Franck, Max; Yingchareonthawornchai, Sorrachai

Engineering Nearly Linear-Time Algorithms for Small Vertex Connectivity

Published in:
ACM Journal of Experimental Algorithmics

DOI:
10.1145/3564822

Published: 13/12/2022

Document Version
Publisher's PDF, also known as Version of record

Engineering Nearly Linear-time Algorithms for Small Vertex Connectivity

MAX FRANCK and SORRACHAI YINGCHAREONTHAWORNCHAI, Department of Computer Science, Aalto University, Finland

Vertex connectivity is a well-studied concept in graph theory with numerous applications. A graph is k-connected if it remains connected after removing any $k-1$ vertices. The vertex connectivity of a graph is the maximum k such that the graph is k-connected. There is a long history of algorithmic development for efficiently computing vertex connectivity. Recently, two near linear-time algorithms for small k were introduced by Forster et al. [SODA 2020]. Prior to that, the best-known algorithm was one by Henzinger et al. [FOCS 1996] with quadratic running time when k is small.

In this article, we study the practical performance of the algorithms by Forster et al. In addition, we introduce a new heuristic on a key subroutine called local cut detection, which we call degree counting. We prove that the new heuristic improves space-efficiency (which can be good for caching purposes) and allows the subroutine to terminate earlier. According to experimental results on random graphs with planted vertex cuts, random hyperbolic graphs, and real-world graphs with vertex connectivity between 4 and 8, the degree counting heuristic offers a factor of 2–4 speedup over the original non-degree counting version for small graphs and almost 20 times for some graphs with millions of edges. It also outperforms the previous state-of-the-art algorithm by Henzinger et al., even on relatively small graphs.

CCS Concepts: • **Theory of computation → Graph algorithms analysis;**

Additional Key Words and Phrases: Algorithm engineering, algorithmic graph theory, sublinear algorithms

ACM Reference format:
Max Franck and Sorrachai Yingchareonthawornchai. 2022. Engineering Nearly Linear-time Algorithms for Small Vertex Connectivity. *ACM J. Exp. Algor.* 27, 4, Article 4.4 (December 2022), 29 pages.
https://doi.org/10.1145/3564822

1 INTRODUCTION

Given an undirected graph, the *vertex connectivity problem* is to compute the minimum size of a vertex set S such that after removing S, the remaining graph is disconnected or a singleton. Such a vertex-set is called a *minimum vertex cut*. Vertex connectivity is a well-studied concept in graph theory with applications in many fields. For example, for network reliability [15, 23], a minimum vertex-cut has the highest chance to disconnect the network, assuming each node fails

M. Franck and S. Yingchareonthawornchai contributed equally to this research. Algorithm implementations are by Max Franck.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 759557.

Authors' address: M. Franck and S. Yingchareonthawornchai, Department of Computer Science, Aalto University, Espoo, Finland; emails: {max.franck, sorrachai.yingchareonthawornchai}@aalto.fi.

1084-6654/2022/12-ART4.4
https://doi.org/10.1145/3564822

independently with the same probability; in sociology, vertex connectivity of a social network measures social cohesion [33].

There is a long history of algorithmic development for efficiently computing vertex connectivity (see Reference [27] for more elaborated discussion of algorithmic development). Let n and m be the number of vertices and edges, respectively, in the input graph. The time complexity for computing vertex connectivity has been $O(n^2)$ since 1970 [19] when $\kappa = O(1)$, where κ is the vertex connectivity of the graph. This bound was not improved on until very recently, when [11] introduced randomized (Monte Carlo)[1] algorithms to compute vertex connectivity in time $O(m + n\kappa^3 \log^2 n)$ (for undirected graphs). The algorithm follows the framework by Reference [27]. This makes progress toward the conjecture (when κ is a constant) by Aho, Hopcroft and Ullman [1] (Problem 5.30) that there exists a linear time algorithm for computing vertex connectivity. Before that, the state-of-the-art algorithm was due to Reference [17], which runs in time $O(n^2\kappa \log n)$.

In this article, we study the practical performance of the near-linear time algorithms by Reference [11] for small vertex connectivity. We briefly describe their framework and point out the potential improvement of the framework. Reference [27] provides a fast reduction from vertex connectivity to a subroutine called *local vertex-cut detection*. Roughly speaking, the framework deals with two extreme cases: detecting balanced cuts and unbalanced cuts. The balanced cuts can be detected using (multiple calls to) a standard st-max flow algorithm; the unbalanced cuts can be detected using (multiple calls to) local vertex-cut detection. Reference [11] follows the same framework and observe that local vertex-cut detection can be further reduced to another subroutine called *local edge-cut detection* as well as provide fast edge cut detection algorithms that finally prove the near-linear time vertex connectivity algorithm for any constant κ. The full algorithm is discussed in Section 5. From our internal testing, we observe that, overall the framework, the performance bottleneck is on the local edge detection algorithm.

Therefore, our focus is on speeding up the local edge-cut detection algorithm. To define the problem precisely, we first set up notations. Let $G = (V, E)$ be a directed graph. Let $E(S, T)$ be the set of edges from vertex-set S to vertex-set T. For any vertex-set S, let $\text{vol}^{\text{out}}(S) := \sum_{v \in S} \deg^{\text{out}}(v)$ denote the volume of S, which is the total number of edges originating in S. Undirected edges are treated as one directed edge in each direction. We now define the interface of the local edge-cut detection algorithm.

Definition 1.1. An algorithm $\mathcal{A}$ is LocalEC if it takes as input a vertex x of a graph $G = (V, E)$, and two parameters v, k such that $vk = O(|E|)$, and output in the following manner:

- either output a vertex-set S such that $x \in S$ and $|E(S, V \setminus S)| < k$ or,
- the symbol $\perp$ certifying that there is no non-empty vertex-set S such that

$$x \in S, \text{vol}^{\text{out}}(S) \leq v, \text{ and } |E(S, V \setminus S)| < k. \tag{1}$$

The algorithm is allowed to have bounded one-sided error in the following sense. If there is a non-empty vertex-set S satisfying Equation (1), then $\perp$ is returned with probability at most 1/2.

Reference [11] introduced two LocalEC algorithms with the running time $O(vk^2)$. The algorithms are very simple: they use repeated DFS (depth-first search) with different conditions for early termination. We note that this running time is enough to get a near-linear time algorithm for small connectivity using the framework by Reference [27].

Our Results and Contribution. We introduce a heuristic called *degree counting* that is applicable to both variants of LocalEC in Reference [11], which we call Local1+ and Local2+. We prove that the degree counting heuristic version is more space-efficient in terms of *edge-query* complexity and *vertex-query* complexity. Edge-query complexity is defined as the number of distinct edges

[1]With at most $\frac{1}{n^c}$ error rate for any constant c.

Table 1. Comparisons among Various Implementation of LocalEC Algorithms

LocalEC Variants	Time	Edge-query	Vertex-query	Reference
Local1	$O(vk^2)$	$O(vk^2)$	$O(vk^2)$	[11]
Local1+	$O(vk^2)$	$O(vk^2)$	$O(vk)$	This article
Local2	$O(vk^2)$	$O(vk)$	$O(vk)$	[11]
Local2+	$O(vk^2)$	$O(vk)$	$O(v)$	This article

Local1+ denotes Local1 with the degree counting heuristic. Similarly, Local2+ denotes Local2 with the degree counting heuristic.

that the algorithm accesses, and vertex-query complexity is defined as the number of distinct vertices that the algorithm accesses. The results are shown in Table 1. These complexity measures can be relevant in practice. For example, an algorithm with low query complexity may be able to store the accessed data in a smaller cache than an algorithm with high query complexity.

We conducted experiments on three types of undirected graphs: (1) graphs with planted cuts where we have control over size and volume of the cuts, (2) random hyperbolic graphs, and (3) real-world networks. We also conducted experiments in directed graphs with planted cuts. We denote LOCAL1, LOCAL1+, and LOCAL2+ to be the same local-search-based vertex connectivity algorithm [11] (see Section 5 for details) except that the unbalanced part is implemented with different LocalEC algorithms using Local1, Local1+, Local2+, respectively (Section 4). We use Local1 as a baseline for LocalEC algorithms. We denote HRG to be the preflow-push-relabel-based algorithm by Reference [17]. We implement HRG as a baseline, because when k is small (say $k = O(1)$) HRG is the fastest known alternative to References [11, 27]. The implementation details can be found in Appendix B. By running a sparsification algorithm [26], we can assume that the input undirected graph size depends on n and k. The following summarize the key findings of our empirical studies.

(1) **Comparisons to HRG (Section 7).** We compare four vertex connectivity algorithms, namely, HRG, LOCAL1, LOCAL1+, LOCAL2+. For graphs with planted vertex cuts (Sections 7.1 and 7.2), LOCAL1, LOCAL1+, and LOCAL2+ scale with much better than HRG with larger graphs for a fixed κ_G. They start to outperform HRG on graphs as small as $n \leq 500$ (when $\kappa \leq 8$). For random hyperbolic graphs (Section 7.3), HRG performs much better than on the planted cut instances, but is still outperformed by a significant margin. In real-world graphs (Section 7.4), LOCAL1+ and LOCAL2+ are the fastest among the four algorithms with LOCAL2+ being slightly faster than LOCAL1+. We also observe that the performance of all four algorithms is very similar on part of the real-world dataset and graphs with planted cuts with the same size and vertex connectivity.

(2) **Internal Comparisons (Section 8).** We compare three LocalEC algorithms (Local1, Local1+, Local2+). According to the experiments (Figure 9), for any v parameter, Local1+ and Local2+ visit *significantly* fewer edges than Local1. Also, Local2+ visits slightly fewer edges than Local1+ overall. When plugging into full vertex connectivity algorithms, the degree counting heuristics (LOCAL1+ and LOCAL2+) improve the performance over non-degree counting counter part (LOCAL1) by a factor ranging between 2 to almost 20, with greater factors for larger graphs. For some input graphs, depth first search in all three LocalEC algorithms explores significantly more edges per vertex explored than in other graphs, which makes the advantage of Local1+ and Local2+ smaller. Finally, according to CPU sampling (Table 4), the local search is the main bottleneck for the performance of LOCAL1 at roughly at least 90% for large instances. On the other hand, for the degree counting versions (LOCAL1+ and LOCAL2+), the CPU usage of local search part is improved to be almost the same as the other main component (i.e., finding a balanced cut using the Ford-Fulkerson's max-flow algorithm).

Organization. We discuss related work in Section 2, and preliminaries in Section 3. Then, we review two variants of LocalEC algorithms (Local1, Local2) [11], and describe new degree counting heuristic versions (Local1+, Local2+) in Section 4. We review the full vertex connectivity algorithm based on LocalEC in Section 5. Then, we describe setups and datasets for the experiments in Section 6. We discuss experimental results in Section 7 and internal comparisons in Section 8. Finally, we conclude in Section 9.

2 RELATED WORK

Vertex Connectivity Algorithms. We consider a decision version where the problem is to decide if G has a vertex cut of size at most $k-1$ (the general optimisation version to find the minimum cut can be solved with $O(\log \kappa)$ calls to the decision version). We highlight only recent state-of-the-art algorithms. For more elaborated discussion, see Reference [27]. When $k = O(1)$, the fastest known algorithm is by the authors of Reference [11], with running time $O(m + nk^3 \log^2 n)$ for undirected and $O(mk^2 \log^2 n)$ for directed graphs. The algorithm is based on local search approach. For larger k, the fastest known algorithm are based on preflow-push-relabel by Reference [17] with the running time $O(n^2 k \log n)$ for undirected graphs and $O(mn \log n)$ for directed graphs, and based on algebraic techniques by the authors of Reference [22] with the running time $O(n^\omega \log^2 n + k^\omega n \log n)$ for undirected graphs (the same bound is also known in directed graphs by the algorithm in Reference [7]) where ω denotes the matrix multiplication exponent, currently $\omega \leq 2.37286$ [2]. When k is small (say $k = O(1)$) and the graph is sparse, the preflow-push-relabel-based algorithm in Reference [17] is the theoretically fastest alternative to References [11, 27] among combinatorial algorithms. Therefore, we implement the preflow-push-relabel-based algorithm [17] as a baseline for performance comparisons. We note both all aforementioned algorithms are randomized. Deterministic algorithms are much slower than the randomized ones. The fastest known deterministic algorithms are by Reference [12] for large k and by Reference [13] for $k = O(1)$. Very recently, there is a reduction from undirected vertex connectivity to max-flows with $O(\log^{O(1)}(n))$ factor overhead in running time [21]. Also, there is a recent algorithm that runs in $\tilde{O}(kn^2)$-time[2] in directed graphs [6], which improves on Reference [17] when the graph is sufficiently dense, and has the same complexity when the graph is sparse (up to polylog factors).

Deciding (k, s, t)-vertex Connectivity. We mention another related problem, which is to decide if the there is a vertex cut separating s and t of size at most $k-1$. By a standard reduction [9], it can be solved by st-maximum flow. st-maximum flow can be solved in time $O(mk)$ by augmenting paths algorithm by Ford-Fulkerson algorithm [10]. For larger k, a simple blocking flow algorithm by Reference [8] runs in time $O(m\sqrt{n})$. The current state-of-the art algorithms are $O(m^{4/3+o(1)})$-time algorithm by Reference [24], and $\tilde{O}(m + n^{1.5})$-time algorithm by the authors of Reference [31]. Note that when k is small (e.g., $k = O(1)$), then Ford-Fulkerson algorithm [10] is the fastest, and we thus implement Ford-Fulkerson algorithm as a subroutine to find vertex cut for the balanced case. The algorithms also work for directed graphs with the same running time.

Local Search. There are quite a few local search algorithm with different running times. The first LocalEC algorithm by the authors of Reference [4] is a deterministic algorithm with a running time of $O(vk^k)$. Reference [11] introduced a randomised local search algorithm with improved time $O(vk^2)$. Reference [11] also provide a reduction to the local vertex cut detection problem, which we called LocalVC (similar to Definition 1.1, but uses vertex cut instead of edge cut). Therefore, there is a LocalVC algorithm with running time $O(vk^2)$. This improved the previous bound for LocalVC with running time $O(v^{1.5}k)$ by the authors of Reference [27] when k is small. For our

[2] $\tilde{O}(f(n)) = O(\text{poly}(\log n) f(n))$.

purpose, when k is small (say $k = O(1)$), the algorithm in Reference [11] is the fastest, and thus we consider the LocalEC algorithm in Reference [11].

