Computing Vertex-Disjoint Paths in Large Graphs Using MAOs

Johanna E. Preißer
Institute of Mathematics, TU Ilmenau, Germany

Jens M. Schmidt
Institute of Mathematics, TU Ilmenau, Germany

Abstract

We consider the problem of computing $k \in \mathbb{N}$ internally vertex-disjoint paths between special vertex pairs of simple connected graphs. For general vertex pairs, the best deterministic time bound is, since 42 years, $O(\min\{k, \sqrt{n}\}m)$ for each pair by using traditional flow-based methods.

The restriction of our vertex pairs comes from the machinery of maximal adjacency orderings (MAOs). Henzinger showed for every MAO and every $1 \leq k \leq \delta$ (where δ is the minimum degree of the graph) the existence of k internally vertex-disjoint paths between every pair of the last $\delta - k + 2$ vertices of this MAO. Later, Nagamochi generalized this result by using the machinery of mixed connectivity. Both results are however inherently non-constructive.

We present the first algorithm that computes these k internally vertex-disjoint paths in linear time $O(m)$, which improves the previously best time $O(\min\{k, \sqrt{n}\}m)$. Due to the linear running time, this algorithm is suitable for large graphs. The algorithm is simple, works directly on the MAO structure, and completes a long history of purely existential proofs with a constructive method. We extend our algorithm to compute several other path systems and discuss its impact for certifying algorithms.

2012 ACM Subject Classification Mathematics of computing → Graph theory, Mathematics of computing → Graph algorithms

Keywords and phrases Computing Disjoint Paths, Large Graphs, Vertex-Connectivity, Linear-Time, Maximal Adjacency Ordering, Maximum Cardinality Search, Big Data, Certifying Algorithm

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.13

Funding This research is supported by the grant SCHM 3186/1-1 (270450205) from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation).

Acknowledgements We wish to thank On-Hei S. Lo for pointing out a connection between MAOs and Mader's proof about pendant pairs.

1 Introduction

Vertex-connectivity is a fundamental parameter of graphs that, by a result due to Menger [12], can be characterized by the existence of internally vertex-disjoint paths between vertex pairs. Thus, much work has been devoted to the following question: Given a number k, a simple graph $G = (V, E)$, and two vertices of G, compute k internally vertex-disjoint paths between these vertices if such paths exist. Despite all further efforts, the traditional flow-based approach by Even and Tarjan [3] and Karzanov [7] gives still the best deterministic bound $O(\min\{k, \sqrt{n}\}m)$ for this task, where $n := |V|$ and $m := |E|$.

© Johanna E. Preißer and Jens M. Schmidt;
licensed under Creative Commons License CC-BY
29th International Symposium on Algorithms and Computation (ISAAC 2018).
Editors: Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao; Article No. 13; pp. 13:1–13:12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our research is driven by the question whether k internally vertex-disjoint paths can be computed faster deterministically. This question has particular impact for large graphs, as we aim for linear-time algorithms. We have no general answer, but show for specific pairs of vertices that this can actually be done using *maximal adjacency orderings* (*MAOs*, also known under the name *maximum cardinality search*). MAOs order the vertices of a graph and can be computed in time $O(n+m)$ [18] (we will define MAOs in detail in Section 2).

One of the key properties of MAOs is that their last vertices are highly vertex-connected, i.e., have pairwise many internally vertex-disjoint paths. In more detail, let G be a simple unweighted graph of minimum degree δ and let $<$ be a MAO of G. Then $<$ decomposes G into edge-disjoint forests $F_1, \dots, F_m$ in a natural way (we will give the precise background on MAOs and such *forest decompositions* later). Let a subset of vertices be *k-connected* if G contains k internally vertex-disjoint paths between every two vertices of this subset. Henzinger proved for every $1 \leq k \leq \delta$ that the last $\delta - k + 2$ vertices of $<$ are k-connected [6].

In order to appreciate Henzinger's result, it is important to mention that its special case $k = \delta$ alone was predated by many results in the (weaker) realm of edge-connectivity: a well-known line of research [14, 4, 17] proved that the last two vertices of $<$ are δ-edge-connected. In fact, we exhibit the following forgotten link to a result by Mader [10, 9] in 1971, who used a preliminary variant of MAOs over one decade before MAOs were introduced and proved that their last two vertices are even δ-connected. In 2006, Nagamochi generalized all the mentioned results as follows.

▶ **Theorem 1** ([13][15, Thm. 2.28])**.** *Let $<$ be a MAO of a simple graph G and let $F_1, \dots, F_m$ be the forests into which $<$ partitions E. For every two vertices s and t that are in the same component of some F_k, G contains k internally vertex-disjoint paths between s and t.*

Theorem 1 specializes to Henzinger's result by taking the component T_k of F_k that contains the last vertex of $<$ (this tree contains the last $\delta - k + 2$ vertices of $<$). Its proof depends heavily on the machinery of mixed connectivity, and so does its most general statement (which we omit here, although all our results extend to this setting). Theorem 1 may be seen as the currently strongest result on MAOs regarding vertex-connectivity. However, all proofs known so far about vertex-connectivity in MAOs (including the ones by Henzinger and Nagamochi) are non-constructive and thus do not give any faster algorithm than the flow-based one for the initial question of computing internally vertex-disjoint paths.

The main result of this paper is an algorithm that computes the k paths of Theorem 1 in linear time $O(n+m)$. This improves upon the previously best time $O(\min\{k, \sqrt{n}\}m)$. To our surprise, its key idea is simple; the details of its correctness proof however are subtle. We therefore explain the algorithm in two incremental variants: The slightly weaker variant in Section 3 computes internally vertex-disjoint paths between one vertex s and a fixed set of k vertices of the forest decomposition; it does so by performing a right-to-left sweep through the MAO, in which the k paths are switched cyclically whenever one of the k paths would be lost. Section 4 then invokes two of these computations (one for s and one for t) in parallel in order to obtain our main result. We show also how the computation can be extended to find the k internally vertex-disjoint paths between a vertex and a vertex set, and between two vertex sets, whose existence was shown by Menger [12].

