for the entire run, initializes $G := (V, \emptyset)$ and $(A, B) := (\emptyset, V)$. Runs in time $\widetilde{\mathcal{O}}(\ell d^2 \cdot n)$.*
- `update`$(V^{moved}, E^{mod}, \texttt{nonEdgesA}, \texttt{edgesB})$*: modifies the edge set of the underlying graph to $E' := E \triangle E^{mod}$ and modifes the partition to $(A', B') := (A \triangle V^{moved}, B \triangle V^{moved})$, provided that after the update* splittance$_{G'}(A', B') \leqslant k$ *for the modified graph $G' = (V, E')$.*

Requires two lists of edges $\texttt{nonEdgesA}, \texttt{edgesB} \subseteq \binom{V}{2}$ *storing the sets of all non-edges of* $G'[A']$ *and all edges of* $G'[B']$*, respectively. Note that both sets are of size at most* k *since* $\mathsf{splittance}_{G'}(A', B') \leqslant k$*. The running time is* $\widetilde{\mathcal{O}}(kd + \ell d^2 \cdot (|V^{moved}| + |E^{mod}|))$.

Moreover, the data structure can answer the following queries:

- $\texttt{sampleEdges}(a)$*: given a vertex* $a \in A$*, returns a subset*

$$N_B^{sample}(a) \subseteq N_G(a) \cap B$$

of size $\min(\ell, |N_G(a) \cap B|)$*;*

- $\texttt{sampleNonEdges}(b)$*: given a vertex* $b \in B$*, returns a subset*

$$\overline{N}_A^{sample}(b) \subseteq \overline{N}_G(b) \cap A$$

of size $\min(\ell, |\overline{N}_G(b) \cap A|)$.

Both queries run in time $\widetilde{\mathcal{O}}(k\ell d^2)$ *and return correct answers with probability* $1 - \mathcal{O}(n^{-d})$.

Proof. The data structure $\mathsf{PromiseNS}[n, k, \ell, d]$ stores:

- the adjacency lists representation of G and representation of A and B as sets;
- the values of the parameters k, ℓ, and d;
- the sets $\texttt{nonEdgesA}, \texttt{edgesB} \subseteq \binom{V(G)}{2}$ given in the updates, as lists;
- a single instance $\mathsf{PromiseNL}[n, k, \ell', d, S]$ of the data structure from Lemma 5 running on the same dynamic graph G, where $\ell' := 72\ell$ and $S := V(G)$;
- for every $i \in [\log n]$ and $j \in [d \log n]$, a subset of vertices $V_{i,j} \subseteq V(G)$, fixed upon initialization and not modified later on; and
- for every $i \in [\log n]$ and $j \in [d \log n]$, an instance $\mathsf{PromiseNL}_{i,j}[n, k, \ell', d, S_{i,j}]$ of the data structure from Lemma 5 running on the same dynamic graph G, where $S_{i,j} := V_{i,j}$.

Initialization. Intuitively, we want the set $V_{i,j}$ to be a random subset containing roughly $n/2^i$ vertices of G. To achieve this, for every $i \in [\log n]$ and $j \in [d \log n]$, and every vertex $v \in V(G)$, we put v into the set $V_{i,j}$ with probability 2^{-i}, independently. Then, we initialize all the data structures $\mathsf{PromiseNL}$ and $\mathsf{PromiseNL}_{i,j}$. The total running time is bounded by:

$$\log n \cdot d \log n \cdot \widetilde{\mathcal{O}}(\ell' d \cdot n) \subseteq \widetilde{\mathcal{O}}(\ell d^2 \cdot n).$$

Update and Sampling. Due to space limitations, the detailed description of updates and sampling is presented in the full version of the paper [29]. As for sampling, the idea is to query all the data structures $\mathsf{PromiseNL}_{i,j}$ and argue, using Chernoff's inequality, that at least one of those data structures returns at least ℓ (non-)neighbors of the vertex in question. ◀

4 Lifting the promise

In the previous section we presented data structures that work under the promise that $\mathsf{splittance}_G(A, B) \leqslant k$. Now, we focus on the general case when no such condition is guaranteed to hold. The main responsibility of our next data structure, $\mathsf{Wrapper}[n, k, d]$, is to implement the same functionality as data structures $\mathsf{PromiseNL}$ and $\mathsf{PromiseNS}$, but to work without the assumption that $\mathsf{splittance}_G(A, B) \leqslant k$ and, additionally, to offer methods for listing non-edges with both endpoints in A and edges with both endpoints in B (these were needed for queries in the data structures $\mathsf{PromiseNL}$ and $\mathsf{PromiseNS}$). All those methods are required to give correct outputs only when the inequality $\mathsf{splittance}_G(A, B) \leqslant k$ holds, but the data structure should persist also when $\mathsf{splittance}_G(A, B) > k$, and return to giving correct answers again once the splittance decreases to at most k.