Implementation and Experimental Studies. To the best of our knowledge, this article is the first experimental study on vertex connectivity algorithms; there were no prior experimental studies on vertex connectivity algorithms.[3] This is in stark contrast to the edge-connectivity problem (which is considered as a sibling problem) where we compute the minimum number of edges to be removed to disconnect the graph. For edge-connectivity, there are many experimental studies [5, 16, 18, 28]. More recently, the work by the authors of Reference [14] implemented the local search framework in Reference [11] to compute directed edge-connectivity.

3 PRELIMINARIES

Let $G = (V, E)$ be a directed graph. In general, we denote $m = |E|$ and $n = |V|$. We denote $E(S, T)$ be the set of edges from vertex-set S to vertex-set T. We say that $S \subset V$ is an xy-vertex cut if x cannot reach y in $G - S$. S is a vertex cut, if it is an xy-vertex cut for some $x, y \in V$. If no vertex cut of size k exists, then the graph is k-(vertex)-connected. Let κ_G be the vertex connectivity of G, i.e., the size of the minimum vertex-cut (or $n - 1$ if no cut exists). Let $\kappa_G(x, y)$ denote the size of the minimum xy-vertex cut in G or $n-1$ if the xy-vertex cut does not exist. If the graph G is obvious from context, then we can omit it from the notation. We say that a triplet (L, S, R) is a *separation triple* if L, S, and R form a partition of V, L, and R are not $\emptyset$ and $E(L, R) = \emptyset$. In this case, S is a vertex-cut in G. The decision problem for vertex connectivity, which we call the k-connectivity problem, is the following: Given $G = (V, E)$, and integer k, decide if G is k-connected, and if not, output a vertex-cut of size $< k$. For the optimisation version, we omit k as input and output a minimum vertex cut.

Sparsification. For an undirected graph $G = (V, E)$, the algorithm by Nagamochi and Ibaraki [26] runs in $O(m)$ time and partitions E into a sequence of forests $E_1, \ldots, E_n$ (possibly $E_i = E_{i+1} = \ldots = E_n = \emptyset$ for some i). For each $k \leq n$, the subgraph $FG_k := (V, \bigcup_{i \leq k} E_i)$ has the property that FG_k is k-connected if and only if G is k-connected. Moreover, any vertex cut of size $< k$ in FG_k is also a vertex cut in G. Clearly, $|E(FG_k)| \leq nk$.

With preprocessing in $O(m)$ time, we can assume that the input graph to the k-connectivity problem is FG_k. In particular, we can assume that the number of edges is $O(nk)$. We can also assume that the minimum degree is at least k (because, otherwise, we can output the neighbor of the vertex with minimum degree).

Split Graph. The split graph construct is a standard reduction from vertex-connectivity-based problems to edge-connectivity-based problems, used in the algorithms featured in this article, among others [9, 11, 17]. Given graph G, we define the split graph SG as follows. For each vertex x in G, we replace x with an "in-vertex" x_{in} and an "out-vertex" x_{out}, with an edge $(x_{\text{in}}, x_{\text{out}})$. We will use this index notation to relate vertices to the corresponding out- and in-vertices in the split graph. For each edge (x, y) in G, we add an edge $(x_{\text{out}}, y_{\text{in}})$ in SG. Any path in G corresponds to a path in SG with an additional edge for every vertex in G that the path passes through. The reduction follows easily from the observation that edge-disjoint paths in SG correspond to vertex-disjoint paths in G.

4 LOCALEC ALGORITHMS AND DEGREE COUNTING HEURISTICS

In this section, we review two variants of LocalEC algorithms in Reference [11], and describe their corresponding new version using the degree counting heuristic. Note that their original versions of Local1 and Local2 in Section 4.1 apply a multiplicative factor of 8 instead of our 2 to ν. The

[3]The experimental work by the authors of Reference [29] mentioned k-vertex connectivity problem. However, in the experiment, they studied only the algorithm for deciding (k, s, t)-vertex connectivity where the source s and sink t are given as inputs.

ALGORITHM 1: AbstractLocalEC$_G(x, v, k)$

```
1 repeat k times
2     Grow a DFS tree T starting from x, stopping early at some point to get y ∈ V(T).
3     If the DFS terminates normally, then return V(T).
4     Reverse all edges along the unique path from x to y in the tree T, unless this is the last iteration.
5 return ⊥.
```

smaller factor is more practical. In addition, Section 5.2 provides mathematical justification. The full vertex connectivity algorithm that applies LocalEC is discussed in Section 5. All the algorithms in this section follow a common framework called AbstractLocalEC as described in Algorithm 1. Let $G = (V, E)$ be the graph that we work on. The algorithm takes as inputs $x \in V$ and two integers v, k. The basic idea is to apply **Depth-first Search (DFS)** on the starting vertex x but force early termination. We repeat for k iterations. If DFS terminates normally at some iteration, i.e., without having to apply the early termination condition, then the set of reachable vertices satisfy Equation (1). Otherwise, we certify that no cut satisfying Equation (1) exists. The only main difference is at line 2, where we need to specify the condition for early termination and selection of the vertex $y \in V(T)$ (where $V(T)$ is the set of vertices of the DFS tree T) in such a way that the entire algorithm outputs correctly with constant probability.

Next, we define time and space complexity (in terms of edges and vertices required to run the algorithm) of a LocalEC algorithm.

Definition 4.1. Let $\mathcal{A}(x, v, k)$ be a LocalEC algorithm. $\mathcal{A}$ has (t, s_e, s_v)-complexity if $\mathcal{A}$ terminates in $O(t)$ time and accesses at most $O(s_e)$ distinct edges, and at most $O(s_v)$ distinct vertices.

4.1 Local1, Local2, and Degree Counting Versions

Algorithm for Local1. On line 2 in Algorithm 1, stop growing the DFS tree when the number of accessed edges is exactly $2vk$. Let E' be the set of accessed edges. We sample an edge $(u, v) \in E'$ uniformly at random and set $y \leftarrow u$. If we choose $\tau \in [1, 2vk]$ uniformly at random to sample the τth edge visited, then we can instead stop the DFS after that edge.

Theorem 4.2 (Theorem A.1 in Reference [11]). *Local1(x, v, k) is LocalEC with (vk^2, vk^2, vk^2)-complexity.*

Next, we present the degree counting version of Local1, which we call Local1+.

Algorithm for Local1+. On line 2 in Algorithm 1, stop growing the DFS tree as soon as $\text{vol}^{\text{out}}(V(T)) \geq \tau$, where τ is chosen uniformly at random from $[1, 2vk]$. On the last iteration, instead set $\tau \leftarrow 2vk$. Finally, we set y to be the last vertex added to the DFS tree. Note that the volume can be calculated as the sum of the degrees of the visited vertices.

Theorem 4.3. *Local1+(x, v, k) is LocalEC with (vk^2, vk^2, vk)-complexity.*

We say that an edge is *new* if it has not been accessed in earlier iterations. Otherwise, it is *old*. It follows that reversed edges are old.

Algorithm for Local2.[4] On line 2 in Algorithm 1, stop the DFS when the number of accessed new edges is exactly $2v$. Let E' be the set of new edges accessed in DFS. We sample an edge $(u, v) \in E'$ uniformly at random and set $y \leftarrow u$. To avoid storing E' for sampling, we can choose $\tau \in [1, 2v]$ uniformly at random and set y to be the τth new edge.

[4]The algorithm Local2 described in this article is similar to Algorithm 1 in Reference [11]. Our description here is simpler, and achieves the same properties as Algorithm 1 in Reference [11].

Theorem 4.4 (Equivalent to Theorem 3.1 in Reference [11]). *Local2(x, v, k) is LocalEC with (vk^2, vk, vk)-complexity.*

Next, we present the degree counting version of Local2, which we call Local2+.

Algorithm for Local2+. For each $v \in V$, let $c(v)$ be the remaining capacity for v, representing the number of uncounted outgoing edges. At the start of LocalEC, set $c(v) = \deg^{\text{out}}(v)$. On line 2 in Algorithm 1, stop growing the DFS tree T as soon as $\sum_{v \in V(T)} c(v) \geq 2v$ and update $c(v)$ for each visited vertex as follows. For the last vertex visited u, set $c(u) \leftarrow (\sum_{v \in T} c(v)) - 2v$. For other vertices $v \in T$, set $c(v) \leftarrow 0$.

To select y, choose $\tau \in [1, 2v]$ uniformly at random before DFS. Let T' be the DFS tree at the earliest point where $\sum_{v \in V(T')} c(v) \geq \tau$ (before updating capacities). At this point, we set y to be the latest vertex added to T'.

Intuitively, we count previously uncounted outgoing edges and choose the origin vertex for one of them at random.

Theorem 4.5. *Local2+(x, v, k) is LocalEC with (vk^2, vk, v)-complexity.*

4.2 Optimisations for Local1, Local1+, Local2, and Local2+

We use a few simple heuristics in all featured LocalEC algorithms that deviate slightly from a literal interpretation of Forster et al. [11]. The original version of Local1 explores a number of edges and then samples one of them at random. As we have already included in our description of Local1, we sample a random number $\tau \in [1, 2vk]$ and stop DFS as soon as it has explored τ edges in Local1 and Local1+. This alone should speed up the DFS process by 50% in expectation. Note that Local2 and Local2+ must mark edges, not only explore them, which means this heuristic is not applicable.

The seed vertex x is always included in any $S \ni x$ (as in Definition 1.1). We can leverage the fact that we know one of the vertices in any such S by subtracting the outdegree of x from v and subsequently not counting/marking any outgoing edges from x. Correctness proofs still apply with the modification that instead of using v as an upper bound for the number of outgoing edges for the set S, we use $v - \deg^{\text{out}}(x)$, or the new value of v, to bound the number of unknown edges. This heuristic provides minor speedups in a few ways: For Local1 and Local1+, we can subtract $\deg(x)$ before multiplying by k, which means that the outgoing edges of x are effectively counted k times each in each iteration. For Local2 and Local2+, we effectively allow ourselves to mark the outgoing edges from x in each iteration. For Local1 and Local2 the heuristic also has an effect similar to using degree counting on the vertex x only.

4.3 Potential Speedup in Practice with Degree Counting (Local1+, Local2+)

Here, we explain the intuition behind the degree counting heuristic and why it can speed up LocalEC in practice. We focus on Local1 and Local1+ (the idea is similar for Local2 and Local2+). At any iteration of Local1 or Local1+, the goal is to locate and count τ edges reachable through the DFS tree. When a new vertex is explored, Local1+ counts all its outgoing edges and makes a lot of progress all at once. The only time Local1 makes progress but Local1+ does not is when it explores a back edge that Local1+ has already counted. At worst Local1+ counts the same edges before Local1. In the best case scenario only few of the counted edges are explicitly explored before DFS terminates.

To illustrate, consider the graph shown in Figure 1. It is a 5-regular graph with most edges and vertices omitted. Let $\tau = 20$. Suppose that DFS starts from x and explores vertices a, b, c, and d in that order at which point Local1+ has counted 20 edges excluding outgoing edges of x. In the best case scenario, this can be done by exploring 4 edges, but there can be up to 6 back edges depending on the order in which edges are considered for DFS. Therefore, we only have to explore between

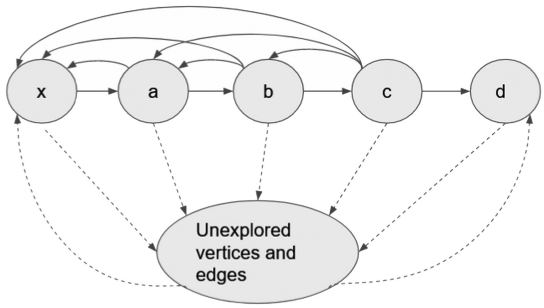

Fig. 1. Depth first search example: 5-regular graph with some edges/vertices omitted.

4 and 10 edges. Local1 would have to explore at least twice as many. Considering cases where DFS needs to backtrack more, the worst case of Local1+ can be much closer to Local1 than in this example, but the difference in best case can also be much greater if there are vertices with high degree.

4.4 Proof of Theorems 4.2 to 4.5

In this section, we address proofs for Theorems 4.2 to 4.5.

Correctness. It can be shown that all four algorithms (Local1, Local1+, Local2, Local2+) are LocalEC through a similar argument as used in Reference [11]. For completeness, we provide the proofs in Appendix A.1.

Complexity. Let $\mathcal{A}$ be an LocalEC algorithm (Definition 1.1), and let v and k be the parameters of the algorithm. We define three measure of complexity $T(\mathcal{A}, G), U_E(\mathcal{A}, G)$, and $U_V(\mathcal{A}, G)$ on input graph G and LocalEC algorithm $\mathcal{A}$ as follows. Let $T(\mathcal{A}, G)$ be the number of times that the algorithm accesses edges on the input graph G. $T(\mathcal{A}, G)$ measures time complexity of the algorithm. Let $U_E(\mathcal{A}, G)$ be the number of unique edges accessed by the algorithm on graph G. This measures how much information (in terms of number of edges) the algorithm needs to run. Let $U_V(\mathcal{A}, G)$ be the number of unique vertices accessed by the algorithm on graph G.

Observation 1. *For any graph G and LocalEC algorithm $\mathcal{A}$, $T(\mathcal{A}, G) \geq U_E(\mathcal{A}, G) \geq U_V(\mathcal{A}, G)$.*

Local1. To see that Local1 has $(O(vk^2), O(vk^2), O(vk^2))$-complexity, it is enough to prove that $T(\text{Local1}, G) = O(vk^2)$. This follows easily, because each iteration we stop the DFS after visiting exactly $2vk$ edges, and there are at most k iterations.

Local1+. We first prove that $T(\text{Local1+}, G) = O(vk^2)$. Since there are k iterations, it is enough to bound one iteration. Let S be the set of vertices visited by the DFS before the step at which it stops early. Clearly, $\text{vol}^{\text{out}}(S) < 2vk$, or we would have stopped earlier. By design, new edges can be only visited within the set $E(S, S)$ or at the last step. Therefore, the number of edges visited is at most $|E(S, S)| + 1 \leq \text{vol}^{\text{out}}(S) + 1 = O(vk)$ per iteration and $O(vk^2)$ in total. We have $T(\text{Local1+}, G) = O(vk^2)$.

