Algorithmica (2024) 86:1623–1656
https://doi.org/10.1007/s00453-023-01203-2

An Efficient Algorithm for All-Pairs Bounded Edge Connectivity

Shyan Akmal[1] · **Ce Jin**[1]

Received: 3 August 2023 / Accepted: 11 December 2023 / Published online: 22 January 2024

Abstract

Our work concerns algorithms for a variant of Maximum Flow in unweighted graphs. In the All-Pairs Connectivity (APC) problem, we are given a graph G on n vertices and m edges, and are tasked with computing the maximum number of edge-disjoint paths from s to t (equivalently, the size of a minimum (s, t)-cut) in G, for all pairs of vertices (s, t). Significant algorithmic breakthroughs have recently shown that over undirected graphs, APC can be solved in $n^{2+o(1)}$ time, which is essentially optimal. In contrast, the true time complexity of APC over directed graphs remains open: this problem can be solved in $\tilde{O}(m^\omega)$ time, where $\omega \in [2, 2.373)$ is the exponent of matrix multiplication, but no matching conditional lower bound is known. Following [Abboud et al. In: 46th International colloquium on automata, languages, and programming, ICALP 2019, July 9-12, 2019, Patras, Greece, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2019], we study a bounded version of APC called the k-Bounded All Pairs Connectivity (k-APC) problem. In this variant of APC, we are given an integer k in addition to the graph G, and are now tasked with reporting the size of a minimum (s, t)-cut only for pairs (s, t) of vertices with min-cut value less than k (if the minimum (s, t)-cut has size at least k, we can just report it is "large" instead of computing the exact value). Our main result is an $\tilde{O}((kn)^\omega)$ time algorithm solving k-APC in directed graphs. This is the first algorithm which solves k-APC faster than simply solving the more general APC problem exactly, for all $k \geq 3$. This runtime is $\tilde{O}(n^\omega)$ for all $k \leq \text{poly}(\log n)$, which essentially matches the optimal runtime for the $k = 1$ case of k-APC, under

This work was originally presented at ICALP 2023. S. Akmal: Supported in part by NSF Grants CCF-2129139 and CCF-2127597. C. Jin: Partially supported by NSF Grant CCF-2129139 and a Siebel Scholarship.

✉ Shyan Akmal
naysh@mit.edu
https://www.shyanakmal.com

Ce Jin
cejin@mit.edu
https://ce-jin.github.io/

[1] MIT EECS and CSAIL, Cambridge, MA, USA

popular conjectures from fine-grained complexity. Previously, this runtime was only achieved for $k \leq 2$ in general directed graphs [Georgiadis et al. In: 44th international colloquium on automata, languages, and programming (ICALP 2017), volume 80 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017], and for $k \leq o(\sqrt{\log n})$ in the special case of directed acyclic graphs [Abboud et al. In: 46th international colloquium on automata, languages, and programming, ICALP 2019, July 9–12, 2019, Patras, Greece, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2019]. Our result employs the same algebraic framework used in previous work, introduced by [Cheung et al. In: FOCS, 2011]. A direct implementation of this framework involves inverting a large random matrix. Our new algorithm is based off the insight that for solving k-APC, it suffices to invert a low-rank random matrix instead of a generic random matrix. We also obtain a new algorithm for a variant of k-APC, the k-Bounded All-Pairs Vertex Connectivity (k-APVC) problem, where we are now tasked with reporting, for every pair of vertices (s, t), the maximum number of internally vertex-disjoint (rather than edge-disjoint) paths from s to t if this number is less than k, and otherwise reporting that there are at least k internally vertex-disjoint paths from s to t. Our second result is an $\tilde{O}(k^2 n^\omega)$ time algorithm solving k-APVC in directed graphs. Previous work showed how to solve an easier version of the k-APVC problem (where answers only need to be returned for pairs of vertices (s, t) which are not edges in the graph) in $\tilde{O}((kn)^\omega)$ time [Abboud et al. In: 46th International colloquium on automata, languages, and programming, ICALP 2019, July 9–12, 2019, Patras, Greece, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2019]. In comparison, our algorithm solves the full k-APVC problem, and is faster if $\omega > 2$.

Keywords Maximum flow · All-pairs · Connectivity · Matrix rank

Mathematics Subject Classification Mathematics of computing · Graph algorithms

1 Introduction

Computing maximum flows is a classic problem which has been extensively studied in graph theory and computer science. In unweighted graphs, this task specializes to computing connectivities, an interesting computational problem in its own right. Given a graph G on n vertices and m edges, for any vertices s and t in G, the *connectivity* $\lambda(s, t)$ from s to t is defined to be the maximum number of edge-disjoint paths[1] from s to t. Since maximum flow can be computed in almost-linear time, we can compute $\lambda(s, t)$ for any given vertices s and t in $m^{1+o(1)}$ time [4].

What if instead of merely returning the value of a single connectivity, our goal is to compute all connectivities in the graph? This brings us to the All-Pairs Connectivity (APC) problem: in this problem, we are given a graph G as above, and are tasked with computing $\lambda(s, t)$ for all pairs of vertices (s, t) in G. In undirected graphs, APC can

[1] By Menger's theorem, $\lambda(s, t)$ is also equal to the minimum number of edges that must be deleted from the graph G to produce a graph with no s to t path.

be solved in $n^{2+o(1)}$ time [2], so that this "all-pairs" problem is essentially no harder than outputting a single connectivity in dense graphs.

In directed graphs, APC appears to be much harder, with various conditional lower bounds (discussed in Sect. 1.2) suggesting it is unlikely this problem can be solved in quadratic time. Naively computing the connectivity separately for each pair yields an $n^2 m^{1+o(1)}$ time algorithm for this problem. Using the flow vector framework (discussed in Sect. 3), it is possible to solve APC in directed graphs in $\tilde{O}(m^\omega)$ time[2] [6], where ω is the exponent of matrix multiplication. Known algorithms imply that $\omega < 2.37286$ [3], so the $\tilde{O}(m^\omega)$ time algorithm is faster than the naive algorithm whenever the input graph is not too dense.

Our work focuses on a bounded version of the APC problem, which we formally state as the k-Bounded All-Pairs Connectivity (k-APC) problem: in this problem, we are given a *directed* graph G as above, and are tasked with computing $\min(k, \lambda(s,t))$ for all pairs of vertices (s,t) in G. Intuitively, this is a relaxation of the APC problem, where our goal is to compute the exact values of $\lambda(s,t)$ only for pairs (s,t) with small connectivity. For all other pairs, it suffices to report that the connectivity is large, where k is our threshold for distinguishing between small and large connectivity values.

When $k = 1$, the k-APC problem is equivalent to computing the transitive closure of the input graph (in this problem, for each pair of vertices (s,t), we are tasked with determining if G contains a path from s to t), which can be done in $\tilde{O}(n^\omega)$ time [7]. Similarly, for the special case of $k = 2$, it is known that k-APC can be solved in $\tilde{O}(n^\omega)$ time, by a divide-and-conquer algorithm employing a cleverly tailored matrix product [9]. As we discuss in Sect. 1.2, there is evidence that these runtimes for k-APC when $k \leq 2$ are essentially optimal.

Already for $k = 3$ however, it is open whether k-APC can be solved faster than computing the *exact values* of $\lambda(s,t)$ for all pairs (s,t) of vertices! Roughly speaking, this is because the known $\tilde{O}(m^\omega)$ time algorithm for APC involves encoding the connectivity information in the inverse of an $m \times m$ matrix, and inverting an $m \times m$ matrix takes $O(m^\omega)$ time in general. This encoding step appears to be necessary for k-APC as well. For $k = 2$, clever combinatorial observations about the structure of strongly connected graphs allow one to skip this computation, but for $k \geq 3$ it is not clear at all from previous work how to avoid this bottleneck. Moreover, it is consistent with existing hardness results that k-APC could be solved in $O(n^\omega)$ time for any constant k.

Open Problem 1 Can k-APC be solved in faster than $\tilde{O}(m^\omega)$ time for $k = 3$?

Due to this lack of knowledge about the complexity of k-APC, researchers have also studied easier versions of this problem. Given vertices s and t in the graph G, we define the *vertex connectivity* $\nu(s,t)$ from s to t to be the maximum number of internally vertex-disjoint paths from s to t. We can consider vertex connectivity analogues of the APC and k-APC problems. In the All-Pairs Vertex Connectivity (APVC) problem, we are given a graph G on n vertices and m edges, and are tasked with computing the value of $\nu(s,t)$ for all pairs of vertices (s,t) in G. In the k-Bounded All-Pairs Vertex

[2] Given a function f, we write $\tilde{O}(f)$ to denote $f \cdot \text{poly}(\log f)$.

Connectivity (k-APVC) problem, we are given the same input G as above, but are now tasked with only computing $\min(k, \nu(s,t))$ for all pairs of vertices (s,t) in G.

The k-APVC problem does not face the $O(m^\omega)$ barrier which existing algorithmic techniques for k-APC seem to encounter, intuitively because it is possible to encode all the vertex-connectivity information of a graph in the inverse of an $n \times n$ matrix instead of an $m \times m$ matrix. As a consequence, [1] was able to present an $\tilde{O}((kn)^\omega)$ time algorithm which computes $\min(k, \nu(s,t))$ for all pairs of vertices (s,t) such that (s,t) is not an edge. Given this result, it is natural to ask whether the more general k-APVC and k-APC problems can also be solved in this same running time.

Open Problem 2 Can k-APVC be solved in $\tilde{O}((kn)^\omega)$ time?

Open Problem 3 Can k-APC be solved in $\tilde{O}((kn)^\omega)$ time?

1.1 Our Contribution

We resolve all three open problems raised in the previous section.

First, we present a faster algorithm for k-APC, whose time complexity matches the runtime given by previous work for solving an easier version of k-APVC.

Theorem 4 *For any positive integer k, k-APC can be solved in $\tilde{O}((kn)^\omega)$ time.*

This is the first algorithm which solves k-APC faster than simply solving APC exactly using the $\tilde{O}(m^\omega)$ time algorithm of [6], for all constant $k \geq 3$.

Second, we present an algorithm for k-APVC, which is faster than the $\tilde{O}((kn)^\omega)$ time algorithm from [1] (which only solves a restricted version of k-APVC) if $\omega > 2$.

Theorem 5 *For any positive integer k, k-APVC can be solved in $\tilde{O}(k^2 n^\omega)$ time.*

1.2 Comparison to Previous Results

Conditional Lower Bounds

The field of fine-grained complexity contains many popular conjectures (which hypothesize lower bounds on the complexity of certain computational tasks) which are used as the basis of conditional hardness results for problems in computer science. In this section, we review known hardness results for APC and its variants. The definitions of the problems and conjectures used in this section are stated in Appendix A.

Assuming that Boolean Matrix Multiplication (BMM) requires $n^{\omega - o(1)}$ time, it is known that k-APC and k-APVC require $n^{\omega - o(1)}$ time to solve, even for $k = 1$ [7]. In particular, this hypothesis implies our algorithms for k-APC and k-APVC are optimal for constant k.

Assuming the Strong Exponential Time Hypothesis (SETH), previous work shows that APC requires $(mn)^{1-o(1)}$ time [11, Theorem 1.8], APVC requires $m^{3/2-o(1)}$ time [14, Theorem 1.7], and k-APC requires $\left(kn^2\right)^{1-o(1)}$ time [11, Theorem 4.3].

Let $\omega(1,2,1)$ be the smallest real number[3] such that we can compute the product of an $n \times n^2$ matrix and $n^2 \times n$ matrix in $n^{\omega(1,2,1)+o(1)}$ time. Assuming the 4-Clique

[3] Known fast matrix multiplication algorithms imply that $\omega(1,2,1) < 3.25669$ [8, Table 2].

Conjecture, the k-APVC problem over directed graphs (and thus the k-APC problem as well) requires $\left(k^2 n^{\omega(1,2,1)-2}\right)^{1-o(1)}$ time [1]. The 4-Clique Conjecture also implies that solving APVC in undirected graphs requires $n^{\omega(1,2,1)-o(1)}$ time [10].

Algorithms for Restricted Graph Classes

As mentioned previously, no nontrivial algorithms for k-APC over general directed graphs were known for $k \geq 3$, prior to our work. However, faster algorithms were already known for k-APC over directed acyclic graphs (DAGs). In particular, [1] presented two algorithms to solve k-APC in DAGs, running in $2^{O(k^2)}mn$ time and $(k \log n)^{4^k+o(k)} n^\omega$ time respectively.