It is not easy to quantify for how many vertex pairs our faster algorithm can be applied. If we require δ internally vertex-disjoint paths, there are δ-regular graphs for which the only component of F_δ consists of one vertex pair joined by an edge and $F_{\delta+1} = \dots = F_m = \emptyset$. In this case, we can apply our algorithm only to a single vertex pair. However, in practice, many more of these sets occur and each of them may have a much larger size. If $k < \delta$ internally vertex-disjoint paths are sufficient, all pairs of a much larger set of size $\delta - k + 2$ can be taken (even in the worst case), at the expense of the linearly decreased pairwise connectivity k.

Certifying Algorithms

Being able to compute k internally vertex-disjoint paths has a benefit that purely existential proofs and algorithms that only argue about vertex separators do not have: It certifies the connectivity between the two vertices. For related problems on edge-connectivity, this has already been used to make algorithms *certifying* (in the sense of [11]).

The perhaps most prominent such result is the minimum cut algorithm of Nagamochi and Ibaraki [14], which refines the work of Mader [10, 9], and was simplified by Frank [4] and by Stoer and Wagner [17]. This algorithm computes iteratively a MAO and then contracts the last two δ(-edge)-connected vertices of it. For unweighted multigraphs, this is easily made certifying by storing the k edge-disjoint paths between these last two vertices in every step; the global k-edge-connectivity then follows by transitivity. In fact, the desired k edge-disjoint paths for every MAO can be obtained by just taking, for every $1 \leq i \leq k$, the unique s-t-path in the tree T_i of F_i that contains t. Using more involved methods, Arikati and Mehlhorn [1] made the algorithm of Nagamochi and Ibaraki certifying even for weighted graphs, again without increasing the quadratic asymptotic running time and space.

For the problem of recognizing k-connectivity, linear-time certifying algorithms are known for every $k \leq 3$ [19, 16]. For arbitrary k, the best known deterministic certifying algorithm is still the traditional flow-based one [3, 5], which achieves a running time of $O((k+\sqrt{n})k\sqrt{n}m)$. By using a geometric characterization of graphs, also a non-deterministic certifying algorithm with running time $O(n^{5/2} + k^{5/2}n)$ is known [8]. For designing faster certifying algorithms, finding a good certificate for k-connectivity seems to be the crucial open graph-theoretic problem, even when k is fixed:

▶ Open Problem. For every $k \in \mathbb{N}$, find a small and easy-to-verify certificate that proves the k-vertex-connectivity of simple graphs.

Our main result plays the same important role for certifying the vertex-connectivity between two vertices, as s-t-flows do for certifying the edge-connectivity between s and t in the results described above. For example, the 2-approximation algorithm for vertex-connectivity [6] by Henzinger can be made certifying using our new algorithm.

2 Maximal Adjacency Orderings

Throughout this paper, our input graph $G = (V, E)$ is simple, unweighted and of minimum degree δ. We assume standard graph theoretic notation as in [2]. A *maximal adjacency ordering* $<$ of G is a total order $1, \dots, n$ on V such that, for every two vertices $v < w$, v has at least as many neighbors in $\{1, \dots, v-1\}$ as w has. For ease of notation, we always identify the vertices of G with their position in $<$.

Every MAO $<$ decomposes G into edge-disjoint forests $F_1, \dots, F_m$ (some of which may be empty)[1] as follows: If $v > 1$ is a vertex of G and $w_1 < \dots < w_l$ are the neighbors of v in $\{1, \dots, v-1\}$, the edge $\{w_i, v\}$ belongs to F_i for all $i \in \{1, \dots, l\}$. For every i, the graph (V, F_i) is an edge-maximal forest of $G \setminus \{E(F_1), \dots, E(F_{i-1})\}$ (we refer to [15, Section 2.2] for a proof). For the sake of conciseness, we identify this forest with its edge set F_i. The partition of E into the non-empty forests is called the *forest decomposition* of $<$. For vertices $v < w$, we say v is *left* of w. If there is an edge between v and w, we call this a *left-edge* of w.

For any k, we allow to compute k internally vertex-disjoint paths between any two vertices that are contained in a tree T_k of the forest F_k. Hence, throughout the paper, let $s > 1$ be an arbitrary but fixed vertex of G and let k be a positive integer that is at most the number

[1] In fact, every forest F_i that satisfies $i > n$ is empty, as G is simple.

of left-edges of s. The vertex s will be the start vertex of the k internally vertex-disjoint paths to find (the end vertex will be left of s). E.g., if we choose s as the last vertex of the MAO (or any other vertex with at least that many left-edges), k can be chosen as any value that is at most the degree of vertex n; in particular, k can be chosen arbitrary in the range $1, \dots, \delta$, as claimed in the introduction.

For $i \in \{1, \dots, k\}$, let T_i be the component of F_i that contains s. As $i \leq k$, T_i is a tree on at least two vertices. Let the smallest vertex r_i of T_i with respect to $<$ be the *root* of T_i. For the purpose of this paper, it suffices to consider the subgraph of G induced by the edges of $T_1, \dots, T_k$.