We can assume that the minimum degree is at least k (otherwise, we have a trivial degree cut). When paths are reversed, only x will have reduced degree. It follows that $k|S| - k = k(|S| - 1) \leq \text{vol}^{\text{out}}(S) \leq 2vk$, which implies $|S| \leq O(v)$ for each iteration and $U_V(\mathit{Local1+}, G) = O(vk)$ in total.

Local2. We first prove that $U_E(\text{Local2}, G) = O(vk)$. By design, for each iteration, we collect at most $2v$ new edges. Since we repeat for k iterations, we collect at most $2vk$ total new edges. Next, we prove $T(\text{Local2}, G) = O(vk^2)$. Since each edge can be explored once per iteration, we have $T(\text{Local2}, G) \leq kU_E(\text{Local2}, G) = O(vk^2)$.

ALGORITHM 2: k-VertexConnectivity-unbalanced(G, SG, k, a)

for $s = 2, 4, 8, \ldots, a/\delta$ **do**
  Let $v \leftarrow s\delta$.
  repeat $\Theta(\frac{m}{v})$ **times**
    Sample a random edge $(x, x') \in E(G)$
    Run LocalEC$_{SG}(x_{\text{out}}, v, k)$
    If set $L' \subseteq V(SG)$ was returned, then compute the corresponding vertex cut $S \subseteq V(G)$ and set $k \leftarrow |S|$.
return the smallest cut found, or $\perp$ if none found.

ALGORITHM 3: k-VertexConnectivity-balanced(G, SG, k, a)

repeat $\Theta(m/a)$ **times**
  Sample random edges $(x, x'), (y, y') \in E(G)$
  Use Ford-Fulkerson on SG to compute maxflow (up to k) from x_{out} to y_{in}
  If max flow was less than k, then compute the corresponding vertex cut $S \subseteq V(G)$ (well-known reduction, e.g., Reference [9]) and set $k \leftarrow |S|$.
return the smallest cut found, or $\perp$ if none found.

ALGORITHM 4: k-VertexConnectivity(G, k)

(If the graph is undirected, then skip Lines 3, 5, 8, and 10)
Compute the split graph (see Section 3) of G, SG.
Compute G^R and its split graph SG^R.
Check for degree cuts in G (vertices with out-degree less than k).
Check for degree cuts in G^R (vertices with out-degree less than k).
Choose a threshold volume $a = \Theta(m/k)$.
Run k-VertexConnectivity-unbalanced(G, SG, k, a).
Run k-VertexConnectivity-unbalanced(G^R, SG^R, k, a).
Run k-VertexConnectivity-balanced(G, SG, k, a).
Run k-VertexConnectivity-balanced(G^R, SG, k, a). (Note: Not SG^R)
return the smallest cut found on Lines 7 to 10, or $\perp$ if none found.

Local2+. We first prove that $U_E(\text{Local2+}, G) = O(vk)$. If true, then we also have $T_E(\text{Local2+}, G) \leq kU_E(\text{Local2+}, G) = O(vk^2)$. We will never visit an outgoing edge of vertex v unless all its capacity has been exhausted. Therefore the total used capacity (at most k times $2v$) is an upper bound for the number of distinct edges visited. For $U_V(\text{Local2+}, G)$, fix any iteration. Let S be the set of vertices visited by the DFS one step before terminating and $S' \subseteq S$ the subset of S that have not been visited before. Clearly, we have $k|S'| \leq \text{vol}^{\text{out}}(S') = \sum_{v \in S'} c(v) \leq \sum_{v \in S} c(v) < 2v$. The first inequality follows, since the minimum degree is at least k. We visit at most $|S'| + 1 = O(v/k)$ distinct vertices per iteration for a total of $O(v)$ distinct vertices.

5 LOCALEC-BASED VERTEX CONNECTIVITY ALGORITHM

We review the full near-linear time small vertex connectivity algorithm by Forster et al. [11] (using LocalEC algorithms described in Section 4 and Ford-Fulkerson max flow). The decision version (find a min-cut smaller than k) of the algorithm is described by Algorithms 2–4. The parameter δ in Algorithm 2 is the minimum degree of G and is found as a side effect of Lines 4 and 5 of Algorithm 4. Algorithm 4 is closely based on the framework by Nanongkai et al. [27]. It finds a

minimum vertex cut of size less than k or returns $\perp$ to certify that $\kappa \geq k$ with constant probability. During the algorithm we reduce the value of k to match the smallest cut found to avoid unnecessary work, since only smaller cuts can improve the result. This is optional. If we are not provided with an upper bound k, then we can start at $k = 1$ and double it until a cut is found.

We provide a correctness proof for our version in Appendix A.2. It is similar to the original proof by the authors of Reference [27].

5.1 Time Complexity

Since Ford-Fulkerson maxflow runs in $\Theta(mk)$ time, the running time for finding balanced cuts is $\Theta(mk)\Theta(m/(m/k)) = \Theta(mk^2)$. Since LocalEC runs in $O(\nu k^2)$ time, the running time for each of the $\Theta(\log(m/k)) = \Theta(\log n)$ values for the parameter ν is $\Theta(\nu k^2)\Theta(m/\nu) = \Theta(mk^2)$, for a total $\Theta(mk^2 \log n)$. For undirected graphs, we can assume $m = \Theta(nk)$ with $\Theta(m)$ additional preprocessing time due to Nagamochi and Ibaraki [26]. This yields $\Theta(m + k^3 n \log n)$ time complexity for a constant error rate. Square the logfactor for correctness with high probability.

5.2 Volume-Sample Tradeoff and Probability Boosting

The volume multiplier 2 (originally 8 in Reference [11]) used in LocalEC pseudocode is an arbitrarily chosen number. Both running time and the error rate bound scale linearly with the multiplier. We can also reduce the overall error rate by increasing the sample size. Multiplying the multiplier by a factor $p > 1$ will turn error rate bound ϵ to $\frac{\epsilon}{p}$ but repeating p times will give error rate ϵ^p. Clearly, increasing the volume multiplier much beyond the point where a single LocalEC call has error rate 0.5 seems inefficient.

Note that except at the lowest/highest values for ν, doubling/halving the multiplier or sample size will result in the same number of calls at any given post-factor volume. This is because of boosting up lower volume calls that have a larger sample size.

5.3 Implementation Details

We choose the parameter a to be $\frac{m}{3k}$, because some cases where post-multiplier LocalEC volume is very close to m resulted in poor practical performance. With the volume multiplier of 2, LocalEC will attempt to explore at most $\frac{2}{3}m$ edges in any given DFS iteration. We use a sample size of $3k(= \frac{m}{a})$ samples for Ford-Fulkerson and $\lfloor \frac{m}{\nu} \rfloor$ for LocalEC.

The graph implementation used for this article is based on adjacency lists with C++ vectors. When we reverse edges along a path, we save the relevant vector indices to enable us to perform the opposite operations later, in order from the newest reversed path to the oldest. We store information such as flags for vertices visited by DFS and the number of uncounted edges/coins in LOCAL2+ per vertex. To avoid resetting this information for every vertex, we also maintain lists of vertices that have been visited within the most recent DFS or LocalEC call.

6 EXPERIMENTAL SETUP

Four algorithms are compared. **LOCAL1**, **LOCAL1+**, and **LOCAL2+** are implementations based on the algorithm by Forster et al. [11]. The full vertex connectivity algorithm based on LocalEC is described in Section 5. LOCAL1, LOCAL1+, and LOCAL2+ use Local1, Local1+, and Local2+ as their LocalEC algorithm. We will refer to these three algorithms collectively as the LOCAL algorithms. We run a directed version of the algorithm on the directed graphs. **HRG** is our implementation of the randomised unit capacity version of the algorithm by Henzinger, Rao, and Gabow [17]. The implementation details are described in Appendix B. We derive an optimisation version (find a min-cut with no upper bound k as input) for the LOCAL algorithms by running the decision version at $k = 1, 2, 4, 8, \ldots$ until a cut is found, similarly to HRG as described by [17]. The experiments

use the optimisation version unless otherwise specified. The algorithms were implemented with C++17 and compiled with g++. All experiments were run on an Ubuntu computer with i7-7700HQ CPU (2.80 GHz) and 2×8 GB DDR4-2400 RAM.

All algorithms were implemented using parameters that bound theoretical success probability from below by a similar constant. In Appendix A.2, we calculate an upper bound of 75% for the error rate for the LOCAL algorithms. However, observed error rate is in the single digits. For undirected graphs, the sparsification algorithm by Nagamochi and Ibaraki [26] is used for preprocessing before each algorithm. The $O(m)$ partitioning of the edges into disjoint forests is not included in the measured time. Construction of the sparse graphs in $O(nk)$ time is included. The theoretical running time therefore only depends on n and k. Due to sparsification, we report graph size in vertices for undirected graphs. For directed graphs, we use the number of edges.

6.1 Input Graphs

The input graphs feature directed and undirected graphs with planted vertex cuts, undirected random hyperbolic graphs and undirected real-world networks. Each data point is an average of 25 running times. For real-world data, we run the algorithms 25 times on each graph. For generated data, we generate five graphs with the same parameters and run 5 times on each. Experiments on the different algorithms are run on the same set of pregenerated graphs.

6.1.1 Undirected Graphs with Planted Cuts. We use planted cuts for undirected graphs in Section 7.1. A planted vertex cut is a vertex partition (L, S, R) in a graph that is constructed to guarantee that S is the unique minimum vertex cut, such that there is no edge between L and R. The sizes of L, S and R are fully configurable. Note that $n = |L| + |S| + |R|$. We also use an additional parameter $\eta > S$ such that for non-adjacent x, y such that S does not restrict connectivity from x to y, the smallest vertex cut that separates x from y should have size at least η. This guarantees the uniqueness of the minimum vertex cut S.

We now describe the construction of random undirected graphs with planted cuts. We use a randomised version of the sparsification algorithm by Nagamochi and Ibaraki [26], which partitions the edges into forests $E_1, E_2, \ldots$ such that $(x, y) \in E_i$ implies that there is a path between x and y in $E_1, E_2, \ldots, E_{i-1}$. A sparse graph using only the edges $E_1 \cup E_2 \cup \cdots \cup E_\eta$ according to the original algorithm guarantees that if vertices x and y are not separated by a vertex cut smaller than η vertices in the original graph, then neither are they in the sparse graph. This, when applied to a complete graph with edges between L and R removed, fulfills the requirements. For the randomised version, we add random edges to the graph as long as it is possible to add edges to the η first forests. The original proof for the algorithm by Nagamochi and Ibaraki [26] does not apply to this version but in practice, S is a unique minimum vertex cut for all random graphs that were generated for these experiments. We generate undirected graphs with planted cuts using the parameter value $\eta = 64$ (64 edge-maximal forests).

6.1.2 Directed Graphs with Planted Cuts. We use planted cuts for directed graphs in Section 7.2. In directed graphs, we use the same definition for planted vertex cuts as the one for undirected graphs; but, we can (and in fact do) include edges from R to L (but not the other direction). Paths may enter, but not leave, L without passing through a vertex in S. This means that $G \setminus S$ is not a strongly connected graph, but it may be weakly connected. The parameters for the graphs are $|L|$, $|S|$, $|R|$, and η.

We generate directed graphs with planted cuts by creating a deterministic base construct that fulfills the definition of a planted cut. The number of edges in the base construct is approximately ηn. We then add random edges that do not violate the definition (none from L to R, but possibly R

to L) until we have the desired number of edges. The construction and its correctness is discussed in Appendix C.

For all experiments (except for Section 7.2.4, where we scale κ), we have constant $\kappa \in \{4, 8\}$ and observe behavior when we change the size of the graph. For these, we set the generator parameter to be $\eta = \kappa + 1$ and $m = 2\kappa n$, which is the lowest power of two above κ. This means that just under half of the edges are random. These values approximately match the largest sparsified graphs in the undirected graph experiments with corresponding parameters. In Section 7.2.3, we instead fix the number of vertices and change the density (Figure 6); we use $\eta = t(\kappa + 1), m = 2t\kappa n$ for $t \in \{1, 2, 3, 4, 5\}$, which preserves the ratio of random to nonrandom edges.

6.1.3 Random Hyperbolic Graphs and Real-world Data. We generate random hyperbolic graphs using NetworKIT [30], which provides an implementation of the generator by von Looz et al. [32]. The properties of random hyperbolic graphs include a degree distribution that follows a power law and small diameter, which are common in real-world graphs [3]. The graphs are generated with a power law exponent of 5 and an average degree of 32 (for $\kappa = 4$) or 40 (for $\kappa = 8$). Graphs are repeatedly generated until the number of graphs with the desired vertex connectivity is 5 for each data point.

The real-world data is based on three graphs from the SNAP dataset [20], soc-Epinions1, com-LiveJournal, and web-BerkStan. The LiveJournal dataset is originally undirected. The other two are directed graphs read as undirected, which means that we compute weak vertex connectivity for these graphs. We preprocess these graphs by taking the largest connected component for a k-core. A k-core is defined as the edge-maximal subgraph with minimum degree at least k. Only k-cores whose vertex connectivity is over 1 but less than the minimum degree are used. This data was collected by manually searching for k-cores with nontrivial ($\kappa > 1$) vertex connectivity instances. Due to time constraints, these data points are few and do not include directed instances read as directed graphs.

7 EXPERIMENTAL RESULTS

7.1 Planted Cuts in Undirected Graphs

7.1.1 Balanced Versus Unbalanced Cuts. We study the performance of all four algorithms (LOCAL1, LOCAL1+, LOCAL2+, HRG) when we vary the size of L, i.e., the balanceness of the optimal vertex cuts. Our key finding is that every algorithm is faster when the cuts are balanced than when the cuts are unbalanced (with the exception when the cuts are extremely unbalanced for LOCAL1, LOCAL1+, and LOCAL2+). This suggests that the easy instances are those whose optimal cuts are balanced. Next, we discuss in details.

