Practical Expander Decomposition

Lars Gottesbüren
Karlsruhe Institute of Technology, Germany

Nikos Parotsidis
Google Research, Zürich, Switzerland

Maximilian Probst Gutenberg
ETH Zürich, Switzerland

Abstract

The expander decomposition of a graph decomposes the set of vertices into clusters such that the induced subgraph of each cluster is a subgraph with high conductance, and there is only a small number of inter-cluster edges. Expander decompositions are at the forefront of recent theoretical developments in the area of efficient graph algorithms and act as a central component in several state-of-the-art graph algorithms for fundamental problems like maximum flow, min-cost flow, Gomory-Hu trees, global min-cut, and more. Despite this crucial role and the existence of theoretically efficient expander decomposition algorithms, little is known on their behavior in practice.

In this paper we explore the engineering design space in implementations for computing expander decompositions. We base our implementation on the near-linear time algorithm of Saranurak and Wang [SODA'19], and enhance it with practical optimizations that accelerate its running time in practice and at the same time preserve the theoretical runtime and approximation guarantees.

We evaluate our algorithm on real-world graphs with up to tens of millions of edges. We demonstrate significant speedups of up to two orders of magnitude over the only prior implementation. To the best of our knowledge, our implementation is the first to compute expander decompositions at this scale within reasonable time.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Expander Decomposition, Clustering, Graph Algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.61

Supplementary Material *Software (Source Code and Experiment Scripts)*: `https://github.com/larsgottesbueren/practical-expander-decomposition`

Acknowledgements We thank Isaac Arvestad for open-sourcing a clean and extensible C++ implementation of the Saranurak-Wang algorithm.

1 Introduction

The problem of computing the *expander decomposition* of a graph G is to decompose the set of vertices of G into clusters such that the induced subgraph of each cluster is a subgraph that is well-connected, and there is only a small number of inter-cluster edges. Since its introduction by Kannan, Vempala, and Vetta [18] over 20 years ago, the notion of expander decomposition has been widely studied [34, 38, 29, 33, 11, 16, 3, 35]. Recently, Saranurak and Wang [33] presented the first near-linear time algorithm for computing expander decompositions.

Expander decompositions find numerous applications such as graph sketching and sparsification [5, 17, 10], Laplacian solvers [34, 12, 11, 24], maximum flow algorithms [21, 11] including the recent breakthrough almost-linear time algorithm [9, 36, 8], computing Gomory-Hu trees [1, 28, 2], global min-cut [20, 15, 32, 26], vertex connectivity [27], dynamic connectivity and minimum spanning forest algorithms [38, 29, 30], and many more.

The following is a quote from Saranurak and Wang [33]: "we note that our algorithm is developed from first principles based on relatively simple and basic techniques, thus making it very likely to be practical". Perhaps surprisingly, no published work has studied the practical performance of expander decomposition algorithms. To the best of our knowledge, there is only a Master's thesis by Arvestad [6], who provides a faithful implementation of Saranurak and Wang's framework [33]. Contrary to the quote above, they note that there is substantial room for improvement in practice, due to high constant and poly-log factors.

In this paper, we explore the algorithm engineering space for computing expander decompositions. We base our approach on the same framework [33]. We propose five optimizations that speed up the algorithm in practice without affecting the theoretical running time and approximation guarantees. Our experimental findings show that our implementation significantly outperforms the basic implementation by up to two orders of magnitude, and allows the algorithm to scale to graphs with tens of millions of edges within reasonable time.

We denote by $G = (V, E)$ the input graph and let $n = |V|, m = |E|$. We let $\deg_G(v)$ denote the degree of vertex v in G and define for any $S \subseteq V$, the volume of S by $\mathrm{vol}_G(S) = \sum_{s \in S} \deg_G(s)$. We denote by $E_G(A, B)$ for any disjoint sets $A, B \subseteq V$ the edges in G with exactly one endpoint in A and one endpoint in B. We denote by $(S, V \setminus S)$ a cut in G, and often denote the cut only by S.