▶ **Lemma 2** ([15, Lemma 2.25]). *Let $i \in \{1, \dots, k\}$. Then $V(T_i)$ consists of the consecutive vertices $r_i, r_i + 1, \dots, w$ in $<$ such that $s \leq w$. Moreover, for each vertex $v \in T_i \setminus \{r_i\}$, the vertex set $\{r_i, r_i + 1, \dots, v\}$ induces a connected subgraph of T_i.*

Hence, for every $i \in \{1, \dots, k\}$, every vertex $v > r_i$ of T_i has exactly one left-edge that is in T_i and thus at least i left-edges that are in G. Let $left_i(v)$ be the end vertex of the left-edge of v in F_i. The root r_i of T_i has left-degree exactly $i - 1$, as if it had more, r_i would have a left-edge in F_i and thus not be the root of T_i and, if it had less, the left-degree of $r_i + 1$ cannot be at least i, as this violates the MAO (this uses that G is simple). We conclude that $r_1 < r_2 \dots < r_k$. Thus, the definition of F_i and Lemma 2 imply the following corollary.

▶ **Corollary 3.** *Let $i < j \leq k$ and let v be a vertex with $r_j < v < s$. Then v is in T_j and T_i, $r_i \leq \text{left}_i(v) < \text{left}_j(v) < v$ and $r_j \leq \text{left}_j(v)$.*

For a vertex-subset $S \subseteq V$, let $\overline{S} := V \setminus S$. For convenience, we will denote sets $\overline{\{v\}}$ by $\overline{v}$. For a vertex-subset $S \subseteq V$, a set of paths is *S-disjoint* if no two of them intersect in a vertex that is contained in S. Thus, V-disjointness is the usual vertex-disjointness and a set of paths is *$\overline{v}$-disjoint* if every two of them intersect in either the vertex v or not at all. We represent paths as lists of vertices. The *length* of a path is the number of edges it contains. For a path A, let *end*(A) be the last vertex of this list and, if the path has length at least one, let *sec*(A) be the second to last vertex of this list.

3 The Loose Ends Algorithm

We first consider the slightly weaker problem of computing k internally vertex-disjoint paths between s and the root set $\{r_1, \dots, r_k\}$. We will extend this to compute k internally vertex-disjoint paths between two vertices in the next section.

▶ **Lemma 4.** *Algorithm 1 computes k $\overline{s}$-disjoint paths in $T_1 \cup \dots \cup T_k$ from s to $\{r_1, \dots, r_k\}$ in time $O(|E(T_1 \cup \dots \cup T_k)|) \subseteq O(n + m)$.*

The outline of our algorithm is as follows. We initialize each A_i to be the path that consists of the two vertices s and $left_i(s)$ (in that order). The vertices $left_i(s)$ are marked as *active*; throughout the algorithm, let a vertex be *active* if it is an end vertex of an unfinished path A_i.

So far the A_i are $\overline{s}$-disjoint. We aim for augmenting each A_i to r_i. Step by step, for every active vertex v from $s - 1$ down to r_1 in $<$, we will modify the A_i to longer paths, similar as in sweep line algorithms from computational geometry. The modification done at an active vertex v is called a *processing step.* From a high-level perspective, the end vertices of several paths A_i may be replaced or augmented by new end vertices w such that $r_i \leq w < v$ during the processing step of v. Such vertices w are again marked as active, which results in a

Algorithm 1 LooseEnds($G, <, s, k$).

```
1: for all i do                                              ▷ initialize all A_i
2:     A_i := (s, left_i(s))
3:     Mark left_i(s) as active
4: while there is a largest active vertex v do               ▷ process v
5:     Let j_1 < j_2 < ··· < j_l be the indices of the paths A_{j_i} that end at v
6:     for i := 2 to l do                                     ▷ replace end vertices
7:         Replace end(A_{j_i}) with left_{j_{i-1}}(sec(A_{j_i}))
8:         Mark left_{j_{i-1}}(sec(A_{j_i})) as active
9:     Perform a cyclic downshift on A_{j_1}, ..., A_{j_l}    ▷ A_{j_i} := A_{j_{i+1}}, A_{j_l} := A_{j_1}
10:    if v = r_{j_l} then
11:        A_{j_l} is finished                                ▷ r_{j_l} is reached
12:    else
13:        Append left_{j_l}(v) to A_{j_l}                    ▷ append predetermined vertex
14:        Mark left_{j_l}(v) as active
15:    Unmark v from being active
16: Output A_1, ..., A_k
```

continuous modification of each A_i to a longer path. By the above restriction on w, each path A_i will have strictly decreasing vertices in $<$ throughout the algorithm. At the end of the processing step of v, we unmark v from being active.

Let v be the active vertex that is largest in $<$. Assume that v is the end vertex of exactly one A_i. If $v = r_i$, A_i is finished. Otherwise, we append the vertex $\mathit{left}_i(v)$ to A_i (see Algorithm 1). The important aspect of this approach is that the index of the path A_i predetermines the vertex that augments A_i. Clearly, this way A_i will reach r_i at some point, according to Lemma 2.

However, if at least two paths end at v, this approach does not ensure vertex-disjointness. Let $A_{j_1}, \ldots, A_{j_l}$ be these $l \geq 2$ paths and assume $j_1 < j_2 < \cdots < j_l$. We first replace the end vertex v of A_{j_i} with the vertex $\mathit{left}_{j_{i-1}}(\mathit{sec}(A_{j_i}))$ for all $i \neq 1$. We will show that these modified end vertices are strictly smaller than v, which will re-establish the vertex-disjointness. The key idea of the algorithm is then to switch the indices of the l paths appropriately such that the appended vertices are again predetermined by the path index.