In this experiment, we use graphs where $n = 10{,}000$ and $\kappa = 4$ with different values of $|L|$. For each algorithm, we show the running time for every instance being normalized by the average time over all instances of the same algorithm (as shown in Figure 2(b)). According to Figure 2(b), all four algorithms perform reasonably well both for graphs with unbalanced cuts and those with balanced cuts, although there is some variance in performance. The difference between the highest and lowest running time is a factor of 1.89 for HRG, 1.71 for LOCAL1 and LOCAL1+, and 1.79 for LOCAL2+. Internal testing suggests that the running time of HRG is roughly proportional to $|L|^2 + |R|^2$. The running time for LOCAL1 is the fastest for very balanced cuts, which are found fast by max flow before running LocalEC. For unbalanced cuts mainly found by LocalEC, the running time is faster the more unbalanced the cut. LocalEC is run with gradually increasing ν parameter. LOCAL1+ and LOCAL2+ have similar performance for balanced cuts but instead have stable running time for moderately unbalanced cuts, dropping off fast for very unbalanced cuts. $|L| = 5$, which is used as a fixed parameter in most of our experiments, is close to the average result for all the LOCAL algorithms.

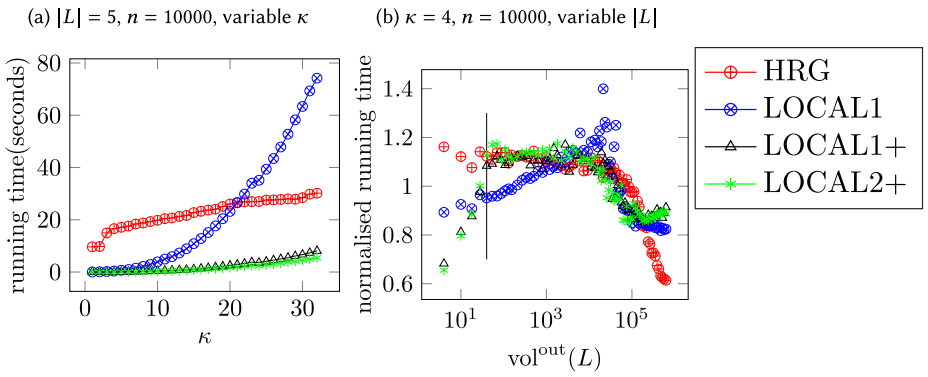

Fig. 2. Undirected Planted Cuts with variable $|L|$ or κ (Decision version with $k = \kappa + 1$). A vertical line marks $|L| = 5$ in Figure 2(a).

7.1.2 Running Time Comparison. We conducted the experiments using the input graphs on vertex connectivity $\kappa \in \{4, 8\}$ up to 16 million edges. Based on discussion above, the difficult (and interesting) instances are the ones with unbalanced cuts. Thus, we use unbalanced cuts in this setting. That is, we set $|L| = 5$ (this corresponds to the vertical line in Figure 2(b)). We found that the LOCAL algorithms outperform the quadratic-time HRG even on very small graphs. At $\kappa = 8$ in Figure 3(d), HRG overtakes LOCAL1 at 250–300 vertices and approximately 30 ms running time. LOCAL1+ and LOCAL2+ are the fastest even for the smallest datapoint. As HRG approaches an hour in running time the LOCAL algorithms are faster by more than an order of magnitude. In Figure 3(e), HRG takes roughly 50 min at $n = 50{,}000$, $\kappa = 8$ and LOCAL1 takes 73 s while LOCAL1+ and LOCAL2+ take fewer than 8 s. As Figures 3(c) and 3(f) show, LOCAL1+ and LOCAL2+ stay well under an hour even as the number of vertices reaches a million. LOCAL1 exceeds an hour for $\kappa = 8$ but not $\kappa = 4$.

LOCAL1+ and LOCAL2+ are faster than LOCAL1 by an increasing margin as we grow the size of the input graph. The speedup relative to LOCAL1 for $\kappa = 4$ and 500, 50,000, and 1,000,000 vertices, respectively (the largest datapoints in Figures 3(a) to 3(c)), are 2.8, 5.7, and 8.6 for LOCAL1+ and 2.6, 5.8, and 9.6 for LOCAL2+. For $\kappa = 8$, the ratios are 1.3–2 times higher with greater additional speedup for larger graphs. The largest observed speedup ratio compared to LOCAL1 in all included experiments is 19.8 for LOCAL2+ on undirected graphs with planted cuts with $\kappa = 8$, $n = 1{,}000{,}000$.

7.1.3 Scalability in k (Decision Version, $k = \kappa + 1$). The theoretical running time for HRG is $O(nm) = O(n^2k)$ and the LOCAL algorithms are approximately cubic in k. Therefore, we should expect HRG to scale better than LOCAL algorithms in terms of k when n is fixed. According to the experiments where the graphs with $n = 10{,}000$, and $|L| = 5$, Figure 2(a) shows that that the running time for HRG indeed grows slower with k. The running time for LOCAL1 exceeds that of HRG at $k = 22$. For $k > 12$, LOCAL1+ settles around being 9–10 times faster than LOCAL1 and LOCAL2+ 14–15 times faster than LOCAL1. For small k the ratios are smaller. Unlike the decision version, the running time of the optimisation version goes up sharply when κ is an even power of two as the decision version has to be run on an additional larger value for k.

7.2 Planted Cuts in Directed Graphs

7.2.1 Balanced Versus Unbalanced Cuts. We study the performance of all four algorithms (LOCAL1, LOCAL1+, LOCAL2+, HRG) when we vary the size of L, i.e., the balanceness of the optimal vertex cuts. As with undirected graphs, we observe that HRG and LOCAL1 are faster for balanced cuts (with the exception of degree cuts for LOCAL1). LOCAL1+ and LOCAL2+ are not

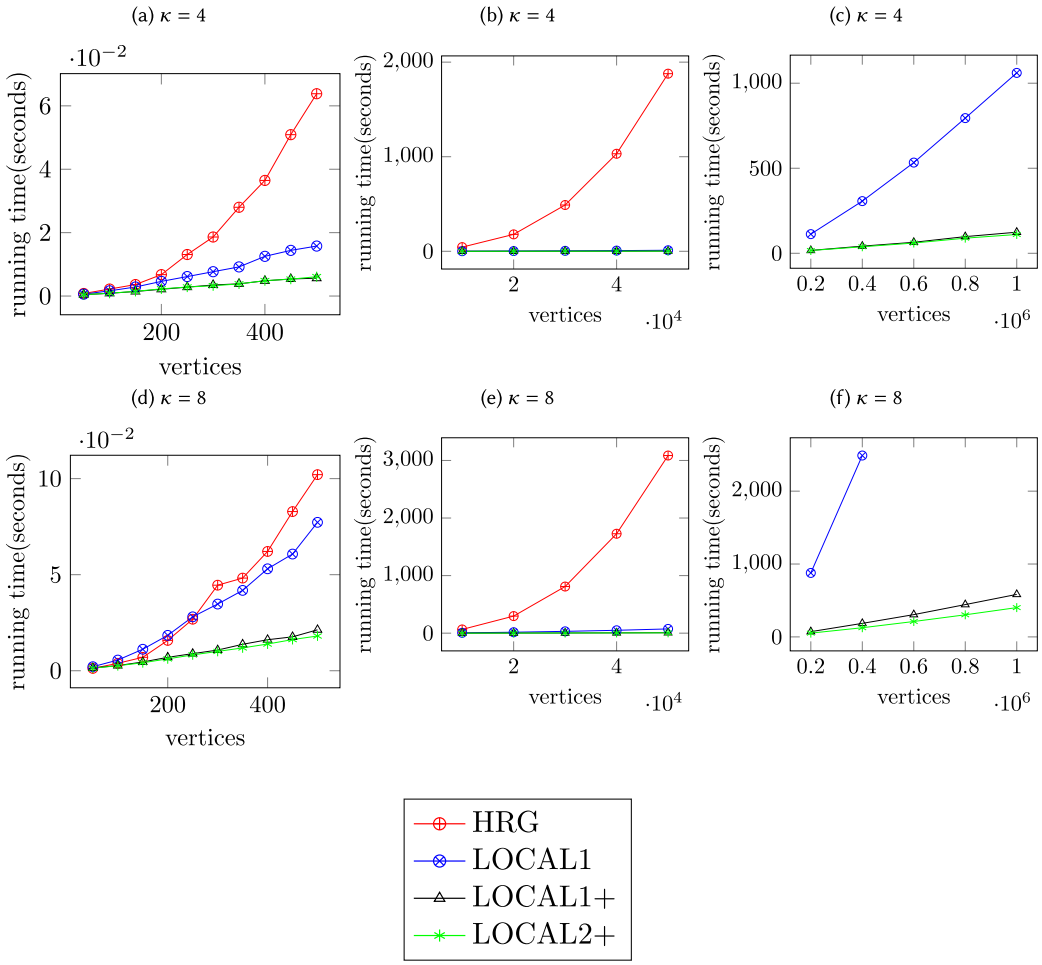

Fig. 3. Undirected Planted Cuts with fixed $|L| = 5$ and κ.

slower for unbalanced cuts so the general trend is still that the fast instances are those whose optimal cuts are balanced. Next, we discuss in details.

In this experiment, we use graphs where $n = 10{,}000$ and $\kappa = 4$ with different values of $|L|$. For each algorithm, we show the running time for every instance being normalized by the average time over all instances of the same algorithm. Figure 4 shows the observed performance for directed graphs with unbalanced and balanced cuts. We show separately the case where a planted vertex cut restricts path into the smaller side of the cut ($|L| > |R|$), here called "in-cuts," or out of the smaller side ($|L| < |R|$), here called "out-cuts." These perform slightly differently, because we look for out-cuts before in-cuts. They are normalised based on the combined dataset. We can see that performance of the LOCAL algorithms is better for unbalanced out-cuts than unbalanced in-cuts. The difference between the highest and lowest running time in Figures 4(a) and 4(b) is a factor of 1.36 for HRG, 1.22 for LOCAL1 and LOCAL1+ and 1.21 for LOCAL2+. Compared to undirected graphs in Figure 2(b), we can see that variance is lower in this experiment for each algorithm.

An interesting discrepancy from the undirected counterpart is that LOCAL1+ and LOCAL2+ are not consistently faster for very balanced cuts (which are searched for first) than unbalanced cuts in

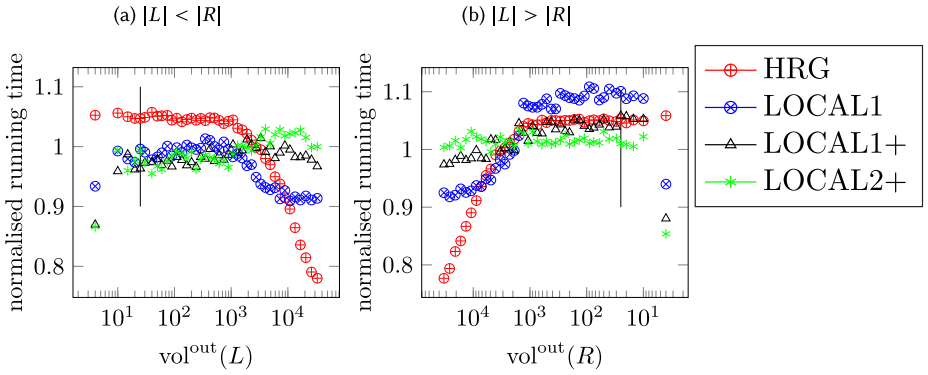

Fig. 4. Directed Planted Cuts with $n = 10{,}000$, $\kappa = 4$, and variable $|L|$, normalised 1 = average. A vertical line marks $|L| = 5$ and $|R| = 5$.

Figure 4. Here is a possible explanation regarding the difference between $\text{vol}^{\text{out}}(L)$ and $\text{vol}^{\text{in}}(L)$. For example, in the "out"-category, the graph may contain edges from any vertex to a vertex in L but an edge that leaves L must go to the planted vertex cut. Due to this restriction, we add about 60% more random edges that enter L than edges than leave L. As a consequence, L has lower average outdegree than the outside, and vertices in it may be unproportionally likely to be explored by LocalEC DFS, which means that they are more likely to have been already explored. It would then be expected that degree counting is somewhat less effective at reducing running time, particularly when a significant portion of the graph has been searched.

7.2.2 Running Time Comparison: Varying Edges, Fixed Density. In this section, we use directed planted cuts with $\kappa \in \{4, 8\}$ and scale the number of edges up to 16 million edges while fixing the density for a given value for κ ($m = 2\kappa n$). Based on the discussion above, the difficult (and interesting) instances are the ones with unbalanced cuts. Thus, we use unbalanced cuts in this setting. That is, we set $|R| = 5$ (this corresponds to the vertical line in Figure 4(b)). We choose this over $|L| = 5$, because it is a slightly slower case for the LOCAL algorithms but not for HRG.

According to Figure 5, the LOCAL algorithms outperform HRG for fairly small instances, although not as small as we observed with undirected graphs. At $\kappa = 8$ in Figure 5(d) LOCAL1 matches the running time of HRG at around 28,800 edges (1,800 vertices) and 1.25 s of running time. LOCAL1+ and LOCAL2+ become faster than HRG at 4,800 edges (300 vertices) and 26–30 ms running time. In Figures 5(b) and 5(e), we see that the LOCAL algorithms are again faster by orders of magnitude when HRG approaches an hour in running time. At 400,000 edges (50,000 vertices) HRG runs in 42 min while LOCAL1 takes 163 s, LOCAL1+ takes 28 s and LOCAL2+ takes 22 s at $\kappa = 8$. In the largest graph going up to a million vertices and 16 million edges, LOCAL1+ and LOCAL2+ still run in under an hour, although LOCAL1 does not.

LOCAL1+ and LOCAL2+ are faster than LOCAL1 and their advantage grows consistently as we add vertices and edges. For example, the speedup factor relative to LOCAL1 for the largest datapoints with $\kappa = 4$ in Figures 5(a) to 5(c) grows as follows: The speedup factor for LOCAL1+ is 2.5 at 500 vertices and 4,000 edges, 3.5 for 5,0000 vertices and 400,000 edges and 4.0 at 1,000,000 vertices and 8,000,000 edges. The corresponding ratios for LOCAL2+ are 2.6, 3.9, and 4.6. LOCAL2+ is faster than LOCAL1+ in the experiments with planted cuts in directed graphs. For $\kappa = 8$ the speedup ratios for directed graphs are 1.4–2 times higher.