▶ **Definition 1** (Conductance and Expander). *Given graph G, and a* cut $\emptyset \subsetneq S \subsetneq V$, *we define the conductance of S by $\Phi_G(S) = \frac{|E_G(S, V \setminus S)|}{\min\{vol_G(S), vol_G(V \setminus S)\}}$. We say S is a ϕ-sparse cut if $\Phi_G(S) < \phi$. We say G is a ϕ-expander if it contains no ϕ-sparse cut.*

▶ **Definition 2** (Expander Decomposition). *A partition $X_1, X_2, \ldots, X_k$ of the vertex set V is a ϕ-expander decomposition of quality q if:*

1. *every* cluster $G[X_i]$ *is a ϕ-expander, and*
2. *the number of inter-cluster edges is at most $q \cdot \phi m$.*

Recently, the first near-linear time algorithm to compute expander decompositions was presented by Saranurak and Wang [33]. We summarize the result in the theorem below.

▶ **Theorem 3** (see [33], Theorem 1.2). *For any $\phi < 1/\log^2(m)$, there is an algorithm that runs in time $O(m \log^4 m/\phi)$ and outputs a ϕ-expander decomposition of quality $O(\log^3 m)$.*

The main algorithmic ingredients in [33] are the cut-matching game [22, 31], which either finds a low-conductance cut or certifies expansion, as well as a trimming procedure [33] to extract an expander from a *near-expander*. Both trimming and cut-matching internally use single-commodity maximum flow. One contribution of [33] is to show that a height-constrained version of push-relabel [15] can be used to efficiently solve these flow problems to achieve near-linear time overall. In order to present clearly our optimizations in the context of the algorithmic framework from [33], and to keep the paper as self-contained as possible, we give a primer on the theoretical background in Section 2, which is all prior work. In Section 3 we then present our contributions: optimizations that make the theoretically efficient techniques fast in practice.

2 Expander Decomposition

Our implementation is inspired by the following framework suggested in [29, 30, 38, 33] to compute expander decompositions. The framework follows the natural idea to recursively partition V into sets $X_1, X_2, \ldots, X_k$ and whenever there is a $\tilde{O}(\phi)$-sparse cut $S \subseteq X_i$ in $G[X_i]$, we replace the cluster X_i by S and $X_i \setminus S$.

However, the recursion depth of this meta-algorithm might be large, e.g. $\Omega(n)$. To control the recursion depth, it is natural to use the most balanced $\tilde{O}(\phi)$-sparse cuts when decomposing. Formally, we say that a cut $S \subseteq X_i$ is b-balanced if $\frac{1}{b}\text{vol}_G(X_i) \leq \text{vol}_G(S) \leq (1-\frac{1}{b})\text{vol}_G(X_i)$ and thus again, we have that S is b-balanced if and only if $X_i \setminus S$ is b-balanced. To obtain recursion depth $\tilde{O}(1)$, we need to only decompose along $\tilde{\Omega}(1)$-balanced cuts.

Unfortunately, there may not always be a $\tilde{\Omega}(1)$-balanced $\tilde{O}(\phi)$-sparse cut in a graph $G[X_i]$ even though it might also not be a ϕ-expander. However, in this case, $G[X_i]$ is very close to being a ϕ-expander. We thus need to only *trim* a very small set $S \subseteq X_i$ from $G[X_i]$ and can then conclude that $G[X_i \setminus S]$ is a $\phi/6$-expander. Thus, when decomposing along S, we only need to recurse on S which is much smaller than $G[X_i]$ and this again yields small recursion depth. Finally, we re-parameterize ϕ and scale it by factor 6 to ensure that all clusters $G[X_i]$ with X_i in the final partition set of V are ϕ-expanders.

Let us now discuss the components that we need to implement this framework: we first need an algorithm to find $\tilde{\Omega}(1)$-balanced $\tilde{O}(\phi)$-sparse cuts, or output that no such cut exists. To this end, we turn to the cut-matching framework introduced in [22] whose guarantees are summarized below. Here, we crucially require the generalization of conductance below.