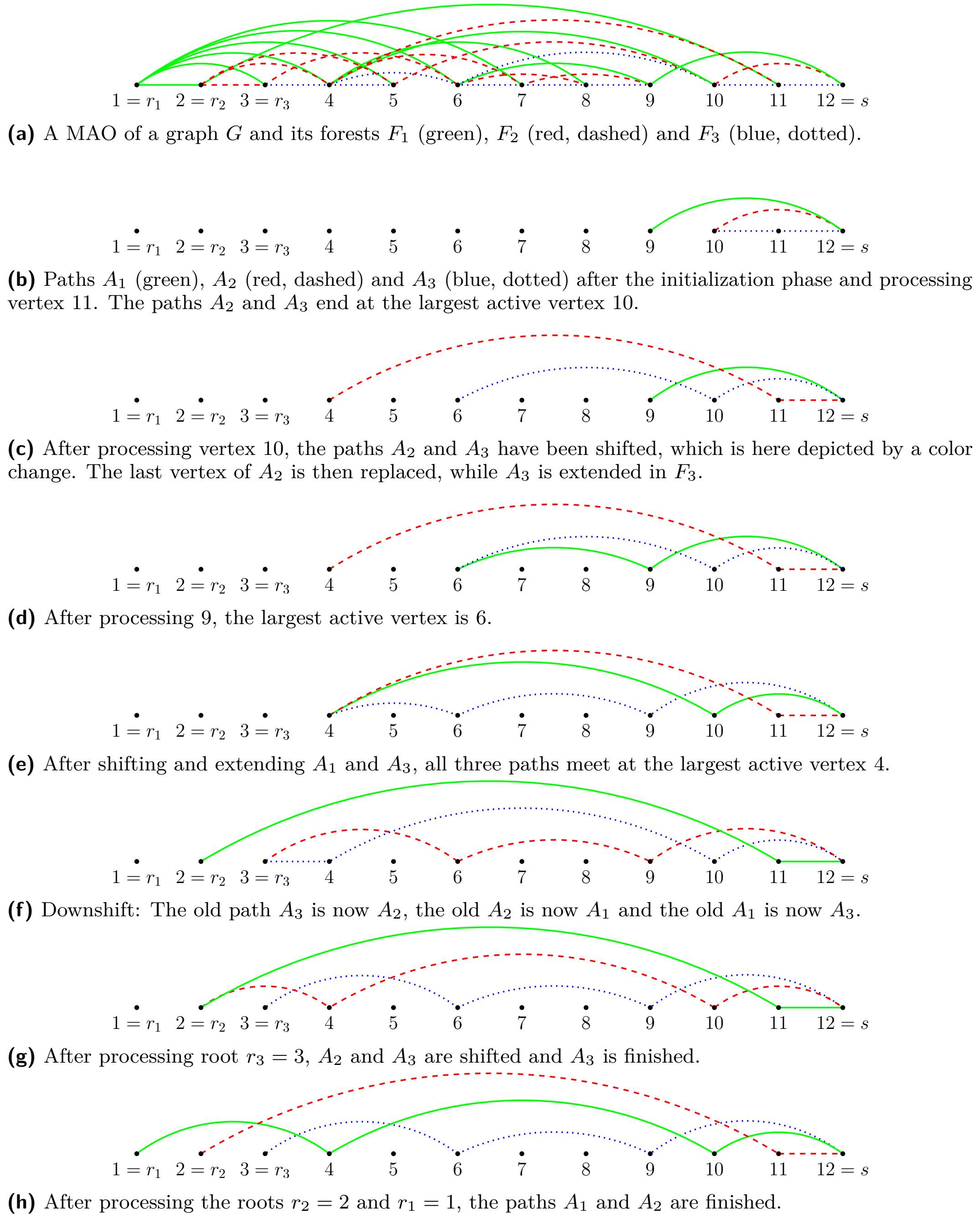
Let a *cyclic downshift* on $A_{j_1}, \ldots, A_{j_l}$ replace the index of each path by the next smaller index of a path in this set (where the next smaller index of j_1 is j_l), i.e. we set $A_{j_i} := A_{j_{i+1}}$ for every $i \neq l$ and then replace A_{j_l} with the old path A_{j_1}. We perform a cyclic downshift on $A_{j_1}, \ldots, A_{j_l}$. Note that we did not alter the path A_{j_l} (which was named A_{j_1} before) yet. If $v = r_{j_l}$, A_{j_l} is finished; otherwise, we append the vertex $\mathit{left}_{j_l}(v)$ to A_{j_l}. See Algorithm 1 for a description of the algorithm in pseudo-code. Figure 1 shows a run of Algorithm 1.

We prove the correctness of Algorithm 1. Before the processing step of any active vertex v, the A_i satisfy several invariants, the most crucial of which are that they are $\{v+1, \ldots, s-1\}$-disjoint and that the vertices of every A_i are decreasing in $<$. In detail, we have the following invariants.

▶ **Invariants.** *Let $v < s$ be the largest active vertex, or $v := 0$ if there is no active vertex left. Before processing v, the following invariants are satisfied for every $1 \leq i \leq k$:*

(1) *The vertices of A_i start with s and are strictly decreasing in $<$.*

(2) *The path A_i is finished if and only if $\mathrm{end}(A_i) > v$. In this case, $\mathrm{end}(A_i) = r_i$. If A_i is not finished, $r_i \leq \mathrm{end}(A_i) \leq v$ and the last edge of A_i is in T_i.*



(a) A MAO of a graph G and its forests F_1 (green), F_2 (red, dashed) and F_3 (blue, dotted).

(b) Paths A_1 (green), A_2 (red, dashed) and A_3 (blue, dotted) after the initialization phase and processing vertex 11. The paths A_2 and A_3 end at the largest active vertex 10.

(c) After processing vertex 10, the paths A_2 and A_3 have been shifted, which is here depicted by a color change. The last vertex of A_2 is then replaced, while A_3 is extended in F_3.

(d) After processing 9, the largest active vertex is 6.

(e) After shifting and extending A_1 and A_3, all three paths meet at the largest active vertex 4.