7.2.3 Running Time Comparison: Varying Edges, Fixed Vertices. In directed graphs, we can no longer assume that the input graph is sparse, since there is no known linear-time algorithm for

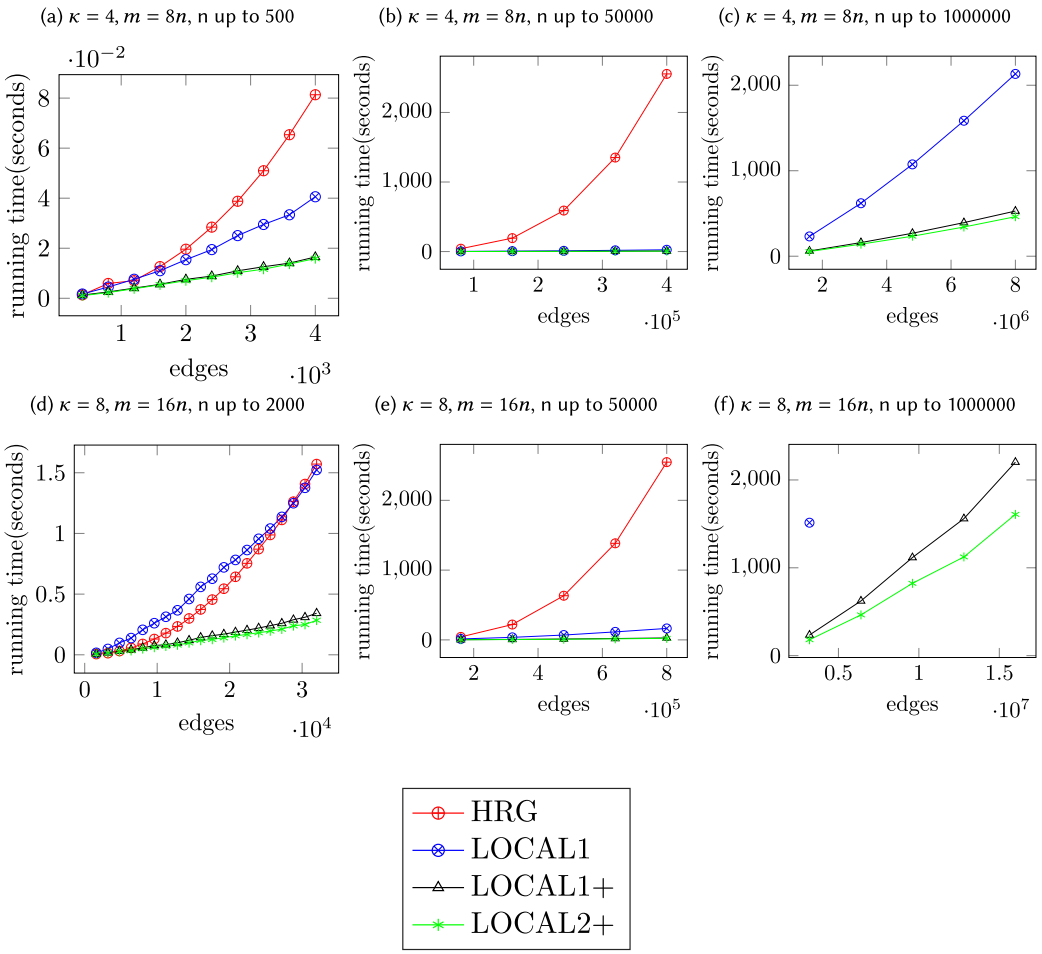

Fig. 5. Directed planted in-cuts with fixed $|R| = 5$ and κ and varying m, n where $m = 2\kappa n$.

sparsifying directed graphs. In this section, we study the running times in Figure 6, where we vary the density of graphs while keeping the number of vertices constant. We achieve this by scaling the generator parameter η and number of added random edges as described in Section 6.1.2. We also compare the performance to corresponding graphs in Figure 5 where the same number of edges is achieved by scaling the number of vertices with fixed density.

LOCAL1 behaves similarly when scaling density or the number of vertices, although it is slightly faster when the same number of edges is achieved by scaling density. LOCAL1 runs for approximately 1500 s at $m = 8{,}000{,}000, n = 200{,}000, \kappa = 4$, which is the largest point in Figure 6(b) but around 2100 s with the same number of edges and 1,000,000 vertices in Figure 5(c).

The running times of LOCAL1+ and LOCAL2+ grow significantly slower when we increase density instead of adding vertices, which is expected considering that the degree counting heuristic benefits from high degree vertices. For the same graphs referenced above, LOCAL1+ and LOCAL2+ take approximately 100 s at $n = 200{,}000$ and 500 s with $n = 1{,}000{,}000$.

Since the running time of LOCAL1+ and LOCAL2+ scales so slowly with density, we can expect greater speedup compared to LOCAL1. At $\kappa = 4, m = 400{,}000, n = 10{,}000$ in Figure 6(a) LOCAL1 is 9.7 times slower than LOCAL1+ and 11.1 times slower than LOCAL2+, which is a significantly

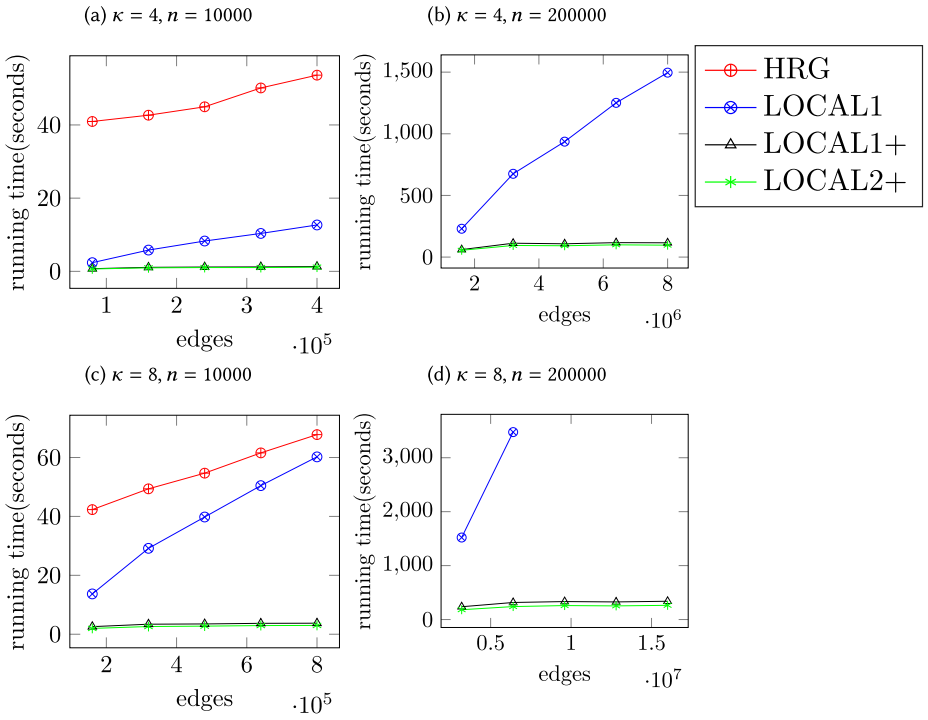

Fig. 6. Directed Planted In-Cuts with fixed $|R| = 5$ and κ. Variable average degree instead of number of vertices.

greater difference than 3.5 and 3.9 at $\kappa = 4, m = 400{,}000, n = 50{,}000$ in Figure 5(b) where density is constant.

For HRG, the theoretical worst-case running time is linear instead of quadratic when we only scale density in a graph. In Figure 6(a), we observe that the scaling appears even slower than one might expect from a linear relation. The running time grows from 41 s at 10,000 vertices and 80,000 edges to 54 s for 5 times as many edges in Figure 6(a). For $n = 50{,}000, m = 400{,}000$ in Figure 5(b) the running time is 2,550 s. Clearly, the number of vertices is very important for the practical running time of HRG. Furthermore, for dense graphs of moderate size, it may be significantly less advantageous to use LOCAL1 over HRG. LOCAL1+ and LOCAL2+ seem more likely to maintain the advantage even for dense graphs.

7.2.4 Scalability in k (Decision Version, $k = \kappa + 1$). In theory, the running time is constant in k for HRG and approximately quadratic for the LOCAL algorithms when m is independent from k (no sparsification). In Figure 7, we fix $n = 10000, m = 64000$ and $\eta = 33$ and vary κ up to $\kappa = 32$ for generating planted cuts in directed graphs. As anticipated, the running time for HRG does not change when κ is variable, and LOCAL algorithms run slower for larger κ. Like for undirected graphs, the running time for LOCAL1 exceeds that of HRG at $k = 16$. For $k > 7$ LOCAL1+ settles around being 11–15 times faster than LOCAL1 and LOCAL2+ 16–19 times faster than LOCAL1. For small k the ratios are smaller. Unlike the decision version, the running time of the optimisation version goes up sharply when κ is an even power of two as the decision version has to be run on an additional larger value for k (with the exception of HRG).

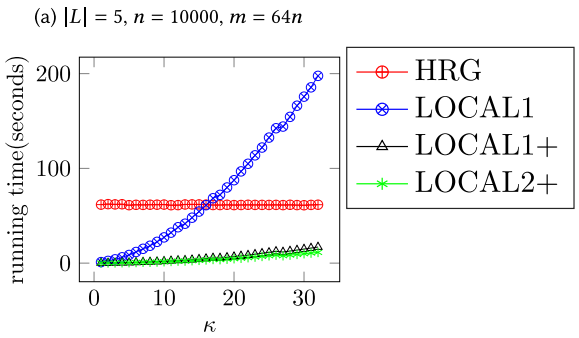

Fig. 7. Directed Planted In-Cuts with variable κ, decision version with $k = \kappa + 1$.

7.3 Random Hyperbolic Graphs

Figure 8 shows performance of all four algorithms on large random hyperbolic graphs described in Section 6.1.3 up to 1 million vertices and 32 million edges. HRG is by far the slowest algorithm with 46 min at $\kappa = 8, n = 600{,}000$ in Figure 8(b). LOCAL1 is about 8 times faster at slightly under 6 min. LOCAL1+ is 3.2 times faster than LOCAL1 at 110 s and LOCAL2+ is 2.3 times faster than LOCAL1 with 150 s of running time in Figure 8(b). Interestingly, the ratio between LOCAL1 and LOCAL1+ or LOCAL2+ is fairly stable as we grow the size of the graph, unlike planted cuts where the ratio grows with size.

Next, we compare the performance of each algorithm with respect to undirected planted cut results. HRG is much faster on random hyperbolic graphs than on the undirected planted cut data. We can compare to Figure 3(e), where the quadratic HRG has similar running time with only 50,000 vertices. The LOCAL algorithms are also faster on the random hyperbolic graphs than on the graphs with planted cuts, but by a smaller margin. LOCAL1 is slower than LOCAL1+ and LOCAL2+ by a notably smaller margin than on graphs with planted cuts.

We can explain the smaller degree counting advantage with observations from Section 8. For the geometric-based random hyperbolic graphs, we explore noticeably more edges for each vertex explored by DFS. This may be computationally cheaper for LOCAL1 while simultaneously counteracting the benefits of degree counting. Note also that LOCAL2+ is slower than LOCAL1+ in this experiment. LOCAL2+ benefits even more than LOCAL1+ from mostly visiting new vertices, as it makes progress when exploring parts of the graph that have not been explored in previous iterations. It turns out that the random hyperbolic graphs used in these experiments have very balanced minimum cuts, which may speed up all three LOCAL algorithms, since a cut is probably found in balanced search, which enables decreasing the value of the k parameter for the rest of the algorithm, resulting in less computation (see Section 7.1.1).

7.4 Real-world Networks

Table 2 presents undirected real-world network data. Each row represents a k-core, where $k = \delta$ is the minimum degree of the resulting graph. Note that in general, minimum degree for a k-core can exceed k. Table 3 shows data for graphs with planted cuts with similar parameters to the real-world graphs. The size of L for the generated data is not a constant 5, but instead corresponds to the size of the small side of the cut of the corresponding real-world graph.

LOCAL1+ and LOCAL2+ clearly outperform LOCAL1 on this selection of real-world networks, as on the artificial data. The k-cores of soc-Epinions1 have very similar performance in real-world networks and graphs with planted cuts. For the other graphs, all algorithms are faster on

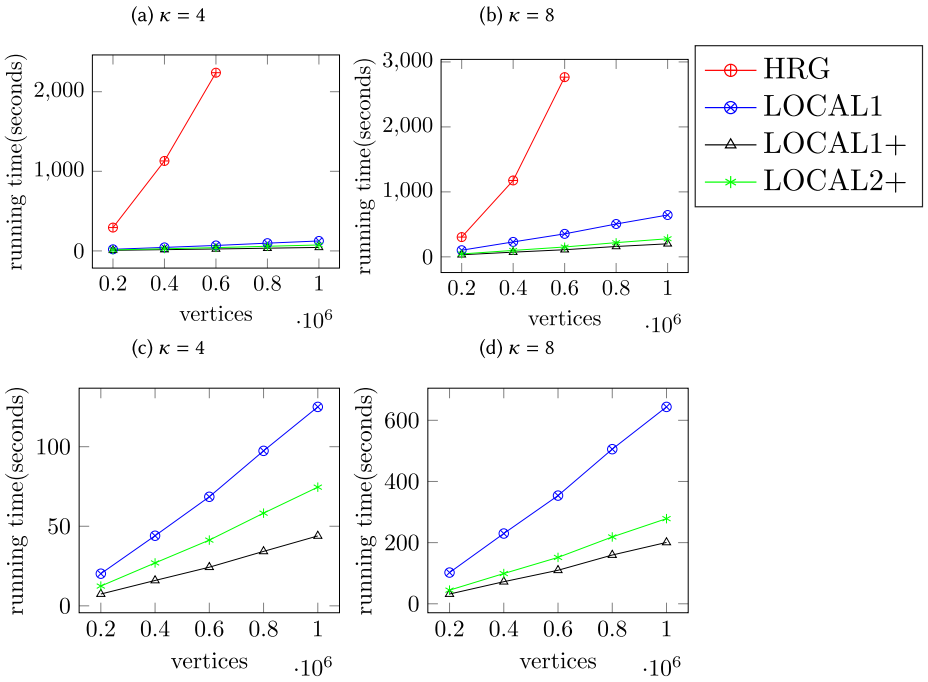

Fig. 8. Running time (seconds) for Random Hyperbolic Graphs.

