Approximating Small Balanced Vertex Separators in Almost Linear Time

Sebastian Brandt and Roger Wattenhofer

ETH Zürich, Switzerland,
brandts@ethz.ch,
wattenhofer@ethz.ch

Abstract. For a graph G with n vertices and m edges, we give a randomized Las Vegas algorithm that approximates a small balanced vertex separator of G in almost linear time. More precisely, we show the following, for any $\frac{2}{3} \leq \alpha < 1$ and any $0 < \varepsilon < 1 - \alpha$: If G contains an α-separator of size K, then our algorithm finds an $(\alpha + \varepsilon)$-separator of size $\mathcal{O}(\varepsilon^{-1} K^2 \log^{1+o(1)} n)$ in time $\mathcal{O}(\varepsilon^{-1} K^3 m \log^{2+o(1)} n)$ w.h.p. In particular, if $K \in \mathcal{O}(\operatorname{polylog} n)$, then we obtain an $(\alpha + \varepsilon)$-separator of size $\mathcal{O}(\varepsilon^{-1} \operatorname{polylog} n)$ in time $\mathcal{O}(\varepsilon^{-1} m \operatorname{polylog} n)$ w.h.p. The presented algorithm does not require knowledge of K.

1 Introduction

1.1 Motivation

In order to solve a large computational problem, the problem is typically divided into smaller parts, and each part is solved on a single processor, in parallel. Some problems can be chopped into pieces in a straight-forward way, e.g., using MapReduce or Spark. Other computational problems cannot be partitioned easily. Such difficult problems can frequently be represented as graphs: Each vertex represents some piece of work whereas an edge between two vertices denotes a relation between the two pieces, i.e., change at one vertex will directly affect the other (and possibly vice versa). There are dozens of software packages for distributed graph processing, e.g., Google's Pregel or PowerGraph.

In order to use multiple processors, the input graph has to be partitioned into multiple components, ideally of similar size. Then the vertices of a component are simulated on a single processor whereas edges between two vertices in different components are handled by the two processors responsible for the two components by exchanging messages. A natural objective of designing such a partition is to reduce the inter-processor communication as it is the expensive part in terms of runtime.

We argue that an input graph should be partitioned by means of a balanced vertex separator (and not a balanced edge cut), since vertex separators are often more efficient. For a simple example, consider a star graph, a tree where one single root is connected to all leaves. We want to partition the star for two processors. A star graph does not feature a small balanced edge cut, whereas

the root is a perfectly good vertex separator. The root is simply replicated on both processors, and communication is reduced to the exchange between the two copies of the root vertex. In general, the computation and communication overhead of a vertex separator is asymptotically never worse than that of a balanced edge cut, whereas in some cases (such as the star graph) it can be a factor of n better, where n is the number of vertices in the graph.

In the last decades, algorithms research has made a lot of progress regarding balanced vertex separators, cf. [2,6,7,8,9,14]. To a large extent, these works focus on the fundamental case of dividing the input graph into two parts. For the remainder of this paper, we will also exclusively consider vertex separators that cut the input graph into two similar-sized pieces.

Even though the algorithms given in the works cited above only need polynomial time, this is often too slow for practical purposes, as partitioning the input graph is the only non-parallel part of the whole process. What is needed is a "quick and dirty" way to compute a balanced vertex separator, i.e., an algorithm that (apart from a polylogarithmic factor) only reads the input once. So far, to the best of our knowledge, it is not known how to compute a balanced vertex separator for general graphs quickly.

Our goal is to *find* a reasonably small balanced vertex separator if there *exists* a small balanced vertex separator, e.g., of polylogarithmic size, and we want to achieve this in almost linear time. For a graph with n vertices and m edges, we show the following, for any $\frac{2}{3} \leq \alpha < 1$ and any $0 < \varepsilon < 1 - \alpha$: If the graph contains an α-balanced vertex separator of size K, then our randomized Las Vegas algorithm finds an $(\alpha + \varepsilon)$-balanced vertex separator of size $\mathcal{O}(\varepsilon^{-1} K^2 \log^{1+o(1)} n)$ in time $\mathcal{O}(\varepsilon^{-1} K^3 m \log^{2+o(1)} n)$ w.h.p. Of course, this result can also be used for other practical applications related to balanced vertex separators, e.g., for determining quickly if a network has serious bottlenecks and locating them in the affirmative case. If no fixed K is considered, by successive doubling we can quickly reach a size K for which an α-separator exists, yielding only an additional small constant factor for the time complexity. In particular, using this technique, our algorithm does not require knowledge of K.

If, on the other hand, the input graph does not contain a small separator, our algorithm will report the lack of such a separator. Note that input graphs without small vertex separators may not be amenable to distributed graph processing in the first place, and one may wonder whether parallelism can speed up processing such input graphs at all.

1.2 Related Work

As discussed above, finding a balanced edge separator does not yield a balanced vertex separator with a similar approximation guarantee in general. Since there is an abundance of results regarding edge separators, we will only mention them if they are also related to vertex separators.

Let $G = (V, E)$ be a graph with n vertices and m edges. An α-separator of G is a triple (A, S, B) of disjoint subsets of V such that $V = A \cup S \cup B$, there are no edges between A and B, and $\max\{|A|, |B|\} \leq \alpha|V|$. Its size is $|S|$.

The problem of finding an α-separator of minimum size is NP-hard, as shown by Bui and Jones [4]. Therefore, one main focus of research in the context of balanced vertex separators has been to find approximation algorithms, cf., e.g., [2,6,7]. In their seminal paper [14], Leighton and Rao gave an $\mathcal{O}(\log n)$-approximation for balanced edge separator, incurring only an arbitrarily small loss in the balance. As they showed, their result extends to the case of directed edge separators and thereby to vertex separators. Feige, Hajiaghayi and Lee [8] proved that a $\frac{3}{4}$-separator of size $K \log^{\frac{1}{2}} K$ can be found in polynomial time if the input graph contains a $\frac{2}{3}$-separator of size K. Subsequently, Feige and Mahdian [9] showed, for any $\frac{2}{3} \leq \alpha < 1$, how to find an α-separator of size K if such a separator exists, except when there is an $(\alpha + \varepsilon)$-separator of smaller size in which case they find the latter. Their runtime is polynomial if $K \in \mathcal{O}(\log n)$, for fixed ε.

As shown by Marx [17], the problem of finding an α-separator of minimum size is even $W[1]$-hard. In their work [9] mentioned above, Feige and Mahdian solve this issue by showing that the problem becomes fixed parameter tractable if the balance requirement is relaxed, obtaining a runtime of $n^{\mathcal{O}(1)}2^{\mathcal{O}(K)}$ which is polynomial for $K \in \mathcal{O}(\log n)$. In our work, we show that if we relax both the balance requirement and the requirement on the size of the separator, then we can achieve an almost linear runtime.

The techniques used in the works listed above, e.g., linear or semidefinite programming, focus on achieving as good approximation ratios as possible while having polynomial time complexity. By applying their primal-dual approach for semidefinite programs [3] to the problem of approximating minimum balanced separators, Arora and Kale achieved a runtime of $\tilde{\mathcal{O}}(m^{\frac{3}{2}} + n^{2+\varepsilon})$ (resp. $\tilde{\mathcal{O}}(m^{\frac{3}{2}})$), for obtaining an approximation ratio of $\mathcal{O}(\log^{\frac{1}{2}} n)$ (resp. $\mathcal{O}(\log n)$). Note that the runtimes and approximation ratios given in their work apply directly to our problem of undirected vertex separators although they are achieved in the context of directed edge separators.

A different line of research consists in searching for primarily fast algorithms that yield separators of not necessarily near-optimal size. For graph classes with certain restrictions, there are a number of results obtaining good runtimes, often at the expense of the separator size depending polynomially on n. Gilbert, Hutchinson and Tarjan [10] gave a linear-time algorithm for finding a $\frac{2}{3}$-separator of size $\mathcal{O}((gn)^{1/2})$ where g is the genus of the given graph, thereby extending the famous planar separator theorem by Lipton and Tarjan [15]. The same linear runtime was achieved independently by Djidjev [5].

A further extension to graphs excluding certain minors was given by Alon, Seymour and Thomas [1]. They showed how to find, for a graph containing no minor K_j for some fixed integer j, a $\frac{2}{3}$-separator of size $\mathcal{O}(n^{\frac{1}{2}})$ in time $\mathcal{O}(n^{\frac{3}{2}})$. Reed and Wood [19] gave an algorithm which solves the same problem in linear time except that the separators are of somewhat larger size $\mathcal{O}(n^{\frac{2}{3}})$. Furthermore, they showed how to trade runtime for separator size in a parametrized way, bounded by those two results. Kawarabayashi and Reed [13] improved the runtime for finding a separator of size $\mathcal{O}(n^{\frac{1}{2}})$ to $\mathcal{O}(n^{1+\varepsilon})$, for any $\varepsilon > 0$, ad-

ditionally improving the dependency of the separator size on the number j of vertices of the excluded minor. Unfortunately, the runtime depends heavily on j, making the algorithm infeasible in practice. Wulff-Nilsen [20] gave an algorithm which depends only polynomially on j, at a slight expense of runtime and separator size. Moreover, he showed how to find, for constant $c < 1$ and fixed j, a separator of size $\mathcal{O}(n^c)$ in linear time. We are not aware of any results for general graphs (regarding balanced vertex separators) that focus on achieving a near-linear time complexity.

As mentioned earlier, recently various software packages to handle large graphs have been introduced, e.g., Pregel [16] or PowerGraph [11]. Some of them include simple heuristics to partition the input graph into pieces. PowerGraph, for instance, merely removes vertices with large degrees until the graph falls into small enough pieces. In practice, this seems to work well on power-law graphs, which include many interesting application areas such as, e.g., social networks. We believe that our work will help to find a theoretical foundation for this practical problem while also providing an implementable solution.

1.3 Our Approach

In the following, we give a descriptive explanation of our approach without providing formal accuracy. Exact definitions will follow in the next section. The approach we take is based on maximum s-t-flows. By the very nature of flows, it is likely that such an approach can only find a near-optimally sized balanced vertex separator *quickly* if the considered graph actually contains a reasonably small balanced vertex separator. As explained before, this restricted problem is still very important in practice, thus we deem the presented approach to be a worthwile endeavour while having the advantage of (conceptual) simplicity.

Assume we are given a graph G containing a small vertex separator and we have vertices s and t "on different sides" of the separator. Then, by Menger's Theorem (cf. [18]), the maximum number of pairwise vertex-disjoint s-t-paths is also small.

We start by computing a set of maximum cardinality of pairwise vertex-disjoint s-t-paths. By using the Ford-Fulkerson algorithm (cf. [12]), this can be done in almost linear time as such a path collection corresponds to a maximum s-t-flow in an unweighted directed graph obtained from G by a simple transformation. From this collection of k paths we extract s-t-vertex cuts of the same cardinality k by taking one vertex from each path. These vertices have to be chosen carefully in order to actually separate s and t, but the existence of the s-t-vertex cuts is ensured, again, by Menger's Theorem.

Using binary search, we determine two of the "best-balanced" of all these s-t-cuts, one closer to s and one closer to t. If one of these two cuts is sufficiently balanced, then we have found the desired small balanced vertex separator. Otherwise, consider the connected components cut off by the two s-t-cuts. We contract the connected component containing s into a new vertex s' and the component containing t into the vertex t'.

All s'-t'-vertex cuts in the newly obtained graph are also s-t-cuts in G and additionally better-balanced than the above two s-t-cuts. We will prove that the maximum number of pairwise vertex-disjoint s'-t'-paths is larger than k (and therefore the same is true for the cardinality of any s'-t'-cut corresponding to such a path collection). We iterate the above process of finding vertex-disjoint paths, extracting some of the best-balanced s-t-cuts and contracting vertex sets, until we obtain s-t-cuts whose cardinality is equal to some predetermined value K (or observe that no such cut of cardinality K exists).

