



Good r -Divisions Imply Optimal Amortized Decremental Biconnectivity

Jacob Holm  

University of Copenhagen, Denmark

Eva Rotenberg  

Technical University of Denmark, Lyngby, Denmark

Abstract

We present a data structure that, given a graph G of n vertices and m edges, and a suitable pair of nested r -divisions of G , preprocesses G in $O(m + n)$ time and handles any series of edge-deletions in $O(m)$ total time while answering queries to pairwise biconnectivity in worst-case $O(1)$ time. In case the vertices are not biconnected, the data structure can return a cutvertex separating them in worst-case $O(1)$ time.

As an immediate consequence, this gives optimal amortized decremental biconnectivity, 2-edge connectivity, and connectivity for large classes of graphs, including planar graphs and other minor free graphs.

2012 ACM Subject Classification Theory of computation \rightarrow Dynamic graph algorithms

Keywords and phrases Dynamic graphs, 2-connectivity, graph minors, r -divisions, graph separators

Digital Object Identifier 10.4230/LIPIcs.STACS.2021.42

Related Version *Preliminary Full Version*: <https://arxiv.org/abs/1808.02568> [26]

Funding *Jacob Holm*: Partially supported by the VILLUM Foundation grant 16582, “BARC”.

Eva Rotenberg: Partially supported by Independent Research Fund Denmark grants 2020-2023 (9131-00044B) “Dynamic Network Analysis” and 2018-2021 (8021-00249B), “AlgoGraph”, and the VILLUM Foundation grant 37507 “Efficient Recomputations for Changeful Problems”.

Acknowledgements We are thankful to Adam Karczmarz and Jakub Łącki for their encouragement and interest in this work.

1 Introduction

Dynamic graph problems concern maintaining information about a graph, as it undergoes changes. In this paper, the changes we allow are deletions of edges or vertices by an adaptive adversary. The information we maintain is a representation that reflects biconnectivity of vertices, that is, whether they are connected after the removal of any vertex of the graph.

A static (non-changing) graph may in $O(n + m)$ time be pre-processed to answer biconnectivity queries in worst-case $O(1)$ time. This is done by finding the *blocks*, i.e. the biconnected components. We show, for a large class of graphs including minor free graphs, that in the same asymptotic total time, we can handle any sequence of edge- and vertex deletions, while still answering biconnectivity queries, 2-edge connectivity queries, and connectivity queries, in worst-case $O(1)$ time.

If a pair of vertices are not biconnected, then there exists a certificate for this in form of a cutvertex separating them. A natural question, if a pair of vertices are not biconnected, is thus to ask for such a certificate. There may be many cutvertices separating a pair of vertices, so an even more advanced and desired functionality is the ability to point to the cutvertex furthest towards either one of them. Again, for a large class of graphs, our running time for a *decremental* graph matches the state of the art for *non-changing graphs*, by revealing the nearest cutvertex in $O(1)$ worst-case time, while spending only $O(n + m)$ total time for both preprocessing the graph and handling any sequence of deletions.



© Jacob Holm and Eva Rotenberg;

licensed under Creative Commons License CC-BY 4.0

38th International Symposium on Theoretical Aspects of Computer Science (STACS 2021).

Editors: Markus Bläser and Benjamin Monmege; Article No. 42; pp. 42:1–42:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



When edges and vertices are both deleted and inserted, there are non-trivial lower bounds [36] saying that no data structure for connectivity has both update- and query-time in $o(\log n)$. This is in stark contrast to the incremental situation, where only edge-insertions are allowed, in which the α -time algorithm for union-find is tight [39, 11]. When restricted to deletions, however, even for general graphs, there are no known lower bounds beyond the trivial $O(|G|)$. The research in this paper is inspired by the fundamental open question of whether decremental (deletion-only) connectivity [41], 2-edge connectivity, biconnectivity, or even minimum cut for general graphs can be solved in amortized constant time per edge-deletion, or whether non-trivial lower bounds do exist.

The following table shows how we improve state-of-the-art for planar graphs and minor-free graphs. Here, we present maximum amortized time *per operation*, that is, we do not require $O(1)$ query time. When restricted to constant query time, the best biconnectivity algorithms for non-planar sparse graphs were fully dynamic and had an update time of $\tilde{O}(\sqrt{n})$ [18].

■ **Table 1** Our improvements (**now**) in relation to previous results (*previous*). The table shows amortized time per operation. The table compares with state-of-the-art amortized deterministic algorithms. Allowing randomization, the previous best decremental connectivity algorithm runs in time $\tilde{O}(\log n)$ [42].

	planar <i>previous</i>	bd. genus <i>previous</i>	minor-free graphs <i>previous</i>	now
connectivity	$O(1)$ [34]	$O(\log n)$ [4]	$O(\frac{\log^2 n}{\log \log n})$ [44]	O(1)
2-edge-connectivity	$O(1)$ [25]		$\tilde{O}(\log^2 n)$ [27]	O(1)
biconnectivity	$O(\log n)$ [25]		$\tilde{O}(\log^3 n)$ [27]	O(1)

Dynamic graph connectivity has been studied for decades. Most general is fully dynamic connectivity for general graphs [8, 21, 20, 23, 42, 29, 28, 44, 30, 35], where edges are allowed to be both inserted and deleted. Similarly, fully dynamic two-edge connectivity and biconnectivity have been studied [10, 18, 5, 19, 23, 42, 27] and have algorithms with polylogarithmic update- and query time. For special graph classes, such as planar graphs, graphs of bounded genus, and minor-free graphs, there has been a bulk of work on connectivity and higher connectivity, e.g. [7, 22, 13, 15, 6, 33, 34, 25, 24].

An r -division is, intuitively, a family of $O(n/r)$ subgraphs called the *regions*, with $O(r)$ vertices each, such that the regions partition the edges, and each region shares $O(\sqrt{r})$ *boundary vertices* with the rest of the graph. The concept of r -divisions was introduced in [9] as a tool for finding shortest paths in planar graphs. It naturally generalizes the notion of a separator: a small set of vertices that cause the graph to fall apart into *two* regions, each containing a constant fraction of the original graph [32].

Later, Henzinger et al. [17] generalized this to the concept of a *strict (r, s) -division*, which is a family of $O(n/r)$ subgraphs called the regions, each with at most r vertices, that partition the edges, and where each region has at most s boundary vertices. An r -division is thus a strict $(O(r), O(s))$ -division. For the rest of our paper, we will use the term r -division to mean any strict $(r, O(r^{1-\epsilon}))$ -division for some suitable r, ϵ . There are algorithms [2, 3, 14, 31] for computing r -divisions of planar graphs in linear time.

Our results. We give a data structure for maintaining biconnectivity for a large class of graphs. In order to state our theorem in its fullest generality, we need to define what it means for a pair of r -divisions to be a *suitable pair*.

Given a graph G with n vertices, we call a pair $(\mathcal{A}, \mathcal{R})$ where \mathcal{A} is a strict (r_1, s_1) -division and \mathcal{R} is a strict (r_2, s_2) -division a *t -suitable pair of r -divisions* if:

- there exists an algorithm for fully dynamic biconnectivity in general graphs with amortized time $t(n)$ per operation¹, such that:
- each boundary vertex of \mathcal{A} is also a boundary vertex of \mathcal{R} ($\partial\mathcal{A} \subseteq \partial\mathcal{R}$); and
- for each region $A \in \mathcal{A}$, \mathcal{R} contains a partition of A into $O(\frac{r_1}{r_2})$ regions of size at most r_2 , each having at most s_2 boundary vertices²; and
- $r_1, s_1 \in O(\text{poly}(\log n))$ and $\frac{r_1}{s_1} \in \Omega(t(n) \log n)$; and
- $r_2, s_2 \in O(\text{poly}(\log \log n))$ and $\frac{r_2}{s_2} \in \Omega(t(r_1) \log r_1)$.