Table 2. Running Times (Seconds) on k-cores of Real-world Networks

n	$\lvert L\rvert$	δ	κ	LOCAL1	LOCAL1+	LOCAL2+	HRG	graph
781	383	250	8	0.103	0.035	0.033	0.042	com-lj.ungraph
853	392	231	8	0.131	0.041	0.037	0.055	com-lj.ungraph
1205	400	200	9	0.218	0.06	0.057	0.106	com-lj.ungraph
1493	418	160	10	0.299	0.077	0.072	0.145	com-lj.ungraph
5352	12	93	12	1.095	0.198	0.184	4.981	web-BerkStan
5480	127	80	2	0.043	0.017	0.017	1.534	web-BerkStan
5719	18	19	17	17.427	1.616	1.116	27.732	soc-Epinions1
6246	2	17	16	18.25	1.702	1.107	33.257	soc-Epinions1
8106	2	12	11	6.903	0.713	0.503	40.765	soc-Epinions1
10147	10	9	5	1.274	0.225	0.208	44.06	soc-Epinions1

real-world graphs than the artificial data, but by different factors. For com-lj.ungraph, the difference is only 10–38% for the LOCAL algorithms but a factor of 3.3–5.1 for HRG. For web-BerkStan, the difference is similar for HRG but larger, ratios between 1.5–3.5, for the LOCAL algorithms.

7.5 Success Rate

We define the success rate of a vertex connectivity algorithm as the percentage of attempts that yields an optimal cut. The observed success rate for HRG is at or near 100% on all featured datasets. The success rate of LOCAL1 is slightly lower at 98% for graphs with planted cuts and 99–100% for other graphs. LOCAL1+ has about 96–97% success rate for both kinds of graphs with planted cuts and 98–99% for the other graphs. LOCAL2+ has noticeably lower accuracy when configured to the same parameters as the other LOCAL algorithms: 98% for random hyperbolic graphs, 96% for

Table 3. Running Times (Seconds) Per Vertex on Undirected Planted Cuts with Parameters $(n, |L|, \kappa)$ that Correspond to Table 2

n	$\|L\|$	δ	κ	LOCAL1	LOCAL1+	LOCAL2+	HRG
781	383	99	8	0.13	0.041	0.037	0.147
853	392	97	8	0.157	0.045	0.041	0.184
1205	400	95	9	0.275	0.07	0.061	0.435
1493	418	91	10	0.413	0.094	0.082	0.742
5352	12	23	12	3.83	0.52	0.419	18.282
5480	127	88	2	0.094	0.026	0.027	7.321
5719	18	34	17	16.755	2.694	2.262	29.536
6246	2	17	16	17.551	1.746	1.209	35.358
8106	2	12	11	6.445	0.698	0.5	41.723
10147	10	14	5	1.255	0.211	0.196	43.679

real-world networks, 90% for undirected graphs with planted cuts, and 85% for directed graphs with planted cuts. In addition to generally lower accuracy, LOCAL2+ also shows more variance in accuracy on different graph families.

The success rate of the LOCAL algorithms is significantly better than the bounds shown in Appendix A.2. Their success rate can be boosted by the standard repetition method or, equivalently, by increasing internal sample sizes (see Section 5.2). By repeating the algorithms two to three times, we would near 100% observed probability of finding the optimal cut with all three. After $O(\log n)$ repetitions the algorithms are correct with high probability.

8 EXPERIMENTAL RESULTS: INTERNAL MEASUREMENT

8.1 Edges Explored in LocalEC DFS

In this section, we study internal measurements from LocalEC in the different algorithms to explore the context of their performance on different graphs. In Figure 9, we study planted cuts in undirected graphs with $n = 200{,}000, \kappa \in \{4, 8\}$ to see how many edges are explored in DFS compared to theoretical performance. We use the decision version at $k = \kappa$, which means that no cut is found and the internal value of k stays constant. Note that we apply a multiplicative factor of 2 to ν in LocalEC. The value for ν used here includes that increase. Therefore, with no optimisations, we would expect Local1 to explore exactly νk^2 edges. The figure features the average number of edges explored normalised by νk^2 to show performance relative to this theoretical worst-case bound. Note that the number of calls is inversely proportional to ν and k is a constant, so this metric also shows the total number of edges explored at that ν times a constant.

From the values for Local1, we can see that the optimisations used cause fewer edges to be explored at all values of ν, but have the biggest impact for low values. When we double k, the values of the normalised metric decrease in this experiment. The decrease is approximately 10% for Local1, 23% for Local1+, and 50% for Local2+ for the higher values for ν, which contribute the most explored edges and are therefore the most significant. The decrease is somewhat larger for the low values. The large decrease for Local2+ in particular is consistent with a possible practical performance closer to νk, although this may not be the case for different graphs. As stated in Section 4.3, Local2+ can potentially run faster than Local1+ by a $\Theta(k)$ factor if it rarely visits the same vertex in different depth first searches for the same LocalEC call.

Figure 10 shows the same metric as in Figure 9, average number of edges explored normalised by νk^2, in random hyperbolic graphs with $n = 200{,}000, \kappa \in \{4, 8\}$. We can see that degree counting is much less effective at reducing the number of edges explored in this experiment. Although the

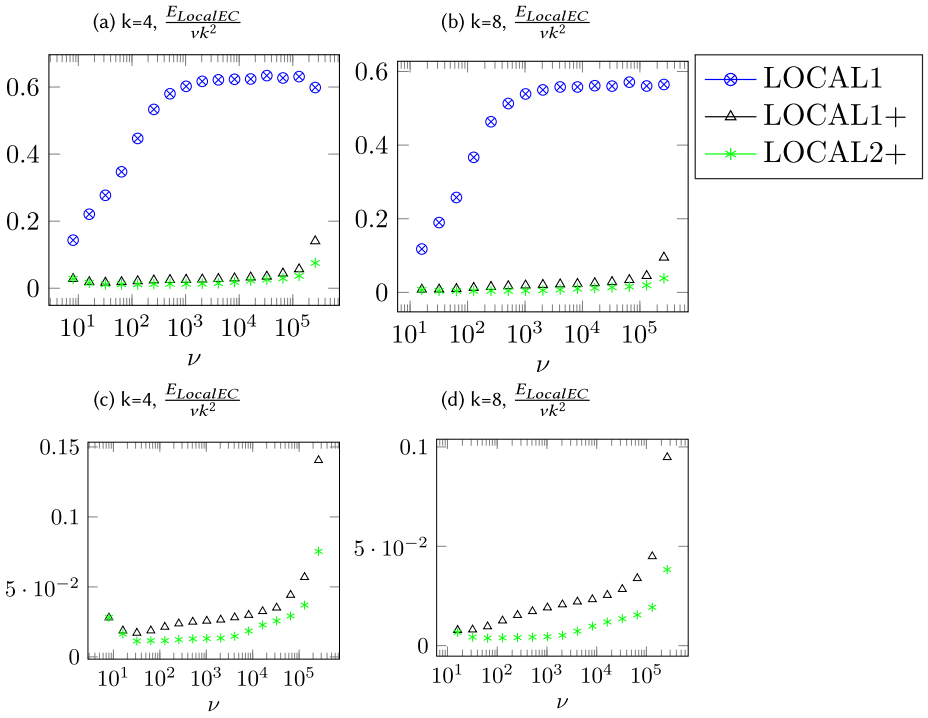

Fig. 9. Undirected Planted cuts with $n = 200{,}000$, $|L| = 5$ (decision version with $k = \kappa$). Average (non-unique) edges explored in DFS per LocalEC call, normalised by νk^2.

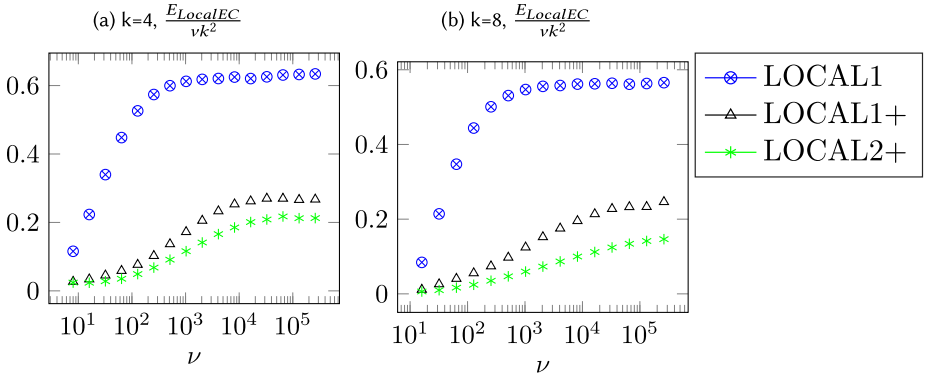

Fig. 10. Random hyperbolic graphs with $n = 200{,}000$ (decision version with $k = \kappa$). Average (non-unique) edges explored in DFS per LocalEC call, normalised by νk^2.

normalised metrics for Local1 converges to very similar values, the values for Local1+ and Local2+ are larger by an order of magnitude compared to Figure 9 and within an order of magnitude of the values for Local1. The metric decreases when we double k. The decrease is approximately 10% for Local1, 80–90% for Local1+ and 50–70% for Local2+ for the higher values for ν, with smaller decreases for the very highest values.

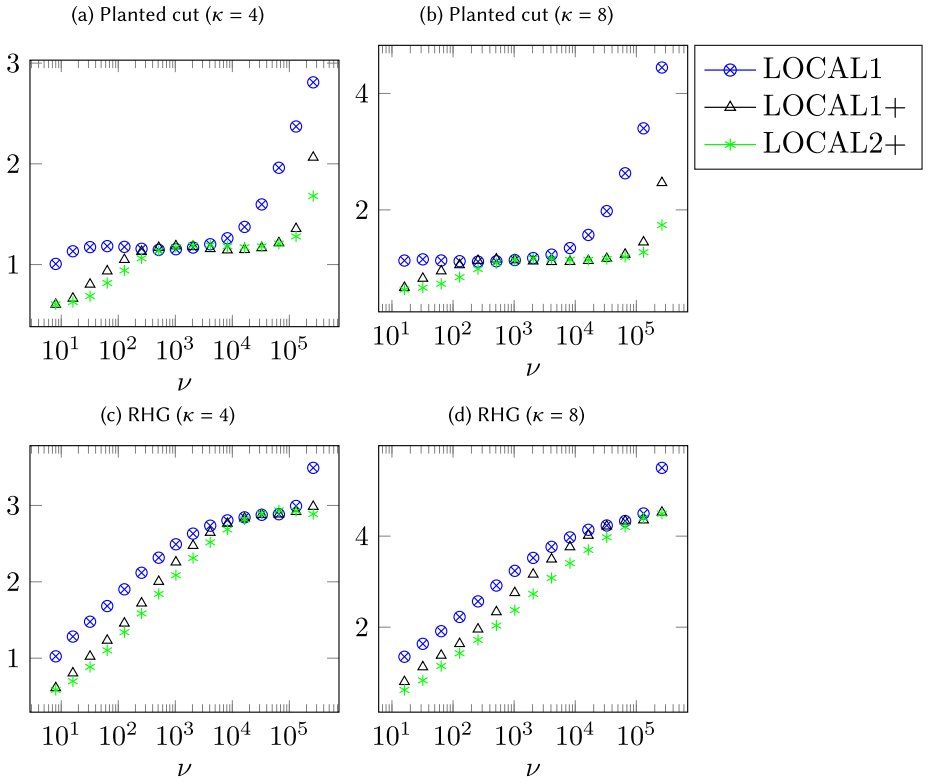

Fig. 11. Edges explored per vertex explored for Undirected Planted cuts with $n = 200{,}000, |L| = 5$, and Random hyperbolic graphs with $n = 200{,}000$ (RHG) (decision version with $k = \kappa$).

8.2 Edges Explored per Vertex in LocalEC DFS

Another internal metric that is useful for evaluating degree counting is the average number of edges that we need to explore in DFS for each vertex that is explored. For degree counting, we want to explore a vertex-set with sufficient edge volume as fast as possible. In theory, the fewer edges per vertex explored, the better for degree counting. For a non-degree counting variant, we may prefer fewer vertices per edge. Exploring a new vertex requires more operations and may access memory in a more computationally expensive way than iterating over edges to already visited vertices.

Figure 11 shows explored edges per explored vertex in undirected graphs with planted cuts and random hyperbolic graphs with $n = 200{,}000, \kappa \in \{4, 8\}$. For planted cuts, most edges lead to a new vertex except for very high ν where we explore a significant portion of the graph. For random hyperbolic graphs, we visit significantly more edges per vertex. In particular, the edges per vertex metric grows starting from the low ν values instead of staying low for most values. We can explain the difference with how the graph families are constructed. Vertex adjacency in random hyperbolic graphs is based on a distance function, which causes high correlation for which other vertices two adjacent vertices are adjacent to. In this setting, it is more likely for already explored vertices to be explored again in DFS compared to our graphs with planted cuts, which feature more uniformly random edges.

Table 4. CPU Use: Balanced Cuts/Ford-Fulkerson(FF) Versus Unbalanced Cuts/LocalEC(Local)

(a) $n = 10{,}000, \kappa = 4$

algorithm	FF	Local	Other	Seconds
LOCAL1	12.9%	84%	3.2%	0.93
LOCAL1+	49.8%	37.5%	12.7%	0.19
LOCAL2+	48.2%	40%	11.8%	0.19

(b) $n = 50{,}000, \kappa = 4$

algorithm	FF	Local	Other	Seconds
LOCAL1	10.6%	87.7%	1.7%	10.85
LOCAL1+	58.8%	33.6%	7.6%	1.91
LOCAL2+	57.4%	35.6%	7%	1.86

(c) $n = 10{,}000, \kappa = 8$

algorithm	FF	Local	Other	Seconds
LOCAL1	8.2%	90.9%	0.9%	5.86
LOCAL1+	52.9%	41.5%	5.6%	0.73
LOCAL2+	52.5%	41.9%	5.6%	0.6

(d) $n = 50{,}000, \kappa = 8$

algorithm	FF	Local	Other	Seconds
LOCAL1	7.3%	92.3%	0.4%	73.59
LOCAL1+	60.1%	36.7%	3.1%	7.54
LOCAL2+	59.8%	36.8%	3.4%	5.89

Undirected planted cuts with $\kappa \in \{4, 8\}$, $n \in \{10{,}000, 50{,}000\}$, Running time was measured separately.