▶ **Lemma 10.** *There exists a data structure* Wrapper$[n, k, d]$ *parametrized by integers* $k, d \in \mathbb{N}$ *that runs on a dynamic graph* $G = (V, E)$ *with* $V = [n]$ *and maintains a partition* (A, B) *of* V *satisfying* $\mathsf{splittance}_G(A, B) = \mathsf{splittance}(G)$ *by supporting the following operations:*

- `initialize`(n, k, d)*: sets* $V \coloneqq [n]$*, fixes the values of* k *and* d *for the entire run, initalizes* $G \coloneqq (V, \emptyset)$ *and* $(A, B) \coloneqq (\emptyset, V)$*. Runs in time* $\widetilde{\mathcal{O}}(kd^2 \cdot n)$*.*
- `update`(uv)*: inserts edge* uv *if* $uv \notin E(G)$ *or removes edge* uv *if* $uv \in E(G)$*. Runs in* $\widetilde{\mathcal{O}}(kd(k + d))$ *amortized time.*
- `splittance`$()$*: returns* $\mathsf{splittance}(G)$ *in time* $\mathcal{O}(1)$*.*

Moreover, if $\mathsf{splittance}(G) \leqslant k$*, the data structure answers the following queries:*

- `listNonEdgesA`$()$*: returns the list of all non-edges of* G *with both endpoints in* A*;*
- `listEdgesB`$()$*: returns the list of all edges of* G *with both endpoints in* B*;*
- `sampleEdges`(a)*: given a vertex* $a \in A$*, returns a subset*

$$N_B^{sample}(a) \subseteq N_G(a) \cap B$$

of size $\min(10k, |N_G(a) \cap B|)$*;*

- `sampleNonEdges`(b)*: given a vertex* $b \in B$*, returns a subset*

$$\overline{N}_A^{sample}(b) \subseteq \overline{N}_G(b) \cap A$$

of size $\min(10k, |\overline{N}_G(b) \cap A|)$*.*

The queries `listNonEdgesA` *and* `listEdgesB` *run in time* $\widetilde{\mathcal{O}}(k)$*, and the queries* `sampleEdges` *and* `sampleNonEdges` *run in time* $\widetilde{\mathcal{O}}(k^2d^2)$*. The data structure is randomized and returns correct answers with probability* $1 - \mathcal{O}(n^{-d})$*.*

Proof. The data structure Wrapper$[n, k, d]$ stores:

- the adjacency list representation of G and representation of A and B as sets;
- the values of k and d;
- an instance DSplit$[n]$ of the data structure from Lemma 3 running on G;
- an instance PromiseNL$[n, k, \ell, d', S]$ of the data structure from Lemma 5, where $\ell = k$, $d' = d + 3$ and $S = V(G)$;
- an instance PromiseNS$[n, k, \ell', d]$ of the data structure from Lemma 9, where $\ell' = 10k$;
- a set `verticesUpd` $\subseteq V(G)$ and sets `edgesUpd`, `edgesB`, `nonEdgesA` $\subseteq \binom{V(G)}{2}$ described in the next paragraph.

Let $G_0, G_1, \ldots, G_t$ be a sequence of consecutive graphs maintained by our data structure, that is, for every $i = 0, 1, \ldots, t - 1$, G_{i+1} is obtained from G_i by adding or removing a single edge. Moreover, let (A_i, B_i) be the corresponding partition of $V(G_i)$ maintained by the data structure, and let E_i be the set of edges of G_i, for $i = 0, \ldots, t - 1$. Define $s \coloneqq \max\{0 \leqslant i \leqslant t \mid \mathsf{splittance}(G_i) \leqslant k\}$. Note that s is well-defined as the initial graph G_0 is a split graph. Then, sets `verticesUpd`, `edgesUpd`, `edgesB`, `nonEdgesA` are defined by the following invariants that must be satisfied at all times:

- (*edges*) `edgesB` is the set of all edges of the graph $G_s[B_s]$, and `nonEdgesA` is the set of all non-edges of the graph $G_s[A_s]$, both with probability $1 - \mathcal{O}(n^{-d})$;
- (*updates*) `verticesUpd` $= A_s \triangle A_t = B_s \triangle B_t$, and `edgesUpd` $= E_s \triangle E_t$;
- (*promise*) the data structures PromiseNL and PromiseNS store the graph G_s with the partition (A_s, B_s) of its vertices.