When t is understood, we will refer to them as simply *suitable*.

Our data structure answers queries to biconnectivity, i.e, a pair of vertices are biconnected if they are connected and not separated by any bridge or cutvertex. If the vertices u and v are connected but not biconnected, we can output a cutvertex separating them, in fact, we can output that of the possibly many cutvertices that is *nearest* to u – we call this the *nearest cutvertex* – or detect the special case where uv is a bridge.

► **Theorem 1.** *There exists a data structure that given a graph G with n vertices and m edges, and given a suitable pair of r -divisions, preprocesses G in $O(m + n)$ time and handles any series of edge-deletions in $O(m)$ total time while answering queries to pairwise biconnectivity and queries to nearest cutvertex in $O(1)$ time.*

This can immediately be combined with any algorithm for finding suitable r -divisions in linear time, to obtain optimal decremental biconnectivity data structures for graphs that are planar, bounded genus, or minor free.

The data structure is easily extended to maintain information about connectivity, so as to answer queries to pairwise connectivity in $O(1)$ time, and our techniques can easily be used to obtain a decremental data structure for 2-edge connectivity with the same update- and query times.

For completeness, the full version of this paper presents linear time algorithms for finding r -divisions in minor-free graphs using techniques from [37, 40, 43]³

► **Lemma 2.** *Given a graph G that does not have a K_ℓ -minor, for any $r \in \Omega(\log n)$ we can compute a strict $(r, O(r^{\frac{2}{3}} \log^{\frac{1}{3}} n))$ -division in linear time⁴.*

Thus, we have the following consequence as a corollary to Theorem 1:

► **Corollary 3.** *There exists a data structure that given a minor-free graph G with n vertices, preprocesses G in $O(n)$ time and handles any series of edge- and vertex deletions in $O(n)$ total time while answering queries to pairwise connectivity, 2-edge connectivity, biconnectivity, nearest separating bridge in $O(1)$, and nearest separating cutvertex, in $O(1)$ time.*

¹ e.g. $t(n) = O(\log^5 n)$ using [23], and $t(n) = O(\log^3 n \cdot \log^2 \log n)$ using [27]

² This is slightly weaker than requiring \mathcal{R} to contain a strict (r_2, s_2) -division of A .

³ This result was first claimed by Henzinger et al. [17], but their solution only works for planar graphs, and h -minor-free graphs of bounded degree.

⁴ We failed to find a reference in the literature for this fact, but we would not be surprised if it is common knowledge.

Even for graphs where no linear-time algorithms for finding r -divisions are known, our results may still be of interest: as soon as the r -divisions have been computed once, they may be used for several different edge-deletion sequences on the same graph.

Our paper can be seen as a generalization of and improvement upon [34], who showed optimal amortized decremental connectivity for planar graphs, that is, amortized constant update time, and worst-case constant query time.

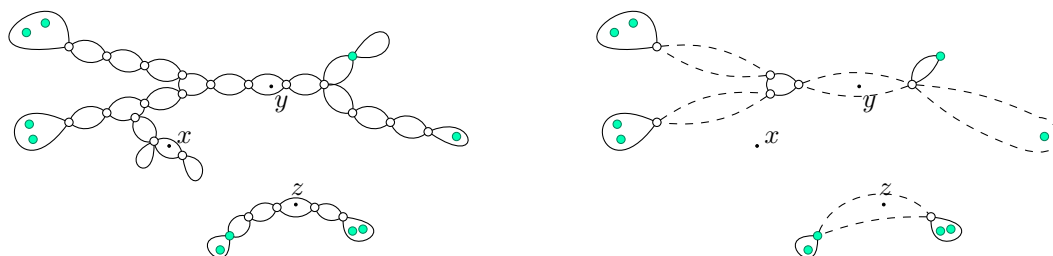
Note that we generalise [34] in two important ways: We generalise from planar graphs to r -divisible graphs, and we generalise from plain connectivity to handle 2-edge-connectivity and biconnectivity. Our generalisation works by getting rid of some unnecessarily planar-specific techniques and replacing them with a more general framework.

We expect this general framework to be of future interest for deriving optimal decremental algorithms from other dynamic algorithms that have polylog update time: as long as a compact representation of each region can be maintained efficiently, with respect to the the graph property of interest, this can be used in our framework for speeding up decremental algorithms.

1.1 Techniques

Since the property of being an r -division is not violated as edges are deleted, it is natural to use r -divisions to get better decremental data structures for graphs. The idea is to have a top-level graph with size only proportional to the number of boundary vertices, and to handle the regions efficiently simply because they are smaller.

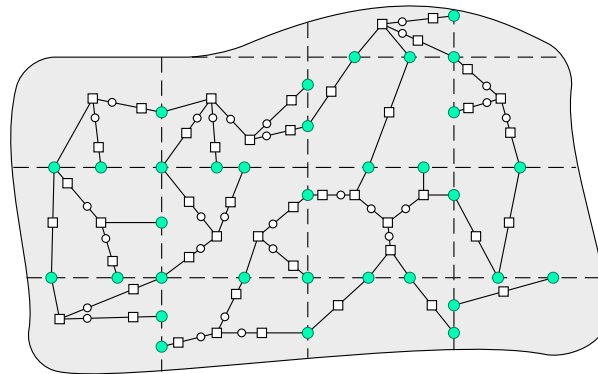
With biconnectivity, the first challenge is to design the top-level graph: a vertex may be not biconnected to any boundary vertex in its region, but yet be biconnected with some other vertex in another region via two separate boundary vertices (see Figure 1). Even vertices from the same region may be biconnected in G although they are not biconnected, or even connected, within the region.



■ **Figure 1** Left: A region R with 10 boundary vertices (green). There is a vertex separating x from the boundary, so x is never biconnected with anything in $G \setminus R$. Vertices y and z however, are not even connected in R , but may be biconnected in G . Right: The structure may be compressed in the sense depicted: x is not represented at all, while y and z are represented in pseudo-blocks (dashed).

We thus need to store an efficient representation of the biconnectivity of the region as seen from the perspective of the boundary vertices. We call this efficient representation the *compressed BC-forest* (see Section 4). It is obtained from the forest of BC-trees (also known as the block-cutpoint trees, see Section 2) by first marking certain blocks and cutvertices as *critical*, and then, basically, contracting the paths that connect them. The critical blocks and cutvertices are sparsely chosen, such that the total size of all the compressed BC-forests is only proportional to the boundary itself. We stitch the compressed BC-forests together by the boundary vertices they share, and obtain the *patchwork graph* (see Figure 2), in which all vertices that are biconnected to anything outside their region are *represented*, and we

use the *representatives* of vertices to reveal when they are biconnected by paths that go via boundary vertices. A construction very similar to our compressed BC-forests appears in [12], where it is used in a separator tree for a planar graph, but the rules for what to contract are subtly different.