Consider an α-separator of size at most K which separates s and t, for some $\frac{2}{3} \leq \alpha < 1$. If the iterative process described above does not yield cuts whose (combined) balance is at least as good as α, then, as we will prove, at least one vertex of the α-separator must have been involved in one of the performed contractions. Thus, by iterating the whole (iterative) process at most K times (with newly chosen s, t in each iteration), we obtain a balanced vertex separator (by collecting all the relevant cuts obtained in the process). We will show that if G contains a small balanced vertex separator, then the obtained balanced vertex separator is also small.

Up to now, we assumed that we can find vertices s and t "on different sides" of a balanced separator. But because of the balance of the separator, this is actually the case with a large enough probability. By choosing s and t uniformly at random, applying the iterative process described above and then iterating the whole procedure always on the largest obtained connected component, we obtain an almost linear runtime for finding a reasonably small balanced vertex separator, provided the given graph contains a small balanced vertex separator.

2 Conventions and Basic Definitions

In this work, we always assume that a considered graph is simple, undirected, connected and contains no self-loops, if not specified otherwise.[1] Let $G = (V, E)$ be a graph. When we consider a graph denoted differently, say by H, we denote the vertex set correspondingly by $V(H)$ and the edge set by $E(H)$. We denote the number of vertices of G by n and the number of edges of G by m. Since G is connected, we have $n \in \mathcal{O}(m)$.

We call a triple (A, S, B) of pairwise disjoint subsets of V a *vertex separator* of G if $V = A \cup S \cup B$ and there is no $\{u, v\} \in E$ such that $u \in A, v \in B$. We call $|S|$ the *size* of (A, S, B). Let $0 < \alpha < 1$. If $\max\{|A|, |B|\} \leq \alpha|V|$, then we call (A, S, B) *α-balanced* or, equivalently, an α-separator. Let $s, t \in V$ such that $s \in A$, $t \in B$. Then we call (A, S, B) an *s-t-vertex separator*.

Let G be an undirected or directed graph and $v_0, ..., v_j \in V, j \in \mathbb{N}$. Then we call the tuple $q = (v_0, ..., v_j)$ a *path* from v_0 to v_j (or, equivalently, a v_0-v_j-path) if we have $\{v_i, v_{i+1}\} \in E$ (resp. $(v_i, v_{i+1}) \in E$) for all $0 \leq i \leq j-1$ and $v_i \neq v_{i'}$ for all $0 \leq i < i' \leq j$. We call j the *length* of q. For all pairs (i, i') satisfying

[1] Note that the main result holds also for graphs that are not simple as we study vertex separators which, by their nature, do not care if there are multiple edges between two vertices.

$0 \leq i < i' \leq j$, we call v_i a *predecessor* of $v_{i'}$ in q and $v_{i'}$ a *successor* of v_i in q. If $i' = i + 1$, then we call $v_{i'}$ the *direct successor* of v_i in q. Furthermore, for all $0 \leq i \leq i' \leq j$, we call $q' = (v_i, v_{i+1}, ..., v_{i'})$ a *subpath* of q.

Let $s, t \in V, s \neq t$ and let $\{f_1, ..., f_k\}, k \in \mathbb{N}_{>0}$ be a set of s-t-paths in G. Then we say that $f_1, ..., f_k$ are *pairwise vertex-disjoint* if there are no vertices except s and t that appear in more than one of these paths. Moreover, in the case of a directed graph, we say that $f_1, ..., f_k$ are *pairwise edge-disjoint* if there are no two vertices $x, y \in V$ such that y is the direct successor of x in at least two of these paths.

Let H be a subgraph of a not necessarily connected graph G. Then we call H an *induced subgraph* of G if each edge of G between vertices of H is also an edge of H. We call an induced subgraph H a *connected component* of G if there is a path from v to w in H for all pairs of vertices $v, w \in V(H)$ and there is no path in G from any vertex in $V(H)$ to any vertex in $V \backslash V(H)$. If a connected component of G contains at least as many vertices as any other connected component of G, we call it a *largest* connected component. Furthermore, for all subsets $X \subseteq V(G)$, we denote the induced subgraph of G whose vertex set is X by $G[X]$.

It will occur quite frequently that we consider two special vertices s, t. For the remainder of this work, we will assume that s and t are different, non-adjacent vertices if not specified otherwise. Furthermore, for convenience, we will be not too technical regarding the distinction between sets and tuples. Thus, e.g., we may consider a tuple as a subset of some set, forgetting the order of the elements in the tuple, or we may consider a set as a tuple when the intended order of the elements of the set is clear.

In 1927, Karl Menger [18] stated a famous theorem which we will use at various points in this work. It can be formulated as follows:

Theorem 1. *The maximum number of pairwise vertex-disjoint s-t-paths in a graph G is equal to the minimum number of vertices v, $s \neq v \neq t$, which have to be removed from G in order that there is no s-t-path in the resulting graph.*

Consider a set of maximum cardinality of pairwise vertex-disjoint s-t-paths. By Menger's Theorem, we can disconnect s from t by removing one vertex from each of those paths. Of course, if we choose these vertices arbitrarily (but still one per path), then it is not ensured that there is no s-t-path left. We call such a set of arbitrarily chosen vertices a *slice* whereas we call it a *cut* if its removal results in a disconnection of s from t. In more formal terms:

Definition 2. *Let G be a graph and $s, t \in V$. Let $\{f_1, ..., f_k\}$ be a set of pairwise vertex-disjoint s-t-paths in G. Then we call a tuple $(w_1, ..., w_k)$ a* slice *(with respect to $(f_1, ..., f_k)$) if $w_i \in f_i$, $s \neq w_i \neq t$ for all $1 \leq i \leq k$. Let X be an arbitrary subset of V such that $s, t \notin X$. If there is no s-t-path in $G[V \backslash X]$, then we say that X* separates *s and t. We call a slice that separates s and t a* cut.[2]

[2] For ease of presentation, we refrain from calling it an s-t-cut which would be the technically precise term.

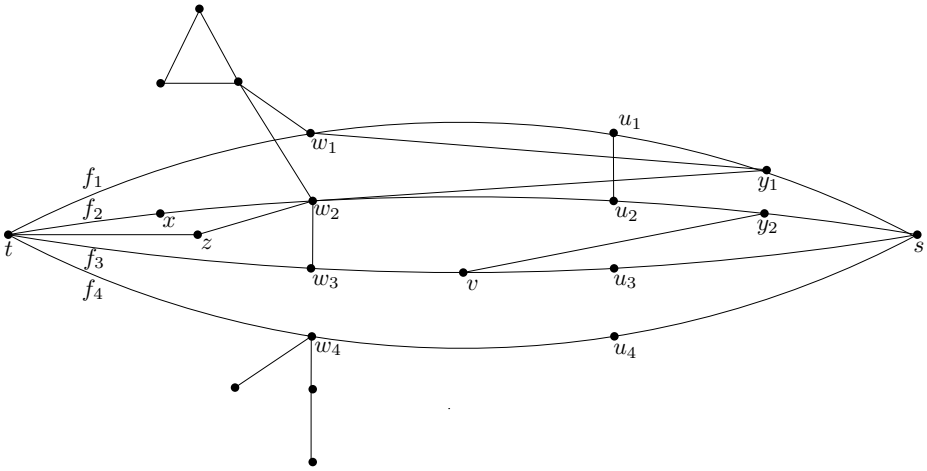

Fig. 1. $U := (u_1, ..., u_4)$ and $W := (w_1, ..., w_4)$ are slices with respect to $(f_1, ..., f_4)$. W is a cut whereas U does not separate s and t. We have $U \prec W$, since U is strictly closer to s than W. Furthermore, we have $V_s(W) = \{s, y_1, y_2, u_1, u_2, u_3, u_4, v\}$ and $V_t(W) = \{t, x, z\}$. $V_r(W)$ contains exactly the unnamed vertices.

Note that the existence of a cut with respect to a set of pairwise vertex-disjoint s-t-paths $(f_1, ..., f_k)$ implies that $(f_1, ..., f_k)$ is of maximum cardinality, i.e., there is no set of cardinality larger than k of pairwise vertex-disjoint s-t-paths. This is the case since a set that separates s and t (which includes cuts) must contain at least one vertex of each s-t-path.

Thus, in the following, by considering a cut we specify implicitly that the respective set of pairwise vertex-disjoint s-t-paths is of maximum cardinality. Moreover, whenever we consider a set $\{f_1, ..., f_k\}$ of s-t-paths, we assume that they are pairwise vertex-disjoint if not specified otherwise.

Following Marx [17], the set of slices (with respect to some fixed set of s-t-paths) can be partially ordered by their relative "closeness" to s. The following definition adapts the definition of the "dominance relation" given in [17] to our setting.

Definition 3. *Let $\{f_1, ..., f_k\}$ be a set of pairwise vertex-disjoint s-t-paths in G. Let $U = (u_1, ...u_k)$ and $W = (w_1, ..., w_k)$ be slices with respect to $(f_1, ..., f_k)$ such that, for all $1 \leq i \leq k$, u_i is a predecessor of w_i in f_i or $u_i = w_i$. Then we say that U is* closer to s *than W and write $U \preceq W$. If we have additionally $u_i \neq w_i$ for some $1 \leq i \leq k$, then we say that U is* strictly closer to s *than W and write $U \prec W$. Analogously, we say that W is* (strictly) closer to t *than U. For convenience, we define the above analogously for the tuples $(s, s, ..., s)$ and $(t, t, ..., t)$. Thus we can, e.g., say that $(s, s, ..., s)$ is closer to s than any slice. Note that "$\preceq$" does not define a total order in general.*

The removal of a cut decomposes G into at least two connected components as s and t are not connected anymore. The component containing s and the component containing t are of special interest to us since we will develop a method to make them larger (by choosing "better" cuts) which, in turn, aids in finding cuts (or, more precisely, s-t-vertex separators) of "better balance".

Definition 4. *Let U be an arbitrary cut. We define $V_s(U)$ as the vertex set of the connected component of $G[V \backslash U]$ containing s, $V_t(U)$ as the vertex set of the connected component of $G[V \backslash U]$ containing t and $V_r(U)$ as the union of the vertex sets of the remaining connected components of $G[V \backslash U]$, i.e., those containing neither s nor t (so $V_r(U)$ may be empty).*

An s-t-vertex separator differs from a cut in that it specifies (in addition to the "cut vertices") how the connected components, that we obtain by removing these cut vertices from G, are divided into two groups (which then constitute the two "sides" of the separator). As it is often more convenient not to specify such a fixed division, we will frequently work with cuts instead of using vertex separators. For an illustration of the definitions given above, consider Fig. 1.

Furthermore, for the remainder of this paper, let $\{f_1, ..., f_k\}$ be a set of maximum cardinality of pairwise vertex-disjoint s-t-paths.

3 Slices and Cuts

In this section, we collect a number of lemmas which we require for the analyses of the algorithms provided in Section 4. We start with a short examination of the properties of cuts. Then we proceed by introducing the notion of a "closest cut" for a given slice and prove a number of results regarding such closest cuts. In particular, we show that we can compute these cuts in time $\mathcal{O}(m)$.

3.1 Properties of Cuts

We observe that it follows directly from the definitions of $V_s(U)$ and $V_t(U)$, in which of the two sets $V_s(U)$ and $V_t(U)$ the predecessors and successors of a vertex in a cut U lie.