In other words, sets `edgesB` and `nonEdgesA` store the answers to the queries `listEdgesB`() and `listNonEdgesA`() that were correct at the last time step when the splittance was bounded by k. Sets `verticesUpd` and `edgesUpd` store all the updates that got accumulated since

this time step. Also, note that the data structures PromiseNL and PromiseNS run on the graph $G_s = (A_s \uplus B_s, E_s)$, not G_t, as they require that the stored graph G and its partition (A, B) satisfy $\mathsf{splittance}_G(A, B) \leqslant k$. Let us observe at this point that one can assume that the inequality $s \leqslant n^2$ holds. This comes from the fact that after n^2 updates, we can recompute the correct values of `edgesB` and `nonEdgesA` for G_s (and renumber $G_0 := G_s$) which amortizes to $\widetilde{\mathcal{O}}(1)$ cost per update. The initialization and query procedures of Wrapper are straightforward, we refer the reader to the full version of the paper [29] for details.

Update. Consider an operation `update`(uv), where uv is an edge to be inserted or deleted. First, we pass the same update to the inner structure DSplit which returns a subset of vertices V^{moved} of size $\mathcal{O}(1)$ which need to be moved between A and B. Then, we can update the sets A and B, accordingly. Next, we update sets `verticesUpd` and `edgesUpd` as follows:

$$\texttt{verticesUpd}' = \texttt{verticesUpd} \triangle V^{\text{moved}} \quad \text{and} \quad \texttt{edgesUpd}' = \texttt{edgesUpd} \triangle \{e\}.$$

Let $G_s, G_{s+1}, \ldots, G_t = G$ be the suffix of the sequence of consecutive graphs given to the data structure, where s is the previous moment our dynamic graph G was of splittance at most k. If $\mathsf{splittance}(G_t) > k$, we can finish the update as all the invariants (*edges*), (*updates*) and (*promise*) hold. The running time is $\widetilde{\mathcal{O}}(1)$.

Now, assume that $\mathsf{splittance}(G_t) \leqslant k$. Then $\max\{0 \leqslant i \leqslant t \mid \mathsf{splittance}(G_i) \leqslant k\} = t$, and we need to perform additional modifications so that the invariants (*edges*), (*updates*) and (*promise*) hold. Let $r := t - s$ be the number of updates that were applied since the last time our graph was of splittance at most k. Our goal now is to recompute the sets `edgesB` and `nonEdgesA`, so that the invariant (*edges*) holds, in time proportional to r. Note the following.

▷ Claim 11 (♠). It holds that $|A_s \cap B_t| \leqslant \mathcal{O}(\sqrt{r+k})$ and $|A_t \cap B_s| \leqslant \mathcal{O}(\sqrt{r+k})$.

Now, let us focus on updating the set `edgesB`, so that after the update it reflects time step t instead of s (the set `nonEdgesA` is handled analogously). In other words, we need to locate all the edges of G_t with both endpoints in B_t. Denote this new set by `edgesB`$'$. All such edges can be classified into the following three types (see Figure 1 for illustration). In the following, we use variables `edgesB` and `nonEdgesA` as they are in the data structure before updating, i.e., their content refers to the time step s.

Type (1): Edges b_1b_2, where $b_1, b_2 \in B_s \cap B_t$.

Observe that every edge in G_t of this form is either

- an edge of $G_s[B_s]$, but there were at most k such ones as $\mathsf{splittance}(G_s) \leqslant k$; or
- an edge inserted during an update after time step s, but there are at most r such ones.

So using the sets `edgesB` and `edgesUpd` we can enumerate all such edges in time $\widetilde{\mathcal{O}}(k + r)$.

Type (2): Edges a_1a_2, where $a_1, a_2 \in A_s \cap B_t$.

We can list the set $A_s \cap B_t \subseteq$ `verticesUpd` in time $\widetilde{\mathcal{O}}(r)$, and by Claim 11 we can check all the pairs of vertices of $A_s \cap B_t$ in time $\widetilde{\mathcal{O}}(r + k)$.

Type (3): Edges a_1b_1, where $a_1 \in A_s \cap B_t$ and $b_1 \in B_s \cap B_t$.

To list all such edges we start with applying on PromiseNL a temporary modification by calling

$$\texttt{update}(\emptyset,\ E^{\text{mod}},\ \texttt{nonEdgesA},\ \texttt{edgesB}),$$

where $E^{\text{mod}} = E_1^{\text{mod}} \cup E_2^{\text{mod}}$ consists of:

- E_1^{mod}: all edges of `edgesUpd` of the form ab, where $a \in A_s \cap B_t$ and $b \in B_s \cap B_t$; there are at most r such edges.

- E_2^{mod}: all edges of the graph G_s of the form ab, where $a \in A_s \cap B_t$ and $b \in B_s \cap A_t$; we can list these edges brutally in time $\widetilde{\mathcal{O}}(r+k)$ by Claim 11.