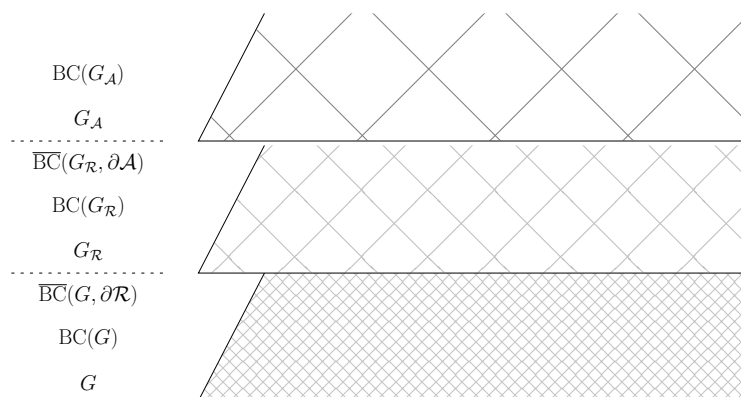
■ **Figure 2** An r -division and its corresponding patchwork graph. The graph is bipartite between, on one hand, round boundary vertices and cutvertices, and, on the other hand, square blocks and contracted (pseudo) blocks.

If decremental changes to a region only gave rise to decremental changes to its forest of BC-trees, we would be close to done. However, and this is the second challenge, the deletion of an edge can cause a block to fall apart into a chain of blocks. Luckily, the damage to the compressed BC-forest is containable: only $O(n/\text{polylog } n)$ vertices can be present in the compressed BC-forest, and the changes can be modeled by only three operations: edge- or path deletions, certain forms of vertex splits, and contractions of paths. These operations, we show, are of a form that can be handled in polylogarithmic time by one of the fully-dynamic biconnectivity data structures (see Section 4).

While using r -divisions once would obtain an improvement from polylog to polyloglog, which might, in practice, be useful already, it is tempting to form r -divisions of the regions themselves and use recursion in order to obtain an even faster speedup (see Figure 3). This would mean that each region should again contain a patchwork made from the compressed BC-forests of its subregions (and, luckily, these patchwork operations compose beautifully). Thus, via recursion, one can obtain a purely combinatorial data structure with $O(\log^* n)$ update- and query time. But in fact, with standard RAM-tricks, if the subregions are of only polyloglog size, one can handle any operation in constant time – simply by using a look-up table. Thus, in the practical RAM-model (i.e. the RAM-model with standard AC^0 operations such as addition, subtraction, bitwise and/or/xor), we can make do with only 3 levels (top, middle and bottom), and obtain $O(1)$ update- and query-time.

Here, as our third challenge, we face that one does not simply recurse into optimality – we need to assure ourselves that when a deletion of an edge causes changes in the compressed BC-trees of the subregion, the changes to the patchwork graph on the level above are manageable. Here, we show that our carefully chosen forms of vertex splits and path contractions do indeed only give rise to the same variant of splits and contractions on the parent level.

Finally, when a pair of vertices u, v are connected but not biconnected, we can in constant time find the nearest cutvertex on any path from u to v – this is called the *nearest cutvertex problem* (see Figure 4). While outputting *some* cutvertex separating u and v is



■ **Figure 3** We use nested r -divisions and obtain a levelled structure. Each level maintains a graph, its BC-tree, and, for the non-top levels, the compressed BC-tree with relation to the boundary.

easy, augmenting the BC-tree with enough information to facilitate nearest cutvertex queries is technically more demanding. We show that the nearest cutvertex can be determined by at most one nearest cutvertex and one biconnected query in the patchwork graph, and at most one nearest cutvertex and one biconnected query in the region. We also show how to augment an explicit representation of the BC-tree subject to certain splits, contractions, and deletions such that we can still access the nearest cutvertex - a problem that reduces to first-on-path on a dynamic tree subject to certain vertex splits, and certain edge contractions and deletions. Specifically, we exploit an intricate flavour of monotonicity: Although blocks can be split arbitrarily, once an element of the structure has participated in a contraction, it will not be subject to further splitting. We solve this by solving a seemingly harder problem on such trees, namely that of answering an extended form of the nearest common ancestor query, known as the *characteristic ancestor* query. This may be of independent interest.



■ **Figure 4** An edge-deletion (red) in the graph can lead to a split of a block which changes the nearest cutvertex from y towards x .

Related techniques. The idea of using recursive separators stems from the sparsification techniques from [5, 6], where it secured $O(\sqrt{n})$ update algorithms for a series of problems, and the idea of using two levels of regions of size $O(\text{polylog } n)$ and $O(\text{poly log log } n)$, respectively, was introduced in [34] where the idea, together with a union-find structure in the dual graphs, was used to obtain amortized $O(1)$ decremental connectivity for planar graphs.

Paper outline. Section 2 is dedicated to preliminaries and terminology. Then, in Section 3, we introduce the notion of *capacitated biconnectivity*, which is a tool for overcoming the third challenge of making the recursion work. Section 4 is dedicated to an understanding of the patchwork graph in a static setting: how it is defined, how it reflects biconnectivity, and how it behaves when there is not one but two or more nested r -divisions of the same

graph. Finally, in Section 5, we show how to maintain the patchwork graph decrementally, thus enabling us solve decremental biconnectivity. The extension to handle nearest cutvertex queries is deferred to Section 6. The extension uses our characteristic ancestors structure, which is deferred to the full version (preliminary version available as [26]). Also deferred to the full version is the reductions that handle 2-edge-connectivity and connectivity and the linear time construction of r -divisions for minor-free graph classes.

2 Preliminaries

Given a graph with vertices u and v , we say they are *connected* if there is a path connecting them. A pair of connected vertices are *2-edge connected* unless there is an edge whose removal would disconnect them. Such an edge is called a *bridge*. A pair of 2-edge connected vertices u and v are (*locally*) *biconnected* unless there exists a vertex (other than u and v) whose removal would disconnect them. Such a vertex is called a *cutvertex*. For an ordered pair (u, v) of connected but not biconnected vertices, the *nearest cutvertex* separating them is uniquely defined as the first cutvertex on a path – any path – from u to v . In the special case where u and v are separated by the bridge uv , we say that the nearest cutvertex is **nil**.

The *blocks* of a graph are the maximal biconnected subgraphs. Each block is either a bridge or a maximal set of biconnected vertices. For each connected component of a graph, the *block-cutpoint tree* [16, p. 36], or *BC-tree* for short, reflects the biconnectivity among the vertices. This tree has all the vertices of the graph and, furthermore, a vertex for each block. Its edges are those that connect each vertex to the block or blocks it belongs to. If the graph G is not necessarily connected, its *BC-forest* $BC(G)$ has a BC-tree for each connected component of the graph. The BC-forest of a graph can be found in linear time [38].

If each BC-tree in the BC-forest is rooted at an arbitrary block, each non-root block has one unique cutvertex separating it from its parent. Then, a pair of vertices are biconnected if and only if they either have the same non-bridge block as parent, or one is the parent of the non-bridge block that is parent of the other.

A dynamic data structure for biconnectivity in general graphs is developed in [23, 42, 27]; it maintains an n -vertex graph and handles deletions and insertions of edges in $t(n) = O(\log^3 n \cdot \log^2 \log n)$ amortized time, and answers queries in $O(\log^2 n \cdot \log^2 \log n)$ worst-case time. The data structure is easily modified to give the first cutvertex separating a pair of vertices in $O(\log^2 n \cdot \log^2 \log n)$ time, but even without this modification, one can find the first cutvertex via a binary search along a spanning tree in $O(\log n)$ queries in $O(\log^3 n \cdot \log^2 \log n)$ worst case time. Note however that for our purposes, the original [23] data structure with $O(\log^5 n)$ amortized update- and query time is sufficient. For the rest of this paper, we will just use $t(n)$ to denote the amortized time per operation (queries included) of a fully dynamic biconnectivity structure for general graphs.

For (not necessarily distinct) vertices v, u, w in a tree, we use $v \longleftrightarrow u$ to denote the tree-path connecting v and u , and we use $\text{meet}(u, v, w)$ to denote the unique common vertex of all three tree-paths connecting them.