In comparison, our algorithm from Theorem 4 solves k-APC in *general* directed graphs, is faster than the former algorithm whenever $m \geq n^{\omega-1}$ or $k \geq \omega(\sqrt{\log n})$ (for example), is always faster than the latter algorithm, and is significantly simpler from a technical perspective than these earlier arguments. However, these algorithms for k-APC on DAGs also return cuts witnessing the connectivity values, while our algorithm does not.

In the special case of undirected graphs, APVC can be solved in $m^{2+o(1)}$ time [14, Theorem 1.8], which is faster than the aforementioned $\tilde{O}(m^\omega)$ time algorithm if $\omega > 2$. Over undirected graphs, k-APVC can be solved in $k^3 m^{1+o(1)} + n^2 \operatorname{poly}(\log n)$ time. In comparison, our algorithm from Theorem 5 can handle k-APVC in both undirected *and* directed graphs, and is faster for large enough values of k in dense graphs.

In directed planar graphs with maximum degree d, [6, Theorem 1.5] proves that APC can be solved in $O\left(d^{\omega-2} n^{\omega/2+1}\right)$ time.

Additional Related Work

In [15], the authors consider a symmetric variant of k-APC. Here, the input is a directed graph G on n vertices and m edges, and the goal is to compute for all pairs of vertices (s, t), the value of $\min(k, \lambda(s, t), \lambda(t, s))$. This easier problem can be solved in $O(kmn)$ time [15].

1.3 Organization

The rest of this paper is devoted to proving Theorems 4 and 5. In Sect. 2 we introduce notation, some useful definitions, and results on matrix computation which will be useful in proving correctness of our algorithms. In Sect. 3 we provide an intuitive overview of our algorithms for k-APC and k-APVC. In Sect. 4 we describe a framework of "flow vectors" for capturing connectivity values, and in Sect. 5 use this framework to prove Theorem 4. In Sect. 6 we present helpful results about vertex-connectivity, and in Sect. 7 use these results to prove Theorem 5. We conclude in Sect. 8, highlighting some interesting open problems suggested by our work.

In Appendix A, we include definitions of problems and conjectures mentioned in Sect. 1.2. In Appendix B, we discuss how the treatment of k-APVC in [1] differs from our own, and present the proof details for one of the results stated in Sect. 6.

2 Preliminaries

Graph Assumptions Throughout, we let G denote a directed graph on n vertices and m edges. Without loss of generality, we assume that the underlying undirected graph of G is connected, i.e., G is weakly connected (since, if not, we could simply run our algorithms separately on each weakly connected component of G), so we have $m \geq n-1$. We assume G has no self-loops, since these do not affect the connectivity or vertex-connectivity values between distinct vertices.

In Sects. 4 and 5 we focus on the k-APC problem, and so allow G to have parallel edges between vertices (i.e., G can be a multigraph). We assume however, without loss of generality, that for any distinct vertices s and t, there are at most k edges from s to t (since if there were more than k parallel edges from s to t, we could delete some and bring the count of parallel edges down to k without changing the value of $\min(k, \lambda(s,t))$). In Sects. 6 and 7 we focus on the k-APVC problem, and so assume that G is a simple graph with no parallel edges, since parallel edges from u to v cannot affect the value of a vertex connectivity $\nu(s,t)$, unless $u = s$ and $v = t$, in which case the value of $\nu(s,t)$ is simply increased by the number of additional parallel edges from s to t.

Graph Terminology and Notation Given an edge e from u to v in G, we write $e = (u,v)$. We call u the tail of e and v the head of e. Vertices which are tails of edges entering a vertex v are called *in-neighbors* of v. Similarly, vertices which are heads of edges exiting v are called *out-neighbors* of v. Given a vertex u in G, we let $E_{\text{in}}(u)$ denote the set of edges entering u, and $E_{\text{out}}(u)$ denote the set of edges exiting u. Similarly, $V_{\text{in}}(u)$ denotes the set of in-neighbors of u, and $V_{\text{out}}(u)$ denotes the set of out-neighbors of u. Furthermore, we define $V_{\text{in}}[u] = V_{\text{in}}(u) \cup \{u\}$ and $V_{\text{out}}[u] = V_{\text{out}}(u) \cup \{u\}$. Finally, let $\deg_{\text{in}}(u) = |E_{\text{in}}(u)|$ and $\deg_{\text{out}}(u) = |E_{\text{out}}(u)|$ denote the indegree and outdegree of u respectively.

Given vertices s and t, an (s,t)-cut is a set C of edges, such that deleting the edges in C produces a graph with no s to t path. By Menger's theorem, the size of a minimum (s,t)-cut is equal to the connectivity $\lambda(s,t)$ from s to t. Similarly, an (s,t)-vertex cut is a set of C' of vertices with $s, t \notin C'$, such that deleting C' produces a graph with no s to t path. Clearly, a vertex cut exists if and only if (s,t) is not an edge. When (s,t) is not an edge, Menger's theorem implies that the size of a minimum (s,t)-vertex cut is equal to the vertex connectivity $\nu(s,t)$ from s to t.

Matrix Notation Let A be a matrix. For indices i and j, we let $A[i,j]$ denote the (i,j) entry of A. More generally, if S is a set of row indices and T a set of column indices, we let $A[S,T]$ denote the submatrix of A restricted to rows from S and columns from T. Similarly, $A[S,*]$ denotes A restricted to rows from S, and $A[*,T]$ denotes A restricted to columns from T. We let $A^\top$ denote the transpose of A. If A is a square matrix, then we let $\text{adj}(A)$ denote the adjugate of A. If A is invertible, we let A^{-1} denote its inverse. If a theorem, lemma, or proposition statement refers to A^{-1}, it is generally asserting that A^{-1} exists (or if A is a random matrix, asserting that A^{-1} exists with some probability) as part of the statement. We let I denote the identity matrix (the dimensions of this matrix will always be clear from context). Given a vector $\vec{v}$,

for any index i we let $\vec{v}[i]$ denote the i^{th} entry in $\vec{v}$. We let $\vec{0}$ denote the zero vector (the dimensions of this vector will always be clear from context). Given a positive integer k, we let $[k] = \{1, \ldots, k\}$ denote the set of the first k positive integers.

Matrix and Polynomial Computation Given a prime p, we let $\mathbb{F}_p$ denote the finite field on p elements. Arithmetic operations over elements of $\mathbb{F}_p$ can be performed in $\tilde{O}(\log p)$ time.

We now recall some well-known results about computation with matrices and polynomials, which will be useful for our algorithms.

Proposition 6 *Let A be an $a \times b$ matrix, and B be a $b \times a$ matrix. If $(I - BA)$ is invertible, then the matrix $(I - AB)$ is also invertible, with inverse*

$$(I - AB)^{-1} = I + A(I - BA)^{-1}B.$$

Proof It suffices to verify that the product of $(I - AB)$ with the right hand side of the above equation yields the identity matrix. Indeed, we have

$$\begin{aligned}
&(I - AB)\left(I + A(I - BA)^{-1}B\right) \\
&\quad = I + A(I - BA)^{-1}B - AB - ABA(I - BA)^{-1}B \\
&\quad = I + A(I - BA)^{-1}B - AB - A\big(I - (I - BA)\big)(I - BA)^{-1}B \\
&\quad = I + A(I - BA)^{-1}B - AB - A(I - BA)^{-1}B + AB,
\end{aligned}$$

which simplifies to I, as desired. □

Proposition 7 *Let A be an $a \times a$ matrix over $\mathbb{F}_p$. We can compute the inverse A^{-1} (if it exists) in $O(a^\omega)$ field operations.*

Proposition 8 ([5, Theorem 1.1]) *Let A be an $a \times b$ matrix over $\mathbb{F}_p$. Then for any positive integer k, we can compute* $\min(k, \text{rank } A)$ *in $O(ab + k^\omega)$ field operations.*

Proposition 9 (Schwartz-Zippel Lemma [12, Theorem 7.2]) *Let $f \in \mathbb{F}_p[x_1, \ldots, x_r]$ be a degree d, nonzero polynomial. Let $\vec{a}$ be a uniform random point in $\mathbb{F}_p^r$. Then $f(\vec{a})$ is nonzero with probability at least $1 - d/p$.*

3 Proof Overview

3.1 Flow Vector Encodings

Previous algorithms for APC [6] and its variants work in two steps:

Step 1: Encode
In this step, we prepare a matrix M which implicitly encodes the connectivity information of the input graph.

Step 2: Decode
In this step, we iterate over all pairs (s, t) of vertices in the graph, and for each pair run a small computation on a submatrix of M to compute the desired connectivity value.

The construction in the **encode** step is based off the framework of *flow vectors*, introduced in [6] as a generalization of classical techniques from network-coding. We give a high-level overview of how this method has been previously applied in the APC problem.[4]

Given the input graph G, we fix a source vertex s. Let $d = \deg_{\text{out}}(s)$, and let $\mathbb{F}$ be some ground field.[5] Our end goal is to assign to each edge e in the graph a special vector $\vec{e} \in \mathbb{F}^d$ which we call a *flow vector*.

First, for each edge $e \in E_{\text{out}}(s)$, we introduce a d-dimensional vector $\vec{v}_e$. These vectors intuitively correspond to some starting flow that is pumping out of s. It is important that these vectors are linearly independent (and previous applications have always picked these vectors to be distinct d-dimensional unit vectors). We then push this flow through the rest of the graph, by having each edge get assigned a vector which is a random linear combination of the flow vectors assigned to the edges entering its tail. That is, given an edge $e = (u, v)$ with $u \neq s$, the final flow vector $\vec{e}$ will be a random linear combination of the flow vectors for the edges entering u. If instead the edge $e = (s, v)$ is in $E_{\text{out}}(s)$, the final flow vector $\vec{e}$ will be a random linear combination of the flow vectors for the edges entering s, added to the initial flow $\vec{v}_e$.

The point of this random linear combination is to (with high probability) preserve linear independence. In this setup, for any vertex v and integer ℓ, if some subset of ℓ flow vectors assigned to edges in $E_{\text{in}}(v)$ is independent, then we expect that every subset of at most ℓ flow vectors assigned to edges in $E_{\text{out}}(v)$ is also independent. This sort of behavior turns out to generalize to preserving linear independence of flow vectors across cuts, which implies that (with high probability) for any vertex t, the rank of the flow vectors assigned to edges in $E_{\text{in}}(t)$ equals $\lambda(s, t)$.

Intuitively, this is because the flow vectors assigned to edges in $E_{\text{in}}(t)$ will be a linear combination of the $\lambda(s, t)$ flow vectors assigned to edges in a minimum (s, t)-cut, and the flow vectors assigned to edges in this cut should be independent.

Collecting all the flow vectors as column vectors in a matrix allows us to produce a single matrix M_s, such that computing the rank of $M_s[*, E_{\text{in}}(t)]$ yields the desired connectivity value $\lambda(s, t)$ (computing these ranks constitutes the **decode** step mentioned previously). Previous work [1, 6] set the initial pumped $\vec{v}_e$ to be distinct unit vectors. It turns out that for this choice of starting vectors, it is possible to construct a single matrix M (independent of a fixed choice of s), such that rank queries to submatrices of M correspond to the answers we wish to output in the APC problem and its variants.

In Sect. 3.2 we describe how we employ the flow vector framework to prove Theorem 4. Then in Sect. 3.3, we describe how we modify these methods to prove Theorem 5.

[4] For the APVC problem we employ a different, but analogous, framework described in Sect. 3.3.

[5] In our applications, we will pick $\mathbb{F}$ to be a finite field of size poly(m).

3.2 All-Pairs Connectivity

Our starting point is the $\tilde{O}(m^\omega)$ time algorithm for APC presented in [6], which uses the flow vector encoding scheme outlined in Sect. 3.1.

Let K be an $m \times m$ matrix, whose rows and columns are indexed by edges in the input graph. For each pair (e, f) of edges, if the head of e coincides with the tail of f, we set $K[e, f]$ to be a uniform random field element in $\mathbb{F}$. Otherwise, $K[e, f] = 0$. These field elements correspond precisely to the coefficients used in the random linear combinations of the flow vector framework. Define the matrix

$$M = (I - K)^{-1}. \tag{1}$$

Then [6] proves that with high probability, for any pair (s, t) of vertices, we have

$$\text{rank } M[E_{\text{out}}(s), E_{\text{in}}(t)] = \lambda(s, t). \tag{2}$$

With this setup, the algorithm for APC is simple: first compute M (the **encode** step), and then for each pair of vertices (s, t), return the value of rank $M[E_{\text{out}}(s), E_{\text{in}}(t)]$ as the connectivity from s to t (the **decode** step).

