An Efficient Algorithm for All-Pairs Bounded Edge Connectivity

Shyan Akmal
MIT EECS and CSAIL, Cambridge, MA, USA

Ce Jin
MIT EECS and CSAIL, Cambridge, MA, USA

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). Over undirected graphs, it is known that APC can be solved in essentially optimal $n^{2+o(1)}$ time. 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., ICALP 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 popular conjectures from fine-grained complexity. Previously, this runtime was only achieved for general directed graphs when $k \leq 2$ [Georgiadis et al., ICALP 2017]. Our result employs the same algebraic framework used in previous work, introduced by [Cheung, Lau, and Leung, 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 for every pair of vertices (s,t), we are now tasked with reporting 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 this number is at least k.

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, ICALP 2019]. In comparison, our algorithm solves the full k-APVC problem, and is faster if $\omega > 2$.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases maximum flow, all-pairs, connectivity, matrix rank

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.11

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2305.02132 [3]

Funding *Shyan Akmal*: Supported in part by NSF grants CCF-2129139 and CCF-2127597.
Ce Jin: Partially supported by NSF Grant CCF-2129139 and a Siebel Scholarship.

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

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 [5].

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 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 Section 1.2) suggesting it is unlikely this problem can be solved in quadratic time. Naively computing the connectivity separately for each pair yields an $n^2m^{1+o(1)}$ time algorithm for this problem. Using the flow vector framework (discussed in Section 3), it is possible to solve APC in directed graphs in $\tilde{O}(m^\omega)$ time[2] [7], where ω is the exponent of matrix multiplication. Known algorithms imply that $\omega < 2.37286$ [4], 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 [8]. 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 [10]. As we discuss in Section 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.

---

[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.

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

▶ **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 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 [7], 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 can be found in...

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 $k = 1$ [8]. 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 [12, 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 [12, 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 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]. This conjecture also implies that solving APVC even in undirected graphs requires $n^{\omega(1,2,1)-o(1)}$ time [11].

Algorithms for Related Problems

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, [7, Theorem 1.5] proves that APC can be solved in $O\left(d^{\omega-2} n^{\omega/2+1}\right)$ time.

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 Section 2 we introduce notation, some useful definitions, and results on matrix computation which will be useful in proving correctness of our algorithms. In Section 3 we provide an intuitive overview of our algorithms for k-APC and k-APVC. In Section 4 we describe a framework of "flow vectors" for capturing connectivity values, and in Section 5 use this framework to prove Theorem 4. In Section 6 we present helpful results about vertex-connectivity, and in Section 7 use these results to prove Theorem 5.

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.

[3] Known fast matrix multiplication algorithms imply that $\omega(1,2,1) < 3.25669$ [9, Table 2].

In Sections 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 Sections 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

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

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

▶ **Proposition 9** (Schwartz-Zippel Lemma [13, 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 [7] 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 [7] 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

[4] For the APVC problem we employ a different, but analogous, framework described in Section 3.3.
[5] In our applications, we will pick $\mathbb{F}$ to be a finite field of size $\text{poly}(m)$.

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 [7, 1] 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 Section 3.2 we describe how we employ the flow vector framework to prove Theorem 4. Then in Section 3.3, we describe how we modify these methods to prove Theorem 5.

3.2 All-Pairs Connectivity

Our starting point is the $\tilde{O}(m^\omega)$ time algorithm for APC presented in [7], which uses the flow vector encoding scheme outlined in Section 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 [7] proves that with high probability, for any pair (s, t) of vertices, we have

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

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 $\operatorname{rank} M[E_{\text{out}}(s), E_{\text{in}}(t)]$ as the connectivity from s to t (the **decode** step).

By Equation (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 Section 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 Equation (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 Equation (1) for this choice of K satisfies

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

This equation is just a k-bounded version of Equation (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 $\tilde{O}\left((kn)^\omega\right)$ 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 Section 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 Equation (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 $\tilde{O}\left((kn)^\omega\right)$ 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 Section 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.

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

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

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

[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.

construction increases the number of vertices in the graph 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 Equation (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, Equation (5) implies that

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

So we can compute $\min(k, \nu(s,t))$ from the value of $\operatorname{rank} M_{s,t}$. This idea is similar to the preconditioning method used in algorithms for computing matrix rank efficiently (see [6] 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 Section 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 [7, 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 Equations (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 Equation (7) and the discussion immediately following it, we know that the diagonal entries of the $m \times m$ matrix Z are zero. By Equation (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 Equation (11) to get

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

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*

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

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 S be the set of nodes reachable from s without using an edge in C, and let T be the set of nodes which can reach t without using an edge in C. By definition of an (s, t)-cut, S and T partition the vertices in G.

Now, 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 Equations (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 Equation (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, $\operatorname{rank} H \leq \lambda$. So by the above equation, $\operatorname{rank} F' \leq \lambda$. 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 $\operatorname{rank} F[*, E_{\text{in}}(t)] \leq \lambda$, 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*

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

Proof. Abbreviate $\lambda = \min(k, \lambda(s,t))$. Intuitively, our proof will argue that the presence of edge-disjoint paths from s to t will lead to certain edges in $E_{\text{in}}(t)$ being assigned linearly independent flow vectors (with high probability), which will then imply the desired rank 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 $i \leq \lambda$ and edge $e = (u, v)$, we set variable $x_{i,ev} = 1$ if e is an edge in P_i. Similarly, for each index $i \leq \lambda$ 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_iY_i (whose rows and columns are indexed by edges in the graph) has the property that $(X_iY_i)[e, f] = 1$ if f is the edge following e on path P_i, and all other entries are set to zero.