That is, structure PromiseNL works now on the graph $G^{\mathrm{tmp}} = (A_s \uplus B_s, E^{\mathrm{tmp}})$ such that

- the graphs $G^{\mathrm{tmp}}[A_s]$ and $G^{\mathrm{tmp}}[B_s]$ are isomorphic $G_s[A_s]$ and $G_s[B_s]$, respectively,
- there are no edges between $A_s \cap B_t$ and $A_t \cap B_s$, and
- the edges between $A_s \cap B_t$ and $B_s \cap B_t$ coincide in the graphs G^{tmp} and G_t.

The cost of this temporary update is $\widetilde{\mathcal{O}}(k+\ell d'(k+r)) = \widetilde{\mathcal{O}}(kd(k+r))$. Observe that `edgesB` and `nonEdgesA` store currently respectively the edges within B_s and non-edges within A_s in G^{tmp}, and thus $\mathsf{splittance}_{G^{\mathrm{tmp}}}(A_s, B_s) \leqslant k$, as required by the PromiseNL data structure.

Now, we iterate over all vertices $a \in A_s \cap B_t$; recall that their number is bounded by r as they are all contained in `verticesUpd`. Recall that $\mathsf{splittance}_{G_t}(A_t, B_t) \leqslant k$, hence a has at most k neighbors in $B_s \cap B_t$, and we would like to list them all. However, this can be done precisely by calling `listNeighborsBS`(a) on PromiseNL. The total cost of such queries for all $a_1 \in A_s \cap B_t$ is

$$\widetilde{\mathcal{O}}(|A_s \cap B_t| \cdot k\ell d') \leqslant \widetilde{\mathcal{O}}(rk^2 d).$$

Finally, we revert PromiseNL to the state before the temporary update, by calling

`update`$(\emptyset,\ E^{\mathrm{mod}},$ `nonEdgesA`, `edgesB`$)$.

Summing up, the amortized cost of computing the set `edgesB`$'$ (and `nonEdgesA`$'$) is upper-bounded by

$$\frac{\widetilde{\mathcal{O}}(k+r) + \widetilde{\mathcal{O}}(r+k) + \widetilde{\mathcal{O}}(kd(k+r)) + \widetilde{\mathcal{O}}(rk^2 d)}{r} \subseteq \widetilde{\mathcal{O}}(k^2 d).$$

Now, we bound the probability of incorrectly computing the sets `edgesB`$'$ and `nonEdgesA`$'$. In the process of recomputation we do at most $\mathcal{O}(n)$ queries to PromiseNL, each having an error probability $\mathcal{O}(n^{-d'}) = \mathcal{O}(n^{-(d+3)})$. Hence, with probabilty $1 - \mathcal{O}(n^{d+2})$ all these queries return correct answers. Furthermore, we rely on the fact that the sets `edgesB` and `nonEdgesA` were computed correctly for the graph G_s. Recall that we can assume $s \leqslant n^2$, and thus by the union bound these sets were computed correctly for all the graphs $G_0, \ldots, G_t$ with probability at least

$$1 - n^2 \cdot \mathcal{O}(n^{-(d+2)}) = 1 - \mathcal{O}(n^{-d}).$$

Next, we update PromiseNL and PromiseNS so that their inner states correspond to the graph G_t. We run on both structures

`update`$($`verticesUpd`, `edgesUpd`, `nonEdgesA`$'$, `edgesB`$'$$)$.

Then, (*promise*) holds, and the amortized cost of these updates is

$$\frac{\widetilde{\mathcal{O}}(k + r\ell d') + \widetilde{\mathcal{O}}(kd + r\ell' d^2)}{r} \subseteq \widetilde{\mathcal{O}}(kd^2).$$

Finally, we set `verticesUpd`$' := \emptyset$ and `edgesUpd`$' := \emptyset$, so that invariant (*updates*) holds. ◀

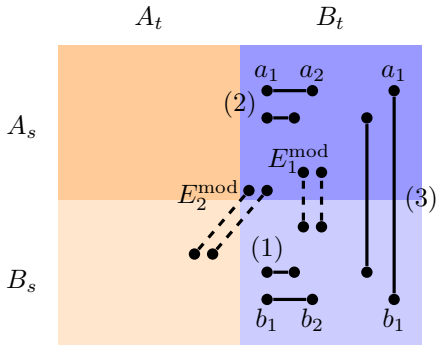

Figure 1 Edge types needed for recomputing `edgesB` in the proof of Lemma 10.