By Eq. (1), we can complete the **encode** step in $\tilde{O}(m^\omega)$ time, simply by inverting an $m \times m$ matrix with entries from $\mathbb{F}$. It turns out we can also complete the **decode** step in the same time bound. So this gives an $\tilde{O}(m^\omega)$ time algorithm for APC.

Suppose now we want to solve the k-APC problem. A simple trick (observed in the proof of [1, Theorem 5.2] for example) in this setting can allow us to speed up the runtime of the **decode** step. However, it is not at all obvious how to speed up the **encode** step. To implement the flow vector scheme of Sect. 3.1 as written, it seems almost inherent that one needs to invert an $m \times m$ matrix. Indeed, an inability to overcome this bottleneck is stated explicitly as part of the motivation in [1] for focusing on the k-APVC problem instead.

Our Improvement

The main idea behind our new algorithm for k-APC is to work with a low-rank version of the matrix K used in Eq. (1) for the **encode** step.

Specifically, we construct certain random sparse matrices L and R with dimensions $m \times kn$ and $kn \times m$ respectively. We then set $K = LR$, and argue that with high probability, the matrix M defined in Eq. (1) for this choice of K satisfies

$$\text{rank } M[E_{\text{out}}(s), E_{\text{in}}(t)] = \min(k, \lambda(s, t)). \tag{3}$$

This equation is just a k-bounded version of Eq. (2). By Proposition 6, we have

$$M = (I - K)^{-1} = (I - LR)^{-1} = I + L(I - RL)^{-1}R.$$

Note that $(I - RL)$ is a $kn \times kn$ matrix. So, to compute M (and thus complete the **encode** step) we no longer need to invert an $m \times m$ matrix! Instead we just need to

invert a matrix of size $kn \times kn$. This is essentially where the $\tilde{O}((kn)^\omega)$ runtime in Theorem 4 comes from.

Conceptually, this argument corresponds to assigning flow vectors through the graph by replacing random linear combinations with random "low-rank combinations." That is, for an edge $e \in E_{\text{out}}(u)$ exiting a vertex u, we define the flow vector at e to be

$$\vec{e} = \sum_{i=1}^{k} \left(\sum_{f \in E_{\text{in}}(u)} L_i[f, u]\vec{f} \right) \cdot R_i[u, e],$$

where the inner summation is over all edges f entering u, $\vec{f}$ denotes the flow vector assigned to edge f, and the $L_i[f, u]$ and $R_i[u, e]$ terms correspond to random field elements uniquely determined by the index i and some (edge, vertex) pair.

Here, unlike in the method described in Sect. 3.1, the coefficient in front of $\vec{f}$ in its contribution to $\vec{e}$ is not uniquely determined by the pair of edges f and e. Rather, if edge f enters node u, then it has the same set of "weights" $L_i[f, u]$ it contributes to every flow vector exiting u. However, since we use k distinct weights, this restricted rule for propagating flow vectors still suffices to compute $\min(k, \lambda(s, t))$.

A good way to think about the effect of this alternate approach is that now for any vertex v and any integer $\ell \leq k$, if some subset of ℓ flow vectors assigned to edges in $E_{\text{in}}(v)$ is independent, then we expect that every subset of at most ℓ flow vectors assigned to edges in $E_{\text{out}}(v)$ is also independent. In the previous framework, this result held even for $\ell > k$. By relaxing the method used to determine flow vectors, we achieve a weaker condition, but this is still enough to solve k-APC.

This modification makes the **encode** step more complicated (it now consists of two parts: one where we invert a matrix, and one where we multiply that inverse with other matrices), but speeds it up overall. To speed up the **decode** step, we use a variant of an observation from the proof of [1, Theorem 5.2] to argue that we can assume every vertex in our graph has indegree and outdegree k. By Proposition 8 and Eq. (3), this means we can compute $\min(k, \lambda(s, t))$ for all pairs (s, t) of vertices in $\tilde{O}(k^\omega n^2)$ time. So the bottleneck in our algorithm comes from the **encode** step, which yields the $\tilde{O}((kn)^\omega)$ runtime.

3.3 All-Pairs Vertex Connectivity

Our starting point is the $\tilde{O}((kn)^\omega)$ time algorithm in [1], which computes $\min(k, \nu(s, t))$ for all pairs of vertices (s, t) which are not edges. That algorithm is based off a variant of the flow vector encoding scheme outlined Sect. 3.1. Rather than assign vectors to edges, we instead assign flow vectors to vertices (intuitively this is fine because we are working with vertex connectivities in the k-APVC problem). The rest of the construction is similar: we imagine pumping some initial vectors to s and its out-neighbors, and then we propagate the flow through the graph so that at the end, for any vertex v,

the flow vector assigned to v is a random linear combination of flow vectors assigned to in-neighbors of v.[6]

Let K be an $n \times n$ matrix, whose rows and columns are indexed by vertices in the input graph. For each pair (u, v) of vertices, if there is an edge from u to v, we set $K[u, v]$ to be a uniform random element in $\mathbb{F}$. Otherwise, $K[u, v] = 0$. These entries correspond precisely to coefficients used in the random linear combinations of the flow vector framework.

Now define the matrix

$$M = (I - K)^{-1}. \tag{4}$$

Then we argue that for any pair (s, t) of vertices, we have

$$\operatorname{rank} M[V_{\text{out}}[s], V_{\text{in}}[t]] = \begin{cases} \nu(s,t) + 1 & \text{if } (s,t) \text{ is an edge} \\ \nu(s,t) & \text{otherwise.} \end{cases} \tag{5}$$

Previously, [1, Proof of Lemma 5.1] sketched a different argument, which shows that $\operatorname{rank} M[V_{\text{out}}(s), V_{\text{in}}(t)] = \nu(s, t)$ when (s, t) is not an edge. As we discuss in Appendix B.1, this claim does not necessarily hold when (s, t) is an edge.

We use Eq. (5) to solve k-APVC. For the **encode** step, we compute M. By Eq. (4), we can do this by inverting an $n \times n$ matrix, which takes $\tilde{O}(n^\omega)$ time. For the **decode** step, by Eq. (5) and Proposition 8, we can compute $\min(k, \nu(s, t))$ for all pairs (s, t) of vertices in asymptotically

$$\sum_{s,t} \left(\deg_{\text{out}}(s) \deg_{\text{in}}(t) + k^\omega\right) = m^2 + k^\omega n^2$$

time, where the sum is over all vertices s and t in the graph. The runtime bound we get here for the **decode** step is far too high – naively computing the ranks of submatrices is too slow if the graph has many high-degree vertices.

To avoid this slowdown, [1] employs a simple trick to reduce degrees in the graph: we can add layers of k new nodes to block off the ingoing and outgoing edges from each vertex in the original graph. That is, for each vertex s in G, we add a set S of k new nodes, replace the edges in $E_{\text{out}}(s)$ with edges from s to all the nodes in S, and add edges from every node in S to every vertex originally in $V_{\text{out}}(s)$. Similarly, for each vertex t in G, we add a set T of k new nodes, replace the edges in $E_{\text{in}}(t)$ with edges from all the nodes in T to t, and add edges from every vertex originally in $V_{\text{in}}(t)$ to every node in T.

It is easy to check that this transformation preserves the value of $\min(k, \nu(s, t))$ for all pairs (s, t) of vertices in the original graph where (s, t) is not an edge. Moreover, all vertices in the original graph have indegree and outdegree exactly k in the new graph. Consequently, the **decode** step can now be implemented to run in $\tilde{O}(k^\omega n^2)$ time. Unfortunately, this construction increases the number of vertices in the graph

[6] Actually, this behavior only holds for vertices $v \notin V_{\text{out}}[s]$. The rule for flow vectors assigned to vertices in $V_{\text{out}}[s]$ is a little more complicated, and depends on the values of the initial pumped vectors.

from n to $(2k+1)n$. As a consequence, in the **encode** step, the matrix K we work with is no longer $n \times n$, but instead is of size $(2k+1)n \times (2k+1)n$. Now inverting $I - K$ to compute M requires $\tilde{O}((kn)^\omega)$ time, which is why [1] obtains this runtime for their algorithm.

Our Improvement

Intuitively, the modification used by [1] to reduce degrees in the graph feels very inefficient. This transformation makes the graph larger in order to "lose information" about connectivity values greater than k. Rather than modify the graph in this way, can we modify the flow vector scheme itself to speed up the **decode** step? Our algorithm does this, essentially modifying the matrix of flow vectors to simulate the effect of the previously described transformation, without ever explicitly adding new nodes to the graph.

Instead of working directly with the matrix M from Eq. (4), for each pair (s, t) of vertices we define a $(k+1) \times (k+1)$ matrix

$$M_{s,t} = B_s \left(M[V_{\text{out}}[s], V_{\text{in}}[t]]\right) C_t$$

which is obtained from multiplying a submatrix of M on the left and right by small random matrices B_s and C_t, with $k+1$ rows and columns respectively. Since B_s has $k+1$ rows and C_t has $k+1$ columns, we can argue that with high probability, Eq. (5) implies that

$$\text{rank } M_{s,t} = \begin{cases} \min(k+1, \nu(s,t)+1) & \text{if } (s,t) \text{ is an edge} \\ \min(k+1, \nu(s,t)) & \text{otherwise.} \end{cases}$$

So we can compute $\min(k, \nu(s,t))$ from the value of rank $M_{s,t}$. This idea is similar to the preconditioning method used in algorithms for computing matrix rank efficiently (see [5] and the references therein). Conceptually, we can view this approach as a modification of the flow vector framework. Let $d = \deg_{\text{out}}(s)$. As noted in Sect. 3.1, previous work

1. starts by pumping out distinct d-dimensional unit vectors to nodes in $V_{\text{out}}(s)$, and then
2. computes the rank of all flow vectors of vertices in $V_{\text{in}}(t)$.

In our work, we instead

1. start by pumping out $(d+1)$ random $(k+1)$-dimensional vectors to nodes in $V_{\text{out}}[s]$, and then
2. compute the rank of $(k+1)$ random linear combinations of flow vectors for vertices in $V_{\text{in}}[t]$.

This alternate approach suffices for solving the k-APVC problem, while avoiding the slow $\tilde{O}((kn)^\omega)$ **encode** step of previous work.

So, in the **decode** step of our algorithm, we compute $\min(k, \nu(s,t))$ for each pair (s,t) of vertices by computing the rank of the $(k+1)\times(k+1)$ matrix $M_{s,t}$, in $\tilde{O}(k^\omega n^2)$ time overall.

Our **encode** step is more complicated than previous work, because not only do we need to compute the inverse $(I-K)^{-1}$, we also have to construct the $M_{s,t}$ matrices. Naively computing each $M_{s,t}$ matrix separately is too slow, so we end up using an indirect approach to compute all entries of the $M_{s,t}$ matrices *simultaneously*, with just $O(k^2)$ multiplications of $n \times n$ matrices. This takes $\tilde{O}(k^2 n^\omega)$ time, which is the bottleneck for our algorithm.

4 Flow Vector Encoding

The arguments in this section are similar to the arguments from [6, Section 2], but involve more complicated proofs because we work with low-rank random matrices as opposed to generic random matrices.

Fix a source vertex s in the input graph G. Let $d = \deg_{\text{out}}(s)$ denote the number of edges leaving s. Let $e_1, \ldots, e_d \in E_{\text{out}}(s)$ be the outgoing edges from s.

Take a prime $p = \Theta(m^5)$. Let $\vec{u}_1, \ldots, \vec{u}_d$ be distinct unit vectors in $\mathbb{F}_p^d$.

Eventually, we will assign each edge e in G a vector $\vec{e} \in \mathbb{F}_p^d$, which we call a *flow vector*. These flow vectors will be determined by a certain system of vector equations. To describe these equations, we first introduce some symbolic matrices.

For each index $i \in [k]$, we define an $m \times n$ matrix X_i, whose rows are indexed by edges of G and columns are indexed by vertices of G, such that for each edge $e = (u,v)$, entry $X_i[e,v] = x_{i,ev}$ is an indeterminate. All entries in X_i not of this type are zero.