A *strict* (r, s) -*division* is a set of $O(n/r)$ subgraphs $\mathcal{R} = \{R_1, R_2, \dots\}$ called *regions*, that partition the edges. Each region $R \in \mathcal{R}$ has at most r vertices, and a set ∂R of at most s *boundary vertices*, such that only boundary vertices appear in more than one region. We denote by $\partial \mathcal{R}$ the set of all boundary vertices $\bigcup_{R \in \mathcal{R}} \partial R$. Note that with these definitions, $\sum_{R \in \mathcal{R}} |\partial R| \leq O(n/r) \cdot O(s) = O(n \cdot \frac{s}{r})$.

An r -division usually means a strict (r, s) -division with $s = O(\sqrt{r})$, but we will be using it more broadly to include any strict (r, s) -division, where $s = O(r^{1-\varepsilon})$ for some $\varepsilon > 0$.

We say that a pair $(\mathcal{A}, \mathcal{R})$ consisting of an r_1 -division and an r_2 -division are *nested*, if $\partial\mathcal{A} \subseteq \partial\mathcal{R}$, and \mathcal{R} contains an r_2 -division of each region of \mathcal{A} . With a slight abuse of notation, for any $A \in \mathcal{A}$ we will let $\mathcal{R} \cap A$ denote this r_2 -division.

3 Bicapacitated biconnectivity

Consider the BC-forest of a graph. It may be viewed as a *bicapacitated graph*, where non-bridge blocks have capacity 2 and bridge blocks and vertices have capacity 1; then, vertices u and v in G are biconnected exactly when there exists a flow of value 2 from u to v in the BC-forest of G . (Disregarding the capacity of the source and sink vertices.) We denote by *bicapacitated biconnectivity* the query to the existence of such a flow.

Recall that we want to be able to use the framework recursively: we want to build and maintain BC-trees for small graphs and stitch them, or rather, compressed versions of them together, thus obtaining a patchwork graph. So, we need to extend our definitions so that they can handle a bicapacitated input graph corresponding to the BC-trees of an underlying region. The resulting patchwork graphs are always bipartite, with vertices on one side all having capacity 1, and vertices on the other side having capacity either 1 or 2. We will restrict our definition of bicapacitated graph to mean such graphs.

Now, we can introduce the problem of *fully dynamic bicapacitated biconnectivity*, as that of facilitating bicapacitated biconnectivity queries between vertices in a bicapacitated graph as it undergoes insertions and deletions of edges. Note that (fully) dynamic bicapacitated biconnectivity has an easy reduction to (fully) dynamic biconnectivity:

► **Lemma 4.** *Given a fully dynamic data structure for biconnectivity in general graphs using amortized $t_u(n)$ time per link or cut and (amortized/worst case) $t_q(n)$ per pairwise biconnectivity or nearest cutvertex query, there is a fully dynamic data structure for bicapacitated graphs that uses $O(t_u(2n))$ amortized time per edge insert/delete, and answers pairwise biconnectivity and nearest-cutvertex queries in (amortized/worst case) $O(t_q(2n))$ time.*

4 The patchwork graph

We are given an r -division $\mathcal{R} = \{R_1, \dots, R_k\}$ of G , and we want to define a graph $G_{\mathcal{R}}$ of size $O(|\partial\mathcal{R}|)$ that somehow captures all the biconnectivity relations that cross multiple regions. We call the resulting $G_{\mathcal{R}}$ a *patchwork graph*, because it is built by stitching together a suitable *patch graph* for each region.

Our patch graph for each region is in turn based on the BC-forest for the region. We *compress* the BC-forest of the region similarly to [12] as follows:

► **Definition 5.** *Given a bicapacitated graph $G = (V, E)$, its BC-forest $F = \text{BC}(G)$, and a subset of vertices $S \subseteq V$, define a node⁵ $x \in T$, where the tree T is a component of F , to be*

- *S -critical if $x = \text{meet}_T(s_1, s_2, s_3)$ for some $s_1, s_2, s_3 \in S$,*
- *S -disposable if $x \notin s_1 \longleftrightarrow_T s_2$ for all $s_1, s_2 \in S$, and*
- *S -contractible otherwise.*

► **Definition 6.** *The compressed BC-forest $\overline{\text{BC}}(G, S)$ is the forest obtained from its forest of BC-trees by deleting all S -disposable nodes, and replacing each maximal path of S -contractible nodes that start and end in distinct blocks, with a single so-called pseudoblock node with capacity 1.*

⁵ Throughout the text we consistently denote vertices of G by *vertices*, and vertices of BC-trees and SPQR-trees as *nodes*.

► **Definition 7.** Given an r -division $\mathcal{R} = \{R_1, \dots, R_k\}$ of a graph G , define the patchwork graph $G_{\mathcal{R}} = \bigcup_{R \in \mathcal{R}} \overline{BC}(R, \partial R)$ to be the bicapacitated graph obtained by taking the (non-disjoint) union of compressed BC-forests $\overline{BC}(R, \partial R)$ for each region $R \in \mathcal{R}$.

Any vertex of G corresponds to a BC-vertex in $BC(R)$ for some R . Some of these BC-vertices are either present or represented in $G_{\mathcal{R}}$. We thus want to define the representation of a vertex as the vertex in $G_{\mathcal{R}}$ representing its BC-node, when it exists:

► **Definition 8.** Given a patchwork graph $G_{\mathcal{R}}$ and a vertex v of G , we define the representative $\overline{B}(v)$ of v as follows:

- If v is a vertex of $G_{\mathcal{R}}$, then $\overline{B}(v) = v$; else
- Let $R \in \mathcal{R}$ be the unique region containing v . If v is incident to a block in $BC(R)$ that is not S -disposable, then v is represented either by that block or the pseudoblock representing it.
- Otherwise, v is not represented.

Overloading notation slightly, say that a vertex of the graph is *critical*, *disposable*, or *contractible*, if the BC-node representing it is.

► **Observation 9.** There is a linear time algorithm for building the compressed BC-forest of a graph with respect to a given subset of vertices, and for finding the representatives of the vertices.

► **Lemma 10.** Distinct vertices u, v are biconnected in G if and only if either

1. At least one of u, v is not a boundary vertex, and u, v are biconnected in the at most one region R containing both; or
2. $\overline{B}(u) = \overline{B}(v)$ is a pseudo-block whose unique neighbours are biconnected in $G_{\mathcal{R}}$; or
3. $\overline{B}(u)$ and $\overline{B}(v)$ are different and are biconnected in $G_{\mathcal{R}}$.

Proof. We will show that u and v are *not* biconnected if and only if all three conditions are false. Assume u and v are not biconnected. Then they can clearly not be biconnected within some region R , so condition 1 is false. If $\overline{B}(u) = \overline{B}(v)$, then this is a pseudo-block contracted from a chain containing the neighbors of u, v in $BC(R)$, and a cutpoint c that separates them within R . Consider the neighbors u' and v' to this pseudoblock in $G_{\mathcal{R}}$. If they were biconnected in $G_{\mathcal{R}}$ there would be a u, v path in $G \setminus \{c\}$ contradicting our choice of u, v . Thus u' and v' are not biconnected in $G_{\mathcal{R}}$ and condition 2 is false. Finally, if $\overline{B}(u)$ and $\overline{B}(v)$ are different, then any cutvertex c separating u and v in G will either be a cutvertex in $G_{\mathcal{R}}$, or will be in a pseudoblock $\overline{B}(c)$ with neighbors u' and v' . Since c is a cutvertex in G , $(u', \overline{B}(c))$ and $(\overline{B}(c), v')$ are bridges in $G_{\mathcal{R}}$, and $\overline{B}(u)$ and $\overline{B}(v)$ will be separated by at least one of them and are therefore not biconnected in $G_{\mathcal{R}}$ and condition 3 is false.