(f) Downshift: The old path A_3 is now A_2, the old A_2 is now A_1 and the old A_1 is now A_3.

(g) After processing root $r_3 = 3$, A_2 and A_3 are shifted and A_3 is finished.

(h) After processing the roots $r_2 = 2$ and $r_1 = 1$, the paths A_1 and A_2 are finished.

Figure 1 A run of Algorithm 1 on the graph depicted in (a) when $s = 12$ and $k = 3$.

(3) $\sec(A_i) > v$
(4) *Every vertex $w \in A_i$ satisfying $v < w < s$ is not contained in any $A_j \neq A_i$.*
(5) $A_i \subseteq T_1 \cup \dots \cup T_k$

We first clarify the consequences. Invariant (2) implies that the algorithm has finished all paths A_i precisely after processing r_1, and that every A_i ends at r_i. The Invariants (1) and (3) are necessary to prove Invariant (4), which in turn implies that the A_i are $\{v+1,\dots,s-1\}$-disjoint before processing an active vertex v. Hence, the final paths A_i are $\overline{s}$-disjoint. With Invariant (5) this gives the claim of Lemma 4.

It remains to prove Invariants (1)–(5). Immediately after initializing $A_1,\dots,A_k$, the next active vertex is $end(A_k) < s$. It is easy to see that all five invariants are satisfied for $v = end(A_k)$, i.e. before processing the first active vertex. We will prove that processing any largest active vertex v preserves all five invariants for the active vertex v' that follows v (where $v' := 0$ if v is the only remaining active vertex). For this purpose, let A'_i be the path with index i immediately before processing v' and let A_i be the path with index i before processing v; by hypothesis, the paths A_i satisfy all invariants for v.

For Lines 7 and 13 in the processing step of v, we have to prove the existence of $left_{j_{i-1}}(sec(A_{j_i}))$ and $left_{j_l}(v)$ respectively. In Line 7, we have $i \geq 2$ and $end(A_{j_i}) = v$ as can be seen in the pseudo-code. Then Invariant (2) implies that A_{j_i} is not finished and $v = end(A_{j_i}) = left_{j_i}(sec(A_{j_i}))$. Thus, $left_{j_{i-1}}(sec(A_{j_i}))$ exists. In Line 13, we have $v \neq r_{j_l}$ and $end(A_{j_l}) = v$ (here, A_{j_l} refers by definition to the path with index j_l *before* the cyclic downshift; note this is not the path dealt with in Line 13). Then Invariant (2) implies that $r_{j_l} \leq v$. This proves $r_{j_l} < v$ and the existence of $left_{j_l}(v)$.

We prove $v' < v$ next. Consider the vertices that are newly marked as active in the processing step of v. According to Line 5 of Algorithm 1, every such vertex is the new end vertex of some path A_{j_i} with end vertex v that was modified in the processing step of v (we do not count index transformations as modifications). There are exactly two cases how A_{j_i} may have been modified, namely either by Line 7 (then $2 \leq i \leq l$ and $left_{j_{i-1}}(sec(A_{j_i}))$ is the vertex that is newly marked as active) or by Line 13 (then $left_{j_l}(v)$ is the vertex that is newly marked as active); in particular, A_{j_i} was not modified by both lines. In the first case, A_{j_i} satisfies Invariant (2) before the processing step of v by hypothesis. In fact, we have $r_{j_i} \leq v$, as $v < r_{j_i}$ implies that A_{j_i} is finished and since $end(A_{j_i}) > v$ would contradict $end(A_{j_i}) = v$.

Hence, the last edge of A_{j_i} is in T_{j_i}, which shows $v = left_{j_i}(sec(A_{j_i}))$. Since $j_{i-1} < j_i$ by Line 5 and due to Corollary 3, we conclude $left_{j_{i-1}}(sec(A_{j_i})) < v$. In the second case, Corollary 3 implies $left_{j_l}(v) < v$. Thus, in both cases, every new active vertex is strictly smaller than v, which proves $v' < v$.

This gives Invariant (1), as every A'_{j_i} starts with s and every new vertex is left of its predecessor in the path by Corollary 3.

For Invariant (2), consider the path A'_i for any i. First, assume that A'_i is finished. Then either A_i is finished or $v = r_i$, according to Line 11 of Algorithm 1 in the processing step of v. In the former case, A_i satisfies Invariant (2) for v and so does A'_i for $v' < v$. In the latter case, we have $v' < v = r_i$ and $end(A'_i) = end(A_{j_1}) = v$.

Second, assume that A'_i was not modified in the processing step of v and is not finished. Then $end(A'_i) < v$, as every path with end vertex at least v is modified or finished in the processing step of v or finished before. In particular, processing v did not change the index of $A_i = A'_i$. As A_i satisfies Invariant (2) for v by hypothesis, the only condition of Invariant (2) that may be violated for v' is $end(A'_i) \leq v'$. However, as $end(A'_i) < v$ was marked as active in some previous step of Algorithm 1 and since v' is the largest active vertex, $end(A'_i) \leq v'$. Thus, A'_i satisfies Invariant (2) for v'.

Third, assume that A'_{j_i} was modified in the processing step of v and is not finished. Then A'_{j_i} was modified either by Line 7 or 13. If A'_{j_i} was modified by Line 7, we have $i < l$ and

$2 \leq l$ after the cyclic downshift, as the path A_{j_1} is not modified by Line 7. In addition, we know $end(A'_{j_i}) = left_{j_i}(sec(A_{j_{i+1}})) < left_{j_{i+1}}(sec(A_{j_{i+1}})) = v$ by Corollary 3 and that the last edge of A'_{j_i} is in T_{j_i}. Thus, $r_{j_i} \leq end(A'_{j_i})$. If A'_{j_i} was modified by Line 13, we have $i = l$ and $r_{j_l} \leq left_{j_l}(v) = end(A'_{j_l})$ by Corollary 3. Then the last edge of A'_{j_l} is in T_{j_l}. In both cases, $end(A'_{j_l})$ is active before processing v' and it follows $end(A'_{j_l}) \leq v'$.