Observation 5. *Let $U = (u_1, ..., u_k)$ be a cut with respect to $(f_1, ..., f_k)$. Then, for all $1 \leq i \leq k$, all predecessors of u_i in f_i are elements of $V_s(U)$ and all successors of u_i in f_i are elements of $V_t(U)$.*

Whenever two cuts are comparable with respect to "$\preceq$", we obtain a number of useful set inclusions regarding the images of those cuts under $V_s()$, $V_t()$ and $V_r()$.

Lemma 6. *Let $U = (u_1, ..., u_k)$ and $W = (w_1, ..., w_k)$ be cuts such that $U \preceq W$. Then we have $V_s(U) \subseteq V_s(W)$ and $V_r(W) \cup V_t(W) \subseteq V_r(U) \cup V_t(U)$. Furthermore, we have $V_t(W) \subseteq V_t(U)$ and $V_r(U) \cup V_s(U) \subseteq V_r(W) \cup V_s(W)$.*

Proof. Let $x \in V_s(U)$. Then, by the definition of $V_s(U)$, there is a path $q = (v_0, ..., v_j)$ in $G[V \backslash U]$ where $v_0 = s$ and $v_j = x$.

Suppose that the path q contains a vertex which is also an element of W, i.e., there are some $0 \leq h \leq j$ and $1 \leq i \leq k$ such that $v_h = w_i$. Since we have $U \preceq W$, no successor of w_i in f_i is an element of U. Thus, there is a w_i-t-path in $G[V \backslash U]$. Now $v_h = w_i$ ensures that there is an s-w_i-path in $G[V \backslash U]$ yielding the existence of an s-t-path in $G[V \backslash U]$. But since U separates s and t, this is a contradiction. Therefore, q contains no element of W which implies that there is an s-x-path in $G[V \backslash W]$. Thus, by the definition of $V_s(W)$, $x \in V_s(W)$ and we obtain $V_s(U) \subseteq V_s(W)$.

Now, consider some $u_l \in U$. By Observation 5, all predecessors of w_l in f_l are elements of $V_s(W)$. Since $U \preceq W$, we obtain that either $u_l \in W$ or $u_l \in V_s(W)$. Combined with $V_s(U) \subseteq V_s(W)$ (as shown above), we get $V_s(U) \cup U \subseteq V_s(W) \cup W$ which implies $V_r(W) \cup V_t(W) \subseteq V_r(U) \cup V_t(U)$. The remaining inclusions given in the lemma follow analogously.

3.2 U^+ and U^-

Consider a slice U. Among all cuts that are closer to t (resp. s) than U, we would like to single the "closest one" out. Our partial order "$\preceq$" provides a very intuitive way to do so, but before we can actually define the "closest cut" accordingly, we have to show the uniqueness of such a cut. Similar results have been shown in the literature, but for completeness sake we give a full proof in our setting.

For that, we need to define another intuitive concept. If two cuts are not comparable with respect to "$\preceq$", there is a slice which is closer to s than both of the aforementioned cuts, but still as close to them as possible under this constraint. We formalize this with the following definition (for arbitrary pairs of cuts) and show in the subsequent lemma that the defined slice separates s and t, i.e., it is a cut.

Definition 7. *Let $U = (u_1, ..., u_k)$ and $W = (w_1, ..., w_k)$ be cuts with respect to $(f_1, ..., f_k)$. Then, for all $1 \leq i \leq k$, we define*

$$\psi_s(u_i, w_i) := \begin{cases} u_i & \text{if } u_i \text{ is a predecessor of } w_i \text{ in } f_i \text{ or } u_i = w_i \\ w_i & \text{if } w_i \text{ is a predecessor of } u_i \text{ in } f_i \end{cases}$$

and

$$\psi_t(u_i, w_i) := \begin{cases} u_i & \text{if } u_i \text{ is a successor of } w_i \text{ in } f_i \text{ or } u_i = w_i \\ w_i & \text{if } w_i \text{ is a successor of } u_i \text{ in } f_i \end{cases}$$

We set $\lim_s(U, W) := (\psi_s(u_1, w_1), ..., \psi_s(u_k, w_k))$ *and analogously* $\lim_t(U, W) := (\psi_t(u_1, w_1), ..., \psi_t(u_k, w_k))$.

Consider, e.g., the cuts $X := (y_1, u_2, v, w_4)$ and $Y := (w_1, w_2, v, u_4)$ from Fig. 1. We obtain $\lim_s(X, Y) = (y_1, u_2, v, u_4)$ and $\lim_t(X, Y) = (w_1, w_2, v, w_4)$. Both of these tuples are again cuts. As the following lemma shows, this is the case in general.

Lemma 8. *Both* $\lim_s(U, W)$ *and* $\lim_t(U, W)$ *are cuts.*

Proof. Suppose there is an s-t-path q in $G[V \setminus \lim_s(U, W)]$. Let v be the first vertex in q such that there is a $1 \leq i \leq k$ such that $v \in f_i$ and $\psi_s(u_i, w_i)$ is a predecessor of v in f_i.[3] Consider the respective i. There are two cases:

First suppose that u_i is a predecessor of w_i in f_i or $u_i = w_i$. Then the subpath of q from s to v contains no element of U, by the definition of v. Furthermore, the subpath of f_i from v to t also contains no element of U since $u_i = \psi_s(u_i, w_i)$ is a predecessor of v in f_i. Thus, we obtain a path from s to t which contains no element of U contradicting the fact that U separates s and t.

Now consider the complementary case, i.e., that w_i is a predecessor of u_i in f_i. With an argumentation analogous to the first case we obtain a contradiction to the fact that W separates s and t. Thus, there is no s-t-path in $G[V \setminus \lim_s(U, W)]$ and $\lim_s(U, W)$ is a cut. The statement for $\lim_t(U, W)$ follows by an analogous proof.

The definitions of $\lim_s(U, W)$ and $\lim_t(U, W)$ enable us to prove the following lemma. It states that if there is a cut which is closer to t than a given slice, then there is a unique "closest" cut which is closer to t than the given slice.

Lemma 9. *Let U be a slice with respect to $(f_1, ..., f_k)$. If there is a cut which is closer to t than U, then there is exactly one such cut X with the property that there is no cut X' satisfying $U \preceq X' \prec X$.*

Proof. If there is a cut which is closer to t than U, then there is at least one cut with the property described in the lemma since "$\preceq$" defines a partial order. Thus, it is enough to show that there are not two such cuts.

Let W and X be two cuts with this property. Consider $\lim_s(W, X)$. The definition of $\lim_s(W, X)$ ensures that $U \preceq \lim_s(W, X) \preceq W$ and $U \preceq \lim_s(W, X) \preceq X$. Since W and X satisfy the aforementioned property and since, by Lemma 8, $\lim_s(W, X)$ is a cut, we have $\lim_s(W, X) \not\prec W$ and $\lim_s(W, X) \not\prec X$ which implies $W = \lim_s(W, X) = X$. Therefore, there is exactly one cut with the property described in the lemma.

Of course, there is also an analogous statement for "closest cuts" that are closer to s than the given slice. It can be proved analogously.

Lemma 10. *Let U be a slice with respect to $(f_1, ..., f_k)$. If there is a cut which is closer to s than U, then there is exactly one such cut X with the property that there is no cut X' satisfying $X \prec X' \preceq U$.*

Justified by Lemma 9 and Lemma 10, we define in the following the notion of a "closest cut" for a given slice.

Definition 11. *Let U be a slice with respect to $(f_1, ..., f_k)$. Let X be a cut such that $U \preceq X$ and there is no cut X' satisfying $U \preceq X' \prec X$. Then we define $U^+ := X$. If there exists no X as described above, then set $U^+ := (t, t, ..., t)$.*

[3] Such a v must exist since for $v = t$ an i as described above exists.

Algorithm 1 Compute U^+

Initialization: Given the maximum number of pairwise vertex-disjoint s-t-paths $f_1, ..., f_k$ and a slice $U = (u_1, ..., u_k)$ with respect to $(f_1, ..., f_k)$, set $X := \{\}$ and $w_i := u_i$, for all $1 \leq i \leq k$.

for $i = 1$ to k **do**
  add all predecessors of w_i in f_i to X // X collects vertices of $V_s(U^+)$
end for
$Y := V \backslash (X \cup U)$ // Y contains all vertices which possibly are not in $V_s(U^+) \cup U^+$
while there is some edge $\{x, y\} \in E$ with $x \in X$, $y \in Y$ **do**
  if $y = t$ **then**
    return $(t, t, ..., t)$
  else if y is an element of f_i for some $1 \leq i \leq k$ **then**
    add to X all predecessors of y in f_i that are also successors of w_i in f_i
    delete these predecessors from Y
    add vertex w_i to X // w_i must be in $V_s(U^+)$
    delete vertex y from Y // y must be in $V_s(U^+) \cup U^+$
    $w_i := y$
  else
    add vertex y to X and delete it from Y // y must be in $V_s(U^+)$
  end if
end while
return $(w_1, ..., w_k)$

Analogously, let Y be a cut such that $Y \preceq U$ and there is no cut Y' satisfying $Y \prec Y' \preceq U$. Then we define $U^- := Y$. If there exists no Y as described above, then set $U^- := (s, s, ..., s)$.

For all slices U, we obtain $U^- \preceq U \preceq U^+$. Note that if U is a cut, then we have $U^- = U = U^+$. Consider once again Fig. 1. We have $W \neq U^+$ since there are cuts which are closer to t than U and strictly closer to s than W, e.g., $Y := (w_1, w_2, v, u_4)$. Indeed, we have $U^+ = Y$. For the slice $Z := (w_1, x, w_3, w_4)$ we obtain $Z^+ = (t, t, ..., t)$ since there is no cut which is closer to t than Z.

3.3 Computing U^+ and U^-

Lemma 12. *For any slice U with respect to $(f_1, ..., f_k)$, we can compute U^+ and U^- in time $\mathcal{O}(m)$.*

Proof. For reasons of symmetry, it is enough to show that we can compute U^+ in time $\mathcal{O}(m)$. In the following we show that Algorithm 1 returns U^+ and runs in time $\mathcal{O}(m)$, thus proving the lemma. For convenience, set $W := (w_1, ..., w_k)$. Furthermore, since $V_s(U^+)$ is not defined if $U^+ = (t, t, ..., t)$, set $V_s(U^+) := V \backslash \{t\}$ in this case.[4]

[4] This applies not only for the remainder of the proof, but also for the comments in Algorithm 1.

The basic idea of Algorithm 1 is to add to the set X more and more vertices of $V_s(U^+)$. For this, we observe that if a vertex in some f_i is in $V_s(U^+) \cup U^+$, then all predecessors of this vertex in f_i are in $V_s(U^+)$, by Observation 5. Furthermore, if there is an edge $\{v, w\} \in E$ with $v \in V_s(U^+)$, then we have $w \in V_s(U^+) \cup U^+$ (by the definition of $V_s(U^+)$) and if, additionally, none of the s-t-paths f_i contains w, then it cannot be an element of U^+ which implies $w \in V_s(U^+)$. Therefore, the vertices added to X in the while loop of Algorithm 1 are in $V_s(U^+)$ since at the start of the first iteration of the while loop we have $X \subseteq V_s(U^+)$, by Observation 5 (and since, by induction, we have $X \subseteq V_s(U^+)$ at the start of each iteration).

The algorithm terminates when it looks at an edge in G from some vertex in X to t or when there is no edge from X to Y. In the first case, recalling the above considerations we see that $t \in V_s(U^+) \cup U^+$ which implies $U^+ = (t, t, ..., t)$ and Algorithm 1 returns U^+. In the second case, t is contained in Y, thus W separates s and t and we obtain $U^+ = W$. Note that since each predecessor of some w_i in f_i is contained in $V_s(U^+)$ (as shown above), there cannot be a cut Z such that $U \preceq Z \prec W$.