If, on the other hand, none of the three conditions are true, then, if $\overline{B}(u) = \overline{B}(v)$ is a pseudoblock whose neighbours in $G_{\mathcal{R}}$ are not biconnected, then any cutvertex separating u from v in their region also separates them in G . If $\overline{B}(u) \neq \overline{B}(v)$ are separable by some cutvertex c in $G_{\mathcal{R}}$, then c is also a cutvertex separating u from v in G , and hence they are not biconnected. If $\overline{B}(u) \neq \overline{B}(v)$ are the endpoints of a bridge in $G_{\mathcal{R}}$ then one of them must be a pseudoblock containing a cutvertex or a bridge in G separating them. ◀

Lemma 10 above almost enables us to transform a biconnectivity-query in G into a biconnectivity-query in $G_{\mathcal{R}}$ and a biconnectivity inside a region R . However, item 1 is only directly useful when neither of the vertices belong to the boundary; when one is a boundary vertex we do not know which vertex in the region it corresponds to. Fortunately, when the non-boundary vertex is represented, we may query biconnectivity in $G_{\mathcal{R}}$ to obtain the answer. To handle disposable vertices, we introduce the notion of the *nearest represented vertex*:

► **Definition 11.** When an S -disposable vertex v is connected to at least one boundary vertex b , it knows its nearest represented vertex $\text{nr}(v)$ which is the first non-disposable node in the BC-tree of the region on the path from b to v (note that this node is one unique cutvertex). When an S -disposable vertex v is not connected to the boundary, it has $\text{nr}(v) = \mathbf{nil}$.

Note also that item 3 requires the pseudo-block to know its exactly two neighbours.

- **Lemma 12.** Vertices u and v are biconnected if and only if either
- u and v are non-boundary vertices of the same region and are biconnected in the region,
 - u is a non-boundary vertex that is biconnected in its region R with $\text{nr}(u)$ and $\text{nr}(u) = v$,
 - $\overline{B}(u) = \overline{B}(v)$ is a pseudo-block and its neighbours are biconnected in $G_{\mathcal{R}}$, or
 - $\overline{B}(u) \neq \overline{B}(v)$ are biconnected in $G_{\mathcal{R}}$

Proof. Follows from Lemma 10 by expanding item 1 into the two cases of whether both or only one vertex is non-boundary. ◀

Note that patchwork graphs are well-behaved and respect sub-divisions of r -divisions in the following sense:

► **Lemma 13.** If $S \subseteq \partial\mathcal{R}$, then $\overline{BC}(G, S) = \overline{BC}(G_{\mathcal{R}}, S)$

Proof. There is a correspondence between the critical, disposable, and contractible BC-nodes.

Consider an S -critical BC-node x of G . It may overlap with several regions. However, in each region, each vertex of x lies on some $r_1 \longleftrightarrow r_2$ path for $r_1, r_2 \in \partial\mathcal{R}$, so they are never disposable. But then, since $S \subseteq \partial\mathcal{R}$, x is also $\partial\mathcal{R}$ -critical, and thus, present in $G_{\mathcal{R}}$. Clearly, once the block is present in $G_{\mathcal{R}}$, it is also S -critical in $G_{\mathcal{R}}$.

If a BC-node of G is S -disposable, we only need to observe that its $\partial\mathcal{R}$ -contractible and $\partial\mathcal{R}$ -critical parts, for each path $r_1 \longleftrightarrow r_2$ they lie on, at most one endpoint is not S -disposable.

Finally, if a BC-node x of G is S -contractible, then it lies on some path $s_1 \longleftrightarrow s_2$, which $\partial\mathcal{R}$ cuts up into subpaths $r_1 \longleftrightarrow r_2 \longleftrightarrow r_3 \longleftrightarrow \dots$ in (not necessarily different) regions R_1, R_2, R_3, \dots . But then, all parts of x are preserved as either $\partial\mathcal{R}$ -critical or $\partial\mathcal{R}$ -contractible $\overline{BC}(R_i)$ -vertices, and thus, survive in $\overline{BC}(G_{\mathcal{R}}, S)$. On the other hand, if a vertex in R_i does not belong in x , then it does not lie on any of the paths $r_j \longleftrightarrow r_{j+1}$, and can thus not be represented by a vertex or a pseudo-block on that path. ◀

The same lines of thought can be used to make the following observation about how nested r -divisions behave with respect to patchwork graphs:

► **Observation 14.** If $\partial\mathcal{R}_1 \subseteq \partial\mathcal{R}_2$, then $G_{\mathcal{R}_1} = (G_{\mathcal{R}_2})_{\mathcal{R}_1}$.

5 Decremental Biconnectivity in Patchwork Graphs

Given the BC-forest for (the patchwork graph associated with) each region of G in an r -division \mathcal{R} , we want to explicitly maintain $G_{\mathcal{R}}$ and $\text{BC}(G_{\mathcal{R}})$.

Let R' be a bicapacitated graph associated with region R , and suppose that $\overline{BC}(R', \partial R) = \overline{BC}(R, \partial R)$. We will arrange things so either $R' = R$ (with all vertices having capacity 1), or $R' = R_{\mathcal{R}'}$ for some r -division \mathcal{R}' of R with $\partial R \subseteq \partial\mathcal{R}'$, so the equality follows from Lemma 13.

We will maintain a fully dynamic biconnectivity structure for R' with amortized time $t(n) \in O(\text{poly}(\log n))$ per operation, e.g. using [23]⁶. We use this structure to explicitly maintain $\text{BC}(R')$ under the following operations:

⁶ A faster algorithm here would just make more pairs of r -divisions suitable.

path deletion – given a path between two vertices of capacity 1, whose internal vertices all have degree 2, deletes all edges and internal vertices on the path.

block split – given a vertex u of capacity 2, and an adjacent vertex v of capacity 1, split u into two vertices u_1, u_2 of capacity 2 connected by a path with 2 edges via v , with u_1, u_2 partitioning the remaining neighbors of u .

pseudoblock contraction – given a path of 3 vertices, all having degree 2 and the middle having capacity 1, contract the path to a single vertex with capacity 1.

The point is that if one of these operations is applied to R' , then the change to $\text{BC}(R')$ and $\overline{\text{BC}}(R', \partial R)$ can also be described by a sequence of these operations.⁷

► **Lemma 15.** *There is a data structure that explicitly maintains $\text{BC}(R')$ that can be initialized, and support any sequence of $O(|R'|)$ path deletions, block splits, and pseudoblock contractions, in $O(|R'|t(n') \log n')$ total time, where n' is the number of vertices in R' .*

Proof. Use the data structure from Lemma 4 as a subroutine. Start by inserting all the edges. Each pseudoblock contraction can be simulated using a constant number of edge insertions or deletions. The total number of edges participating in path deletions is upper bounded by $O(n')$. Each block split either takes only a constant number of edge insertions or deletions, or makes a non-trivial partition of the adjacent edges. In the latter case, we still do a constant number of edge insertions and deletions, followed by one edge move (deletion and insertion) for each edge that ends up in a non-largest set in the partition. Each edge is moved in this way $O(\log n')$ times, so the total number of update operations done on the fully dynamic structure is $O(|R'| \log n')$. Once an update to R' has been simulated in the fully dynamic structure, we can use queries in that structure to find any new cutvertices that we need to update $\text{BC}(R')$. If the update in R' was a path deletion, then the corresponding update to $\text{BC}(R')$ is either a path deletion, or a sequence of block splits. Each of these block splits can be found using the cutvertices given by the fully dynamic structure: Do a parallel search from both endpoints, and use the nearest cutvertex-query from Lemma 4 to guide the search and to know when a whole block has been found. If the update in R' was a block split, this will either do nothing in $\text{BC}(R')$ or cause a single block split. If the update in R' is a pseudoblock contraction, the corresponding update to $\text{BC}(R')$ is at most one edge deletion (because the leaf corresponding to the cutvertex disappears), at most one pseudoblock contraction (corresponding to the same pseudoblock contraction), or nothing happens (because the pseudoblocks were disposable). ◀

The point is that we will be using this with $|R'| = O(n/(t(n) \log n))$, where n is the number of vertices in R , which means the total time used on R is $O(n) \subseteq O(|R|)$.