5 Localizing the obstructions

Let us recall that the graph is split if and only if it is $\mathcal{F}$-free, where $\mathcal{F} = \{2K_2, C_4, C_5\}$. We call the members of $\mathcal{F}$ simply *obstructions*. In this section, using the interface of the data structure $\mathsf{Wrapper}[n, k, d]$ from Lemma 10, we show how to extend this data structure with a method that is able to either conclude that G is split, or report an obstruction in G witnessing that it is not split. The following statement encapsulates this result.

▶ **Lemma 12.** *The data structure* $\mathsf{Wrapper}[n, k, d]$ *of Lemma 10 can be extended by a method* `findObstruction`() *which returns Split if* $\mathsf{splittance}(G) = 0$*; or a subset of vertices* $U \subseteq V(G)$ *such that the induced subgraph* $G[U]$ *is isomorphic to one of the graphs of* $\mathcal{F} = \{2K_2, C_4, C_5\}$ *if* $0 < \mathsf{splittance}(G) \leqslant k$.

The query runs in time $\widetilde{\mathcal{O}}(k^{\mathcal{O}(1)} \cdot d^2)$ *and is correct with probability* $1 - \mathcal{O}(n^{c-d})$ *for some constant* c *whenever* $\mathsf{splittance}(G) \leqslant k$*; without this assumption, there are no guarantees on the correctness.*

Proof. We assume that $\mathsf{splittance}(G) = \mathsf{splittance}_G(A, B) \leqslant k$, otherwise we can provide any answer. In what follows, we use extensively the following two observations.

▷ Claim 13 (♠). Suppose $a_1, a_2 \in A$ and $a_1a_2 \notin E(G)$. Then one of the following holds:
- there is a subset $U \subseteq V(G)$ such that $\{a_1, a_2\} \subseteq U$ and $G[U]$ is isomorphic to $2K_2$ or C_4;
- at least one of the vertices a_1, a_2 has at most $3\sqrt{k}$ neighbors in B.

Moreover, in time $\widetilde{\mathcal{O}}(k^2d^2)$ we can find out which of the above two cases holds, and return either a set U in the first case, or the neighborhood of a_1 or a_2 in B in the second case. The answer is correct with probability at least $1 - \mathcal{O}(n^{-d})$.

The second observation is symmetric to the first one. We include the statement for clarity.

▷ Claim 14. Let $b_1, b_2 \in B$ be such that $b_1b_2 \in E(G)$. Then at least one of the following holds:
- there is a subset $U \subseteq V(G)$ such that $\{b_1, b_2\} \subseteq U$ and $G[U]$ is isomorphic to $2K_2$ or C_4;
- at least one of the vertices b_1, b_2 has at most $3\sqrt{k}$ non-neighbors in A.

Moreover, in time $\widetilde{\mathcal{O}}(k^2d^2)$ we can return a case that holds, together with a witnessing set U for the first case, or the non-neighborhood of b_1 or b_2 in A for the second case. The answer is correct with probability at least $1 - \mathcal{O}(n^{-d})$.

We proceed with the description of the `findObstruction`($\cdot$) query. Suppose that there is a set $U \subseteq V(G)$ such that $G[U]$ is an obstruction. We examine all three cases for $G[U] \in \{2K_2, C_4, C_5\}$, and in each case we provide a procedure that, assuming the existence

of U as above, detects some set U' (possibly different from U) that induces an obstruction. If no case yields an obstruction, we conclude that G is split. Recall that by running `listNonEdgesA`() and `listEdgesB`(), we have access to the sets `nonEdgesA` containing all non-edges within A and `edgesB` containing all edges within B, and each of these sets has cardinality at most k. For brevity, a set is *small* if its size is at most $3\sqrt{k}$. Due to space limit, we focus here on the case $G[U] = 2K_2$, and present the remaining cases in the full version of the paper [29]. So suppose that $G[U] = G[\{x, y, z, t\}]$ is isomorphic to $2K_2$, where $xy, zt \in E(G)$. We consider all possible alignments of vertices x, y, z, t with respect to the partition (A, B). We split the analysis into five cases reflecting the intersection $U \cap A$ (see Figure 2).

(a) $|U \cap A| = 4$. Then, in particular, xt and yz are two non-edges within A. Hence, U can be detected by exhaustive search through all the pairs of non-edges in the set `nonEdgesA`.