Similarly, we define $n \times m$ matrices Y_i, with rows indexed by vertices of G and columns indexed by edges of G, such that for every edge $f = (u,v)$, the entry $Y_i[u,f] = y_{i,uf}$ is an indeterminate. All entries in Y_i not of this type are zero.

Let X be the $m \times kn$ matrix formed by horizontally concatenating the X_i matrices. Similarly, let Y be the $kn \times m$ matrix formed by vertically concatenating the Y_i matrices. Then we define the matrix

$$Z = XY = X_1Y_1 + \cdots + X_kY_k. \tag{6}$$

By construction, Z is an $m \times m$ matrix, with rows and columns indexed by edges of G, such that for any edges $e = (u,v)$ and $f = (v,w)$, we have

$$Z[e,f] = \sum_{i=1}^{k} x_{i,ev} y_{i,vf} \tag{7}$$

and all other entries of Z are set to zero.

Consider the following procedure. We assign independent, uniform random values from $\mathbb{F}_p$ to each variable $x_{i,ev}$ and $y_{i,uf}$. Let L_i, L, R_i, R, and K be the matrices over $\mathbb{F}_p$ resulting from this assignment to X_i, X, Y_i, Y, and Z respectively. In particular,

we have

$$K = LR. \tag{8}$$

Now, to each edge e, we assign a flow vector $\vec{e} \in \mathbb{F}_p^d$, satisfying the following equalities:

1. Recall that $e_1, \ldots, e_d$ are all the edges exiting s, and $\vec{u}_1, \ldots, \vec{u}_d$ are distinct unit vectors in $\mathbb{F}_p^d$. For each edge $e_i \in E_{\text{out}}(s)$, we require its flow vector satisfy

$$\vec{e}_i = \left(\sum_{f \in E_{\text{in}}(s)} \vec{f} \cdot K[f, e_i] \right) + \vec{u}_i. \tag{9}$$

2. For each edge $e = (u, v)$ with $u \neq s$, we require its flow vector satisfy

$$\vec{e} = \sum_{f \in E_{\text{in}}(u)} \vec{f} \cdot K[f, e]. \tag{10}$$

A priori it is not obvious that flow vectors satisfying the above two conditions exist, but we show below that they do (with high probability). Let H_s be the $d \times m$ matrix whose columns are indexed by edges in G, such that the column associated with e_i is $\vec{u}_i$ for each index i, and the rest of the columns are zero vectors. Let F be the $d \times m$ matrix, with columns indexed by edges in G, whose columns $F[*, e] = \vec{e}$ are flow vectors for the corresponding edges. Then Eqs. (9) and (10) are encapsulated by the simple matrix equation

$$F = FK + H_s. \tag{11}$$

The following lemma shows we can solve for F in the above equation, with high probability.

Lemma 10 *We have* $\det(I - K) \neq 0$*, with probability at least* $1 - 1/m^3$.

Proof Since the input graph has no self-loops, by Eq. (7) and the discussion immediately following it, we know that the diagonal entries of the $m \times m$ matrix Z are zero. By Eq. (7), each entry of Z is a polynomial of degree at most two, with constant term set to zero. Hence, $\det(I - Z)$ is a polynomial over $\mathbb{F}_p$ with degree at most $2m$, and constant term equal to 1. In particular, this polynomial is nonzero. Then by the Schwartz-Zippel Lemma (Proposition 9), $\det(I - K)$ is nonzero with probability at least

$$1 - 2m/p \geq 1 - 1/m^3$$

by setting $p \geq 2m^4$. □

Suppose from now on that $\det(I - K) \neq 0$ (by Lemma 10, this occurs with high probability). Then with this assumption, we can solve for F in Eq. (11) to get

$$F = H_s(I - K)^{-1} = \frac{H_s\,(\text{adj}(I - K))}{\det(I - K)}. \tag{12}$$

This equation will allow us to relate ranks of collections of flow vectors to connectivity values in the input graph.

Lemma 11 *For any vertex t in G, with probability at least $1 - 2/m^3$, we have*

$$\text{rank } F[*, E_{\text{in}}(t)] \leq \lambda(s, t).$$

Proof Abbreviate $\lambda = \lambda(s, t)$. Conceptually, this proof works by arguing that the flow vectors assigned to all edges entering t are linear combinations of the flow vectors assigned to edges in a minimum (s, t)-cut of G.

Let C be a minimum (s, t)-cut. By Menger's theorem, $|C| = \lambda$.

Let T be the set of nodes which can reach t without using an edge in C. Let S be set of nodes in G not in T. Let E' be the set of edges $e = (u, v)$ in G with $v \in T$.

Let E' be the set of edges $e = (u, v)$ with $v \in T$.

Set $K' = K[E', E']$ and $F' = F[*, E']$. Finally, let H' be a matrix whose columns are indexed by edges in E', such that the column associated with an edge $e \in C$ is $\vec{e}$, and all other columns are equal to $\vec{0}$.

Then by Eqs. (9) and (10), we have

$$F' = F'K' + H'.$$

Indeed, for any edge $e = (u, v) \in E'$, if $u \in S$ then $e \in C$ so $H'[*, e] = \vec{e}$, and there can be no edge $f \in E'$ entering u, so $(F'K')[*, e] = \vec{0}$. If instead $u \in T$, then $H'[*, e] = \vec{0}$, but every edge f entering u is in E', so by Eq. (10), we have $(F'K')[*, e] = F'[*, e]$ as desired.

Using similar reasoning to the proof of Lemma 10, we have $\det(I - K') \neq 0$ with probability at least $1 - 1/m^3$. If this event occurs, we can solve for F' in the previous equation to get

$$F' = H'(I - K')^{-1}.$$

Since H' has at most λ nonzero columns, rank $H \leq \lambda$. So by the above equation, rank $F' \leq \lambda$. By definition, $E_{\text{in}}(t) \subseteq E'$. Thus $F[*, E_{\text{in}}(t)]$ is a submatrix of F'. Combining this with the previous results, we see that rank $F[*, E_{\text{in}}(t)] \leq \lambda$, as desired. The claimed probability bound follows by a union bound over the events that $I - K$ and $I - K'$ are both invertible. □

Lemma 12 *For any vertex t in G, with probability at least $1 - 2/m^3$, we have*

$$\text{rank } F[*, E_{\text{in}}(t)] \geq \min(k, \lambda(s, t)).$$

Proof Abbreviate $\lambda = \min(k, \lambda(s,t))$. Intuitively, our proof argues that the presence of edge-disjoint paths from s to t leads to certain edges in $E_{\text{in}}(t)$ being assigned linearly independent flow vectors (with high probability), which then implies the desired lower bound.

By Menger's theorem, G contains λ edge-disjoint paths $P_1, \ldots, P_\lambda$ from s to t.

Consider the following assignment to the variables of the symbolic matrices X_i and Y_i. For each index $i \le \lambda$ and edge $e = (u, v)$, we set variable $x_{i,ev} = 1$ if e is an edge in P_i. Similarly, for each index $i \le \lambda$ and edge $f = (u, v)$, we set variable $y_{i,uf} = 1$ if f is an edge in P_i. All other variables are set to zero. In particular, if $i > \lambda$, then X_i and Y_i have all their entries set to zero. With respect to this assignment, the matrix $X_i Y_i$ (whose rows and columns are indexed by edges in the graph) has the property that $(X_i Y_i)[e, f] = 1$ if f is the edge following e on path P_i, and all other entries are set to zero.

Then by Eq. (6), we see that under this assignment, $Z[e, f] = 1$ if e and f are consecutive edges in some path P_i, and all other entries of Z are set to zero. For this particular assignment, because the P_i are edge-disjoint paths, Equations (9) and (10) imply that the last edge of each path P_i is assigned a distinct d-dimensional unit vector. These vectors are independent, so, rank $F[*, E_{\text{in}}(t)] = \lambda$ in this case.

With respect to this assignment, this means that $F[*, E_{\text{in}}(t)]$ contains a $\lambda \times \lambda$ full-rank submatrix. Let F' be a submatrix of $F[*, E_{\text{in}}(t)]$ with this property. Since F' has full rank, we have $\det F' \neq 0$ for the assignment described above.

Now, before assigning values to variables, each entry of $\text{adj}(I - Z)$ is a polynomial of degree at most $2m$. So by Eq. (12), $\det F'$ is equal to some polynomial P of degree at most $2\lambda m$, divided by $(\det(I - Z))^\lambda$. We know P is a nonzero polynomial, because we saw above that $\det F'$ is nonzero for some assignment of values to the variables (and if P were the zero polynomial, then $\det F'$ would evaluate to zero under every assignment).

By Lemma 10, with probability at least $1 - 1/m^3$, a random evaluation to the variables will have $\det(I - Z)$ evaluate to a nonzero value. Assuming this event occurs, by Schwartz-Zippel Lemma (Proposition 9), a random evaluation to the variables in Z will have $\det F' \neq 0$ with probability at least $1 - (2\lambda m)/p \ge 1 - 1/m^3$ by setting $p \ge 2\,m^5$.

So by union bound, a particular $\lambda \times \lambda$ submatrix of $F[*, E_{\text{in}}(t)]$ will be full rank with probability at least $1 - 2/m^3$. This proves the desired result. □

Lemma 13 *Fix vertices s and t. Define $\lambda = \text{rank}\ (I - K)^{-1}[E_{\text{out}}(s), E_{\text{in}}(t)]$. With probability at least $1 - 4/m^3$, we have $\min(k, \lambda) = \min(k, \lambda(s,t))$.*

Proof The definition of H_s together with Eq. (12) implies that

$$F[*, E_{\text{in}}(t)] = (I - K)^{-1}[E_{\text{out}}(s), E_{\text{in}}(t)]. \tag{13}$$

By union bound over Lemmas 12 and 11, with probability at least $1 - 4/m^3$ the inequalities

$$\lambda = \text{rank}\ (I - K)^{-1}[E_{\text{out}}(s), E_{\text{in}}(t)] = \text{rank}\ F[*, E_{\text{in}}(t)] \le \lambda(s,t)$$

and

$$\lambda = \text{rank } (I - K)^{-1}[E_{\text{out}}(s), E_{\text{in}}(t)] = \text{rank } F[*, E_{\text{in}}(t)] \geq \min(k, \lambda(s, t))$$

simultaneously hold. The desired result follows. □

5 Connectivity Algorithm

In this section, we present our algorithm for k-APC.

Graph Transformation We begin by modifying the input graph G as follows. For every vertex v in G, we introduce two new nodes v_{out} and v_{in}. We replace each edge (u, v) originally in G is by the edge $(u_{\text{out}}, v_{\text{in}})$. We add k parallel edges from v to v_{out}, and k parallel edges from v_{in} to v, for all u and v. We call vertices present in the graph before modification the *original vertices*.

Suppose G originally had n nodes and m edges. Then the modified graph has $n_{\text{new}} = 3n$ nodes and $m_{\text{new}} = m + 2kn$ edges. For any original vertices s and t, edge-disjoint paths from s to t in the new graph correspond to edge disjoint paths from s to t in the original graph. Moreover, for any integer $\ell \leq k$, if the original graph contained ℓ edge-disjoint paths from s to t, then the new graph contains ℓ edge-disjoint paths from s to t as well.

Thus, for any original vertices s and t, the value of $\min(k, \lambda(s, t))$ remains the same in the old graph and the new graph. So, it suffices to solve k-APC on the new graph. In this new graph, the indegrees and outdegrees of every original vertex are equal to k. Moreover, sets $E_{\text{out}}(s)$ and $E_{\text{in}}(t)$ are pairwise disjoint, over all original vertices s and t.

Additional Definitions We make use of the matrices defined in Sect. 4, except now these matrices are defined with respect to the modified graph. In particular, K, L, and R are now matrices with dimensions $m_{\text{new}} \times m_{\text{new}}$, $m_{\text{new}} \times kn_{\text{new}}$, and $kn_{\text{new}} \times m_{\text{new}}$ respectively.

Moreover, we work over a field $\mathbb{F}_p$ for some prime $p = \Theta(m_{\text{new}}^5)$.

Define $\tilde{L}$ to be the $kn \times kn_{\text{new}}$ matrix obtained by vertically concatenating $L[E_{\text{out}}(s), *]$ over all original vertices s. Similarly, define $\tilde{R}$ to be the $kn_{\text{new}} \times kn$ matrix obtained by horizontally concatenating $R[*, E_{\text{in}}(t)]$ over all original vertices t.

The Algorithm Using the above definitions, we present our approach for solving k-APC in Algorithm 1.