For Invariant (3), assume to the contrary that $sec(A'_i) \leq v'$. Since $v' < v < sec(A_j)$ for all $j \in \{1, \dots, k\}$, a new end vertex was appended to A'_i in the processing step of v (the end vertex was not replaced, as this would not have changed $sec(A'_i)$). This must have been done in Line 13 of Algorithm 1 and we conclude $v' < v = sec(A'_i)$, which contradicts the assumption.

For Invariant (4), consider Line 7 of the processing step of v. As showed in the proof of $v' < v$ above, we have $left_{j_{i-1}}(sec(A_{j_i})) < v$ for all $1 < i \leq l$. Thus, Invariants (1) and (3) imply that exactly the path A'_{j_l} of the paths $A'_1, \dots, A'_k$ contains v.

Invariant (5) follows directly from the definition of $left_i$. This concludes the correctness part of the proof of Lemma 4.

So far we have shown an algorithmic proof for the existence of k $\bar{s}$-disjoint paths from s to the roots $r_1, \dots, r_k$. It remains to show the running time for Lemma 4. At every point in time, we maintain the order $A_1 < \cdots < A_i$ on our $i \leq k$ internally vertex-disjoint paths, where i is the index of the root vertex r_i that will be visited next. This ordered list can be updated in constant time after each cyclic downshift by modifying the position of one element.

Let v be the currently active vertex and let $r_i \leq v$ be the root vertex that will be visited next. Consider the ordered list of unfinished paths $A_1 < \cdots < A_i$ just before invoking Line 5. For Line 5, we need to sort the subset $A_{j_1}, \dots, A_{j_l}$ ($j_l \leq i$) of such paths paths ending at v according to $<$. In order to do this, we run through the i paths $A_1 < \cdots < A_i$ in that order, check for each entry whether its end vertex is v, and if so, append it to the sorted list $A_{j_1} < A_{j_2} < \dots$. Since v has precisely i (or $i-1$ in case of $v = r_i$) left-edges in $T_1 \cup \cdots \cup T_k \subseteq G$, this running time is upper-bounded by the number of such left-edges plus one. Summing the number of these left-edges for every visited v thus gives a running time bound of $O(|E(T_1 \cup \cdots \cup T_k)|)$ for all invocations of Line 5. Since the algorithm visits every edge only a constant number of times, this implies a total running time of $O(|E(T_1 \cup \cdots \cup T_k)|) = O(n + m)$.

4 Computing Vertex-Disjoint Paths Between Two Vertices

We use the algorithm of the last section to prove our following main result.

▶ **Theorem 5.** *Let $t < s$ be a vertex in T_k. Then k internally vertex-disjoint paths between s and t can be computed in time $O(|E(T_1 \cup \cdots \cup T_k)|) \subseteq O(n + m)$.*

This theorem is directly implied by the following lemma.

▶ **Lemma 6.** *Let $t < s$ be a vertex in T_k. Then there are k paths $A_1, \dots, A_k$ with start vertex s and k paths $B_1, \dots, B_k$ with start vertex t such that $\mathrm{end}(A_i) = \mathrm{end}(B_i)$ for every i and $\{A_1 \cup B_1, \dots, A_k \cup B_k\}$ is a set of k internally vertex disjoint paths from s to t. Moreover, all paths are contained in $T_1 \cup \cdots \cup T_k$ and can be computed by Algorithm 2 in time $O(|E(T_1 \cup \cdots \cup T_k)|)$.*

A first idea would be to use the loose ends-algorithm twice, once for the start vertex s and once for the start vertex t, in order to find the paths A_i and B_i for all i. However, in general

this is bound to fail. In some cases, the union of both outputs is a graph in which s and t are not k-connected. A second attempt may try to finish two paths A_i and B_j whenever they end at the same active vertex. However, this may fail when $i \neq j$, as then two single paths $A_{i'}$ and $B_{j'}$ may remain that end at the respective roots $r_{i'}$ and $r_{j'} > r_{i'}$ such that $B_{j'}$ cannot be extended to $r_{i'}$ without violating the index scheme of Invariant (2).

We will nevertheless use Algorithm 1 to prove Lemma 6, but in a more subtle way, as outlined next. First, we compute the paths $A_1, \dots, A_k$ with start vertex s using Algorithm 1, until the largest active vertex v is less or equal t (i.e. the parts of the A_i between s and t are just computed by Algorithm 1). As soon as $v \leq t$, we additionally construct a second set of paths $B_1, \dots, B_k$ with start vertex t using Algorithm 1.

The main difference to Algorithm 1 from this point on is that we extend the paths A_i and the paths B_i in parallel (i.e. we take the largest active vertex of both running constructions) such that, after the processing step of v, the vertex v is not contained in any two paths A_i and B_j with $i \neq j$. This ensures the vertex-disjointness.

If no A-path or no B-path ends at v, we again just perform Algorithm 1; then at most one path contains v after the processing step. Otherwise, some A-path and some B-path ends at v. After the processing step at v, we want to have exactly two paths A_j and B_j (i.e. having the same index) that end at v; such a pair of paths is then finished. In order to ensure this, we choose j as the largest index such that A_j *or* B_j ends at v before processing v. If both A_j and B_j end at v, we perform one processing step of Algorithm 1 at v for the A-paths and the B-paths, respectively, which implies that no other path is ending at v.