(b) $|U \cap A| = 3$. Without loss of generality assume that $U \cap A = \{x, y, z\}$. Since xz and yz are two non-edges with both endpoints in A we can list all $\mathcal{O}(k^2)$ candidates for (x, y, z) by inspecting all the pairs of non-edges in `nonEdgesA`. Having (x, y, z) fixed, it remains to locate (any) vertex $t' \in B$ that is a neighbor of z and a non-neighbor of x and y (the original vertex $t \in U$ witnesses that such t' exists). We call `sampleEdges`(z) on Wrapper to obtain a set $N_B^{\text{sample}}(z)$ consisting of up to $10k$ neighbors of z in B. If $|N_B^{\text{sample}}(z)| < 10k$, then $N_B^{\text{sample}}(z)$ contains all neighbors of z in B, and we can test all of them for a vertex t'. So assume now that $|N_B^{\text{sample}}(z)| = 10k$. Then, we run the subroutine from Claim 13 for pairs (x, z) and (y, z). If any of these calls detects and returns an obstruction, we are done. Otherwise, since z has more than $3\sqrt{k}$ neighbors in B, both x and y must have at most $3\sqrt{k}$ neighbors in B and the subroutine returns the corresponding sets. Denote them by $N_B(x)$ and $N_B(y)$. Since

$$|N_B^{\text{sample}}(z)| = 10k > 3\sqrt{k} + 3\sqrt{k} \geqslant |N_B(x)| + |N_B(y)|,$$

there exists a vertex $t' \in N_B^{\text{sample}}(z) \setminus (N_B(x) \cup N_B(y))$. Then x, y, z, t' induce a $2K_2$.

(c) $|U \cap A| = 2$ and the vertices of $U \cap A$ are adjacent in G. Without loss of generality assume that $U \cap A = \{x, y\}$ and $U \cap B = \{z, t\}$. Then zt is an edge with both endpoints in B, hence using the set `edgesB` we can list $\mathcal{O}(k)$ candidates for (z, t) and thus fix (z, t). Next, we run the subroutine from Claim 14 for (z, t), and assume that it did not detect an obstruction (for otherwise we are done). Then, the subroutine discovers that one of z or t (say z) has small non-neighborhood $\overline{N}_A(z)$ in A. Since $x, y \in \overline{N}_A(z)$, it is enough to check at most $\binom{3\sqrt{k}}{2} = \mathcal{O}(k)$ candidates for (x, y).

Due to the space constraints, we present the remaining cases in the full version of the paper [29].

This finishes the case study. Let us analyze the running time of `findObstruction`() query. It issues single calls `listNonEdgesA`() and `listEdgesB`(), $k^{\mathcal{O}(1)}$ calls `sampleEdges`$(\cdot)$ and `sampleNonEdges`$(\cdot)$ (either directly, or via the calls to the subroutines from Claim 13 and Claim 14), and spends $\widetilde{\mathcal{O}}(k^{\mathcal{O}(1)})$ time on internal computation. Therefore, the total running time of `findObstruction`() is bounded by

$$k^{\mathcal{O}(1)} \cdot \widetilde{\mathcal{O}}(k^2 d^2) + \widetilde{\mathcal{O}}(k^{\mathcal{O}(1)}) \subseteq \widetilde{\mathcal{O}}(k^{\mathcal{O}(1)} \cdot d^2).$$

Further, as we may assume that $k \leqslant n^2$, the probability of incorrectly answering the query is:

$$\mathcal{O}\left(n^{-d}\right) + \mathcal{O}(k^{\mathcal{O}(1)}) \cdot \mathcal{O}\left(n^{-d}\right) = \mathcal{O}\left(n^{\mathcal{O}(1)} \cdot n^{-d}\right) \subseteq \mathcal{O}\left(n^{c-d}\right)$$

for some fixed constant c. ◀

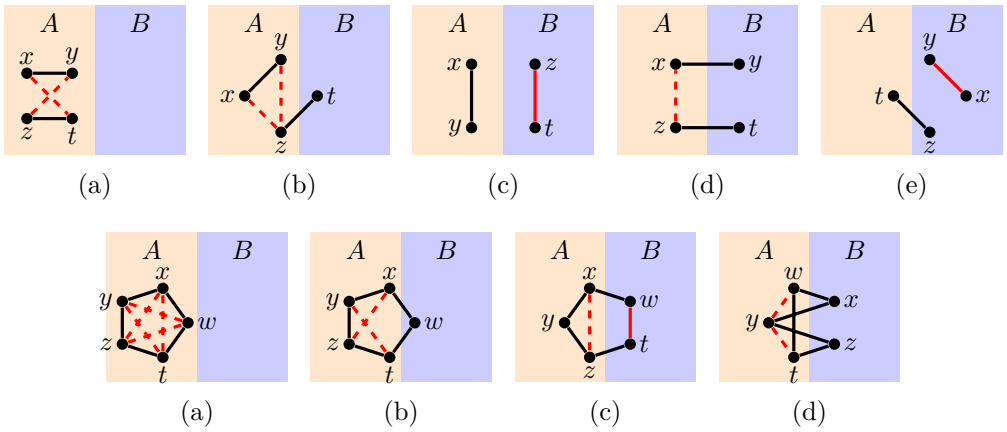

Figure 2 Non-symmetric cases in $2K_2$ (top row) and C_5 (bottom row) localization. Edges and non-edges (dashed) marked in red are present on the lists `edgesB` and `nonEdgesA`, respectively, hence they can be selected among $\mathcal{O}(k)$ candidates and fixed.