8.3 CPU Time Allocation Between Unbalanced and Balanced Search

Local1 clearly visits more edges than Local1+ and Local2+, especially for graphs with planted cuts in Figure 9. In Table 4, we show how much CPU time is spent looking for unbalanced cuts with LocalEC compared to looking for balanced cuts with Ford-Fulkerson max flow on undirected graphs with planted cuts with $\kappa \in \{4, 8\}$, $n \in \{10{,}000, 50{,}000\}$. Most of the running time of LOCAL1 is used searching for unbalanced cuts with LocalEC. On the other hand, LOCAL1+ and LOCAL2+ spend a similar amount of time on balanced and unbalanced cuts. Note that only LocalEC is different between the versions. These results suggest that degree counting improves the practical performance of LocalEC significantly and there is not much more room for improvement through LocalEC without also further optimising x-y max flow to search for balanced cuts. Furthermore, when we increase κ or n, the percentage for local search goes up for LOCAL1 but even further down for LOCAL1+ and LOCAL2+. As the running time increases with κ or n, time spent on the "other" category decreases for all three algorithms. This category is dominated by setup for data structures.

9 CONCLUSION

We study the experimental performance of the near-linear time algorithm in Reference [11] when the input graph connectivity is small. The algorithm is based on local search. We also introduce a new heuristic for the local search algorithm, which we call degree counting. Based on experimental results, the degree counting heuristic significantly improves the empirical running time of the algorithm over its non-degree counting counterpart.

We point out new theoretical results that have potential to be practical and worth studying the practical performance. We believe that new interesting heuristics for practical speedup remain totally unexplored. This includes vertex connectivity in polylog max-flows [21], improved directed vertex connectivity algorithms [6] or even new linear-time minimum edge-cut algorithms by the authors of Reference [25].

APPENDICES

A OMITTED PROOFS

A.1 Correctness of LocalEC

To show that any algorithm among Local1, Local1+, Local2, and Local2+ is LocalEC, it is enough to prove that it satisfies two properties:

Property 1. *If $V(T)$ is returned, then $|E(V(T), V - V(T))| < k$ and $\emptyset \neq V(T) \subsetneq V$.*

Property 2. *If there is a vertex-set S satisfying Equation (1), then $\perp$ is returned with probability at most $1/2$.*

The following simple observation is due to Reference [4].

Observation 2. *Let S be a vertex-set in graph G and $x \in S$. Let P be a path from x to y. Let G' be G after reversing all edges along P. If $y \in S$, then $|E_{G'}(S, V-S)| = |E_G(S, V-S)|$. Otherwise, $|E_{G'}(S, V-S)| = |E_G(S, V-S)| - 1$.*

For the first property, the following argument works for all four algorithms.

Lemma A.1. *Local1, Local1+, Local2, and Local2+ satisfy Property 1.*

Proof. Let $S = V(T)$ be the cut the algorithm returned. Observe that $x \in S$ by design. By Observation 2, each iteration can only reduce the number of crossing edges by at most one. This can happen at most $k-1$ times before the final iteration, which implies that initially $|E(S, V-S)| \leq k-1$. □

For the second property, the following argument works for Local1 and Local1+.

Lemma A.2. *Local1 and Local1+ satisfy Property 2.*

Proof. We focus on proving that LOCAL1 satisfies Property 2 (the proof for Local1+ will be essentially identical). If the algorithm terminates before the kth iteration, then it outputs $V(T)$, and thus $\perp$ is never returned. So now we assume that the algorithm terminates at the kth iteration. Let $y_1, \ldots y_{k-1}$ be the sequence of chosen path endpoints y in DFS iterations. We first bound the probability that $y_i \in S$. Let $\text{vol}_i^{\text{out}}(S)$ be the volume of S at iteration i. So,

$$\Pr(y_i \in S) \leq \frac{\text{vol}_i^{\text{out}}(S)}{2\nu k} \leq \frac{\text{vol}^{\text{out}}(S)}{2\nu k} \leq \frac{\nu}{2\nu k} = \frac{1}{2k}. \tag{2}$$

The first inequality follows by design. The second inequality follows by Observation 2.

By Observation 2, the algorithm can only return $\perp$ at the final iteration if at least one of the y_i's is in S (or if there is not viable cut). Let $\mathbb{1}[y_i \in S]$ be an indicator function. Let $Y = \sum_{i \leq k-1} \mathbb{1}[y_i \in S]$. Observe that $Y \geq 1$ if and only if the algorithm outputs $\perp$. We now bound the probability that $Y \geq 1$. By linearity of expectation, we have $\mathbb{E}[Y] = \sum_{i \leq k-1} \mathbb{E}[\mathbb{1}[y_i \in S]] = \sum_{i \leq k-1} \Pr(y_i \in S) \leq \frac{1}{2}$. Therefore, by Markov's inequality, we have

$$\Pr(Y \geq 1) = \Pr\left(Y \geq 2 \cdot \frac{1}{2}\right) \leq \Pr(Y \geq 2\mathbb{E}[Y]) \leq \frac{1}{2}. \tag{3}$$

This completes the proof for LOCAL1. To see that the same proof works for LOCAL1+, observe that it (Equation (2) in particular) does not use the identity of the edges. Outgoing edges of a vertex are interchangible. The degree counting variant counts edges ensures that each outgoing edge for visited vertices is included in the collection of edges without collecting explicitly. The precomputed random number τ corresponds to a random edge from the collection. □

It remains to prove the second property for Local2 and Local2+. However, the arguments for Local2 and Local2+ are very similar to Local1 and Local1+:

Lemma A.3. *Local2 and Local2+ satisfy Property 2.*

Proof. For Local2, each edge in $E(S, V)$ has a $\frac{1}{2\nu}$ probability to be chosen if the edge is visited. The probabilities are not independent but can be used for Markov's inequality. If Y is the number of edges in $E(S, V)$ that are chosen, or equivalently the number of times a vertex in S is chosen, then we have $\mathbb{E}[Y] \leq \frac{\nu}{2\nu} = \frac{1}{2}$, resulting in the same equation as Equation (3). If we consider the case where all edges in $E(S, V)$ are visited in a single iteration, then we can see that the bound is tight. For Local2+, apply the same logic to $c(v)$ instead of edges. □

We finish with an observation on the effect of increasing ν on the probability bound.

Observation 3. *The proofs above use the volume multiplier 2 in LocalEC (LocalEC uses 2ν instead of ν). It is easy to show that if we substitute a different multiplier C, the bound in Equation (3) will be $\frac{1}{C}$. In particular, if we can show that $\nu \geq C\text{vol}^{\text{out}}(L)$ for some constant C, we effectively substitute the multiplier with $2C$ for a bound of $\frac{1}{2C}$.*

A.2 Correctness of LocalEC-based Vertex Connectivity (Algorithm 4)

If $\kappa \geq k$, then the algorithm cannot find a cut and correctly returns $\perp$. Suppose that there is a vertex cut in G, represented by a separation triplet (L, S, R) such that $|S| < k$ and all paths from L to R contain a vertex in S. We define a partition (L', R') of the split graph (see Section 3) of G as: $L' = \{x_{\text{in}}, x_{\text{out}} | x \in L\} \cup \{x_{\text{in}} | x \in S\}$, $R' = \{x_{\text{in}}, x_{\text{out}} | x \in R\} \cup \{x_{\text{out}} | x \in S\}$. If $\text{vol}^{\text{out}}(L') < a$ or $\text{vol}^{\text{in}}(R') < a$, then we say that the cut is *unbalanced*. If $\text{vol}^{\text{out}}(L'), \text{vol}^{\text{in}}(R') \geq a$, then it is *balanced*.

Lemma A.4. *If there is an unbalanced cut, then Algorithm 4 returns it with constant probability.*

Proof. Suppose that the cut is unbalanced. If we do not have $\text{vol}^{\text{out}}(L') < a$, then the graph is directed and we also run the algorithm on the reverse graph, where we have $\text{vol}^{\text{out}}(R') < a$. We can relabel L and R as needed to assume without loss of generality for both directed and undirected graphs with unbalanced vertex cuts that $\text{vol}^{\text{out}}(L') < a$.

If $|L| = 1$, then the cut is found on Lines 4 and 5 of Algorithm 4, so we can assume that $|L| > 1$. We can also assume $\delta \geq k$, because otherwise a minimum degree cut is trivial to find. It follows that $\text{vol}^{\text{out}}(L) \geq \delta|L| \geq k|L| \geq 2k$. The case $k = 1$ is trivial, so we also assume that $k \geq 2$. We can use the above to show that

$$\begin{aligned} \text{vol}^{\text{out}}(L) \leq \text{vol}^{\text{out}}(L') &= \text{vol}^{\text{out}}(L) + |L| + |S| < \text{vol}^{\text{out}}(L) + \frac{1}{k}\text{vol}^{\text{out}}(L) + k \\ &\leq \left(1 + \frac{1}{k} + \frac{1}{2}\right)\text{vol}^{\text{out}}(L) \leq 2\text{vol}^{\text{out}}(L). \end{aligned} \tag{4}$$

For some iteration of the main loop in Algorithm 2, there exists $C \in [1, 2]$ such that, $\frac{\nu}{2} \leq \text{vol}^{\text{out}}(L') = \frac{\nu}{C} \leq \nu$. For this value of ν and each sampled edge (x, x'), we have

$$\Pr(x \in L) = \frac{\text{vol}^{\text{out}}(L)}{m} \geq \frac{\text{vol}^{\text{out}}(L')}{2m} = \frac{\nu}{2Cm}. \tag{5}$$

By Observation 3 and $\text{vol}^{\text{out}}(L') = \frac{\nu}{C}$, we have

$$\Pr(\text{LocalEC does not find cut} | x \in L) \leq \frac{1}{2C}. \tag{6}$$

Now, for any given sampled edge, we have

$$\Pr(x \in L \wedge \text{LocalEC finds cut}) \geq \frac{\nu}{2Cm}\left(1 - \frac{1}{2C}\right) \geq \frac{\nu}{4m}\left(1 - \frac{1}{4}\right) = \frac{3/16}{m/\nu}. \tag{7}$$

Local minima for an expression of the form $x(\text{const} - x)$ can be found at the endpoints of its domain. $C = 2$ minimizes this expression within the interval $C \in [1, 2]$, which gives us the last inequality. Finally, define $Y = \sum_{\text{sampled edge } (x,x')} \mathbb{1}(x \in L \wedge \text{LocalEC finds cut})$. Clearly, $Y = 0$ if and only if the algorithm fails to find the cut with this value of ν. With a sample size of $\frac{m}{\nu}$, we have

$$\Pr(Y = 0) \leq \left(1 - \frac{3/16}{m/\nu}\right)^{\frac{m}{\nu}} \leq e^{-3/16} \leq 0.83. \tag{8}$$

The final inequality holds for $\frac{m}{\nu} > \frac{3}{16}$, which is trivially true. (The inequality is easy to prove by showing that $\frac{\partial}{\partial x}(1 - \frac{a}{x})^x$ is positive when $a < x$ and therefore $(1 - \frac{a}{x})^x$ is lesser than its limit e^{-a}.) □

Lemma A.5. *If there is a balanced cut, then Algorithm 4 returns it with constant probability.*

Proof. For balanced cuts, we use an additional assumption that $k < \frac{\sqrt{n}}{2}$. This is a reasonable assumption, since if $k = \Omega(\sqrt{n})$, the HRG algorithm has superior time complexity. For directed graphs at least $m - k^2 \geq m - \frac{n}{4} \geq m - \frac{m}{4} \geq \frac{3m}{4}$ edges are outside of $E(S,S)$. We define $C_{\text{out}} = \frac{\text{vol}^{\text{out}}(L)+\text{vol}^{\text{out}}(R)}{m}$ and $C_{\text{in}} = \frac{\text{vol}^{\text{in}}(L)+\text{vol}^{\text{in}}(R)}{m}$. Clearly, $C_{\text{out}} + C_{\text{in}} \geq \frac{3}{4}$. We sample $(x,x'),(y,y') \in E(G)$ independently. Equation (4) gives us $\text{vol}^{\text{out}}(L) \geq \frac{\text{vol}^{\text{out}}(L')}{2} \geq \frac{a}{2}$. Assuming $a = \frac{m}{3k}$, we have

$$\Pr(x \in L, y \in R) = \frac{\text{vol}^{\text{out}}(L)\,\text{vol}^{\text{out}}(R)}{m^2} = \frac{\text{vol}^{\text{out}}(L)(mC_{\text{out}} - \text{vol}^{\text{out}}(L))}{m^2} \geq \frac{\frac{a}{2}(mC_{\text{out}} - \frac{a}{2})}{m^2}$$
$$= \frac{C_{\text{out}} - \frac{a}{2m}}{2m/a} = \frac{C_{\text{out}} - \frac{1}{6k}}{2m/a}. \tag{9}$$

We have another expression of the form $x(\text{const} - x)$, which is minimized at $\text{vol}^{\text{out}}(L) = \frac{a}{2}$, or equivalently at $\text{vol}^{\text{out}}(R) = (mC_{\text{out}} - \text{vol}^{\text{out}}(L)) = \frac{a}{2}$, which gives us the inequality above. For sampling $(x,x'),(y,y') \in E(G^R)$ in directed graphs, we get a bound of $\frac{C_{\text{in}} - \frac{1}{6k}}{2m/a} = \frac{\frac{3}{4} - C_{\text{out}} - \frac{1}{6k}}{2m/a}$. We define $Y = \sum_{\text{samples in } G \text{ and } G^R} \mathbb{1}(x \in L, y \in R)$, which is 0 if and only if the cut is not found.

$$\Pr(Y = 0) \leq \left(1 - \frac{C_{\text{out}} - \frac{1}{6k}}{2m/a}\right)^{\frac{m}{a}} \left(1 - \frac{\frac{3}{4} - C_{\text{out}} - \frac{1}{6k}}{2m/a}\right)^{\frac{m}{a}} \leq \left(1 - \frac{\frac{3}{8} - \frac{1}{6k}}{2m/a}\right)^{\frac{2m}{a}} \leq \left(1 - \frac{\frac{7}{24}}{2m/a}\right)^{\frac{2m}{a}}$$
$$\leq e^{-7/24} \leq 0.75. \tag{10}$$

For the second inequality, $C_{\text{out}} = C_{\text{in}} = \frac{3}{8}$ maximizes the expression. The third uses $k \geq 2$. The fourth inequality holds for $(6k =)\frac{2m}{a} > \frac{7}{24}$. For undirected graphs, we always have $C_{\text{out}} = C_{\text{in}} \geq \frac{3}{8}$. We do not sample in G^R but if $x \in L, y \in R$ is successful sampling, then so is the mutually exclusive event $x \in R, y \in L$, which gives us a bound of $2\frac{\frac{3}{8} - \frac{1}{6k}}{2m/a}$ in the equivalent of Equation (9) and an identical bound at the equivalent of Equation (10) for undirected graphs. □

Lemmas A.4 and A.5 are sufficient to show that the error rate is bounded by a constant for both balanced (0.75) and unbalanced (0.83) vertex cuts. We will now use independent probabilities to slightly improve the overall probability bound.