Otherwise, exactly one of the paths A_j and B_j ends at v, say A_j. Then B_j is not finished, as we finish only paths having the same index, and the last edge of B_j is in F_j. By assumption, there is an index $i < j$ such that B_i ends at v. We then apply a processing step of Algorithm 1 (including a cyclic downshift) on B_j and all B-paths that end at v, and one on all A-paths, respectively. Then the new paths A_j and B_j (due to cyclic downshifts, these correspond to the former A- and B-paths with lowest index ending at v) end at v afterward, but no other A- or B-path, as desired. Note that the replacement of the last edge of (the old) B_j, which did not end at v but, say, at a vertex w, may cause w to be active although neither an A-path nor a B-path ends at w.

For a precise description of the approach, see Algorithm 2. The following observations follow directly from Algorithm 2.

▶ **Observation 1.** *Throughout Algorithm 2 the paths $A_1, \dots, A_k, B_1, \dots, B_k$ satisfy the following properties.*

(1) *For every $i \in \{1, \dots, k\}$, A_i and B_i are both finished or both unfinished.*
(2) *As long as the largest active vertex is larger than t, $B_1 = B_2 = \dots = B_k = (t)$.*
(3) *The end vertex of every unfinished path is active.*

Before the processing step of any active vertex v, the paths A_i and B_i satisfy several invariants, the most crucial of which are that they are $\{v+1, \dots, s-1\} \setminus \{t\}$-disjoint and that the vertices of every A_i and B_i are decreasing in $<$.

▶ **Invariants.** *Let $v < s$ be the largest active vertex, or $v := 0$ if there is no active vertex left. Before processing v, the following invariants are satisfied for every $1 \leq i \leq k$:*

(1) *A_i starts with s, B_i starts with t, and the vertices of both paths are strictly decreasing in $<$.*
(2) *The paths A_i and B_i are finished if and only if $v < \operatorname{end}(A_i) = \operatorname{end}(B_i)$. If A_i and B_i are not finished, then $r_i \leq \operatorname{end}(A_i) \leq v$, $r_i \leq \operatorname{end}(B_i) \leq v$, and the last edge of A_i as well as the last edge of B_i (if B_i has length at least 1) are in T_i.*

Algorithm 2 MatchingEnds($G, <, s, t, k$). ▷ t is a vertex in T_k, $t < s$

1: **for all** i **do** ▷ initialize all A_i and B_i
2: $A_i := (s, \mathit{left}_i(s))$
3: Mark $\mathit{left}_i(s)$ as active
4: $B_i := (t)$
5: Mark t as active
6: **while** there is a largest active vertex v **do** ▷ process v
7: **if** v=t **then**
8: **for all** i **do** ▷ initialize all A_i
9: **if** $\mathit{end}(A_i) = t$ **then**
10: A_i, B_i are finished
11: **else**
12: Append $\mathit{left}_i(t)$ to B_i
13: Mark $\mathit{left}_i(t)$ as active
14: Unmark t from being active
15: **else**
16: $I_A := \{i|\mathit{end}(A_i) = v\}$
17: $I_B := \{i|\mathit{end}(B_i) = v\}$
18: **if** I_A and I_B are empty **then**
19: Unmark v from being active and go to Line 6
20: $j := \max(I_A \cup I_B)$
21: **for all** pairs (i_1, i_2) of consecutive indices $i_1 < i_2$ in $I_A \cup \{j\}$ **do**
22: Replace $\mathit{end}(A_{i_2})$ with $\mathit{left}_{i_1}(\mathit{sec}(A_{i_2}))$ ▷ replace ends
23: Mark $\mathit{left}_{i_1}(\mathit{sec}(A_{i_2}))$ as active
24: **for all** pairs (i_1, i_2) of consecutive indices $i_1 < i_2$ in $I_B \cup \{j\}$ **do**
25: Replace $\mathit{end}(B_{i_2})$ with $\mathit{left}_{i_1}(\mathit{sec}(B_{i_2}))$ ▷ replace ends
26: Mark $\mathit{left}_{i_1}(\mathit{sec}(B_{i_2}))$ as active
27: Perform a cyclic downshift on all A_i with $i \in I_A \cup j$
28: Perform a cyclic downshift on all B_i with $i \in I_B \cup j$
29: **if** $v = \mathit{end}(A_j) = \mathit{end}(B_j)$ **then** ▷ if and only if $I_A \neq \emptyset \neq I_B$
30: A_j, B_j are finished
31: **else if** $v = \mathit{end}(A_j)$ **then**
32: Append $\mathit{left}_j(v)$ to A_j ▷ append predetermined vertex
33: Mark $\mathit{left}_j(v)$ as active
34: **else if** $v = \mathit{end}(B_j)$ **then**
35: Append $\mathit{left}_j(v)$ to B_j ▷ append predetermined vertex
36: Mark $\mathit{left}_j(v)$ as active
37: Unmark v from being active
38: Output $A_1, \ldots, A_k$, $B_1, \ldots, B_k$

(3) $\sec(A_i) > v$. *If* $v \geq t$, $B_i = (t)$. *If* $v < t$, *either* B_i *is finished with* $B_i = (t)$ *or* B_i *has length at least* 1 *such that* $\sec(B_i) > v$.

(4) *Let* $w \neq t$ *be a vertex with* $v < w < s$. *If* $w \in A_i \cup B_i$, w *is neither contained in a path* $A_j \neq A_i$ *nor in a path* $B_j \neq B_i$. *If* $w \in A_i \cap B_i$, A_i *and* B_i *are finished with* $w = \text{end}(A_i) = \text{end}(B_i)$.