► **Lemma 16.** *The data structure of Lemma 15 can be extended to also report the explicit changes needed to $\overline{\text{BC}}(R', \partial R)$ in the same asymptotic initialisation and update time. Any sequence of $O(|R'|)$ updates to R' cause $O(|\partial R|)$ updates in $\overline{\text{BC}}(R', \partial R)$.*

Proof. For each change to $\text{BC}(R')$, we can update $\overline{\text{BC}}(R', \partial R)$ accordingly. This essentially consists of replaying the same change as in $\text{BC}(R')$, followed by at most two pseudoblock contractions; at most one in each end of the path it possibly unfolds to. Note however, that some operations will end up having no effect on the structure of $\overline{\text{BC}}(R', \partial R)$. For example a trivial block split followed by a pseudoblock contraction will change only which cutvertex

⁷ By *explicit maintenance* is meant that each rooted BC-tree is maintained such that finding the parent of a vertex or block takes constant time.

42:12 Good r -Divisions Imply Optimal Amortized Decremental Biconnectivity

separates the pseudoblock from the block. In this case, rather than doing a split and a contract, we simply update the identity of the cutvertex. With this optimization, the total number of block splits is upper bounded by $O(|\partial R|)$, and so is the number of edges and hence the number of possible path deletions and pseudoblock contractions. ◀

It immediately follows that we are able to efficiently maintain the patchwork graph, by combining the lemma above with the definition of the patchwork graph, $G_{\mathcal{R}} = \bigcup_{R \in \mathcal{R}} \overline{BC}(R, \partial R)$.

► **Lemma 17.** *Given a graph G , and a strict (r, s) -division \mathcal{R} of G , if we can explicitly maintain $\overline{BC}(R, \partial R)$ for each $R \in \mathcal{R}$ in amortized constant time per update after $O(|R|)$ preprocessing, then we can explicitly maintain $G_{\mathcal{R}}$ in amortized constant time per update after $O(|G|)$ preprocessing. Furthermore, any sequence of $O(|G|)$ updates in G cause $O(|G_{\mathcal{R}}|)$ updates in $G_{\mathcal{R}}$.*

Proof. Let G have n vertices and m edges. The first part follows trivially from $\sum_{R \in \mathcal{R}} |R| \in O(n/r)O(r) + m = O(n + m)$. Each block split in $G_{\mathcal{R}}$ either reduces the degree of some block, or adds a pseudoblock. Since we do not add another pseudoblock when there already is one in a given direction, the maximum total number of splits an initial block vertex v can cause is $O(d(v))$. Thus the maximum number of splits is $\sum_{v \in G_{\mathcal{R}}} O(d(v)) = O(|G_{\mathcal{R}}|)$, and so is the maximum number of edges and hence the number of possible path deletions and pseudoblock contractions. ◀

In order to use Lemma 12 to answer biconnected queries, we need to store some auxiliary information: for each pseudoblock, store its neighbours, and for each disposable vertex, store its nearest represented vertex. Thus, these need to be updated as the graph undergoes dynamic updates.

path deletion When a path from x to y is deleted, all vertices represented by internal nodes on the path become disposable. For each such vertex v , its nearest represented vertex becomes either x or y . Furthermore, each vertex u who had v as its nearest represented vertex, now changes its nearest represented vertex to $\text{nr}(u) = \text{nr}(v)$. Thus, the set of vertices having x (or y) as a representative, is now the union of: vertices on the path, vertices represented by blocks or pseudo-blocks on the path, and the sets that these vertices used to represent. These sets of vertices that have the same representative can be maintained via union find in $O(n \log n)$ total merge-time and $O(1)$ worst-case find-time, using the weighted quick-find algorithm [1]. The endpoints x and y may change status from being represented by themselves to being represented by a block or pseudoblock.

block split A block is never the neighbour of a pseudoblock, nor is it the nearest represented vertex, so block splits do not give cause to changes in neighbours and representatives.

pseudoblock contraction does not give rise to changes in the nearest represented vertex - the vertices that were previously represented by a node that is involved in the contraction, are still represented, but now they are represented by the resulting pseudoblock. The set of vertices represented by the resulting pseudoblock is the union of vertices represented by nodes along the contracted path, again, this is done via union-find. Finally, the resulting pseudoblock is updated to remember its two neighbours.

We are now ready to prove:

► **Theorem 18** (first part of Theorem 1). *There exists a data structure that given a graph G with n vertices and m edges, and given a suitable pair of r -divisions, preprocesses G in $O(m + n)$ time and handles any series of edge-deletions in $O(m)$ total time while answering queries to pairwise biconnectivity in $O(1)$ time.*

Proof. Given a fine r -division \mathcal{R} of G , build the BC-forest and compressed BC-forest for each region, and build the patchwork graph $G_{\mathcal{R}}$. Given the coarse division \mathcal{A} , and given the patchwork graph for \mathcal{R} , build the BC-forest and the compressed BC-forest for each region of the patchwork graph, and build the patchwork graph $G_{\mathcal{A}}$. Finally, build the BC-forest for $G_{\mathcal{A}}$. The construction time is linear, due to [38] and Observation 9.

Deletions are handled bottom up: updating the regions of \mathcal{R} , then those of $G_{\mathcal{R}}$ induced by \mathcal{A} , and then $G_{\mathcal{A}}$. The total time for deletions is linear, due to Lemmata 15, 16, and 17.

In detail: Since $r_2 = O(\text{poly}(\log \log n))$, we can afford to precompute and store a table of all simple graphs on r_2 vertices with s_2 boundary vertices, and how their BC-trees and compressed BC-trees change under any possible edge deletion. Using such a table, the region R in \mathcal{R} containing the deleted edge can be updated in constant time.

The updates to $\overline{\text{BC}}(R, \partial R)$ may cause some updates to the patchwork graph $A_{\mathcal{R} \cap \mathcal{A}}$ for the region $A \in \mathcal{A}$ containing the deleted edge. By Lemma 17, we can find these in amortized constant time per edge deletion in A , and there are at most $|A_{\mathcal{R} \cap \mathcal{A}}|$ of them. Since

$$|A_{\mathcal{R} \cap \mathcal{A}}| \leq \sum_{R \in \mathcal{R} \cap \mathcal{A}} |\partial R| \leq s_2 \cdot O\left(\frac{r_1}{r_2}\right) \in O\left(r_1 \frac{s_2}{r_2}\right) \quad \text{and} \quad \frac{r_2}{s_2} = \Omega(t(r_1) \log r_1)$$

we have $|A_{\mathcal{R} \cap \mathcal{A}}| \in O\left(\frac{r_1}{t(r_1) \log r_1}\right)$. By Lemma 15 and 16 we can therefore explicitly maintain $\text{BC}(A_{\mathcal{R} \cap \mathcal{A}})$ and $\overline{\text{BC}}(A_{\mathcal{R} \cap \mathcal{A}}, \partial A)$ in amortized constant time per edge deletion in A .