Lemma A.6. *If $\kappa < k$, then the LocalEC-based vertex connectivity algorithm finds a cut with probability at least 25%.*

Proof. When $\text{vol}^{\text{out}}(L)$ is low enough, the cut can be found by LocalEC calls with greater ν than what was used in the proof for Lemma A.4. If $\text{vol}^{\text{out}}(L') \leq \frac{a}{2}$, then we can run LocalEC at 2ν (with half as many samples) in addition to ν the error rate for unbalanced cuts is lowered to 0.75 (insert $C = 4$ instead of $C = 2$ in the proof to get the error rate at 2ν).

When $\text{vol}^{\text{out}}(L') < a$, we can find cuts with the balanced cut approach with reduced success rate. We can get the error rate by inserting a bound other than $\text{vol}^{\text{out}}(L) \geq \frac{\text{vol}^{\text{out}}(L')}{2} \geq \frac{a}{2}$ at Equation (9). If we insert $\frac{\text{vol}^{\text{out}}(L')}{2} \geq \frac{a}{4}$, then we get $\Pr(Y = 0) \leq e^{-1/6} \leq 0.85$.

When $\text{vol}^{\text{out}}(L') \approx a$, we have a significant independent probability of finding the cut with both the unbalanced and balanced approach. When $\frac{a}{2} \leq \text{vol}^{\text{out}}(L') \leq a$, we have an error rate of at most $e^{-3/16}$ using the unbalanced approach for $\text{vol}^{\text{out}}(L') \leq a$ and an error rate of at most $e^{-1/6}$ using the balanced cut approach for $\frac{a}{2} \leq \text{vol}^{\text{out}}(L')$. The probability of finding the cut with neither is at most $(1 - e^{-3/16}e^{-1/6}) = (1 - e^{-17/48}) \geq 0.29$. For more balanced cuts, we have 0.75 error rate from Lemma A.5, and for more unbalanced cuts, we have 0.75 error rate by considering LocalEC

at $v \in [\frac{1}{2}\,\mathrm{vol}^{\mathrm{out}}(L'), \mathrm{vol}^{\mathrm{out}}(L')]$ and $v \in [\frac{1}{4}\,\mathrm{vol}^{\mathrm{out}}(L'), \frac{1}{2}\,\mathrm{vol}^{\mathrm{out}}(L')]$. Therefore, we have an improved upper bound of 0.75 for the error rate for all vertex cuts. □

We cannot use the same approach as above to improve the bound for balanced cuts. The success rate bound for the unbalanced approach quickly goes to zero for higher $\mathrm{vol}^{\mathrm{out}}(L')$ due to the pessimistic assumption in the probability bound for LocalEC that every edge in $\mathrm{vol}^{\mathrm{out}}(L')$ is explored/counted every time it is possible. It may be possible to improve the bound in the average case with a smart order in DFS edge selection or just more analysis for a randomised edge order.

B PREFLOW-PUSH-BASED VERTEX CONNECTIVITY ALGORITHM

We use the algorithm by Henzinger, Rao, and Gabow [17] with only minor optimisations. For details omitted here, see the original article. The core algorithm (Section 3 in Reference [17]) uses a preflow-based algorithm to calculate the minimum $S_i x_i$-cut, where $S_i = \{x\} \cup \{x_j : j < i\}$, for each vertex x_i not adjacent to x. The algorithm maintains an active set W of vertices from where the current sink may be reachable. If there exists a minimum vertex cut $S \ni x$, which is very probable for small κ, then the minimum of these cuts will be a minimum vertex cut.

On page 10 of Reference [17], in the proof for Theoreom 2.3, Henzinger et al. describe a randomised version that repeats the core algorithm on random seed vertices to achieve a 50% or lower error rate. None of the included test cases need to repeat to achieve this bound. In the same section, Henzinger et al. provide a guaranteed method of doubling k to find some $k \in (\kappa, 4\kappa)$, which we need for undirected graphs. Using their method, the algorithm is run on an arbitrary nonrandom vertex of degree k. To obtain an optimal cut with any probability guarantee, the algorithm needs to be repeated on a random seed vertex. We use random seed vertices during doubling to avoid having to repeat the algorithm after already finding a cut of size less than k.

For undirected graphs, we use the sparsification algorithm by Nagamochi and Ibaraki [26] to reduce the average degree of the graph to at most k, doubling k until we find a cut smaller than k, as we do with the algorithms by Forster et al. [11]. For directed graphs, we can omit the doubling process, since the algorithm itself does not depend on the k parameter. For directed graphs the algorithm is repeated on the reverse graph using the same seed vertices. In case of weighted edges, dynamic trees would be used to improve time complexity, but the experiments in this article only use unweighted edges.

On page 20 of Reference [17], Henzinger et al. describe multiple auxiliary data structures used to achieve the desired time complexity. One of these is a partition of vertices in the awake set W by their current distance values. We add another auxiliary data structure that stores the index of a vertex in this data structure to speed up finding and removing a vertex, which happened frequently enough to create a CPU hotspot.

C CYCLE-BASED DIRECTED GRAPHS WITH PLANTED CUTS

The graphs with planted cuts are graphs with a vertex partition (L, S, R) such that S is a unique minimum vertex cut and every path from L to R contains a vertex in S.

We construct the graph in the following way: Label the vertices $1, 2, \ldots, n$. Let $L = \{1, 2, \ldots, |L|\}, S = \{|L| + 1, |L| + 2, \ldots, |L| + |S|\}, R = V \setminus (L \cup S)$. We ban edges from L to R. This is enough to guarantee that every path from L to R contains a vertex in S. For some integer $\eta > |S|$, let every vertex have an outgoing edge to the η nonbanned vertices after it and an incoming edge from the η nonbanned vertices before it, mod n. For example, the last vertex in L has edges to all of S and the $(\eta - |S|)$ first vertices in L. See Figure 12 for an example.

We can show that S is a unique minimum vertex cut by explicitly constructing η vertex-disjoint paths between non-adjacent vertices x to y when $x \notin L$ or $x \notin R$. Start with an edge from x to the

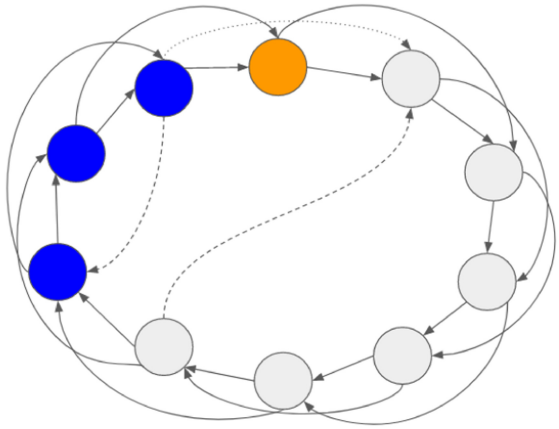

Fig. 12. Example of the deterministic base construct of a directed graph with a planted cut with parameters: $|L| = 3, |S| = 1, |R| = 6, \eta = 2$. The dotted edge crosses the (orange) set S. It is removed and replaced by the edges represented by dashed arrows.

η next nonbanned vertices. We then repeatedly extend the least advanced path, in the sense that for its endpoint v, $(v - y) \mod n$ is the least among the paths. At most $\eta - 1$ vertices "ahead" of it are included in other paths, so we can always extend the paths until they reach y or x again.

When we can end a path by adding an edge to y, we do. For the $(\eta - |S|)$ last vertices in R have a choice between an edge to L or to the beginning of R. We go to R if and only if $y \in R$. In this case only $|S|$ paths enter L. At the end of L, we only use the edges to the beginning of L if necessary, which implies that $y \notin S$ and S paths have already exited L, which then implies $x, y \in L$.

At no point do we include an edge that skips vertices in L if $y \in L$ or R if $y \in R$. Therefore, each path inevitably reaches y, and we have η vertex-disjoint paths from x to y.

REFERENCES

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

[2] Josh Alman and Virginia Vassilevska Williams. 2020. A refined laser method and faster matrix multiplication. Retrieved from https://arxiv.org/abs/2010.05846.

[3] Deepayan Chakrabarti and Christos Faloutsos. 2006. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.* 38, 1 (2006), 2.

[4] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. 2017. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In *Proceedings of the SODA*. SIAM, 1900–1918.

[5] Chandra Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Clifford Stein. 1997. Experimental study of minimum cut algorithms. In *Proceedings of the SODA*. ACM/SIAM, 324–333.

[6] Chandra Chekuri and Kent Quanrud. 2021. Faster algorithms for rooted connectivity in directed graphs. In *Proceedings of the ICALP (LIPIcs)*, Vol. 198. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 49:1–49:16.

[7] Joseph Cheriyan and John H. Reif. 1994. Directed *s-t* numberings, rubber bands, and testing digraph *k*-vertex connectivity. *Comb.* 14, 4 (1994), 435–451.

[8] Yefim Dinitz. 2006. Dinitz' algorithm: The original version and Even's version. In *Essays in Memory of Shimon Even (Lecture Notes in Computer Science)*, Vol. 3895. Springer, 218–240.

[9] Shimon Even. 1975. An algorithm for determining whether the connectivity of a graph is at least k. *SIAM J. Comput.* 4, 3 (1975), 393–396.

[10] Lester Randolph Ford and Delbert Ray Fulkerson. 1956. Maximal flow through a network. *Can. J. Math.* 8 (1956), 399–404.

[11] Sebastian Forster, Danupon Nanongkai, Liu Yang, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2020. Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms. In *Proceedings of the SODA*. SIAM, 2046–2065.

[12] Harold N. Gabow. 2006. Using expander graphs to find vertex connectivity. *J. ACM* 53, 5 (2006), 800–844.
[13] Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2019. Deterministic graph cuts in subquadratic time: Sparse, balanced, and k-vertex. Retrieved from https://arxiv.org/abs/1910.07950.
[14] Loukas Georgiadis, Dionysios Kefallinos, Luigi Laura, and Nikos Parotsidis. 2021. An experimental study of algorithms for computing the edge connectivity of a directed graph. In *Proceedings of the ALENEX*. 85–97.
[15] Olivier Goldschmidt, Patrick Jaillet, and Richard Lasota. 1994. On reliability of graphs with node failures. *Networks* 24, 4 (1994), 251–259.
[16] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. 2018. Practical minimum cut algorithms. *ACM J. Exp. Algorithmics* 23 (2018).
[17] Monika Rauch Henzinger, Satish Rao, and Harold N. Gabow. 1996. Computing vertex connectivity: New bounds from old techniques. In *Proceedings of the FOCS*. IEEE, 462–471.
[18] Michael Jünger, Giovanni Rinaldi, and Stefan Thienel. 2000. Practical performance of efficient minimum cut algorithms. *Algorithmica* 26, 1 (2000), 172–195.
[19] D. Kleitman. 1969. Methods for investigating connectivity of large graphs. *IEEE Trans. Circ. Theory* 16, 2 (1969), 232–233.
[20] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from http://snap.stanford.edu/data.
[21] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2021. Vertex connectivity in poly-logarithmic max-flows. In *Proceedings of the STOC*. ACM, 317–329.
[22] Nathan Linial, László Lovász, and Avi Wigderson. 1988. Rubber bands, convex embeddings and graph connectivity. *Combinatorica* 8, 1 (1988), 91–102.
[23] Shaobin Liu, Kam-Hoi Cheng, and Xiaoping Liu. 2000. Network reliability with node failures. *Networks* 35, 2 (2000), 109–117.
[24] Yang P. Liu and Aaron Sidford. 2020. Faster divergence maximization for faster maximum flow. Retrieved from https://arxiv.org/abs/2003.08929.
[25] Sagnik Mukhopadhyay and Danupon Nanongkai. 2020. Weighted min-cut: Sequential, cut-query, and streaming algorithms. In *Proceedings of the STOC*. ACM, 496–509.
[26] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica* 7, 5-6 (1992), 583–596.
[27] Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2019. Breaking quadratic time for small vertex connectivity and an approximation scheme. In *Proceedings of the STOC*. ACM, 241–252.
[28] Manfred Padberg and Giovanni Rinaldi. 1990. An efficient algorithm for the minimum capacity cut problem. *Math. Program.* 47 (1990), 19–36.
[29] Azzeddine Rigat. 2012. An experimental study of k-vertex connectivity algorithms. In *Proceedings of the INFOCOMP*.
[30] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Netw. Sci.* 4, 4 (2016), 508–530.
[31] Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. 2020. Bipartite matching in nearly linear time on moderately dense graphs. In *Proceedings of the FOCS*. IEEE, 919–930.
[32] Moritz von Looz, Henning Meyerhenke, and Roman Prutkin. 2015. Generating random hyperbolic graphs in subquadratic time. In *Proceedings of the ISAAC (Lecture Notes in Computer Science)*, Vol. 9472. Springer, 467–478.
[33] Douglas R. White and Frank Harary. 2001. The cohesiveness of blocks in social networks: Node connectivity and conditional density. *Sociol. Methodol.* 31, 1 (2001), 305–359.

Received 28 January 2022; revised 24 July 2022; accepted 29 August 2022