Then by Equation (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, $\operatorname{rank} F[*, E_{\text{in}}(t)] = \lambda$ 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 $\operatorname{adj}(I - Z)$ is a polynomial of degree at most $2m$. So by Equation (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 $1 - (2\lambda m)/p \geq 1 - 1/m^3$ by setting $p \geq 2m^5$.

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 $\lambda = \operatorname{rank} (I - K)^{-1}[E_{\text{out}}(s), E_{\text{in}}(t)]$. 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 Equation (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 11 and 12, with probability at least $1 - 4/m^3$ the inequalities

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

and

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

simultaneously hold. The desired result follows. ◀

5 Connectivity Algorithm

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

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.

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

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

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

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

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

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

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

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

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 Equation (14) holds for all original vertices s and t with probability at least $1 - 4/(m_{\text{new}})$.

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

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

Using the above equation in Equation (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 Equation (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_iL_j for each pair of indices $(i,j) \in [k]^2$. The matrix R_iL_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

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

As noted in Section 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 Equation (15), we have

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

Thus, to compute the R_iL_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, $km_{\text{new}} \leq k^2n^2$, we can compute all the R_iL_j products, and hence the entire matrix RL, in $\tilde{O}(n^2k^\omega)$ 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 $kn \times n_{\text{new}}$ matrix. By definition, each row of $\tilde{L}$ has a single nonzero entry. Similarly, $\tilde{R}$ is an $n_{\text{new}} \times kn$ matrix, with a single nonzero entry in each column.

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

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

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

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 Equation (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 Section 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 [7, Section 2]. A proof of this result can be found in the full version of this paper [3, 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*

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

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*

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

[7] The result stated here differs from a similar claim used in [1, Section 5]. See the full version of this paper [3, Appendix B.1] for a comparison of these arguments.

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

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

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

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

Set $r = \min(k+1, \operatorname{rank} M)$. 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).

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 Equation (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*

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

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

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

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

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

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

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

if there is an edge from s to t, and

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

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 Section 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

$$D_{ij} = AP_i(I - K)^{-1}Q_jA^\top.$$

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 $(\operatorname{rank} F_{s,t}) - 1$ as the value for $\min(k, \nu(s, t))$. Otherwise, output $\min(k, \operatorname{rank} F_{s,t})$ 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 Equation (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 Equation (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

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

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

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

as our answer for this pair.

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

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

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 Proposition 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. ◀

References

1 Amir Abboud, Loukas Georgiadis, Giuseppe F. Italiano, Robert Krauthgamer, Nikos Parotsidis, Ohad Trabelsi, Przemysław Uznański, and Daniel Wolleb-Graf. 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*, volume 132 of *LIPIcs*, pages 7:1–7:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.7.

2 Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. 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*, pages 1135–1146. IEEE, 2021. doi:10.1109/FOCS52979.2021.00112.

3 Shyan Akmal and Ce Jin. An efficient algorithm for all-pairs bounded edge connectivity, 2023. arXiv:2305.02132.

4 Josh Alman and Virginia Vassilevska Williams. 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*, pages 522–539. SIAM, 2021. doi:10.1137/1.9781611976465.32.

5 Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623, 2022. doi:10.1109/FOCS54457.2022.00064.

6 Ho Yee Cheung, Tsz Chiu Kwok, and Lap Chi Lau. Fast matrix rank algorithms and applications. *Journal of the ACM*, 60(5):1–25, October 2013. doi:10.1145/2528404.

7 Ho Yee Cheung, Lap Chi Lau, and Kai Man Leung. Graph connectivities, network coding, and expander graphs. *SIAM Journal on Computing*, 42(3):733–751, January 2013. doi:10.1137/110844970.

8 M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory (SWAT 1971)*. IEEE, October 1971. doi:10.1109/swat.1971.4.

9 François Le Gall and Florent Urrutia. 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, pages 1029–1046, USA, 2018. Society for Industrial and Applied Mathematics.

10 Loukas Georgiadis, Daniel Graf, Giuseppe F. Italiano, Nikos Parotsidis, and Przemysław Uznański. 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)*, pages 74:1–74:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2017.74.

11 Zhiyi Huang, Yaowei Long, Thatchaphol Saranurak, and Benyu Wang. Tight conditional lower bounds for vertex connectivity problems, 2022. doi:10.48550/arXiv.2212.00359.

12 Robert Krauthgamer and Ohad Trabelsi. Conditional lower bounds for all-pairs max-flow. *ACM Trans. Algorithms*, 14(4), August 2018. doi:10.1145/3212510.

13 Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, August 1995. doi:10.1017/cbo9780511814075.

14 Ohad Trabelsi. (Almost) ruling out SETH lower bounds for all-pairs max-flow, 2023. doi:10.48550/arXiv.2304.04667.

15 Xiaowei Wu and Chenzi Zhang. Efficient algorithm for computing all low s-t edge connectivities in directed graphs. In *Mathematical Foundations of Computer Science 2015*, pages 577–588. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-48054-0_48.