With Lemma 12 established, the proof of Theorem 1 follows: Assuming access to Wrapper extended as in Lemma 12, upon each query one simply executes the standard branching algorithm for Split Completion, using method `findObstruction`() for finding consecutive obstructions and `update`$(\cdot)$ for modifying the graph on the fly during backtracking. Details can be found in the full version of the paper [29].

6 Conclusions

In this work we gave a dynamic data structure for the Split Completion problem with amortized update time $2^{\mathcal{O}(k)} \cdot (d \log n)^{\mathcal{O}(1)}$, where the error probability is bounded by $1 - \mathcal{O}(n^{-d})$. Our result together with the previous work of Iwata and Oka [23] and of Alman et al. [1] suggest that investigating parameterized dynamic data structures for graph modification problems might turn out to be a particularly fruitful direction. Concrete problems that we suspect to be amenable to dynamization are Planarization [24, 31] and various graph modification problems connected to chordal graphs and their subclasses such as interval, proper interval, trivially perfect, or threshold graphs; these were studied extensively, see e.g. [3–7, 11, 12, 15, 16, 25, 30, 34, 35].

References

1 Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams. Dynamic parameterized problems and algorithms. *ACM Trans. Algorithms*, 16(4):45:1–45:46, 2020. doi:10.1145/3395037.

2 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995. doi:10.1145/210332.210337.

3 Ivan Bliznets, Fedor V. Fomin, Marcin Pilipczuk, and Michał Pilipczuk. A subexponential parameterized algorithm for Proper Interval Completion. *SIAM J. Discret. Math.*, 29(4):1961–1987, 2015. doi:10.1137/140988565.

4 Ivan Bliznets, Fedor V. Fomin, Marcin Pilipczuk, and Michał Pilipczuk. Subexponential parameterized algorithm for Interval Completion. *ACM Trans. Algorithms*, 14(3):35:1–35:62, 2018. doi:10.1145/3186896.

5 Yixin Cao. Linear recognition of almost interval graphs. In *27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 1096–1115. SIAM, 2016. doi:10.1137/1.9781611974331.ch77.

6 Yixin Cao. Unit interval editing is fixed-parameter tractable. *Inf. Comput.*, 253:109–126, 2017. doi:10.1016/j.ic.2017.01.008.

7 Yixin Cao and Dániel Marx. Chordal editing is fixed-parameter tractable. *Algorithmica*, 75(1):118–137, 2016. doi:10.1007/s00453-015-0014-x.

8 Jiehua Chen, Wojciech Czerwiński, Yann Disser, Andreas Emil Feldmann, Danny Hermelin, Wojciech Nadara, Marcin Pilipczuk, Michał Pilipczuk, Manuel Sorge, Bartłomiej Wróblewski, and Anna Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. In *2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 796–809. SIAM, 2021. doi:10.1137/1.9781611976465.50.

9 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.

10 Marek Cygan and Marcin Pilipczuk. Split Vertex Deletion meets Vertex Cover: New fixed-parameter and exact exponential-time algorithms. *Inf. Process. Lett.*, 113(5-6):179–182, 2013. doi:10.1016/j.ipl.2013.01.001.

11 Pål Grønås Drange, Fedor V. Fomin, Michał Pilipczuk, and Yngve Villanger. Exploring the subexponential complexity of completion problems. *ACM Trans. Comput. Theory*, 7(4):14:1–14:38, 2015. doi:10.1145/2799640.

12 Pål Grønås Drange and Michal Pilipczuk. A polynomial kernel for Trivially Perfect Editing. *Algorithmica*, 80(12):3481–3524, 2018. doi:10.1007/s00453-017-0401-6.

13 Zdenek Dvořák, Martin Kupec, and Vojtech Tůma. A dynamic data structure for MSO properties in graphs with bounded tree-depth. In *22th Annual European Symposium on Algorithms, ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 2014. doi:10.1007/978-3-662-44777-2_28.

14 Zdenek Dvořák and Vojtech Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In *13th International Symposium on Algorithms and Data Structures, WADS 2013*, volume 8037 of *Lecture Notes in Computer Science*, pages 304–315. Springer, 2013. doi:10.1007/978-3-642-40104-6_27.

15 Fedor V. Fomin, Saket Saurabh, and Yngve Villanger. A polynomial kernel for Proper Interval Vertex Deletion. *SIAM J. Discret. Math.*, 27(4):1964–1976, 2013. doi:10.1137/12089051X.