▶ **Definition 4** (Generalized Conductance and Expander). *Given graph G, and set $A \subseteq V$ and a cut $\emptyset \subsetneq S \subsetneq A$, we define the conductance of S by $\Phi_G(S, A) = \frac{|E_G(S, V\setminus S)|}{\min\{vol_G(S), vol_G(A\setminus S)\}}$. We say A is ϕ-expander in G if it contains no cut S with $\Phi_G(S, A) < \phi$.*

▶ **Theorem 5** (Cut Matching Framework, see Theorem 4.1 in [22]). *Given input graph $G = (V, E)$, $\phi < 1/\log^2(m)$ and a constant $C > 0$. The algorithm $\textsc{CutMatchingGame}(G, \phi, C)$ takes $O(m \log m/\phi)$ time and either reports*

1. *an $\Omega(1/\log^2 m)$-balanced $\Omega(1/\log^2 m)$-sparse cut S in G, or*
2. *that G is a ϕ-expander, or*
3. *a $(c_0/\log^2 m)$-sparse cut $S \subseteq V$ with $vol_G(S) \leq m/(C \log^2 m)$ for some constant $c_0 > 0$ and $V \setminus S$ is a ϕ-expander in G.*

In the first case, we can decompose immediately. In the second case, we do not have to decompose the graph and thus do not need to recurse. In the third case, we invoke the following trimming routine from [33].

▶ **Theorem 6** (Trimming, see Theorem 2.1 in [33]). *Given a graph G, a cut S and some ϕ such that $vol_G(S) \leq m/(32c_1 \log^2 m)$ and $\Phi_G(S) \leq c_1\phi \cdot \log^2 m$ for any constant $c_1 > 0$ such that $V \setminus S$ is ϕ-expander in G. Then there is an algorithm $\textsc{Trim}(G, V \setminus S, \phi)$ that after time $O(m \log m/\phi)$ returns a set $S' \supseteq S$ such that $G[V \setminus S']$ is $\phi/6$-expander, $vol_G(S') \leq vol_G(V)/2$ and $|E_G(S', V \setminus S')| = O(\phi \cdot m \log^2 m)$.*

Note that $V \setminus S$ being ϕ-expander in G is not equal to $G[V \setminus S]$ being ϕ-expander, since some cut $S' \subset V \setminus S$ might have many edges into S but not many edges into $V \setminus (S \cup S')$ and thus might be sparse. The trimming procedure above recovers the desired property.

In Algorithm 1, we give the pseudocode to implement the framework above. Given the guarantees of Theorem 5 and Theorem 6, we have that all sets added to *ExpanderDecomp* are expanders and eventually, it is a full partition. Further, from these guarantees, we can bound the recursion depth of our algorithm by $O(\log^3 m)$. Since all graphs at the same recursion level are vertex-disjoint, and the number of edges between sets in *Active* increases by at most $O(\phi \cdot \text{vol}_G(X_i))$ whenever we decompose a graph $G[X_i]$, the final expander decomposition is of quality $O(\log^3 m)$. And again, since every edge participates in at most $O(\log^3 m)$ invocations of $\textsc{CutMatchingGame}(\cdot)$ and $\textsc{Trimming}(\cdot)$, we have that the total runtime is $O(m \log^4 m/\phi)$. This concludes our review and proof sketch of Theorem 3.

Algorithm 1 ComputeExpanderDecomposition(G, ϕ).

```
1  Active ← {V}; ExpanderDecomp ← {}.
2  while Active ≠ ∅ do
3  |  Remove arbitrary set X from Active.
4  |  Run procedure CutMatchingGame(G[X], 6φ, 32 · c_0).
5  |  if CutMatchingGame(G[X], 6φ) returned a balanced sparse cut S
6  |  |  Active ← Active ∪ {S, X \ S}.
7  |  else if CutMatchingGame(G[X], 6φ) reports that G[X_i] is (6φ)-expander
8  |  |  Add X to ExpanderDecomp.
9  |  else // i.e. CutMatchingGame(G[X], 6φ) returns unbalanced sparse cut S
10 |  |  S' ← Trim(G, S, 6φ).
11 |  |  Add φ-expander V \ S' to ExpanderDecomp.
12 |  |  Add S' to Active.
13 return ExpanderDecomp.
```

2.1 Cut-Matching Algorithm

Let us next describe the cut-matching algorithm which yields Theorem 5. Here, we describe the algorithm and analysis for obtaining sparse cuts and expanders for sparsity instead of conductance. These two measures differ in the denominator where conductance compares to volume while sparsity compares to vertex count. The result for conductance can be obtained from the one for sparsity by a simple graph transformation.