The updates to $\overline{\text{BC}}(A_{\mathcal{R} \cap \mathcal{A}}, \partial A)$ again trigger some number of updates to $G_{\mathcal{R}}$. By Lemma 17, we can find these in amortized constant time per edge deletion in G , and there are at most $|G_{\mathcal{R}}|$ of them. Since

$$|G_{\mathcal{R}}| \leq \sum_{A \in \mathcal{A}} |\partial A| \leq s_1 \cdot O\left(\frac{n}{r_1}\right) \in O\left(n \frac{s_1}{r_1}\right) \quad \text{and} \quad \frac{r_1}{s_1} = \Omega(t(n) \log n)$$

we have $|G_{\mathcal{R}}| \in O\left(\frac{n}{t(n) \log n}\right)$. By Lemma 15 we can therefore explicitly maintain $\text{BC}(G_{\mathcal{R}})$ in amortized constant time per edge deletion in G .

To handle biconnected-queries, perform the $O(1)$ queries indicated by Lemma 12. \blacktriangleleft

6 Nearest cutvertex in $O(1)$ worst-case time

We have now shown how we handle queries to biconnectivity in a decremental graph subject to deletions. To answer nearest cutvertex queries, we need more structure. We need to augment our explicit representation of the dynamic BC-tree subject to block-splits so that it answers nearest cutvertex queries (subsection 6.1), and we need to show that we need only a constant number of queries in the patchwork graph together with a constant number of queries in regions, to answer nearest-cutvertex in the graph.

6.1 Navigating a dynamic BC-tree

If the vertices u and v are connected but not biconnected, and we have a BC-tree over the component containing them, the nearest cutvertex to u will be the second internal node on the unique BC-tree-path from u to v . So, in order to answer nearest cutvertex queries, it is enough to answer first-on-path queries on a tree (since second-on-path can be found using two first-on-path queries).

► Lemma 19. *There is a data structure for representing a dynamic BC-forest that can be initialized on a forest with n nodes and support any sequence of $O(n)$ path-deletions, block-splits, and pseudoblock-contractions, in $O(n \log n)$ total time, while answering connected and first-on-path queries in worst case constant time.*

Proof. We use the *characteristic ancestor* and *tree connectivity* data structures defined in the full version as a base (preliminary version available as [26]). These both work on rooted trees with black and white nodes, where a white node can be *split* into two white nodes of lower degree, and we can *contract* the endpoints of any edge regardless of color to form a new black node.

First, observe that we can combine these into a single structure, supporting both split, contract, and delete operations and both first-on-path and connected queries. This is because first-on-path(u, v) is only valid when u and v are connected, and the results of valid queries are therefore not affected by edge deletions. So we can maintain the two structures in parallel, and simply ignore deletions in the first-on-path structure, and let each structure answer the query it is designed for.

Second, observe that:

- Each path-deletion can be simulated using contractions and an edge deletion.
- Each block-split can be implemented as two node splits and an edge contraction.
- Each pseudoblock-contraction can be implemented as two edge contractions.

And note that if we color each vertex black and each block white (with pseudoblocks being either black or white depending on their history), then these operations respect the color requirements for our data structures.

Since we do only $O(n)$ operations, and we start with n black nodes, the total time for all updates is $O(n \log n)$. ◀

It follows as a corollary that we can answer nearest cutvertex queries given an explicit representation of the BC-forest:

► **Corollary 20.** *Given a dynamic BC-tree over a connected n -vertex graph, we can answer biconnected and nearest cutvertex queries in $O(1)$ time, spending an additional $O(n \log n)$ time on any sequence of updates.*

Proof. Given a pair of connected and different vertices u, v , let w be the second-on-path vertex found by querying first-on-path(first-on-path(u, v), v). If $w = v$, the vertices are biconnected. Otherwise, w is the nearest cutvertex separating u from v . ◀

6.2 The patchwork graph

In the following, recall that each disposable vertex v knows its nearest represented vertex $\text{nr}(v)$, and each pseudoblock knows its two neighbours.

► **Lemma 21.** *If u, v are connected and not biconnected, then the nearest cutvertex separating u from v can be determined by at most one nearest cutvertex-query and at most one biconnected query in $G_{\mathcal{R}}$ followed by at most one nearest cutvertex-query and at most one biconnected query within a region.*

Proof. If u and v are not both non-boundary vertices, and they are connected within the region R containing both, then the nearest cutvertex within R is the nearest cutvertex in G .

Otherwise, if u is disposable, then it knows its nearest represented vertex $\text{nr}(u)$. If u and $\text{nr}(u)$ are biconnected, then $\text{nr}(u)$ is the nearest cutvertex separating u from v in G . Otherwise, the nearest cutvertex separating u from $\text{nr}(u)$ in their region R is also the nearest cutvertex separating u from v in G .

If u is represented but v is not represented, then v knows its closest represented vertex $\text{nr}(v)$ within its region. If $\overline{\text{B}}(\text{nr}(v))$ is biconnected with $\overline{\text{B}}(u)$, then $\text{nr}(v)$ is the answer, otherwise, $\text{nr}(v)$ is used in place of v in the following.

For the remaining cases, u and v are both represented, and their representatives are different. If the nearest cutvertex query between $\bar{B}(u)$ and $\bar{B}(v)$ in $G_{\mathcal{R}}$ returns a neighbour b of the pseudo-block that either is $\bar{B}(u)$ or is a neighbour of $\bar{B}(u)$, then querying nearest cutvertex between u and b in the region of the pseudo-block will return the nearest cutvertex between u and v in G . Note here, that a pseudo-block is only present in one region, and even if u is a boundary vertex that appears in several regions, the pseudo-block knows the identity of both its endpoints within the region.

Finally, in all other cases, the nearest cutvertex separating $\bar{B}(u)$ and $\bar{B}(v)$ in $G_{\mathcal{R}}$ is the nearest cutvertex separating u and v in G . ◀

► **Theorem 22** (Second part of Theorem 1). *The data structure in Theorem 18 can be augmented to support queries to nearest cutvertex in $O(1)$ worst-case time, while handling any series of edge-deletions in $O(n + m)$ total time.*

Proof. For the patchwork graph of G and for the patchwork graphs of each region, maintain their dynamic BC-forest as indicated in Lemma 19. For the regions of the fine r -division, that is, those of polylog log-size, maintain an explicit table over the answer to nearest cutvertex queries.

Due to Corollary 20, the maintenance of explicit forests of BC-trees over the patchwork graph of G is done in $O(n' \log n')$ total time for $n' = O(n/\log n)$, thus, $O(n)$ total time, while handling intermixed nearest cutvertex-queries in $O(1)$ worst-case time. Same goes for the explicit maintenance of BC-forests of the patchwork graphs in the regions of the coarse r -division.

Finally, to handle nearest-cutvertex(u, v)-queries, perform the $O(1)$ queries indicated by Lemma 21: each of the $O(1)$ queries in the regions of the coarse r -division give rise to $O(1)$ look-ups in the regions of the fine r -division. Thus, the total query-time is constant. ◀

7 Conclusion and implications

We have given a somewhat technical theorem stating that if a graph has suitable r -divisions, there is an efficient data structure for decremental biconnectivity. For minor free graphs, we promised not only that they admit suitable r -divisions, but that such r -divisions can be computed in linear time in the size of the graph (regarding the size of the excluded minor as a constant). In the full version, we prove the following consequence to Theorem 1:

► **Corollary 23.** *There exists a data structure that, given a graph G with n vertices and m edges, and given a suitable pair of r -divisions, preprocesses G in $O(m + n)$ time and handles any series of edge-deletions in $O(m)$ total time while answering connectivity queries, 2-edge connectivity queries, and biconnectivity queries in $O(1)$ time.*

Using [37, Lemma 2] and [40, Lemma 3.4], we also show Lemma 2 which is used in the proof of the following theorem, which in combination with Theorem 1 gives Corollary 3.