Now consider the runtime of Algorithm 1. For checking if there is some edge $\{x, y\} \in E$ with $x \in X$, $y \in Y$, we run through all vertices in X and check for each vertex for all incident edges if the other endpoint is in Y. After we have checked all incident edges of a vertex, we do not have to consider this vertex again as it is (and stays) contained in X and none of its neighbors is contained in Y (anymore) or will be contained in Y at a later stage during the algorithm. Thus, across all the iterations of the while loop each edge is checked at most twice, so the checking can be done in time $\mathcal{O}(m)$. Moreover, each vertex is moved at most twice between the sets in the algorithm, since vertices are moved only from Y to W, from W to X or from Y to X. Therefore, this part of the algorithm can be done in time $\mathcal{O}(n)$, thus also in time $\mathcal{O}(m)$. The rest of the algorithm can be done in time $\mathcal{O}(m)$ as well, so the overall runtime is $\mathcal{O}(m)$.

3.4 Properties of U^+ and U^-

The following lemma shows that the partial order given by "$\preceq$" is preserved under $()^+$ and $()^-$.

Lemma 13. *Let U and W be slices with respect to $(f_1, ..., f_k)$ such that $U \preceq W$. Then $U^+ \preceq W^+$ and $U^- \preceq W^-$.*

Proof. If $W^+ = (t, t, ..., t)$, then $U^+ \preceq W^+$. If $W^+ \neq (t, t, ..., t)$, then W^+ is a cut and $U \preceq W \preceq W^+$ implies that there is a cut X satisfying $U \preceq X \preceq W^+$ such that there is no cut X' satisfying $U \preceq X' \prec X$, by Lemma 9. By definition, we have $U^+ = X$ which implies $U^+ \preceq W^+$. By an analogous proof, we obtain $U^- \preceq W^-$.

Note that if $U \prec W$, then it does not necessarily follow that $U^+ \prec W^+$ since we could have $U^+ = W^+$. Analogously, $U \prec W$ does not imply $U^- \prec W^-$.

Later in this work, we will contract subsets of V in order to obtain a smaller graph. These subsets will contain either s or t and we will call the vertex resulting from a contraction also s, resp. t. In order to keep track of the number of contracted vertices, we will assign weights $g(s)$ and $g(t)$ to s and t which count the number of vertices which are "added" to s, resp. t, in a contraction. The initial weights will be $g(s) = g(t) = 1$. Keeping this in mind, we can proceed to the following lemma. It is constructed specifically for the design and the analysis of the first algorithm given in the next section (Algorithm 2).

Lemma 14. *Let $g(s), g(t)$ be positive integers. Let $U = (u_1, ..., u_k)$ and $W = (u_1, ..., u_{i-1}, u_i^+, u_{i+1}, ..., u_k)$ be slices where $1 \leq i \leq k$ and u_i^+ denotes the direct successor of u_i in f_i. Then the following holds:*

If $U^+ = (t, t, ..., t)$ or $|V_s(U^+)| + g(s) > |V_r(U^+) \cup V_t(U^+)| + g(t)$, then $W^+ = (t, t, ..., t)$ or $|V_s(W^+)| + g(s) > |V_r(W^+) \cup V_t(W^+)| + g(t)$.

Proof. We observe that $U \preceq W$. By Lemma 13, if $U^+ = (t, t, ..., t)$, then $W^+ = (t, t, ..., t)$. Now suppose that $U^+ \neq (t, t, ..., t)$ and $|V_s(U^+)| + g(s) > |V_r(U^+) \cup V_t(U^+)| + g(t)$. Again by Lemma 13, we obtain $U^+ \preceq W^+$ which implies $W^+ = (t, t, ..., t)$ or $V_s(U^+) \subseteq V_s(W^+)$ and $V_r(W^+) \cup V_t(W^+) \subseteq V_r(U^+) \cup V_t(U^+)$, by Lemma 6. In the latter case, it follows that $|V_s(W^+)| + g(s) > |V_r(W^+) \cup V_t(W^+)| + g(t)$.

An analogous version of the lemma regarding U^-, W^-, etc. holds as well.

4 An Algorithm for Finding Good Cuts of Bounded Size

Consider a graph G which contains an s-t-vertex separator of size at most K, for some fixed s, t. The goal of this section is to find a good cut of size at most K. Intuitively, a cut U is "good" if the connected components obtained by removing U from G can be divided into two groups in a balanced way. Unfortunately, it is not easy to find such a cut quickly. Thus, we relax our notion of "good" slightly and say that a cut U, that does not admit a balanced partition of the connected components, is still good if it satisfies the following property: For any s-t-vertex separator (A^*, S^*, B^*) of size at most K with a better balance than U, S^* is not contained in any of the connected components obtained by removing U. The idea is to iterate the process of finding a good cut on the largest component obtained by removing the (previous) good cut and to benefit from the fact that the size of (A^*, S^*, B^*) restricted to this component decreases by at least 1 in each iteration. The details of this idea will be discussed in Section 5.

We begin this section by gathering the tools (lemmas and algorithms) we need for the design and the analysis of an algorithm that finds a good cut. Then we formulate the algorithm (Algorithm 4) itself, analyze its runtime and show that it indeed finds a good cut (or, more precisely, a pair of tuples which contains a good cut).

Algorithm 2 Find Innermost s-sided Cut of Minimum Size

Initialization: Given weights $g(s), g(t) \in \mathbb{N}_{>0}$ and the maximum number of pairwise vertex-disjoint s-t-paths $f_1, ..., f_k$, let $v_{ij}, 0 \leq j \leq \ell_i$, be the jth vertex of the path f_i where ℓ_i is the length of f_i and s is considered to be the 0th vertex of every f_i. Set $w_i := v_{i1}$ for all $1 \leq i \leq k$ and valid := **false**.

for $i = 1$ to k **do**
  $c := 0$ // indexes the start of path f_i
  $d := \ell_i$ // indexes the end of path f_i
  while $d \neq c + 1$ **do**
    $e := \lceil \frac{c+d}{2} \rceil$ // binary search on f_i
    $W := (w_1, ..., w_{i-1}, v_{ie}, w_{i+1}, ..., w_k)$ // obtain new slice by moving vertex on f_i
    if $W^+ \neq (t, t, ..., t)$ and $|V_s(W^+)| + g(s) \leq |V_r(W^+) \cup V_t(W^+)| + g(t)$ **then**
      $c := e$ // W is suitable, continue binary search in direction towards t
      valid := **true** // suitable cut found
    else
      $d := e$ // W is not suitable, continue binary search in direction towards s
    end if
  end while
  $w_i := v_{ic}$ // fix best vertex found on f_i, continue with next path
end for
if valid **then**
  return $(w_1, ..., w_k)$
else
  return $(s, s, ..., s)$
end if

4.1 Cuts of Minimum Size

The first step in order to design such an algorithm is to develop a method for finding a cut U (if it exists) such that $|V_s(U)| \leq |V_r(U) \cup V_t(U)|$ and all cuts which are strictly closer to t than U violate that property. Since the space of slices is possibly quite large, simply checking, for all slices W, if W is a cut and then potentially computing the sizes of $V_s(W)$ and $V_r(W) \cup V_t(W)$ will not achieve this goal quickly. Instead, we solve the problem by searching the space of slices in a more efficient way using binary search, as given by Algorithm 2. Essentially, this algorithm moves a vertex of some initial slice closer to t along an s-t-path, thereby obtaining a new slice, and checks if the cut closest to this slice in direction towards t still satisfies the aforementioned inequality. In the affirmative case, it iterates starting from this new slice, otherwise it goes back and tries another vertex. Taking into account that, later on, we will have to deal with graphs which are the result of a series of contractions on an initially given graph, we design Algorithm 2 in a rather general way where we have weights assigned to s and t.

Lemma 15. *Let $g(s), g(t)$ be positive integers. If there is a cut X such that $|V_s(X)| + g(s) \leq |V_r(X) \cup V_t(X)| + g(t)$, then Algorithm 2 returns such a cut X with the additional property that $|V_s(X')| + g(s) > |V_r(X') \cup V_t(X')| + g(t)$ for all cuts X' satisfying $X \prec X'$. If there is no such cut, then the algorithm returns the tuple $(s, s, ..., s)$. In both cases Algorithm 2 terminates in time $\mathcal{O}(km \log n)$.*

Proof. First, we prove the statements regarding the output of Algorithm 2. If there is no cut as described in the lemma, then our boolean variable valid remains false throughout the whole algorithm and the tuple $(s, s, ..., s)$ is returned. Now consider the case that there is such a cut. In order to avoid confusion, denote the output of Algorithm 2 by $X = (x_1, ..., x_k)$.

Applying Lemma 14, we see that the while loop finds the largest c (if such an c exists) such that $U^+ \neq (t, t, ..., t)$ and $|V_s(U^+)| + g(s) \leq |V_r(U^+) \cup V_t(U^+)| + g(t)$ where $U := (w_1, ..., w_{i-1}, v_{ic}, w_{i+1}, ..., w_k)$. Thus, if at the start of the ith iteration of the for loop (at which point we have $w_i = v_{i1}$) the slice $W' := (w_1, ..., w_k)$ satisfies $W'^+ \neq (t, t, ..., t)$ and $|V_s(W'^+)| + g(s) \leq |V_r(W'^+) \cup V_t(W'^+)| + g(t)$, then at the end of the ith iteration this is also the case. Since each slice is closer to t than the slice $(v_{11}, ..., v_{i1})$, we know, by Lemma 13 and Lemma 6, that at the start of the algorithm we have $W'^+ \neq (t, t, ..., t)$ and $|V_s(W'^+)| + g(s) \leq |V_r(W'^+) \cup V_t(W'^+)| + g(t)$. Thus, by induction, this is also the case at the end of the algorithm, i.e., we have $X^+ \neq (t, t, ..., t)$ and $|V_s(X^+)| + g(s) \leq |V_r(X^+) \cup V_t(X^+)| + g(t)$.

Suppose there is a cut $Y = (y_1, ..., y_k)$ such that $X \prec Y$ and $|V_s(Y)| + g(s) \leq |V_r(Y) \cup V_t(Y)| + g(t)$. Let i' be the smallest index such that $y_{i'} \neq x_{i'}$ (which implies that $x_{i'}$ is a predecessor of $y_{i'}$ in $f_{i'}$). Consider the slice $Z = (x_1, ..., x_{i'-1}, y_{i'}, v_{(i'+1)1}, ..., v_{k1})$. Since $Z \preceq Y$ and $Y = Y^+$ (since Y is a cut), we have $Z^+ \preceq Y$, by Lemma 13. We obtain $V_s(Z^+) \subseteq V_s(Y)$ and $V_r(Y) \cup V_t(Y) \subseteq V_r(Z^+) \cup V_t(Z^+)$, by Lemma 6. It follows that $|V_s(Z^+)| + g(s) \leq |V_r(Z^+) \cup V_t(Z^+)| + g(t)$. Recalling the description of what the while loop accomplishes, we obtain that, at the end of the i'th iteration of the for loop, $w_{i'}$ is a vertex which is a successor of $x_{i'}$. Since $w_{i'}$ remains unchanged throughout the rest of the algorithm, this yields a contradiction.

Therefore, we have $X \not\prec X^+$ (as otherwise X^+ would constitute a Y as described above whose existence we just disproved) which implies $X = X^+$. Thus, X is a cut and combining this fact with the above considerations, we can conclude that the output of Algorithm 2 is indeed as described in the lemma.