Theorem 14 *With probability at least* $1 - 5/(m_{\text{new}})$*, Algorithm 1 correctly solves* k-APC.

Proof By Lemma 10 with probability at least $1 - 1/(m_{\text{new}})^4$ the matrix $I - K$ is invertible (note that here we are using our choice of field size $p = \Theta(m_{\text{new}}^5)$).

Going forward, we assume that $I - K$ is invertible.

Algorithm 1 Our algorithm for solving k-APC.

1: Compute the $n_{\text{new}} \times n_{\text{new}}$ matrix $(I - RL)^{-1}$.
2: Compute the $kn_{\text{new}} \times kn_{\text{new}}$ matrix $M = \tilde{L}(I - RL)^{-1}\tilde{R}$.
3: For each pair (s, t) of original vertices, compute

$$\text{rank } M[E_{\text{out}}(s), E_{\text{in}}(t)]$$

and output this as the value for $\min(k, \lambda(s, t))$.

By Lemma 13, with probability at least $1 - 4/(m_{\text{new}})^3$, we have

$$\text{rank}(I - K)^{-1}[E_{\text{out}}(s), E_{\text{in}}(t)] = \min(k, \lambda(s, t)) \tag{14}$$

for any given original vertices s and t. By union bound over all $n^2 \leq (m_{\text{new}})^2$ pairs of original vertices (s, t), we see that Eq. (14) holds for all original vertices s and t with probability at least $1 - 4/(m_{\text{new}})$.

Since $I - K$ is invertible, by Eq. (8) and Proposition 6 we have

$$(I - K)^{-1} = (I - LR)^{-1} = I + L(I - RL)^{-1}R.$$

Using the above equation in Eq. (14) shows that for original vertices s and t, the quantity $\min(k, \lambda(s, t))$ is equal to the rank of

$$(I + L(I - RL)^{-1}R)[E_{\text{out}}(s), E_{\text{in}}(t)] = L[E_{\text{out}}(s), *](I - RL)^{-1}R[*, E_{\text{in}}(t)]$$

where we use the fact that $I[E_{\text{out}}(s), E_{\text{in}}(t)]$ is the all zeroes matrix, since in the modified graph, $E_{\text{out}}(s)$ and $E_{\text{in}}(t)$ are disjoint sets for all pairs of original vertices (s, t).

Then by definition of $\tilde{L}$ and $\tilde{R}$, the above equation and discussion imply that

$$\min(k, \lambda(s, t)) = \text{rank } (\tilde{L}(I - RL)^{-1}\tilde{R})[E_{\text{out}}(s), E_{\text{in}}(t)] = \text{rank } M[E_{\text{out}}(s), E_{\text{in}}(t)]$$

which proves that Algorithm 1 outputs the correct answers.

A union bound over the events that $I - K$ is invertible and that Eq. (14) holds for all (s, t), shows that Algorithm 1 is correct with probability at least $1 - 5/(m_{\text{new}})$. □

We are now ready to prove our main result.

Theorem 4 *For any positive integer k, k-APC can be solved in $\tilde{O}((kn)^{\omega})$ time.*

Proof By Theorem 14, Algorithm 1 correctly solves the k-APC problem. We now argue that Algorithm 1 can be implemented to run in $\tilde{O}((kn)^{\omega})$ time.

In step 1 of Algorithm 1, we need to compute $(I - RL)^{-1}$.

From the definitions of R and L, we see that to compute RL, it suffices to compute the products $R_i L_j$ for each pair of indices $(i, j) \in [k]^2$. The matrix $R_i L_j$ is $n_{\text{new}} \times n_{\text{new}}$, and its rows and columns are indexed by vertices in the graph. Given vertices u and

v, let $E(u, v)$ denote the set of parallel edges from u to v. From the definitions of the R_i and L_j matrices, we see that for any vertices u and v, we have

$$(R_i L_j)[u, v] = \sum_{e \in E(u,v)} R_i[u, e] L_j[e, v]. \tag{15}$$

As noted in Sect. 2, for all vertices u and v we may assume that $|E(u, v)| \leq k$.

For each vertex u, define the $k \times \deg_{\text{out}}(u)$ matrix R'_u, with rows indexed by $[k]$ and columns indexed by edges exiting u, by setting

$$R'_u[i, e] = R_i[u, e]$$

for all $i \in [k]$ and $e \in E_{\text{out}}(u)$.

Similarly, for each vertex v, define the $\deg_{\text{in}}(v) \times k$ matrix L'_v by setting

$$L'_v[e, j] = L_j[e, v]$$

for all $e \in E_{\text{in}}(v)$ and $j \in [k]$.

Finally, for each pair (u, v) of vertices, define $R'_{uv} = R'_u[*, E(u, v)]$ and $L'_{uv} = L'_v[E(u, v), *]$. Then by Eq. (15), we have

$$(R_i L_j)[u, v] = R'_{uv} L'_{uv}[i, j].$$

Thus, to compute the $R_i L_j$ products, it suffices to build the R'_u and L'_v matrices in $O(km_{\text{new}})$ time, and then compute the $R'_{uv} L'_{uv}$ products. We can do this by computing $(n_{\text{new}})^2$ products of pairs of $k \times k$ matrices. Since for every pair of vertices (u, v), there are at most k parallel edges from u to v, $km_{\text{new}} \leq k^2 n^2$, we can compute all the $R_i L_j$ products, and hence the entire matrix RL, in $\tilde{O}(n^2 k^\omega)$ time.

We can then compute $I - RL$ by modifying $O(kn)$ entries of RL. Finally, by Proposition 7 we can compute $(I - RL)^{-1}$ in $\tilde{O}((kn)^\omega)$ time.

So overall, step 1 of Algorithm 1 takes $\tilde{O}((kn)^\omega)$ time.

In step 2 of Algorithm 1, we need to compute $M = \tilde{L}(I - RL)^{-1}\tilde{R}$.

Recall that $\tilde{L}$ is a $kn \times kn_{\text{new}}$ matrix. By definition, each row of $\tilde{L}$ has k nonzero entries. Similarly, $\tilde{R}$ is an $kn_{\text{new}} \times kn$ matrix, with k nonzero entries in each column.

Thus we can compute M, and complete step 2 of Algorithm 1 in $\tilde{O}((kn)^\omega)$ time.

Finally, in step 3 of Algorithm 1, we need to compute

$$\text{rank } M[E_{\text{out}}(s), E_{\text{in}}(t)] \tag{16}$$

for each pair of original vertices (s, t) in the graph. In the modified graph, each original vertex has indegree and outdegree k, so each $M[E_{\text{out}}(s), E_{\text{in}}(t)]$ is a $k \times k$ matrix. For any fixed (s, t), by Proposition 8 we can compute the rank of $M[E_{\text{out}}(s), E_{\text{in}}(t)]$ in $\tilde{O}(k^\omega)$ time.

So we can compute the ranks from Eq. (16) for all n^2 pairs of original vertices (s, t) and complete step 3 of Algorithm 1 in $\tilde{O}(k^\omega n^2)$ time.

Thus we can solve k-APC in $\tilde{O}((kn)^\omega)$ time overall, as claimed. □

6 Encoding Vertex Connectivities

Take a prime $p = \tilde{\Theta}(n^5)$. Let K be an $n \times n$ matrix, whose rows and columns are indexed by vertices of G. For each pair (u, v) of vertices, if (u, v) is an edge in G, we set $K[u, v]$ to be a uniform random element of $\mathbb{F}_p$. Otherwise, $K[u, v] = 0$.

Recall from Sect. 2 that given a vertex v in G, we let $V_{\text{in}}[v] = V_{\text{in}}(v) \cup \{v\}$ be the set consisting of v and all in-neighbors of v, and $V_{\text{out}}[v] = V_{\text{out}}(v) \cup \{v\}$ be the set consisting of v and all out-neighbors of v. The following proposition[7] is based off ideas from [6, Section 2] and [1, Section 5]. We present a complete proof of this result in Appendix B.2.

Proposition 15 *For any vertices s and t in G, with probability at least $1 - 3/n^3$, the matrix $(I - K)$ is invertible and we have*

$$\text{rank } (I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}[t]] = \begin{cases} \nu(s,t) + 1 & \text{if } (s,t) \text{ is an edge} \\ \nu(s,t) & \text{otherwise.} \end{cases}$$

Proposition 15 shows that we can compute vertex connectivities in G simply by computing ranks of certain submatrices of $(I - K)^{-1}$. However, these submatrices could potentially be quite large, which is bad if we want to compute the vertex connectivities quickly. To overcome this issue, we show how to decrease the size of $(I-K)^{-1}$ while still preserving relevant information about the value of $\nu(s, t)$.

Lemma 16 *Let M be an $a \times b$ matrix over $\mathbb{F}_p$. Let Γ be a $(k + 1) \times a$ matrix with uniform random entries from $\mathbb{F}_p$. Then with probability at least $1 - (k + 1)/p$, we have*

$$\text{rank } \Gamma M = \min(k + 1, \text{rank } M).$$

Proof Since ΓM has $k + 1$ rows, $\text{rank}(\Gamma M) \leq k + 1$.

Similarly, since ΓM has M as a factor, $\text{rank}(\Gamma M) \leq \text{rank } M$. Thus

$$\text{rank } \Gamma M \leq \min(k + 1, \text{rank } M). \tag{17}$$

So, it suffices to show that ΓM has rank at least $\min(k + 1, \text{rank } M)$.

Set $r = \min(k + 1, \text{rank } M)$. Then there exist subsets S and T of row and column indices respectively, such that $|S| = |T| = r$ and $M[S, T]$ has rank r. Now, let U be an arbitrary set of r rows in Γ. Consider the matrix $M' = (\Gamma M)[U, T]$.

We can view each entry of M' as a polynomial of degree at most 1 in the entries of Γ. This means that $\det M'$ is a polynomial of degree at most r in the entries of Γ. Moreover, if the submatrix $\Gamma[U, T] = I$ happens to be the identity matrix, then $M' = M[S, T]$. This implies that $\det M'$ is a nonzero polynomial in the entries of Γ, because for some assignment of values to the entries of Γ, this polynomial has nonzero evaluation $\det M[S, T] \neq 0$ (where we are using the fact that $M[S, T]$ has full rank).

[7] The result stated here differs from a similar claim used in [1, Section 5] We discuss this difference, and related subtleties, in Appendix B.1.

So by the Schwartz-Zippel Lemma (Proposition 9), the matrix ΓM has rank at least r, with probability at least $1 - r/p$.

Together with Eq. (17), this implies the desired result. □

Now, to each vertex u in the graph, we assign a $(k+1)$-dimensional column vector $\vec{b}_u$ and a $(k+1)$-dimensional row vector $\vec{c}_u$.

Let B be the $(k+1) \times n$ matrix formed by concatenating all of the $\vec{b}_u$ vectors horizontally, and let C be the $n \times (k+1)$ matrix formed by concatenating all of the $\vec{c}_u$ vectors vertically. For each pair of distinct vertices (s, t), define the $(k+1) \times (k+1)$ matrix

$$M_{s,t} = B[*, V_{\text{out}}[s]] \left((I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}[t]] \right) C[V_{\text{in}}[t], *]. \tag{18}$$

The following result is the basis of our algorithm for k-APVC.

Lemma 17 *For any vertices s and t in G, with probability at least $1 - 5/n^3$, we have*

$$\text{rank } M_{s,t} = \begin{cases} \min(k+1, \nu(s,t)+1) & \text{if } (s,t) \text{ is an edge} \\ \min(k+1, \nu(s,t)) & \text{otherwise.} \end{cases}$$

Proof Fix vertices s and t. Then, by Proposition 15, we have

$$\text{rank } (I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}[t]] = \begin{cases} \nu(s,t)+1 & \text{if } (s,t) \text{ is an edge} \\ \nu(s,t) & \text{otherwise} \end{cases}$$

with probability at least $1 - 3/n^3$. Assume the above equation holds.

Then, by setting $\Gamma = B[*, V_{\text{out}}[s]]$ and $M = (I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}[t]]$ in Lemma 16, we see that with probability at least $1 - 1/n^3$ we have

$$\text{rank } B[*, V_{\text{out}}[s]](I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}(t)] = \begin{cases} \min(k+1, \nu(s,t)+1) & \text{if } (s,t) \text{ is an edge} \\ \min(k+1, \nu(s,t)) & \text{otherwise.} \end{cases}.$$

Assume the above equation holds.