▶ **Definition 7** (Sparsity). *For $G = (V, E)$, any $A \subseteq V$, any cut $\emptyset \subsetneq S \subsetneq A$, we define the* sparsity *by $\Psi_G(S, A) = \frac{|E_G(S, V \setminus S)|}{\min\{|S|, |A \setminus S|\}}$. We say A is a ψ-expander in G (w.r.t. sparsity) if no $\emptyset \subsetneq S \subsetneq A$ has $\Psi_G(S, A) < \psi$.*

We focus on sparsity, since this allows us to get much cleaner algorithms and proofs. Since we do not use conductance in the rest of this section, we often omit the explicit statement that we work with respect to sparsity (and use ψ instead of ϕ). We walk through the proof the following result in the remainder of the section where the last case combines scenarios 2 and 3 of Theorem 5.

▶ **Theorem 8** (Cut Matching Framework, see Theorem 4.1 in [22]). *Given input graph $G = (V, E)$, $\psi < 1/\log^2(m)$ and a constant $C > 0$. The algorithm CutMatchingGame(G, ψ, C) takes $O(m \log m/\psi)$ time and either reports*

1. *an $\Omega(1/\log^2 m)$-balanced $\Omega(1/\log^2 m)$-sparse cut S in G, or*
2. *a (possibly empty) set $S \subseteq V$, such that $V \setminus S$ is ψ-expander in G and either S is empty or it is $O(1/\log^2 m)$-sparse and has $|S| \leq n/(C \log^2 n)$.*

To recover guarantees on the conductance instead of sparsity, it then suffices to apply the algorithm presented in this section to the subdivision graph of G, defined below, instead of G directly.

▶ **Definition 9.** *Given input graph $G = (V, E)$, we denote by $G_E = (V_E, E_E)$ the graph obtained from replacing each edge $e = (u, v) \in E$ with edges $(u, x_e), (x_e, v)$ where x_e is a newly-added vertex.*

Applying Theorem 8 to G_E then yields Theorem 5 (we also need to slightly re-scale parameters by constant factors).

For most of the section, we describe the original algorithm in [22] that either outputs G being a ψ-expander or a $\tilde{O}(\psi)$-sparse cut. In Section 2.1.5, we describe the necessary adaptations to this algorithm to yield Theorem 8.

2.1.1 Sparsity via Graph Embeddings

The algorithm in Theorem 8 heavily exploits that a good ψ'-expander graph H can be used to prove that G is a ψ-expander (or not) by embedding H into G. In the former case, we also say that H is a *witness graph*. Let us formally establish this claim.

▶ **Definition 10.** *Given two graphs H, G over the same vertex set, we say that function $\mathcal{E}_{H\mapsto G}$ is a graph embedding of H into G if for every edge $e = (u, v) \in E(H)$, $\mathcal{E}_{H\mapsto G}(e)$ is a u-to-v path in G. We define the congestion of the embedding by $cong(\mathcal{E}_{H\mapsto G}) = \max_{e'\in E(G)} |\{e \in E(H) | e' \in \mathcal{E}_{H\mapsto G}(e)\}|$, i.e. the maximum number of occurrences of any edge in G on all embedding paths.*

▶ **Theorem 11** (Folklore). *Given a ψ'-expander A in H and graph embedding $\mathcal{E}_{H\mapsto G}$, then A is a ψ-expander in G for $\psi \geq \psi'/cong(\mathcal{E}_{H\mapsto G})$.*

Proof. For any cut S, then for every edge $e \in E_H(S, V \setminus S)$, the path $\mathcal{E}_{H\mapsto G}(e)$ must contain at least one edge from $E_G(S, V \setminus S)$. But no edge in G occurs on more than $cong(\mathcal{E}_{H\mapsto G})$ many such paths. Thus $|E_G(S, V \setminus S)| \geq |E_H(S, V \setminus S)|/cong(\mathcal{E}_{H\mapsto G}) \geq \psi' \cdot \min\{|S|, |V \setminus S|\}/cong(\mathcal{E}_{H\mapsto G})$ where we used that A is a ψ'-expander in H in the final inequality. ◀

2.1.2 Computing a Witness Graph and Embedding

Theorem 11 inspires the algorithmic idea of taking a graph H that we know to be a good expander, and embed it into G. While there are efficient and simple constructions of such expanders H, embedding them into G turns out to be quite challenging.