The runtime given in the lemma follows from $\ell_i \leq n$ for all $1 \leq i \leq k$ (so there are at most $\lceil \log n \rceil$ iterations of the while loop in each iteration of the for loop), the fact that computing $V_r(W^+)$, $V_s(W^+)$ and $V_t(W^+)$ in the while loop takes time $\mathcal{O}(m)$ and by Lemma 12.

For reasons of symmetry, Algorithm 2 and Lemma 15 also work if s and t are reversed. The respective versions are given in the following.

Algorithm 3 Find Innermost t-sided Cut of Minimum Size

Initialization: Given weights $g(s), g(t) \in \mathbb{N}_{>0}$ and the maximum number of pairwise vertex-disjoint s-t-paths $f_1, ..., f_k$, let $v_{ij}, 0 \leq j \leq \ell_i$, be the jth vertex of the path f_i where ℓ_i is the length of f_i and s is considered to be the 0th vertex of every f_i. Set $w_i := v_{i(\ell_i - 1)}$ for all $1 \leq i \leq k$ and valid := **false**.

for $i = 1$ to k **do**
  $c := 0$ // indexes the start of path f_i
  $d := \ell_i$ // indexes the end of path f_i
  while $d \neq c + 1$ **do**
    $e := \lfloor \frac{c+d}{2} \rfloor$ // binary search on f_i
    $W := (w_1, ..., w_{i-1}, v_{ie}, w_{i+1}, ..., w_k)$ // obtain new slice by moving vertex on f_i
    if $W^- \neq (s, s, ..., s)$ and $|V_t(W^-)| + g(t) \leq |V_r(W^-) \cup V_s(W^-)| + g(s)$ **then**
      $d := e$ // W is suitable, continue binary search in direction towards s
      valid := **true** // suitable cut found
    else
      $c := e$ // W is not suitable, continue binary search in direction towards t
    end if
  end while
  $w_i := v_{id}$ // fix best vertex found on f_i, continue with next path
end for
if valid **then**
  return $(w_1, ..., w_k)$
else
  return $(t, t, ..., t)$
end if

Lemma 16. *Let $g(s), g(t)$ be positive integers. If there is a cut X such that $|V_t(X)| + g(t) \leq |V_r(X) \cup V_s(X)| + g(s)$, then Algorithm 3 returns such a cut X with the additional property that $|V_t(X')| + g(t) > |V_r(X') \cup V_s(X')| + g(s)$ for all cuts X' satisfying $X' \prec X$. If there is no such cut, then the algorithm returns the tuple $(t, t, ..., t)$. In both cases Algorithm 3 terminates in time $\mathcal{O}(km \log n)$.*

Proof. The result follows by a proof analogous to the proof for Lemma 15.

4.2 Replacing Vertices by Edges

In 1956, Ford and Fulkerson [12] devised their famous maximum flow algorithm which can also be used to compute k pairwise edge-disjoint s-t-paths in time $\mathcal{O}(km)$. In order to use it for computing pairwise vertex-disjoint s-t-paths, we present a transformation of a graph into a directed graph where pairwise vertex-disjoint s-t-paths are transformed into pairwise edge-disjoint paths all starting and ending in the same vertices. The basic idea is to transform each vertex v into a directed edge (c_v, d_v) and each edge $\{v, w\}$ into two directed edges (d_v, c_w) and (d_w, c_v). More formally, we define:

Definition 17. *Let $G = (V, E)$ be a graph. We define the directed graph $\overline{G} = (\overline{V}, \overline{E})$ as follows:*

$$\overline{V} := \{c_v | v \in V\} \cup \{d_v | v \in V\} \qquad \overline{E} := \{(c_v, d_v) | v \in V\} \cup \{(d_v, c_w) | \{v, w\} \in E\}$$

Note that, as the edges in E are undirected, the set $\{(d_w, c_v) | \{v, w\} \in E\}$ is a subset of $\overline{E}$.
Let s, t be two vertices of G and $f = (s, v_1, v_2, ..., v_j, t)$ an s-t-path in G. We define

$$\overline{f} := (d_s, c_{v_1}, d_{v_1}, c_{v_2}, d_{v_2}, ..., c_{v_j}, d_{v_j}, c_t) \ .$$

Lemma 18. *Let s, t be two vertices of G and let F be the set of all s-t-paths in G. Define $\overline{F} := \{\overline{f} | f \in F\}$. Then $\overline{F}$ is the set of all d_s-c_t-paths in $\overline{G}$. Moreover, the s-t-paths contained in some subset $F' \subseteq F$ are pairwise vertex-disjoint if and only if the d_s-c_t-paths contained in $\overline{F'} := \{\overline{f'} | f' \in F'\}$ are pairwise edge-disjoint.*

Proof. Consider an arbitrary d_s-c_t-path in $\overline{G}$. The definition of $\overline{E}$ ensures that this path is of the form $(d_s, c_{v_1}, d_{v_1}, c_{v_2}, d_{v_2}, ..., c_{v_j}, d_{v_j}, c_t)$ where v_i and v_{i+1} are adjacent in G for all $1 \leq i \leq j-1$, as well as s and v_1, and v_j and t. Thus, it is an element of $\overline{F}$.

Vice versa, consider some arbitrary $\overline{f} \in \overline{F}$. Then $\overline{f}$ must be of the form $(d_s, c_{v_1}, d_{v_1}, c_{v_2}, d_{v_2}, ..., c_{v_j}, d_{v_j}, c_t)$ and between each two consecutive vertices in this tuple there is a directed edge in $\overline{E}$. Thus, $\overline{f}$ is an d_s-c_t-path.

For the second part of the lemma, it is enough to show that any two s-t-paths f and f' in G are vertex-disjoint if and only if $\overline{f}$ and $\overline{f'}$ are edge-disjoint. If f and f' are not vertex-disjoint, then there is some vertex $v \in V, s \neq v \neq t$ which is contained in both f and f'. It follows that c_v and d_v are consecutive vertices in both $\overline{f}$ and $\overline{f'}$ and therefore $\overline{f}$ and $\overline{f'}$ are not edge-disjoint.

Vice versa, suppose $\overline{f}$ and $\overline{f'}$ are not edge-disjoint. Then there is some vertex c_v or some vertex d_v (as a closer look shows, actually both) which is contained in both $\overline{f}$ and $\overline{f'}$.[5] It follows that v is contained in both f and f' and therefore f and f' are not vertex-disjoint.

4.3 Cuts of Bounded Size

With the tools gathered above, we are now able to design and analyze an algorithm (Algorithm 4) which finds a pair of tuples that contains a good cut.[6] As we will perform contractions on a given graph G in the process, we give a short overview of the technical details and the used terminology. The contraction of a subset U of V transforms G into a graph H where $V(H) := (V \backslash U) \cup \{u\}$ and $E(H)$ contains an edge $\{u, w\}$ for each edge $\{v, w\} \in E$ satisfying $v \in U$, $w \in V \backslash U$ while all edges in G between vertices in $V \backslash U$ remain edges in H. Note that, according to this definition, an edge in G between vertices in U does not produce a self-loop in H. We call vertex u the contraction of subset U.

[5] Note that d_s and c_t are not adjacent in $\overline{G}$ since s and t are not adjacent in G.
[6] Note that after the above digression, we now consider again undirected graphs.

For an illustration of how Algorithm 4 (and, implicitly, Algorithm 2 and Algorithm 3) proceeds, consider Fig. 2–4. Essentially, Algorithm 4 uses Algorithm 2 and Algorithm 3 as subroutines in order to find two cuts that cut a preferably large part containing s, resp. t, off. Then it contracts these two parts into new nodes s and t and iterates on the obtained graph. We show in the following that the number of pairwise vertex-disjoint s-t-paths grows in each iteration and that the performed contractions ensure that s and t remain non-adjacent. Subsequently, we examine the runtime of the algorithm.

Algorithm 4 Find Innermost Cut of Bounded Size

Initialization: Given a positive integer K and two vertices $s, t \in V$, set $g(s) := g(t) := 1$, $k := 0$, $H := G$ and $S := T := \{\}$.

while the maximum number of pairwise vertex-disjoint s-t-paths in H is at most K **do**
  find maximum number of pairwise vertex-disjoint s-t-paths $f_1, ..., f_k$ in H, update k
  execute Algorithm 2, denote output by U // U cuts "large" part containing s off
  execute Algorithm 3, denote output by W // W cuts "large" part containing t off
  if $U \neq (s, s, ..., s)$ **then**
    $M_s := V_s(U) \cup U$ // collect vertex set which is to be contracted
    $S := U$ // U is best "s-sided" cut we found so far
  else
    $M_s := \{s\}$ // no s-sided cut found, so nothing to contract
  end if
  if $W \neq (t, t, ..., t)$ **then**
    $M_t := V_t(W) \cup W$ // collect vertex set which is to be contracted
    $T := W$ // W is best "t-sided" cut we found so far
  else
    $M_t := \{t\}$ // no t-sided cut found, so nothing to contract
  end if
  if $M_s \cap M_t \neq \emptyset$ or there is an edge from M_s to M_t in H **then**
    break // contracting M_s and M_t impossible or problematic for later iterations
  else
    contract M_s and denote the contraction by s (and update H accordingly)
    contract M_t and denote the contraction by t (and update H accordingly)
    replace parallel edges of H by a single edge
    $g(s) := g(s) + |M_s| - 1$ // update total number of vertices contracted into s
    $g(t) := g(t) + |M_t| - 1$ // update total number of vertices contracted into t
  end if
end while
return the pair (S, T)

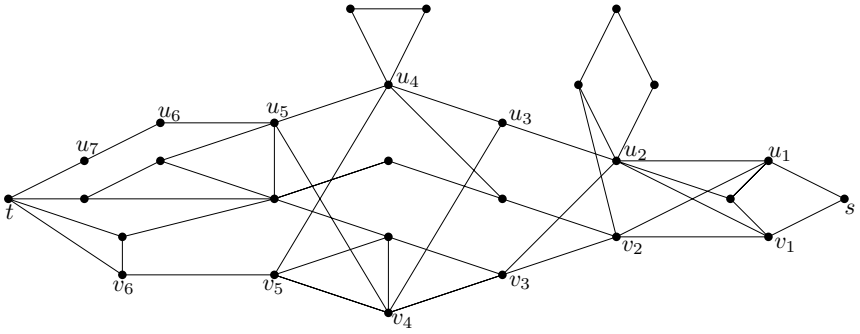

Fig. 2. Assume that the given K is 4. In the first iteration of the while loop, Algorithm 4 determines that the maximum number of pairwise vertex-disjoint s-t-paths is 2 and finds two pairwise vertex-disjoint s-t-paths, say the two paths given by the named vertices. Then executing Algorithm 2 yields output (u_2, v_2) whereas executing Algorithm 3 yields output (t, t). The contractions and the replacement of parallel edges by single edges at the end of the first iteration of the while loop produce the graph given in Fig. 3. Furthermore, we obtain $g(s) = 6$ and $g(t) = 1$.

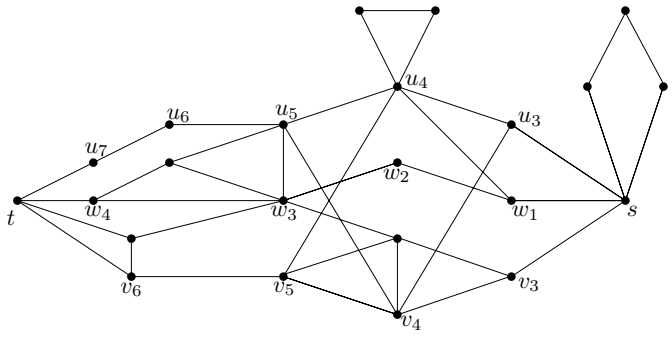

Fig. 3. In the second iteration of the while loop, Algorithm 4 determines that the maximum number of pairwise vertex-disjoint s-t-paths is 3 and finds three pairwise vertex-disjoint s-t-paths, say the three paths given by the named vertices. The subsequent execution of Algorithm 2 yields output (u_3, v_3, w_1) whereas executing Algorithm 3 yields output (u_5, v_5, w_3). The contractions and edge replacements at the end of the second iteration of the while loop produce the graph given in Fig. 4. We obtain $g(s) = 12$ and $g(t) = 10$.