Finally, by setting $\Gamma = C^\top[*, V_{\text{in}}(t)]$ and $M = (B[*, V_{\text{out}}[s]](I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}(t)])^\top$ in Lemma 16 and transposition, we see that with probability at least $1 - 1/n^3$ we have

$$\text{rank } B[*, V_{\text{out}}[s]] \left((I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}(t)] \right) C[V_{\text{in}}(t), *] = \min(k+1, \nu(s,t)+1)$$

if there is an edge from s to t, and

$$\text{rank } B[*, V_{\text{out}}[s]] \left((I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}(t)] \right) C[V_{\text{in}}(t), *] = \min(k+1, \nu(s,t))$$

otherwise. So by union bound, the desired result holds with probability at least $1 - 5/n^3$. □

7 Vertex Connectivity Algorithm

Let A be the adjacency matrix of the graph G with self-loops. That is, A is the $n \times n$ matrix whose rows and columns are indexed by vertices of G, and for every pair (u, v) of vertices, $A[u, v] = 1$ if $v \in V_{\text{out}}[u]$ (equivalently, $u \in V_{\text{in}}[v]$), and $A[u, v] = 0$ otherwise.

Recall the definitions of the $\vec{b}_u$ and $\vec{c}_u$ vectors, and the K, B, C and $M_{s,t}$ matrices from Sect. 6. For each $i \in [k+1]$, let P_i be the $n \times n$ diagonal matrix, with rows and columns indexed by vertices of G, such that $P_i[u, u] = \vec{b}_u[i]$. Similarly, let Q_i be the $n \times n$ diagonal matrix, with rows and columns indexed by vertices of G, such that $Q_i[u, u] = \vec{c}_u[i]$.

With these definitions, we present our approach for solving k-APVC in Algorithm 2.

Algorithm 2 Our algorithm for solving k-APVC.

1: Compute the $n \times n$ matrix $(I - K)^{-1}$.
2: For each pair $(i, j) \in [k+1]^2$ of indices, compute the $n \times n$ matrix

$$D_{ij} = A P_i (I - K)^{-1} Q_j A^{\top}.$$

3: For each pair (s, t) of vertices, let $F_{s,t}$ be the $(k+1) \times (k+1)$ matrix whose (i, j) entry is equal to $D_{ij}[s, t]$. If (s, t) is an edge, output (rank $F_{s,t}$) $- 1$ as the value for $\min(k, \nu(s, t))$. Otherwise, output $\min(k,$ rank $F_{s,t})$ as the value for $\min(k, \nu(s, t))$.

The main idea of Algorithm 2 is to use Lemma 17 to reduce computing $\min(k, \nu(s, t))$ for a given pair of vertices (s, t) to computing the rank of a corresponding $(k+1) \times (k+1)$ matrix, $M_{s,t}$. To make this approach efficient, we compute the entries of all $M_{s,t}$ matrices simultaneously, using a somewhat indirect argument.

Theorem 18 *With probability at least* $1 - 5/n$*, Algorithm 2 correctly solves* k-APVC.

Proof We prove correctness of Algorithm 2 using the following claim.

Claim 19 For all pairs of indices $(i, j) \in [k+1]^2$ and all pairs of vertices (s, t), we have

$$M_{s,t}[i, j] = D_{ij}[s, t],$$

where D_{ij} is the matrix computed in step 2 of Algorithm 2.

Proof By expanding out the expression for D_{ij} from step 2 of Algorithm 2, we have

$$D_{ij}[s, t] = \sum_{u,v} A[s, u] P_i[u, u] \left((I - K)^{-1}[u, v] \right) Q_j[v, v] A[v, t],$$

where the sum is over all vertices u, v in the graph (here, we use the fact that P_i and Q_j are diagonal matrices). By the definitions of A, the P_i, and the Q_j matrices, we

have

$$D_{ij}[s,t] = \sum_{\substack{u \in V_{\text{out}}[s] \\ v \in V_{\text{in}}[t]}} \vec{b}_u[i] \left((I - K)^{-1}[u,v] \right) \vec{c}_v[j]. \tag{19}$$

On the other hand, the definition of $M_{s,t}$ from Eq. (18) implies that

$$M_{s,t}[i,j] = \sum_{\substack{u \in V_{\text{out}}[s] \\ v \in V_{\text{in}}[t]}} B[i,u] \left((I - K)^{-1}[u,v] \right) C[v,j].$$

Since $B[i,u] = \vec{b}_u[i]$ and $C[v,j] = \vec{c}_v[j]$, the above equation and Eq. (19) imply that

$$M_{s,t}[i,j] = D_{ij}[s,t]$$

for all (i,j) and (s,t), as desired. □

By Claim 19, the matrix $F_{s,t}$ computed in step 3 of Algorithm 2 is equal to $M_{s,t}$. So by Lemma 17, for any fixed pair (s,t) of vertices we have

$$\text{rank } F_{s,t} = \begin{cases} \min(k+1, \nu(s,t)+1) & \text{if } (s,t) \text{ is an edge} \\ \min(k+1, \nu(s,t)) & \text{otherwise.} \end{cases} \tag{20}$$

with probability at least $1 - 5/n^3$. Then by a union bound over all pairs of vertices (s,t), we see that Eq. (20) holds for all pairs (s,t), with probability at least $1 - 5/n$.

Assume this event occurs. Then if (s,t) is an edge, by Eq. (20) we correctly return

$$(\text{rank } F_{s,t}) - 1 = \min(k+1, \nu(s,t)+1) - 1 = \min(k, \nu(s,t))$$

as our answer for this pair.

Similarly, if (s,t) is not an edge, by Eq. (20) we correctly return

$$\min(k, \text{rank } F_{s,t}) = \min(k, k+1, \nu(s,t)) = \min(k, \nu(s,t))$$

as our answer for this pair. This proves the desired result. □

With Theorem 18 established, we can prove our result for vertex connectivities.

Theorem 5 *For any positive integer k, k-APVC can be solved in $\tilde{O}(k^2 n^\omega)$ time.*

Proof By Theorem 18, Algorithm 2 correctly solves the k-APVC problem. We now argue that Algorithm 2 can be implemented to run in $\tilde{O}(k^2 n^\omega)$ time.

In step 1 of Algorithm 2, we need to compute $(I - K)^{-1}$. Since K is an $n \times n$ matrix, by Proposition 7 we can complete this step in $\tilde{O}(n^\omega)$ time.

In step 2 of Algorithm 2, we need to compute D_{ij} for each pair $(i, j) \in [k+1]^2$. For each fixed pair (i, j), the D_{ij} matrix is defined as a product of five $n \times n$ matrices whose entries we know, so this step takes $\tilde{O}(k^2 n^\omega)$ time overall.

In step 3 of Algorithm 2, we need to construct each F_{st} matrix, and compute its rank. Since each F_{st} matrix has dimensions $(k+1) \times (k+1)$ and its entries can be filled in simply by reading entries of the D_{ij} matrices we have already computed, by Propostion 8 this step can be completed in $\tilde{O}(k^\omega n^2)$ time.

By adding up the runtimes for each of the steps and noting that $k \leq n$, we see that Algorithm 2 solves k-APVC in $\tilde{O}(k^2 n^\omega)$ time, as claimed. □

8 Conclusion

In this paper, we presented algorithms solving k-APC and k-APVC in $\tilde{O}((kn)^\omega)$ and $\tilde{O}(k^2 n^\omega)$ time respectively. Many open problems remain concerning the exact time complexity of these problems. We highlight some open questions we find particularly interesting:

1. The most relevant question to our work: can we solve k-APC or k-APVC faster? Is it possible to solve k-APC in $\tilde{O}(k^2 n^\omega)$ time, as fast as our algorithm for k-APVC? Could there be some moderately large parameter values $k \geq n^{\Omega(1)}$ for which these problems can be solved in $\tilde{O}(n^\omega)$ time, matching the runtime for constant k?
2. Can we get better conditional lower bounds for k-APC and k-APVC? Currently, no conditional lower bound rules out the possibility that these problems could, for example, be solved in $\tilde{O}(kn^\omega)$ time. For the APC and APVC problems, can the known $n^{\omega(1,2,1)-o(1)}$ conditional lower bounds be improved[8] to $n^{4-o(1)}$ conditional lower bounds?
3. Recently, [14] showed that there is a nondeterministic verifier for the APVC problem, running in $O(n^{\omega(1,2,1)})$ time. Is there a nondeterministic verifier for APC with the same runtime? Are there nondeterministic verifiers for the k-APC and k-APVC problems, which run faster than the algorithms from Theorems 4 and 5?

Acknowledgements The first author thanks Virginia Vassilevska Williams for insightful discussions on algorithms for computing matrix rank.

Funding Open Access funding provided by the MIT Libraries.

Declarations

Conflict of interest The authors declare that we have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

[8] There is some evidence that better lower bounds may be difficult to establish [14].

A Conjectures in Fine-Grained Complexity

In the Boolean Matrix Multiplication (BMM) problem, we are given $n \times n$ matrices A and B with entries in $\{0, 1\}$, and are tasked with computing, for each pair $(i, j) \in [n]^2$, whether there exists an index k such that $A[i, k] = B[k, j] = 1$. Using matrix multiplication, we can solve BMM in $O(n^\omega)$ time.

The BMM hypothesis[9] posits that this is essentially optimal, and asserts that there is no constant $\delta > 0$ such that BMM can be solved in $O(n^{\omega-\delta})$ time.

Let k be a positive integer. In the kSAT problem, we are given a k-CNF (a Boolean formula which can be written as a conjunction of clauses, where each clause is the disjunction of at most k Boolean literals) over n variables, and tasked with determining if there is some assignment of values to the variables which satisfies the k-CNF.

The Strong Exponential Time Hypothesis (SETH) conjectures that for any constant $\delta > 0$, there exists some positive integer k such that kSAT cannot be solved in $2^{(1-\delta)n}\,\mathrm{poly}(n)$ time.

In the 4-Clique problem, we are given a graph G, and tasked with determining if it contains a clique on four vertices (i.e., four distinct vertices which are mutually adjacent).

Let $\omega(1, 2, 1)$ be the smallest positive real such that we can multiply an $n \times n^2$ matrix with an $n^2 \times n$ matrix in $n^{\omega(1,2,1)+o(1)}$ time. It is known that 4-Clique can be solved in $O\left(n^{\omega(1,2,1)}\right)$ time. The 4-Clique Conjecture[10] asserts that this runtime is essentially optimal, in the sense that for any constant $\delta > 0$, the 4-Clique problem cannot be solved in $O\left(n^{\omega(1,2,1)-\delta}\right)$ time.

B Vertex Connectivity Encoding Scheme

In Appendix B.1, we describe the result that [1] obtains for computing vertex cuts, and explain how the claims in [1] differ from the arguments in this work. In Appendix B.2, we present a proof of Proposition 15.

B.1 Discussion of Previous Vertex Cuts Algorithm

In [1], the authors present an $\tilde{O}((kn)^\omega)$ time algorithm for the k-Bounded All-Pairs Minimum Vertex Cut problem. In this problem, we are given a directed graph G on n vertices, and are tasked with returning, for every pair of vertices (s, t), the size of a

[9] The literature also refers to the Combinatorial BMM hypothesis, an informal conjecture that no "combinatorial" algorithm for BMM runs in $O(n^{3-\delta})$ time, for any constant $\delta > 0$.

[10] This conjecture also has informal counterpart in the literature, which states that no "combinatorial" algorithm for 4-Clique runs in $O(n^{4-\delta})$ time, for any constant $\delta > 0$.

minimum (s, t)-vertex cut, if this size is less than k. For pairs (s, t) where the size of a minimum (s, t)-vertex cut is at least k, we simply need to return the value k.

When (s, t) is not an edge, Menger's theorem implies that the size of a minimum (s, t)-vertex cut is equal to the maximum number of internally vertex-disjoint paths from s to t. So, for such pairs (s, t), the k-Bounded All-Pairs Minimum Vertex Cut problem simply requires we compute the value of $\min(k, \nu(s, t))$, as in the k-APVC problem.

However, when (s, t) is an edge, as discussed in Sect. 2, no (s, t)-vertex cut can exist. This is because no matter which vertices outside of s and t we delete, the resulting graph will always have a path of length one from s to t.

In this case, it may be reasonable to define the "size of a minimum (s, t)-vertex cut" to be ∞. With this convention, for pairs of vertices (s, t) which are edges, we simply need to return the value of k in the k-Bounded All-Pairs Minimum Vertex Cut problem. This is precisely what the algorithm of [1] does.