► **Theorem 24.** *Given a graph G with n vertices that does not have a K_ℓ -minor, and any $t(n) \in O(\text{poly}(\log n))$ we can compute a t -suitable pair of r -divisions in $O(n)$ time.*

Finally, since the total number of edges in a minor-free graph is $O(n)$, the data structure above has the optimal amortized update time for edge deletions and vertex deletions, both. The question of whether our data structure generally admits vertex deletions in $O(n)$ total time remains open.

References

- 1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.
- 2 Lyudmil Aleksandrov and Hristo Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *Siam Journal on Discrete Mathematics - SIAMDM*, 9:129–150, February 1996. doi:10.1137/S0895480194272183.
- 3 Lars Arge, Freek van Walderveen, and Norbert Zeh. Multiway simple cycle separators and i/o-efficient algorithms for planar graphs. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 901–918, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics. URL: <http://dl.acm.org/citation.cfm?id=2627817.2627882>.
- 4 David Eppstein. Dynamic generators of topologically embedded graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 599–608, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. URL: <http://dl.acm.org/citation.cfm?id=644108.644208>.
- 5 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, September 1997. doi:10.1145/265910.265914.
- 6 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Thomas H. Spencer. Separator-based sparsification ii: Edge and vertex connectivity. *SIAM Journal on Computing*, 28(1):341–381, February 1999. doi:10.1137/S0097539794269072.
- 7 David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert E. Tarjan, Jeffery R. Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. *Journal of Algorithms*, 13(1):33–54, March 1992. Special issue for 1st SODA.
- 8 Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985. doi:10.1137/0214055.
- 9 Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, December 1987. doi:10.1137/0216064.
- 10 Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing*, 26(2):484–538, 1997. doi:10.1137/S0097539792226825.
- 11 Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 345–354, New York, NY, USA, 1989. ACM. doi:10.1145/73007.73040.
- 12 Zvi Galil, Giuseppe F. Italiano, and Neil Sarnak. Fully dynamic planarity testing with applications. *Journal of the ACM*, 46(1):28–91, January 1999. doi:10.1145/300515.300517.
- 13 Dora Giammarresi and Giuseppe F. Italiano. Decremental 2- and 3-connectivity on planar graphs. *Algorithmica*, 16(3):263–287, 1996. doi:10.1007/BF01955676.
- 14 Michael T. Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 51(3):374–389, 1995. doi:10.1006/jcss.1995.1076.
- 15 Jens Gustedt. Efficient union-find for planar graphs and other sparse graph classes. *Theoretical Computer Science*, 203(1):123–141, 1998. doi:10.1016/S0304-3975(97)00291-0.
- 16 Frank Harary. *Graph Theory*. Addison-Wesley Series in Mathematics. Addison Wesley, 1969.
- 17 Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997. doi:10.1006/jcss.1997.1493.
- 18 Monika R. Henzinger and Han La Poutré. Certificates and fast algorithms for biconnectivity in fully-dynamic graphs. In Paul Spirakis, editor, *Algorithms — ESA '95*, pages 171–184, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- 19 Monika Rauch Henzinger and Valerie King. Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation, 1997.

- 20 Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999. Announced at STOC '95. doi:10.1145/320211.320215.
- 21 Monika Rauch Henzinger and Mikkel Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Struct. Algorithms*, 11(4):369–379, 1997. doi:10.1002/(SICI)1098-2418(199712)11:4<369::AID-RSA5>3.0.CO;2-X.
- 22 John Hershberger, Monika Rauch, and Subhash Suri. Data structures for two-edge connectivity in planar graphs. *Theoretical Computer Science*, 130(1):139–161, 1994. doi:10.1016/0304-3975(94)90156-2.
- 23 Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, July 2001. doi:10.1145/502090.502095.
- 24 Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Lacki, and Eva Rotenberg. Incremental SPQR-trees for Planar Graphs. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 46:1–46:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ESA.2018.46.
- 25 Jacob Holm, Giuseppe F Italiano, Adam Karczmarz, Jakub Lacki, Eva Rotenberg, and Piotr Sankowski. Contracting a planar graph efficiently. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 87. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.
- 26 Jacob Holm and Eva Rotenberg. Good r -divisions imply optimal amortised decremental biconnectivity. *CoRR*, abs/1808.02568, 2018. arXiv:1808.02568.
- 27 Jacob Holm, Eva Rotenberg, and Mikkel Thorup. Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 35–52, 2018. doi:10.1137/1.9781611975031.3.
- 28 Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log \log n)^2)$ amortized expected time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 510–520, 2017. doi:10.1137/1.9781611974782.32.
- 29 Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '13*, pages 1131–1142, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics. URL: <http://dl.acm.org/citation.cfm?id=2627817.2627898>.
- 30 Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Faster Worst Case Deterministic Dynamic Connectivity. In Piotr Sankowski and Christos Zaroliagis, editors, *24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 53:1–53:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ESA.2016.53.
- 31 Philip N. Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing, STOC '13*, pages 505–514, New York, NY, USA, 2013. ACM. doi:10.1145/2488608.2488672.
- 32 Richard J. Lipton and Robert E. Tarjan. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.
- 33 Jakub Łącki and Piotr Sankowski. Min-cuts and shortest cycles in planar graphs in $O(n \log \log n)$ time. In *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, pages 155–166, 2011. doi:10.1007/978-3-642-23719-5_14.

- 34 Jakub Łacki and Piotr Sankowski. Optimal decremental connectivity in planar graphs. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, pages 608–621, 2015. doi:10.4230/LIPIcs.STACS.2015.608.
- 35 Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *Proceedings of the 58th Annual Symposium on Foundations of Computer Science, FOCS 2017*, 2017.
- 36 Mihai Patrascu and Erik D Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.
- 37 Bruce Reed and David R. Wood. Fast separation in a graph with an excluded minor. In Stefan Felsner, editor, *2005 European Conference on Combinatorics, Graph Theory and Applications (EuroComb '05)*, volume DMTCS Proceedings vol. AE, European Conference on Combinatorics, Graph Theory and Applications (EuroComb '05) of *DMTCS Proceedings*, pages 45–50, Berlin, Germany, 2005. Discrete Mathematics and Theoretical Computer Science. URL: <https://hal.inria.fr/hal-01184376>.
- 38 Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.
- 39 Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975. doi:10.1145/321879.321884.
- 40 Siamak Tazari and Matthias Müller-Hannemann. Shortest paths in linear time on minor-closed graph classes, with an application to steiner tree approximation. *Discrete Applied Mathematics*, 157(4):673–684, 2009. doi:10.1016/j.dam.2008.08.002.
- 41 Mikkel Thorup. Decremental dynamic connectivity. In *SODA '97*, pages 305–313. SIAM, 1997.
- 42 Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing, STOC '00*, pages 343–350, New York, NY, USA, 2000. ACM. doi:10.1145/335305.335345.
- 43 Christian Wulff-Nilsen. Separator theorems for minor-free and shallow minor-free graphs with applications. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 37–46, 2011. doi:10.1109/FOCS.2011.15.
- 44 Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Encyclopedia of Algorithms*, pages 738–741. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.