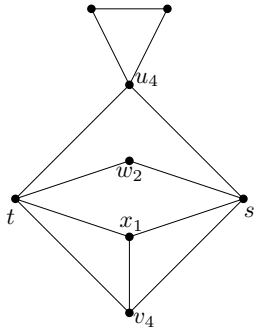

Fig. 4. In the third iteration of the while loop, Algorithm 4 finally finds four pairwise vertex-disjoint s-t-paths. Algorithm 2 and Algorithm 3 both return the same cut $X = (u_4, v_4, w_2, x_1)$. Note that this is the case for Algorithm 2 because there are those two vertices at the top that ensure that $|V_s(X)| + g(s) \leq |V_r(X) \cup V_t(X)| + g(t)$ (or, to be more precise, $|V_s(X)| + g(s) = |V_r(X) \cup V_t(X)| + g(t)$) by being contained in $V_r(X)$. We obtain $M_s \cap M_t \neq \emptyset$, so this was the last iteration of the while loop. Thus, Algorithm 4 returns the pair (X, X).

Lemma 19. *Consider an iteration of the while loop in Algorithm 4 where (not necessarily non-trivial) contractions are performed. Then, at the end of the iteration, there is no edge $\{s, t\} \in E(H)$ and there are (at least) $k + 1$ pairwise vertex-disjoint s-t-paths in H.*

Proof. For ease of presentation, we will denote s, t and H by s_0, t_0 and H_0 at the beginning of the iteration and by s_1, t_1 and H_1 at the end of the iteration. Since contractions are performed in the considered iteration, we have $M_s \cap M_t = \emptyset$ which implies $s_1 \neq t_1$ (and ensures that nothing problematic happens contractionwise), and we know that there is no edge from M_s to M_t in H_0 which implies that there is no edge $\{s_1, t_1\} \in E(H_1)$.

Now suppose that there is no set of $k + 1$ pairwise vertex-disjoint s_1-t_1-paths in H_1. Since there is no edge $\{s_1, t_1\} \in E(H_1)$, this implies, by Menger's Theorem, that there is a set $X = \{x_1, ..., x_k\}, s_1 \neq x_i \neq t_1$, of pairwise different vertices of H_1 such that there is no s_1-t_1-path in $H_1[V(H_1) \backslash X]$. Considering the x_i as vertices of H_0, the construction of H_1 ensures that there is no s_0-t_0-path in $H_0[V(H_0) \backslash X]$. As each of the f_i must contain an element of X, we can assume that $x_i \in f_i$. So X is a cut with respect to $(f_1, ..., f_k)$. Furthermore, we have $U \prec X$ and $X \prec W$, again by the construction of H_1.

By a simple calculation, we see that $|V_{s_0}(X)| + g(s_0) \leq |V_r(X) \cup V_{t_0}(X)| + g(t_0)$ or $|V_{t_0}(X)| + g(t_0) \leq |V_r(X) \cup V_{s_0}(X)| + g(s_0)$ must be satisfied. In the first case we can apply Lemma 15 and obtain $|V_{s_0}(Y)| + g(s_0) > |V_r(Y) \cup V_{t_0}(Y)| + g(t_0)$ for all cuts Y satisfying $U \prec Y$. Since we have $U \prec X$, we get $|V_{s_0}(X)| + g(s_0) > |V_r(X) \cup V_{t_0}(X)| + g(t_0)$ which yields a contradiction to $|V_{s_0}(X)| + g(s_0) \leq |V_r(X) \cup V_{t_0}(X)| + g(t_0)$. In the second case we obtain a contradiction analogously by using Lemma 16.

Lemma 20. *Let K be some positive integer and let s, t be two vertices of G. Then Algorithm 4 terminates in time $\mathcal{O}(K^2 m \log n)$.*

Proof. By Lemma 18, we can find a set of some cardinality of pairwise vertex-disjoint s-t-paths in some graph G by finding a set of the same cardinality of pairwise edge-disjoint d_s-c_t-paths in $\overline{G}$. By the definition of $\overline{G}$, we have $|\overline{V}| = 2|V|$ and $|\overline{E}| = |V| + 2|E|$. Using the Ford-Fulkerson algorithm mentioned earlier, we can check in time $\mathcal{O}(K|\overline{E}|) = \mathcal{O}(Km)$ whether there are $K+1$ pairwise edge-disjoint d_s-c_t-paths in $\overline{H}$ and, if this is not the case, find the maximum number of pairwise edge-disjoint d_s-c_t-paths $\overline{f_1}, ..., \overline{f_k}$, also in time $\mathcal{O}(Km)$.[7] Therefore, the part of the while loop before executing Algorithm 2 can be done in time $\mathcal{O}(Km)$.

Executing Algorithm 2 and Algorithm 3 can be done in time $\mathcal{O}(Km \log n)$, by Lemma 15 and Lemma 16, while the computation of $V_s(U)$ and $V_t(W)$ can be done in time $\mathcal{O}(m)$. The remaining tasks in an iteration of the while loop can be done as fast.

By Lemma 19, in each iteration of the while loop where contractions are performed, k is increased by at least 1. Thus there are at most K iterations of the while loop which results in a runtime of $\mathcal{O}(K^2 m \log n)$.

4.4 A Property of the Returned Pair

What is left to show is that the returned pair contains indeed a good cut. Theorem 23 will take care of that. In order to prove Theorem 23, we need two lemmas.

Lemma 21. *Consider an iteration of the while loop of Algorithm 4 where contractions are performed. Then, for the sets M_s and M_t which are contracted in this iteration, $H[M_s]$ and $H[M_t]$ are connected graphs.*

Proof. For reasons of symmetry, it is enough to show the statement for $H[M_s]$. If Algorithm 2 returns $(s, s, ..., s)$, there is nothing to show. If it does not return $(s, s, ..., s)$, then it returns a cut U, by Lemma 15. By definition, $V_s(U)$ is a connected component of $H[V(H) \backslash U]$. Now if there is a vertex $u \in U$ which is not the neighbor of some vertex in $V_s(U)$, then $V_s(U)$ is also a connected component of $H[V(H) \backslash (U \backslash \{u\})]$ which implies that $U \backslash \{u\}$ separates s and t. But then, by Menger's Theorem, there can be at most $|U| - 1$ pairwise vertex-disjoint s-t-paths in H which is a contradiction to U being a cut. Thus $H[M_s]$ is connected.

Lemma 22. *Let K be some positive integer and let s, t be two vertices of G. Let (S, T) be the pair returned by Algorithm 4 and suppose that there is an s-t-vertex separator (A^*, S^*, B^*) of G of size $K' \leq K$. Then one of S and T is a cut while the other is either a cut or empty.*

[7] Note that we have $|\overline{V(H)}| \leq |\overline{V}|$, $|\overline{E(H)}| \leq |\overline{E}|$ and $k \leq K$ at every time during the algorithm.

Proof. In the first iteration of the while loop of Algorithm 4 the maximum number of pairwise vertex-disjoint s-t-paths is not larger than K since S^* separates s and t which implies, by Menger's Theorem, that there can only be K' pairwise vertex-disjoint s-t-paths. Suppose that, in this first iteration of the while loop of Algorithm 4, Algorithm 2 returns $(s, s, ..., s)$ and Algorithm 3 returns $(t, t, ..., t)$. Then $M_s \cap M_t = \emptyset$ and there is no edge from M_s to M_t in H since s and t are non-adjacent in G. The performed contractions are trivial (i.e., they do not change H) and at the beginning of the subsequent iteration we have exactly the same situation as at the beginning of the first iteration.

This implies that there is no upper bound on the runtime of Algorithm 4 which yields a contradiction to the already established runtime given in Lemma 20. Therefore, in the first iteration we obtain $U \neq (s, s, ..., s)$ or $W \neq (t, t, ..., t)$ which implies that at the end of Algorithm 4 we have $S \neq ()$ or $T \neq ()$. Furthermore, by Lemma 15, Algorithm 2 returns a cut if it does not return $(s, s, ..., s)$ (and Algorithm 3 analogously). Thus, by the design of Algorithm 4 and Lemma 21 (which ensures that a cut found in some iteration separates the initially given s and t), one of S and T is a cut while the other is either a cut or empty.

For ease of presentation, define, for any subset U of V,

$$L_U := \max\{|V(C)| \mid C \text{ is a connected component of } G[V \backslash U]\} \ .$$

Theorem 23. *Let K be some positive integer and let s, t be two vertices of G. Let (S, T) be the pair returned by Algorithm 4 and suppose that there is an s-t-vertex separator (A^*, S^*, B^*) of G of size $K' \leq K$. Now if there is a connected component C of $G[V \backslash (S \cup T)]$ such that $S^* \subseteq V(C)$, then one of the following holds:*

(i) $L_S \leq L_{S^*}$ *(ii)* $L_T \leq L_{S^*}$ *(iii)* $L_S \leq \frac{1}{2}|V|$ *(iv)* $L_T \leq \frac{1}{2}|V|$

Proof. Let s, t, H, $g(s)$ and $g(t)$ be as they are at the end of Algorithm 4. Furthermore, denote by s_0 and t_0 the vertices s and t as they are at the start of Algorithm 4. By Lemma 22, $V_{s_0}(S)$, $V_{t_0}(S)$, $V_{s_0}(T)$ and $V_{t_0}(T)$ are defined if S and T are not empty. For the remainder of this proof, set $V_{s_0}(S) := \{\}$ and $V_{t_0}(S) := V$ if $S = ()$ and $V_{t_0}(T) := \{\}$ and $V_{s_0}(T) := V$ if $T = ()$ (since, as shown in Lemma 22, at most one of S and T can be empty, this is well-defined). Let C be a connected component of $G[V \backslash (S \cup T)]$ such that $S^* \subseteq V(C)$. Then we have $V(C) \subseteq V_{s_0}(S)$ or $V(C) \subseteq V_{t_0}(S)$, as well as $V(C) \subseteq V_{s_0}(T)$ or $V(C) \subseteq V_{t_0}(T)$.

Suppose that $V(C) \subseteq V_{t_0}(S)$ and $V(C) \subseteq V_{s_0}(T)$. Thus, $S^* \subseteq V_{t_0}(S)$ and $S^* \subseteq V_{s_0}(T)$. By design, Algorithm 4 contracts only sets which consist of vertices in $V_{s_0}(S) \cup S$ and vertices which are a contraction of vertices in $V_{s_0}(S) \cup S$ and sets which consist of vertices in $V_{t_0}(T) \cup T$ and vertices which are a contraction of vertices in $V_{t_0}(T) \cup T$. Therefore, as $S^* \subseteq V_{t_0}(S)$ and $S^* \subseteq V_{s_0}(T)$, no vertex contained in S^* is ever in a vertex set that is contracted. Since S^* separates s_0 and t_0, this implies, by Lemma 21, that each set that is contracted is "contained" in A^* or B^*.