(5) $A_i \cup B_i \subseteq T_1 \cup \cdots \cup T_k$

Invariant (2) implies that the algorithm has finished all paths when $v = 0$ and that the end vertices of A_i and B_i match for all i. Invariants (1) and (3) will be necessary to prove Invariant (4), which in turn implies that the paths $A_1 \cup B_1, \ldots, A_k \cup B_k$ are internally vertex-disjoint. Invariant (5) settles the first part of the second claim of Lemma 6. We continue with further consequences of some of these invariants, which can be used to prove the invariants for the next largest active vertex v' after processing v.

▶ **Observation 2.** *Let* $v < s$ *be the largest active vertex, or* $v := 0$ *if there is no active vertex left. Before processing* v*, we have the following observations:*

(1) *Assume Invariants (1) and (3). Then, for every* $1 \leq i \leq k$*, all vertices of the paths* A_i *and* B_i *except* $\text{end}(A_i)$ *and* $\text{end}(B_i)$ *are greater than* v *before processing* v.

(2) *Assume Invariant (2). Then no finished path is modified while processing* v*, as Algorithm 2 modifies* A_i *or* B_i, $1 \leq i \leq k$*, only if at least one of them ends at* v.

(3) *Assume Invariants (2) and (3). Then the largest active vertex after processing* $v > 0$ *is smaller than* v.

Due to space constraints, we omit the proofs of the Invariants (1)–(5) and Observation 2.

As in the loose ends algorithm, the running time of Algorithm 2 is upper bounded by $O(|E(T_1 \cup \cdots \cup T_k)|)$ and thus by $O(n + m)$, as it suffices to visit every edge in the trees $T_1, \ldots, T_k$ a constant number of times.

4.1 Variants

Several variants of Menger's theorem [12] are known. Instead of computing k paths between two vertices, we can compute paths between a vertex and a set of vertices (*fan variant*) and between two sets of vertices (*set variant*). Our algorithm extends to these variants.

▶ **Theorem 7.** *Let* G *be a simple graph and* $<$, s *and* $T_1, \ldots, T_k$ *be defined as in Section 2.*

(i) *(Fan variant) Let* $T = \{t_1, \ldots, t_k\}$ *be a subset of* V *such that* $r_i \leq t_i < s$ *for every* i*. Then* k *internally vertex-disjoint paths between* s *and* T *can be computed in time* $O(|E(T_1 \cup \cdots \cup T_k)|) \subseteq O(n + m)$.

(ii) *(Set variant) Let* $T = \{t_1, \ldots, t_k\}$ *and* $S = \{s_1, \ldots, s_k\}$ *be disjoint vertex sets such that* $r_i \leq t_i < s$ *and* $r_i \leq s_i \leq s$ *for every* i*. Then* k *internally vertex-disjoint paths between* S *and* T *can be computed in time* $O(|E(T_1 \cup \cdots \cup T_k)|) \subseteq O(n + m)$.

Let $\alpha : V \to \mathbb{N}^+$ be a weight function. In the area of *mixed connectivity*, a set of paths connecting two vertices s and t of G is called *α-independent* if every vertex $v \notin \{s, t\}$ is contained in at most $\alpha(v)$ of these paths. For suitable multigraphs G, Nagamochi [13] generalized Theorem 1 by showing that these contain k α-independent s-t-paths. Algorithm 2 can be modified to compute also these paths without increasing its running time, by replacing the two cyclic downshifts by a more complicated algorithm that transforms the path indices.

References

1. S. R. Arikati and K. Mehlhorn. A correctness certificate for the Stoer-Wagner min-cut algorithm. *Information Processing Letters*, 70(5):251–254, 1999.
2. R. Diestel. *Graph Theory*. Springer, fourth edition, 2010.
3. S. Even and R. E. Tarjan. Network Flow and Testing Graph Connectivity. *SIAM Journal on Computing*, 4(4):507–518, 1975.
4. A. Frank. On the edge-connectivity algorithm of Nagamochi and Ibaraki. Laboratoire Artemis, IMAG, Université J. Fourier, Grenoble, March 1994.
5. Z. Galil. Finding the vertex connectivity of graphs. *SIAM Journal on Computing*, 9(1):197–199, 1980.
6. M. Rauch Henzinger. A Static 2-Approximation Algorithm for Vertex Connectivity and Incremental Approximation Algorithms for Edge and Vertex Connectivity. *Journal of Algorithms*, 24:194–220, 1997.
7. A. V. Karzanov. O nakhozhdenii maksimal'nogo potoka v setyakh spetsial'nogo vida i nekotorykh prilozheniyakh (in Russian; On finding a maximum flow in a network with special structure and some applications). *Matematicheskie Voprosy Upravleniya Proizvodstvom*, 5:81–94, 1973.
8. N. Linial, L. Lovász, and A. Wigderson. Rubber bands, convex embeddings and graph connectivity. *Combinatorica*, 8(1):91–102, 1988.
9. W. Mader. Existenz gewisser Konfigurationen in n-gesättigten Graphen und in Graphen genügend großer Kantendichte. *Mathematische Annalen*, 194:295–312, 1971.
10. W. Mader. Grad und lokaler Zusammenhang in endlichen Graphen. *Mathematische Annalen*, 205:9–11, 1973.
11. R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.
12. K. Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.
13. H. Nagamochi. Sparse connectivity certificates via MA orderings in graphs. *Discrete Applied Mathematics*, 154(16):2411–2417, 2006.
14. H. Nagamochi and T. Ibaraki. Computing Edge-Connectivity in Multigraphs and Capacitated Graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.
15. H. Nagamochi and T. Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, 2008.
16. J. M. Schmidt. Contractions, Removals and Certifying 3-Connectivity in Linear Time. *SIAM Journal on Computing*, 42(2):494–535, 2013.
17. M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.
18. R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.
19. H. Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34(1):339–362, 1932.