In more detail, the argument sketched in [1, Proof of Lemma 5.1] argues that

$$\text{rank } (I - K)^{-1}[V_{\text{out}}(s), V_{\text{in}}(t)] \tag{21}$$

is equal to the size of a minimum (s, t)-vertex cut, for all pairs of vertices (s, t) which are not edges, with high probability (where K is defined as in Sect. 3.3).

The algorithm from [1, Proof of Theorem 5.2] first modifies the graph, and then computes the value of Eq. (21) for every pair of original vertices (s, t) with respect to the new graph, to compute the answers to the k-Bounded All-Pairs Minimum Vertex Cut problem.

As described in Sect. 3.3, the graph is transformed by introducing for each vertex s a set S of k new nodes, adding edges from s to each node in S, adding edges from all nodes in S to all out-neighbors of s in the original graph, and erasing all edges from s to its original out-neighbors. Similarly, the transformation also introduces for each vertex t a set T of k new nodes, adds edges from all in-neighbors of t to the nodes in T, adds edges from all nodes in T to t, and erases all edges entering t from its original in-neighbors.

If (s, t) was not an edge in the original graph, then (s, t) is still not an edge in the new graph. In this scenario, the vertex connectivity from s to t in the new graph turns out to be $\min(k, \nu(s, t))$. Since (s, t) is not an edge, the value $\nu(s, t)$ coincides with the size of a minimum (s, t)-vertex cut. So in this case, returning the value of Eq. (21) produces the correct answer for the k-Bounded All-Pairs Minimum Vertex Cut problem.

Suppose now that (s, t) is an edge in the original graph. Then (s, t) will not be an edge in the new graph, since the new out-neighbors of s are all distinct from the new in-neighbors of t. However, the transformation ensures that in the new graph there are k internally vertex-disjoint paths from s to t, because each of the k new out-neighbors of s has an edge to each of the k new in-neighbors of t. Then in this case, returning the value of Eq. (21) just amounts to returning the value k.

So the algorithm of [1, Proof of Theorem 5.2] produces the correct answer for the k-Bounded All-Pairs Minimum Vertex Cut problem when (s, t) is an edge, using the convention that the size of a minimum (s, t)-vertex cut is ∞ when (s, t) is an edge.

In summary: the algorithm from [1] runs in $\tilde{O}((kn)^{\omega})$ time and computes $\min(k, \nu(s,t))$ for all pairs of vertices (s,t) such that (s,t) is not an edge, but for pairs where (s,t) is an edge, does not return any information about the value of $\nu(s,t)$.

Solving k-APVC provides strictly more information than solving k-Bounded All-Pairs Minimum Vertex Cut, and for that reason appears to be the more meaningful analogue of k-APC for vertex connectivity, hence our interest in the former rather than the latter problem. Although vertex connectivity has been defined both in terms of vertex cuts and vertex disjoint paths in the literature, the most recent papers in the area concerning algorithms for vertex connectivity (for example, [10, 13]) generally define $\nu(s,t)$ in terms of the number of internally vertex-disjoint paths from s to t, and our definition of k-APVC is consistent with that choice.

Moving From Vertex Cuts to Vertex Connectivities The quantity in Eq. (21) differs from the expression we use Proposition 15, where we index the rows and columns by $V_{\text{out}}[s]$ and $V_{\text{in}}[t]$, instead of $V_{\text{out}}(s)$ and $V_{\text{in}}(t)$ respectively. For the purpose of solving k-APVC, we need to work with a different submatrix from the one used in Eq. (21), because when (s,t) is an edge, the expression in Eq. (21) is not necessarily equal to $\nu(s,t)$.

For example, suppose G is a graph where $\deg_{\text{in}}(s) = 0$, $\deg_{\text{out}}(t) = 1$, there is an edge from s to t, and $\nu(s,t) = 1$. Then the proof sketch in [1, Proof of Lemma 5.1] (using the terminology of Sect. 3.3) suggests pumping unit vectors to nodes in $V_{\text{out}}(s)$, using these initial vectors to determine flow vectors for all nodes, and then computing the rank of the flow vectors assigned to nodes in $V_{\text{in}}(t)$. In this example, since s has indegree zero and no initial vector is pumped to it, it is assigned the flow vector $\vec{s} = \vec{0}$. Since $V_{\text{in}}(t) = \{s\}$ in this example, the rank of flow vectors in $V_{\text{in}}(t)$ is just zero, even though $\nu(s,t) = 1$.

This issue arises more generally in the proof suggested by [1, Proof of Lemma 5.1] whenever (s,t) is an edge of the graph. Intuitively, this is because a maximum collection of internally vertex-disjoint paths from s to t will always include a path consisting of a single edge from s to t, but if we do not pump out a vector at s, this path will not contribute to the rank of the flow vectors entering t.

To overcome this issue and correctly compute for vertex connectivities for all pairs, we modify the expression from Eq. (21) appropriately (by pumping out an initial vector at the source s, and allowing the flow vector assigned to t to contribute to teh rank), which is why our statement of Proposition 15 involves computing the rank of a different submatrix.

The issue described above does not appear when dealing with edge connectivity, essentially because in that case there is always a set of edges whose removal destroys all s to t paths.

B.2 Proof of Vertex Connectivity Encoding

In this section, we provide a proof of Proposition 15, whose statement we recall below.

Proposition 15 *For any vertices s and t in G, with probability at least $1 - 3/n^3$, the matrix $(I - K)$ is invertible and we have*

$$\text{rank } (I-K)^{-1}[V_{\text{out}}[s], V_{\text{in}}[t]] = \begin{cases} v(s,t)+1 & \text{if } (s,t) \text{ is an edge} \\ v(s,t) & \text{otherwise.} \end{cases}$$

Our proof of Proposition 15 is similar to the outline in [1, Section 5] and is based off ideas from [6, Section 2], but involves several modifications needed to address issues which arise when dealing with vertex connectivities.

Fix a source node s and target node t in the input graph G. Write $v_0 = s$. Suppose s has outdegree $d = \deg_{\text{out}}(s)$. Let $v_1, \ldots, v_d$ be the out-neighbors of s from $V_{\text{out}}(s)$.

Take a prime $p = \Theta(n^5)$ (this is the same prime p from Sect. 6). Let $\vec{u}_0, \ldots, \vec{u}_d$ denote distinct unit vectors in $\mathbb{F}_p^{d+1}$.

Eventually, we will assign each vertex v in G a vector $\vec{v} \in \mathbb{F}_p^{d+1}$, which we call a *flow vector*. These flow vectors are determined by a system of vector equations. To describe these equations, we first need to introduce some symbolic matrices.

Let Z be an $n \times n$ matrix, whose rows and columns are indexed by vertices in G.

If (u, v) is an edge, we set $Z[u, v] = z_{uv}$ to be an indeterminate, and otherwise $Z[u, v] = 0$.

Consider the following procedure. We assign independent, uniform random values from the field $\mathbb{F}_p$ to each variable z_{uv}. Let K be the $n \times n$ matrix resulting from this assignment to the variables in Z (this agrees with the definition of K in Sect. 6).

Now, to each vertex v, we assign a flow vector $\vec{v}$, satisfying the following equalities:

1. Recall that $V_{\text{out}}[s] = \{v_0, \ldots, v_d\}$. For each vertex v_i, we require its flow vector satisfy

$$\vec{v}_i = \left(\sum_{u \in V_{\text{in}}(v_i)} \vec{u} \cdot K[u, v_i] \right) + \vec{u}_i. \tag{22}$$

2. For each vertex $v \notin V_{\text{out}}[s]$, we require its flow vector satisfy

$$\vec{v} = \sum_{u \in V_{\text{in}}(v)} \vec{u} \cdot K[u, v]. \tag{23}$$

It is not obvious that such flow vectors exist, but we show below that they do (with high probability over the random assignment to the z_{uv} variables). Let H_s denote the $(d+1) \times n$ matrix whose columns are indexed by vertices in G, such that the column associated with v_i is $\vec{v}_i$ for each index $0 \le i \le d$, and the rest of the columns are zero vectors. Let F be the $(d+1) \times n$ matrix, with columns indexed by vertices in G, such that the column associated with a vertex v is the corresponding flow vector $\vec{v}$.

Then Eq. (22)and (23) are captured by the matrix equation

$$F = FK + H_s. \tag{24}$$

We will prove that flow vectors $\vec{v}$ satisfying the above conditions exist, by showing that we can solve for F in Eq. (24). To do this, we use the following lemma.

Lemma 20 *We have* $\det(I - K) \neq 0$*, with probability at least* $1 - 1/n^3$.

Proof Each entry of Z is a polynomial of degree at most one with constant term zero. Since the input graph has no self-loops, the diagonal entries of Z are all zeros.

Thus, $\det(I - Z)$ is a polynomial of degree at most n and constant term 1 (which means this polynomial is nonzero). So by the Schwartz-Zippel Lemma (Proposition 9), $\det(I - K)$ is nonzero with probability at least $1 - n/p \geq 1/n^4$ (by taking $p \geq n^5$) as claimed. □

Suppose from now on that $\det(I - K) \neq 0$ (by Lemma 20, this occurs with high probability). In this case, the flow vectors are well-defined, and by Eq. (24) occur as columns of

$$F = H_s(I - K)^{-1} = \frac{H_s\,(\mathrm{adj}(I - K))}{\det(I - K)}. \tag{25}$$

Lemma 21 *For any vertex* t *in* G*, with probability at least* $1 - 1/n^3$*, we have*

$$\operatorname{rank} F[*, V_{\text{in}}[t]] \leq \begin{cases} \nu(s,t) + 1 & \text{if } (s,t) \text{ is an edge} \\ \nu(s,t) & \text{otherwise.} \end{cases}$$

Proof Abbreviate $\nu = \nu(s,t)$. Conceptually, this proof works by arguing that the flow vectors assigned to all in-neighbors of t are linear combinations of the flow vectors assigned to nodes in a minimum (s,t)-vertex cut of the graph G with edge (s,t) removed.

Let G' be the graph G with edge (s,t) removed (if (s,t) is not an edge, then $G' = G$). Let C' be a minimum (s,t)-vertex cut of G'. This means that C' is a minimum size set of nodes (where $s, t \notin C'$) such that deleting C' from G' produces a graph with no s to t path. After removing the nodes in C' from G', let S be the set of vertices reachable from s, and T be the set of vertices which can reach t.

If (s,t) is an edge, set $C = C' \cup \{s,t\}$. Otherwise, set $C = C'$.

By Menger's theorem, $|C| = \nu$ if (s,t) is not an edge in G, and otherwise $|C| = \nu + 1$. This is because in addition to the edge (s,t), we can find $|C'|$ internally-vertex disjoint paths from s to t, so $\nu(s,t) = |C'| + 1 = |C| - 1$.

Let V' be the set of vertices which are in T, or have an edge to a vertex in T. Then set $K' = K[V', V']$ and $F' = F[*, V']$. Now define the matrix $H' = F' - F'K'$. By definition, the columns of H' are indexed by vertices in V'.

Claim 22 For all $v \in V' \setminus C$, we have $H'[*, v] = \vec{0}$.

Proof Take $v \in V' \setminus C$.

Note that $v \neq s$. Indeed, if (s,t) is an edge, then $s \in C$, so $v \neq s$ is forced since $v \notin C$. If (s,t) is not an edge, then by definition of a vertex cut, s can have no edge to T and s is not in T, which means that $s \notin V'$, again forcing $v \neq s$.

We first handle the case where $v \neq t$.

Case 1: $v \neq t$

In this case, by definition of a vertex cut, $v \in T$. Thus, the in-neighbors of v are all members of V'. By the discussion at the beginning of this proof, $v \neq s$.

We claim that $v \notin V_{\text{out}}(s)$. Indeed, suppose to the contrary that $v \in V_{\text{out}}(s)$. By the case assumption, $v \neq t$. By definition, $v \notin C$ and v has an edge to T. This means there is a path of length two from s to a vertex in T, not using any vertices in C. This contradicts the definition of an (s, t)-vertex cut. So it must be the case that $v \notin V_{\text{out}}(s)$ as claimed.