Let A and B be the subsets of $V(H)$ that "remain" from A^*, resp. B^* after applying the series of contractions performed in Algorithm 4. The above considerations ensure that (A, S^*, B) is an s-t-vertex separator of H. Furthermore, since $g(s)$ and $g(t)$ indicate the total number of vertices of G which have been contracted into s, resp. t, we have $|A^*| \leq |B^*|$ if and only if $|A|+g(s) \leq |B|+g(t)$, and $|A^*| \geq |B^*|$ if and only if $|A| + g(s) \geq |B| + g(t)$. Suppose that $|A^*| \leq |B^*|$. Then $|A|+g(s) \leq |B|+g(t)$ and we obtain $|V_s(S^*)|+g(s) \leq |V_r(S^*) \cup V_t(S^*)|+g(t)$ (since, by the definition of $V_s()$, $V_t()$ and $V_r()$, we have $|V_s(S^*)| \leq |A|$ and $|B| \leq |V_r(S^*) \cup V_t(S^*)|$). Let U denote the output of Algorithm 2 in the last iteration of the while loop of Algorithm 4. Since $S^* \subseteq V_{t_0}(S)$ and by Lemma 21, we have $U \prec S^*$ [8] which combined with $|V_s(S^*)|+g(s) \leq |V_r(S^*) \cup V_t(S^*)|+g(t)$ yields a contradiction because in the last iteration of the while loop Algorithm 2 could not have returned U, by Lemma 15.

Analogously to the case $|A^*| \leq |B^*|$, Lemma 16 yields a contradiction for the case $|A^*| \geq |B^*|$. It follows that we cannot have both $V(C) \subseteq V_{t_0}(S)$ and $V(C) \subseteq V_{s_0}(T)$. Thus, we have $V(C) \subseteq V_{s_0}(S)$ or $V(C) \subseteq V_{t_0}(T)$.

Suppose that $V(C) \subseteq V_{s_0}(S)$.[9] If $S \neq ()$, then there is a last iteration of the while loop in which Algorithm 2 returns some $U \neq (s, s, ..., s)$. We obtain $S = U$. Furthermore, Lemma 15 and the fact that the updating of the weights at the end of the while loop corresponds to adding the respective numbers of contracted vertices (excluding s, resp. t, itself) ensure that $|V_{s_0}(S)| \leq |V_{t_0}(S) \cup V_r(S)|$. If $S = ()$, we also have $|V_{s_0}(S)| \leq |V_{t_0}(S) \cup V_r(S)|$. Thus, if $L_S > \frac{1}{2}|V|$, then there is a connected component C_S of $G[V \backslash S]$ with $|C_S| = L_S$ such that $V(C_S) \cap V_{s_0}(S) = \emptyset$. Since $S^* \subseteq V(C) \subseteq V_{s_0}(S)$, it follows that $V(C_S) \cap S^* = \emptyset$. Therefore, C_S is a subgraph of a connected component of $G[V \backslash S^*]$ which implies $L_S \leq L_{S^*}$.

If $V(C) \subseteq V_{t_0}(T)$ then it follows analogously that $L_T \leq \frac{1}{2}|V|$ or $L_T \leq L_{S^*}$.

5 Approximating Small Balanced Vertex Separators Quickly

In this section, we finally design an algorithm that uses Algorithm 4 as a subroutine in order to find a small and balanced vertex separator. We show that it does so in almost linear time w.h.p. and conclude the section by stating and proving our main result.

[8] Note that in this last iteration the found number of pairwise vertex-disjoint s-t-paths must be K' since there cannot be more such paths than K' (because S^* separates s and t and consists of K' vertices) and if the found number would be strictly smaller than K', then there would be at least one further iteration (since S^* separates s_0 and t_0, the last if condition must remain true throughout the whole Algorithm 4 if $S^* \subseteq V_{t_0}(S) \cap V_{s_0}(T)$).

[9] From this point on, deviating from the earlier convention, let s and t be as they are in the respective considered iteration (and not as they are at the end of Algorithm 4).

5.1 The Algorithm

The idea of Algorithm 5 is based on Algorithm 4 in conjunction with Theorem 23. Let $\frac{2}{3} \leq \alpha < 1$, and assume that there is a small α-separator (A^*, S^*, B^*) of G. By removing vertices of the given graph G, we obtain different connected components of which we then choose the largest one and iterate on this component. The goal is to reduce thereby the size of the largest component to approximately $\alpha|V|$ at most while removing only a small number of vertices in the process.

In more detail, Algorithm 5 chooses two vertices s, t in each iteration and then uses Algorithm 4 to find a small number of vertices which it will then remove. There is a "large enough" probability that the chosen vertices are on different sides of the vertex separator whose existence we assumed above since this separator is α-balanced. If the chosen vertices are on different sides, then Theorem 23 ensures that the size of the largest connected component after the vertex removal is at most $\alpha|V|$ or that the aforementioned separator contains strictly fewer vertices in the "separating set" S^* when restricted to this component. By iterating, we can reduce the number of vertices in this "separating set" to 0 (if the latter case occurs repeatedly) and then the balance of the aforementioned separator ensures that the largest component is of size at most $\alpha|V|$.

Since the balance of the separator may decrease radically by restricting it to the largest obtained component, so does the probability of s and t being on different sides of the separator. Thus, in order to obtain a good runtime, we stop when we achieve a certain balance close to α (which is realized in Algorithm 5 by the condition $|V(H)| > (\alpha + \varepsilon)|V|$ of the while loop).

In the analysis of Algorithm 5, we differentiate between iterations where s, t are "on the same side" of some fixed vertex separator and those where this is not the case.

Definition 24. *Let (A^*, S^*, B^*) be a vertex separator. Then we call an iteration of the while loop of Algorithm 5* unsuccessful *(with respect to (A^*, S^*, B^*)) if, for the vertices s, t chosen in that iteration, we have $s, t \in A^*$ or $s, t \in B^*$. If this is not the case, then we call the iteration* successful *(with respect to (A^*, S^*, B^*)).*

The following lemma formalizes some of the above considerations, thus enabling us to prove the subsequent corollary which gives an upper bound for the number of successful iterations.

Lemma 25. *Let $\frac{2}{3} \leq \alpha < 1$ and let (A^*, S^*, B^*) be an α-separator of G of size at most K. Consider a successful iteration of the while loop of Algorithm 5 with respect to (A^*, S^*, B^*). Let H_0 and H_1 denote the graph H at the beginning, resp. the end, of this iteration. Then $|V(H_1) \cap S^*| \leq |V(H_0) \cap S^*| - 1$ or $|V(H_1)| \leq \alpha|V|$.*

Proof. We observe that $V(H_1) \cap S^* \subseteq V(H_0) \cap S^*$. If $s \in S^*$ or $t \in S^*$, then $V(H_1) \cap S^*$ is a proper subset of $V(H_0) \cap S^*$ as the former does not contain s, resp. t, while the latter does. Thus, $|V(H_1) \cap S^*| \leq |V(H_0) \cap S^*| - 1$ and we are finished. Consider the remaining cases for a successful iteration, i.e.,

Algorithm 5 Find Small Balanced Vertex Separator

Initialization: Given a graph G, a positive integer K, some $\frac{2}{3} \leq \alpha < 1$ and some $0 < \varepsilon < 1 - \alpha$, set $H := G$ and $S' := \{\}$.

1: **while** $|V(H)| > (\alpha + \varepsilon)|V|$ **do**
2: choose two vertices s, t in H uniformly at random // possibly identical or adjacent
3: **if** $s = t$ or s and t are adjacent **then**
4: compute a largest connected component C of $H[V(H)\backslash(\{s\} \cup \{t\})]$ // remove s, t
5: $H := C$ // continue on the resulting largest connected component
6: $S' := S' \cup \{s\} \cup \{t\}$ // remember the removed vertices
7: **else**
8: execute Algorithm 4 (with input H, K, s, t) and denote the output by (S, T)
9: compute a largest connected component C of $H[V(H)\backslash(S \cup T \cup \{s\} \cup \{t\})]$
10: $H := C$ // continue on the resulting largest connected component
11: $S' := S' \cup S \cup T \cup \{s\} \cup \{t\}$ // remember the removed vertices
12: **end if**
13: **end while**
14: order the connected components $C_1, C_2, ...$ of $G[V\backslash S']$ such that $|V(C_1)| \geq |V(C_2)| \geq ...$
15: $A := V(C_1)$ // start collecting (vertex sets of) components
16: add the vertex sets $V(C_2), V(C_3), ...$ successively to A as long as the resulting A satisfies $|A| \leq \alpha|V|$ // note that $V(C_1)$ could already contain $(\alpha + \varepsilon)|V|$ vertices
17: $B := V(G)\backslash(A \cup S')$ // collect the "other side" of the vertex separator
18: **return** (A, S', B)

$s \in A^*, t \in B^*$ and $t \in A^*, s \in B^*$. For reasons of symmetry, we can assume w.l.o.g. that $s \in A^*, t \in B^*$. Note that, since (A^*, S^*, B^*) is a vertex separator, $s \in A^*, t \in B^*$ implies that the if-condition in Algorithm 5 is not satisfied.

Consider the s-t-vertex separator $(V(H_0) \cap A^*, V(H_0) \cap S^*, V(H_0) \cap B^*)$ of H_0 which is of size at most K since $|S^*| \leq K$. By Theorem 23, we have

$$V(H_0) \cap S^* \nsubseteq V(H_1)$$

or

$$|V(H_1)| \leq \max\{|V(H_0) \cap A^*|, |V(H_0) \cap B^*|, \frac{1}{2}|V|\} \ .$$

In the first case, we obtain $V(H_0) \cap S^* \nsubseteq V(H_1) \cap S^*$ which implies $|V(H_1) \cap S^*| \leq |V(H_0) \cap S^*| - 1$, since $V(H_1) \cap S^* \subseteq V(H_0) \cap S^*$. For the second case, we recall that $\max\{|A^*|, |B^*|\} \leq \alpha|V|$ and obtain $|V(H_1)| \leq \alpha|V|$.

Corollary 26. *Let $\frac{2}{3} \leq \alpha < 1$ and let (A^*, S^*, B^*) be an α-separator of G of size at most K. Then the number of successful iterations with respect to (A^*, S^*, B^*) is at most K in any execution of Algorithm 5.*

Proof. Suppose there is an execution of Algorithm 5 where at least $K+1$ successful iterations take place. Consider one of the first K successful iterations. By Lemma 25, we know that the number of vertices contained in the largest connected component after this iteration is at most $\alpha|V|$ or this iteration decreases the number of vertices in S^* contained in the largest connected component by at least 1. If the former of these two possibilities occurs, then the algorithm terminates after this iteration since we have $|V(H)| \leq (\alpha+\varepsilon)|V|$. But since we assumed that there are at least $K+1$ successful iterations, this cannot be the case.

Thus, each of the first K iterations decreases the number of vertices in S^* contained in the largest connected component by at least 1 which implies that after those K iterations, the resulting connected component H contains no vertex in S^*. Since (A^*, S^*, B^*) is a vertex separator, it follows that $V(H) \subseteq A^*$ or $V(H) \subseteq B^*$. Thus $|V(H)| \leq (\alpha+\varepsilon)|V|$ and the algorithm terminates before executing $K+1$ successful iterations. This yields a contradiction to our assumption.

5.2 The Runtime

In order to prove an upper bound for the runtime of Algorithm 5 we need the following lemma.

Lemma 27. *For all $0 < \varepsilon \leq 1$, we have $(1-\varepsilon)^{\lceil \varepsilon^{-1} \rceil} \leq \frac{1}{2}$.*

Proof. Using Bernoulli's Inequality, we obtain, for all $0 < \varepsilon \leq 1$,

$$(1-\varepsilon)^{\lceil \varepsilon^{-1} \rceil} = \left(\frac{(1-\varepsilon)(1+\varepsilon)}{(1+\varepsilon)}\right)^{\lceil \varepsilon^{-1} \rceil} \leq \frac{1}{(1+\varepsilon)^{\lceil \varepsilon^{-1} \rceil}} \leq \frac{1}{1+\varepsilon\lceil \varepsilon^{-1} \rceil} \leq \frac{1}{2} \ .$$

Using the already established bound for the number of successful iterations, we prove an upper bound for the runtime of Algorithm 5 in terms of the number of iterations.