16 Fedor V. Fomin and Yngve Villanger. Subexponential parameterized algorithm for Minimum Fill-in. *SIAM J. Comput.*, 42(6):2197–2216, 2013. doi:10.1137/11085390X.

17 Stéphane Földes and Peter L. Hammer. Split graphs. In *Eighth Southeastern Conference on Combinatorics, Graph Theory and Computing*, volume XIX of *Congressus Numerantium*, pages 311–315, 1977.

18 Esha Ghosh, Sudeshna Kolay, Mrinal Kumar, Pranabendu Misra, Fahad Panolan, Ashutosh Rai, and M. S. Ramanujan. Faster parameterized algorithms for deletion to split graphs. *Algorithmica*, 71(4):989–1006, 2015. doi:10.1007/s00453-013-9837-5.

19 Alejandro Grez, Filip Mazowiecki, Michał Pilipczuk, Gabriele Puppis, and Cristian Riveros. Dynamic data structures for timed automata acceptance. *Algorithmica*, 84(11):3223–3245, 2022. doi:10.1007/s00453-022-01025-8.

20 Peter L. Hammer and Bruno Simeone. The splittance of a graph. *Combinatorica*, 1:275–284, 1981.

21 Pinar Heggernes and Federico Mancini. Dynamically maintaining split graphs. *Discret. Appl. Math.*, 157(9):2057–2069, 2009. doi:10.1016/J.DAM.2008.06.028.

22 Louis Ibarra. Fully dynamic algorithms for chordal graphs and split graphs. *ACM Trans. Algorithms*, 4(4):40:1–40:20, 2008. doi:10.1145/1383369.1383371.

23 Yoichi Iwata and Keigo Oka. Fast dynamic graph algorithms for parameterized problems. In *14th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2014*, volume

8503 of *Lecture Notes in Computer Science*, pages 241–252. Springer, 2014. doi:10.1007/978-3-319-08404-6_21.

24 Bart M. P. Jansen, Daniel Lokshtanov, and Saket Saurabh. A near-optimal planarization algorithm. In *25th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*, pages 1802–1811. SIAM, 2014. doi:10.1137/1.9781611973402.130.

25 Yuping Ke, Yixin Cao, Xiating Ouyang, Wenjun Li, and Jianxin Wang. Unit interval vertex deletion: Fewer vertices are relevant. *J. Comput. Syst. Sci.*, 95:109–121, 2018. doi:10.1016/j.jcss.2018.01.001.

26 Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michał Pilipczuk, and Marek Sokołowski. Dynamic treewidth. *CoRR*, abs/2304.01744, 2023. To appear in the proceedings of FOCS 2023. doi:10.48550/arXiv.2304.01744.

27 John M. Lewis and Mihalis Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *J. Comput. Syst. Sci.*, 20(2):219–230, 1980. doi:10.1016/0022-0000(80)90060-4.

28 Konrad Majewski, Michał Pilipczuk, and Marek Sokołowski. Maintaining CMSO_2 properties on dynamic structures with bounded feedback vertex number. In *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023*, volume 254 of *LIPIcs*, pages 46:1–46:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.STACS.2023.46.

29 Konrad Majewski, Michal Pilipczuk, and Anna Zych-Pawlewicz. Parameterized dynamic data structure for Split Completion. *CoRR*, abs/2402.08816, 2024. arXiv:2402.08816, doi:10.48550/arXiv.2402.08816.

30 Dániel Marx. Chordal deletion is fixed-parameter tractable. *Algorithmica*, 57(4):747–768, 2010. doi:10.1007/s00453-008-9233-8.

31 Dániel Marx and Ildikó Schlotter. Obtaining a planar graph by vertex deletion. *Algorithmica*, 62(3-4):807–822, 2012. doi:10.1007/s00453-010-9484-z.

32 Assaf Natanzon, Ron Shamir, and Roded Sharan. Complexity classification of some edge modification problems. *Discret. Appl. Math.*, 113(1):109–128, 2001. doi:10.1016/S0166-218X(00)00391-7.

33 Jędrzej Olkowski, Michał Pilipczuk, Mateusz Rychlicki, Karol Węgrzycki, and Anna Zych-Pawlewicz. Dynamic data structures for parameterized string problems. In *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023*, volume 254 of *LIPIcs*, pages 50:1–50:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.STACS.2023.50.

34 Pim van 't Hof and Yngve Villanger. Proper interval vertex deletion. *Algorithmica*, 65(4):845–867, 2013. doi:10.1007/s00453-012-9661-3.

35 Yngve Villanger, Pinar Heggernes, Christophe Paul, and Jan Arne Telle. Interval Completion is fixed parameter tractable. *SIAM J. Comput.*, 38(5):2007–2020, 2009. doi:10.1137/070710913.