So in this case, we have shown that for $v \in V' \setminus C$, we have $V_{\text{in}}(v) \subseteq V'$ and $v \notin V_{\text{out}}(s)$. It follows from Eq. (23) and the definitions of F' and K' that

$$(F'K')[*, v] = \sum_{u \in V_{\text{in}}(v) \cap V'} \vec{u} \cdot K'[u, v] = \sum_{u \in V_{\text{in}}(v)} \vec{u} \cdot K[u, v] = F[*, v] = F'[*, v] \tag{26}$$

which proves that

$$H'[*, v] = F'[*, v] - (F'K')[*, v] = \vec{0}$$

as desired. It remains to handle the case where $v \in V' \setminus C$ has $v = t$.

Case 2: $v = t$

In this case, by definition of C, there must be no edge from s to t.

Hence $t \notin V_{\text{out}}(s)$. Of course we have $V_{\text{in}}(t) \subseteq V'$ by definition. So the calculation from Eq. (26) applies to $v = t$ as well, which means that $H'[*, t] = \vec{0}$.

Thus for all $v \in V' \setminus C$, we have $H'[*, v] = \vec{0}$ as desired. □

By Claim 22, H' has at most $|C|$ nonzero columns. Thus, rank $H' \leq |C|$.

Similar reasoning to the proof of Lemma 20, shows that matrix $I - K'$ is invertible, with probability at least $1 - 1/n^3$. So we can solve for F' in the equation $H' = F' - F'K'$ to get

$$F' = H'(I - K')^{-1}.$$

Since rank $H' \leq |C|$, the above equation implies that rank $F' \leq |C|$ as well.

By definition, $V_{\text{in}}[t] \subseteq V'$, so $F[*, V_{\text{in}}[t]]$ is a submatrix of F'. It follows that

$$\text{rank } F[*, V_{\text{in}}[t]] \leq |C|.$$

Since $|C| = \nu + 1$ if (s, t) is an edge, and otherwise $|C| = \nu$, the desired result follows. □

Lemma 23 *For any vertex t in G, with probability at least $1 - 2/n^3$, we have*

$$\text{rank } F[*, V_{\text{in}}[t]] \geq \begin{cases} \nu(s, t) + 1 & \text{if } (s, t) \text{ is an edge} \\ \nu(s, t) & \text{otherwise.} \end{cases}$$

Proof Abbreviate $\nu = \nu(s,t)$. Intuitively, our proof will argue that the presence of internally vertex-disjoint paths from s to t will lead to certain nodes in $V_{\text{in}}[t]$ being assigned linearly independent flow vectors (with high probability), which then implies the desired lower bound.

By definition, G contains ν internally-vertex disjoint paths $P_1, \ldots, P_\nu$ from s to t.

Consider the following assignment to the z_{uv} variables of the matrix Z. For each (u,v), we set $z_{uv} = 1$ if (u,v) is an edge on one of the P_i paths. Otherwise, we set $z_{uv} = 0$. For each vertex v, let $\vec{v}$ denote the flow vector for v with respect to this assignment. To show the desired rank lower bound, it will help to first identify the values of some of these flow vectors.

Let $\vec{u}_0$ denote the unit vector initially pumped out at vertex s (as in Eq. (22)).

Take an arbitrary P_i path.

First, suppose that P_i is a path of length at least two. In this case, let a_i denote the second vertex in P_i, and b_i denote the penultimate vertex in P_i (note that if P_i has length exactly two, then $a_i = b_i$). By definition, $a_i \in V_{\text{out}}(s)$. Let $\vec{u}_i$ be the unit vector initially pumped at node a_i (as in Eq. (22)). Then from our choice of assignment to the z_{uv} variables, by Eq. (22) we have

$$\vec{a}_i = \vec{u}_0 + \vec{u}_i.$$

Then by applying Eq. (23) repeatedly to the vertices on the path from a_i to b_i on P_i, we find that

$$\vec{b_i} = \vec{u}_0 + \vec{u}_i \tag{27}$$

as well. The above equation characterizes the flow vectors for the penultimate vertices of P_i paths of length at least two. It remains to consider the case where P_i has length one. If P_i has length one, it consists of a single edge from s to t. Then by Eq. (22) we have

$$\vec{s} = \vec{u}_0. \tag{28}$$

We are now ready to show that the flow vectors for nodes in $V_{\text{in}}[t]$ together achieve the desired rank lower bound, for this particular assignment of values to the z_{uv} variables.

Claim 24 With respect to the assignment where $z_{uv} = 1$ if (u,v) is an edge in a P_i path, and $z_{uv} = 0$ otherwise, the rank of the flow vectors in $V_{\text{in}}[t]$ is at least $\nu + 1$ if (s,t) is an edge, and at least ν otherwise.

Proof We perform casework on whether (s,t) is an edge or not.

Case 1: (s,t) is not an edge

Suppose that (s,t) is not an edge. Then every path P_i has length at least two. Equation (27) shows that the flow vectors in $V_{\text{in}}(t)$ include $\vec{u}_0 + \vec{u}_1, \vec{u}_0 + \vec{u}_2, \ldots, \vec{u}_0 + \vec{u}_\nu$. Since the $\vec{u}_i$ are distinct unit vectors, these flow vectors have rank at least ν as desired.

Case 2: (s, t) **is an edge**

Suppose instead that (s, t) is an edge. Then one of the paths in our maximum collection of vertex-disjoint paths must be a direct edge from s to t. Without loss of generality, let P_v be this path of length one (so that P_i is a path of length two for all $1 \leq i \leq v-1$).

In this case, $t \in V_{\text{out}}(s)$. Let $\vec{u}_v$ denote the unit vector pumped out at $\vec{t}$.

By substituting Eq.s (27) and (28) into Eq. (22), we get that

$$\vec{t} = \vec{s} + \left(\sum_{i=1}^{v-1} \vec{b}_i\right) + \vec{u}_v = v \cdot \vec{u}_0 + \left(\sum_{i=1}^{v-1} \vec{u}_i\right) + \vec{u}_v.$$

By combining the above equation with Eqs. (27) and (28), we see that flow vectors assigned to nodes in $V_{\text{in}}[t]$, which include $\vec{b}_1, \ldots, \vec{b}_{v-1}, \vec{s}$, and $\vec{t}$, span the space containing distinct unit vectors $\vec{u}_0, \vec{u}_1, \ldots, \vec{u}_v$, and hence have rank at least $v+1$ as desired.

For convenience, in the remainder of this proof, we let $\tilde{v}$ denote $\tilde{v} = v + 1$ if (s, t) is an edge, and set $\tilde{v} = v$ otherwise.

By Claim 24, rank $F[*, V_{\text{in}}(t)] = \tilde{v}$. So, $F[*, V_{\text{in}}(t)]$ contains a full rank $\tilde{v} \times \tilde{v}$ submatrix.

Let F' be such a submatrix.

Now, before assigning values to the z_{uv} variables, each entry of $\text{adj}(I - Z)$ is a polynomial of degree at most n. So by Eq. (25), $\det F'$ is equal to some polynomial P of degree at most $n\tilde{v}$, divided by the polynomial $(\det(I - Z))^{\tilde{v}}$. Note that $\det(I - Z)$ is nonzero polynomial, since it has constant term equal to 1.

By the definition of F', we know that under the assignment of values to the variables from the statement of Claim 24, we have $\det(F') \neq 0$. Since as a polynomial

$$\det(F') = P / \left(\det(I - Z)\right)^{\tilde{v}}, \tag{29}$$

it must be the case that P is a nonzero polynomial (because if P was the zero polynomial, then $\det(F')$ would evaluate to zero under every possible assignment).

By Lemma 20, with probability at least $1 - 1/n^3$, the determinant $\det(I - Z) \neq 0$ will be nonzero under a uniform random assignment to the z_{uv} variables. Assuming this event occurs, by the Schwartz-Zippel Lemma (Proposition 9), a uniform random evaluation to the variables will additionally have $P \neq 0$ with probability at least

$$1 - (2\tilde{v}n)/p \geq 1 - 1/n^3$$

by setting $p \geq 2n^5$.

So by union bound, a uniform random assignment of values from $\mathbb{F}_p$ to the z_{uv} variables will make P and $\det(I - Z)$ nonzero simultaneously with probability at least $1 - 2/n^3$.

If this happens, by Eq. (29), we have $\det(F') \neq 0$. This means a particular $\tilde{v} \times \tilde{v}$ submatrix of $F[*, V_{\text{in}}(t)]$ will be full rank with the claimed probability, which proves the desired result. □

With these lemmas established, we can immediately prove the main result of this section.

Proof of Proposition 15 Fix a pair of distinct vertices (s, t).

Substituting the definition of H_s into Eq. (25) implies that

$$F[*, V_{\text{in}}[t]] = (I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}[t]].$$

By union bound over Lemmas 21 and 23, we see that

$$\text{rank } (I - K)^{-1}[V_{\text{out}}[s], V_{\text{in}}[t]] = \text{rank } F[*, V_{\text{in}}[t]] = \begin{cases} \nu(s,t) + 1 & \text{if } (s,t) \text{ is an edge} \\ \nu(s,t) & \text{otherwise} \end{cases}$$

with probability at least $1 - 3/n^3$. □

References

1. Abboud, A., Georgiadis, L., Italiano, G.F., Krauthgamer, R., Parotsidis, N., Trabelsi, O., Uznański, P., Wolleb-Graf, D.: Faster algorithms for all-pairs bounded min-cuts. In: 46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9–12, 2019, Patras, Greece, vol. 132 of LIPIcs, pp. 7:1–7:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ICALP.2019.7
2. Abboud, A., Krauthgamer, R., Trabelsi, O.: APMF < APSP? Gomory–Hu tree for unweighted graphs in almost-quadratic time. In: 62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7–10, 2022, pp. 1135–1146. IEEE (2021). https://doi.org/10.1109/FOCS52979.2021.00112
3. Alman, J., Williams, V.V.: A refined laser method and faster matrix multiplication. In: Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10–13, 2021, pp. 522–539. SIAM (2021). https://doi.org/10.1137/1.9781611976465.32
4. Chen, L., Kyng, R., Liu, Y.P., Peng, R., Gutenberg, M.P., Sachdeva, S.: Maximum flow and minimum-cost flow in almost-linear time. In: 2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS), pp. 612–623 (2022). https://doi.org/10.1109/FOCS54457.2022.00064
5. Cheung, H.Y., Kwok, T.C., Lau, L.C.: Fast matrix rank algorithms and applications. J. ACM **60**(5), 1–25 (2013). https://doi.org/10.1145/2528404
6. Cheung, H.Y., Lau, L.C., Leung, K.M.: Graph connectivities, network coding, and expander graphs. SIAM J. Comput. **42**(3), 733–751 (2013). https://doi.org/10.1137/110844970
7. Fischer, M.J., Meyer, A.R.: Boolean matrix multiplication and transitive closure. In: 12th Annual Symposium on Switching and Automata Theory (SWAT 1971) IEEE (1971). https://doi.org/10.1109/swat.1971.4
8. Gall, F., Urrutia, F.: Improved rectangular matrix multiplication using powers of the Coppersmith-Winograd tensor. In Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '18, pp. 1029–1046. Society for Industrial and Applied Mathematics, USA (2018)
9. Georgiadis, L., Graf, D., Italiano, G.F., Parotsidis, N., Uznański, P.: All-pairs 2-reachability in $O(n^{\omega} log n)$ time. In: 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017), volume 80 of Leibniz International Proceedings in Informatics (LIPIcs), pp. 74:1–74:14. Dagstuhl, Germany (2017). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. http://drops.dagstuhl.de/opus/volltexte/2017/7451. https://doi.org/10.4230/LIPIcs.ICALP.2017.74
10. Huang, Z., Long, Y., Saranurak, T., Wang, B.: Tight Conditional Lower Bounds for Vertex Connectivity Problems (2022). https://doi.org/10.48550/ARXIV.2212.00359
11. Krauthgamer, R., Trabelsi, O.: Conditional lower bounds for all-pairs max-flow. ACM Trans. Algorithms (2018). https://doi.org/10.1145/3212510
12. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, Cambridge (1995). https://doi.org/10.1017/cbo9780511814075

13. Pettie, S., Saranurak, T., Yin, L.: Optimal Vertex Connectivity Oracles (2022). arXiv:2201.00408
14. Trabelsi, O.: (Almost) Ruling Out SETH Lower Bounds for All-Pairs Max-Flow (2023). https://doi.org/10.48550/ARXIV.2304.04667
15. Wu, X., Zhang, C.: Efficient algorithm for computing all low *s*-*t* edge connectivities in directed graphs. In: Mathematical Foundations of Computer Science 2015, pp. 577–588. Springer Berlin Heidelberg (2015). https://doi.org/10.1007/978-3-662-48054-0_48