Lemma 28. *Let $\frac{2}{3} \leq \alpha < 1$. If there exists an α-separator of size at most K, then Algorithm 5 terminates after $\mathcal{O}(\varepsilon^{-1} K \log^{1+o(1)} n)$ iterations w.h.p.*

Proof. Let (A^*, S^*, B^*) be an α-separator of size at most K. Consider an arbitrary iteration in an execution of Algorithm 5. Let p_{succ} be the probability that this iteration is successful (given only the partial execution up to this point). In the following, we give a lower bound for p_{succ}.

Let H be as it is at the start of this iteration. Since we have $|V(H) \cap A^*| \leq |A^*| \leq \alpha|V|$ and $|V(H) \cap B^*| \leq |B^*| \leq \alpha|V|$, there exist disjoint subsets $X, Y \subseteq V(H)$ such that $X \cup Y = V(H)$, $|X| \leq |Y| \leq \alpha|V|$ and neither X nor Y contains vertices from both $V(H) \cap A^*$ and $V(H) \cap B^*$.[10]

[10] We can obtain such X, Y by successively adding the vertices of $V(H) \cap S^*$ to the smaller one of $V(H) \cap A^*$ and $V(H) \cap B^*$.

Let p_X be the probability that a vertex chosen uniformly at random from $V(H)$ is in X. Since $|X| \leq |Y|$, we have $1 - p_X \geq \frac{1}{2}$. If the s, t chosen in our iteration satisfy $s \in X, t \in Y$ or $s \in Y, t \in X$, then the iteration is successful, by definition. We obtain $p_{\text{succ}} \geq p_X(1 - p_X) + (1 - p_X)p_X \geq p_X$. Since $|V(H)| > (\alpha + \varepsilon)|V|$, $|Y| \leq \alpha|V|$ and $0 < \varepsilon < 1 - \alpha$, we have

$$p_{\text{succ}} \geq p_X = \frac{|X|}{|V(H)|} = \frac{|V(H)| - |Y|}{|V(H)|} \geq 1 - \frac{\alpha}{\alpha + \varepsilon} = \frac{\varepsilon}{\alpha + \varepsilon} \geq \varepsilon \ .$$

Thus, the probability that an iteration is successful is always at least ε and the probability that an iteration is unsuccessful is always at most $1 - \varepsilon$.

Let p be the probability that an execution of Algorithm 5 has more than $2K\lceil\varepsilon^{-1}\rceil\lceil\log^{1+\delta} n\rceil$ iterations where δ is an arbitrary positive real number. By Corollary 26 and the above considerations, p is at most as large as the probability that, in a sequence of $2K\lceil\varepsilon^{-1}\rceil\lceil\log^{1+\delta} n\rceil$ tosses of a coin which shows heads with probability ε, we throw heads less than K times.[11]

Consider $\lceil\varepsilon^{-1}\rceil$ successive tosses of such a coin. Applying Lemma 27, we see that the probability that all of these coin tosses result in tails is at most $\frac{1}{2}$. In the following, we consider blocks of $\lceil\varepsilon^{-1}\rceil$ successive coin tosses. As we can group our $2K\lceil\varepsilon^{-1}\rceil\lceil\log^{1+\delta} n\rceil$ coin tosses in $2K\lceil\log^{1+\delta} n\rceil$ non-overlapping blocks, p is at most as large as the probability that less than K of these blocks contain a coin toss which results in heads. Since we can see each block as the toss of a "big" coin (which results in tails if and only if all coin tosses in the block result in tails) and we already bounded the probability of tails for these big coins, we know that p is at most as large as the probability that, in a sequence of $2K\lceil\log^{1+\delta} n\rceil$ tosses of a fair coin, we throw heads less than K times.

For reasons of symmetry, the probability that, in a sequence of $2K$ tosses of a fair coin, we throw heads less than K times is less than $\frac{1}{2}$. Thus, by dividing our above sequence in $\lceil\log^{1+\delta} n\rceil$ subsequences of $2K$ coin tosses, we obtain

$$p \leq \left(\frac{1}{2}\right)^{\lceil\log^{1+\delta} n\rceil} \leq \frac{1}{2^{\log n \log^{\delta} n}} = \frac{1}{n^{\log^{\delta} n}} \ .$$

By choosing $\delta := \frac{1}{\log\log\log n} \in o(1)$, we obtain

$$\log^{\delta} n = \left(2^{\log\log n}\right)^{\frac{1}{\log\log\log n}} = 2^{\frac{\log\log n}{\log\log\log n}} \in \omega(1)$$

which implies

$$p \in \frac{1}{n^{\omega(1)}} \ .$$

Thus, Algorithm 5 terminates after $\mathcal{O}(\varepsilon^{-1} K \log^{1+o(1)} n)$ iterations w.h.p.

[11] The reason we switch to coins instead of iterations is simply that we do not stop throwing coins when we have reached a certain number of heads whereas Algorithm 5 terminates after at most K successful iterations which complicates the analysis.

5.3 The Main Result

By combining the results obtained in this section, we are able to state and prove our main result.

Theorem 29. *Let $\frac{2}{3} \leq \alpha < 1$ and $0 < \varepsilon < 1 - \alpha$. If G contains an α-separator that has size at most K, then Algorithm 5 finds an $(\alpha + \varepsilon)$-separator of size $\mathcal{O}(\varepsilon^{-1}K^2 \log^{1+o(1)} n)$ in time $\mathcal{O}(\varepsilon^{-1}K^3 m \log^{2+o(1)} n)$ w.h.p.*

Proof. By Lemma 28, Algorithm 5 terminates after $\mathcal{O}(\varepsilon^{-1}K \log^{1+o(1)} n)$ iterations w.h.p. Consider the steps performed in an iteration. Computing the largest connected component of a subgraph of G can be done in time $\mathcal{O}(m)$ whereas executing Algorithm 4 takes time $\mathcal{O}(K^2 m \log n)$, by Lemma 20. Since computing and ordering the C_j and adding the $V(C_j)$ can also be done in time $\mathcal{O}(m)$, Algorithm 5 terminates in time $\mathcal{O}(\varepsilon^{-1}K^3 m \log^{2+o(1)} n)$ w.h.p.

By the design of Algorithm 4, the entries of the pair that Algorithm 4 returns are cuts or empty tuples. If an entry is a cut, the input parameter K (and the design of Algorithm 4) ensures that the cardinality of the cut is at most K. Thus, the cardinality of S' increases by at most $2K+2$ in each iteration of the while loop in Algorithm 5. Combining this with the bound on the number of iterations given in Lemma 28, we obtain that the cardinality of S' is in $\mathcal{O}(\varepsilon^{-1}K^2 \log^{1+o(1)} n)$ when Algorithm 5 terminates.

What is left to show is that the output of Algorithm 5 is an $(\alpha+\varepsilon)$-separator. By design, when Algorithm 5 terminates, the number of vertices in the largest connected component is at most $(\alpha+\varepsilon)|V|$. Thus, $|V(C_1)| \leq (\alpha+\varepsilon)|V|$. The set A contained in the output satisfies $|A| = |V(C_1)| + ... + |V(C_{j'})|$ for some $j' \geq 1$. If $|B| > \alpha|V|$, then $|A| \leq (1-\alpha)|V| \leq \frac{1}{3}|V|$ which implies $|V(C_{j'+1})| \leq |V(C_1)| \leq \frac{1}{3}|V|$ and $V(C_{j'+1})$ would have been added to A in Algorithm 5 (since we would then still have $|A| \leq \alpha|V|$). Thus, $|B| \leq \alpha|V|$ and since $|A| \leq (\alpha+\varepsilon)|V|$, we can conclude that (A, S', B) is an $(\alpha+\varepsilon)$-separator.

By Theorem 29, we can find a reasonably small $(\alpha+\varepsilon)$-separator in almost linear time, provided that G contains a small α-separator. In particular, we obtain the following:

- If $K \in \mathcal{O}(\text{polylog}\, n)$, then Algorithm 5 finds an $(\alpha+\varepsilon)$-separator of size $\mathcal{O}(\varepsilon^{-1}\,\text{polylog}\, n)$ in time $\mathcal{O}(\varepsilon^{-1} m\,\text{polylog}\, n)$ w.h.p.
- If $K \in \mathcal{O}(\log n)$, then Algorithm 5 finds an $(\alpha+\varepsilon)$-separator that has size $\mathcal{O}(\varepsilon^{-1}\log^{3+o(1)} n)$ in time $\mathcal{O}(\varepsilon^{-1} m \log^{5+o(1)} n)$ w.h.p.

Throughout this work, we supposed that we have a fixed K which gives us an upper bound for the size of the α-separator whose existence we assume. If we do not want to consider some specific K, but rather find some (ideally small) K for which Algorithm 5 returns a vertex separator as specified in Theorem 29, we can achieve this by successively doubling K, each time executing Algorithm 5. The obtained total runtime is asymptotically the same as the runtime of Algorithm 5.

We consider our approach to be a first step in a new direction. We are confident that future work building on this approach can improve the presented theoretical bounds significantly. One reason (of many) for this is the following: One factor of K in the runtime is due to the possibility that, in each of K successful iterations, the number of vertices in S^* contained in the largest connected component potentially decreases by only 1 (see the proof of Corollary 26). We expect that graphs exhibiting such an incremental decrease must have very specific structures that can be exploited. Moreover, from a practical (and entirely informal) standpoint, we note that the hidden constants are fairly small and that all three factors of K in the runtime should be significantly lower than K on average.

References

1. Noga Alon, Paul D. Seymour, and Robin Thomas. A separator theorem for graphs with an excluded minor and its applications. *STOC 1990*.
2. Eyal Amir, Robert Krauthgamer, and Satish Rao. Constant factor approximation of vertex-cuts in planar graphs. *STOC 2003*.
3. Sanjeev Arora and Satyen Kale. A combinatorial, primal-dual approach to semidefinite programs. *STOC 2007*.
4. Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, 42(3):153–159, 1992.
5. H. N. Djidjev. A linear algorithm for partitioning graphs of fixed genus. *Serdica*, 11(4):369–387, 1985.
6. Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Divide-and-conquer approximation algorithms via spreading metrics. *FOCS 1995*.
7. Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Fast approximate graph partitioning algorithms. *SODA 1997*.
8. Uriel Feige, Mohammad Taghi Hajiaghayi, and James R. Lee. Improved approximation algorithms for minimum-weight vertex separators. *STOC 2005*.
9. Uriel Feige and Mohammad Mahdian. Finding small balanced separators. *STOC 2006*.
10. John R. Gilbert, Joan P. Hutchinson, and Robert Endre Tarjan. A separator theorem for graphs of bounded genus. *Journal of Algorithms*, 5(3):391–407, 1984.
11. Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. *OSDI 2012*.
12. L. R. Ford Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.
13. Ken-ichi Kawarabayashi and Bruce A. Reed. A separator theorem in minor-closed classes. *FOCS 2010*.
14. Frank Thomson Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.
15. Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.
16. Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. *SIGMOD 2010*.

17. Dániel Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351(3):394–406, 2006.
18. Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.
19. Bruce A. Reed and David R. Wood. A linear-time algorithm to find a separator in a graph excluding a minor. *ACM Transactions on Algorithms*, 5(4), 2009.
20. Christian Wulff-Nilsen. Separator theorems for minor-free and shallow minor-free graphs with applications. *FOCS 2011*.