The cut-matching framework constructs H over $T = O(\log^2 m)$ iterations. In the t-th iteration, the algorithm either returns a sparse cut, or finds a new perfect matching M_t of the vertices V from K_n (the complete graph) along with an embedding $\mathcal{E}_{M_t\mapsto G}$ of M_t into G of congestion $O(1/\psi)$. Finally, it constructs the graph $H = \cup_t M_t$, i.e. as the union of the matchings, and takes the embedding $\mathcal{E}_{H\mapsto G} = \cup_t \mathcal{E}_{M_t\mapsto G}$. The crux of the cut-matching game is to choose the matchings $M_1, M_2, \ldots, M_T$ such that H is a $\frac{1}{2}$-expander and $cong(\mathcal{E}_{H\mapsto G}) = 1/(2\psi)$, and thus G is a ψ-expander by Theorem 11. Thus, in the t-th iteration, the matching M_t has to be carefully chosen to make $\cup_{i\leq t} M_i$ much closer to being an expander than $\cup_{i<t} M_i$. To achieve this, intuitively, S is chosen in every round to be an almost sparse cut in $\cup_{i<t} M_i$.

We give the pseudo-code of the cut-matching algorithm in Algorithm 2. Here, we assume that the number of vertices in G is even and implement FindBipartition$(\cdot)$ to always produce a bipartition into equally sized sets.

Then in each iteration, given the bipartition $(S, V \setminus S)$, the algorithm sets up a flow problem on $G[A]$ with demand $\mathbf{d} = -\mathbf{1}_S + \mathbf{1}_{V\setminus S}$ and uniform capacities $\mathbf{c} = \frac{1}{2\psi T} \cdot \mathbf{1}$. It then runs a max-flow algorithm to obtain a flow $\mathbf{f}$ that routes the maximum amount of flow from sources S to the sinks $V \setminus S$. If this flow $\mathbf{f}$ does not route $\mathbf{d}$, the algorithm further finds a min-cut $(X, V \setminus X)$ that is saturated as proof (i.e. all edges from X to $V \setminus X$ carry as much flow as possible). Note that the sum of demands on X is at most $|X|$, and the min-cut needs to route exactly the (positive) sum of demands over the cut, so the fact that it cannot be routed through $(X, V \setminus X)$ for our choice of capacities implies that the cut is $O(T\psi)$-sparse.

Algorithm 2 CutMatchingGame(G, ψ).

for $t = 1, 2, \ldots, T$ **do**
  $(S, V \setminus S) =$ FindBipartition$(M_1, M_2, \ldots, M_{t-1})$.
  Attempt to find a flow $\mathbf{f}$ that routes demand $\mathbf{d} = -\mathbf{1}_S + \mathbf{1}_{V \setminus S}$ on G with uniform capacities $\frac{1}{2\psi T}$.
  if *the flow problem is infeasible* **return** min-cut X (that certifies infeasibility).
  else
    Decompose flow $\mathbf{f}$ into flow-paths $P_1, P_2, \ldots, P_k$.
    Let M_t be the graph obtained by adding an edge $e = \{u, v\}$ for every uv-path P_i and set $\mathcal{E}_{M_t}(e) = P_i$.
return $H = \cup_t M_t$, $\mathcal{E}_{H \mapsto G} = \cup_t \mathcal{E}_{M_t \mapsto G}$.

Otherwise, i.e. if $\mathbf{f}$ routes the demand $\mathbf{d}$, the algorithm decomposes $\mathbf{f}$ into source-sink paths $P_1, P_2, \ldots, P_{n/2}$ and adds for each u-to-v-path P_i an edge (u, v) to M_t. The embedding $\mathcal{E}_{M_t \mapsto G}$ is defined naturally to consist of paths $P_1, P_2, \ldots, P_k$. Note that M_t is a perfect matching.

Saranurak and Wang [33] use a procedure called *Unit-Flow* [15] in place of max-flow. Unit-Flow is essentially height-constrained push-relabel with minimum label selection rule, i.e., vertices with label greater than height h need not be discharged, which leads to $O(mh)$ runtime. It can be shown that $h = 1/\psi \log(m)$ works, which leads to $O(m/(\log m \cdot \psi))$ runtime per cut-matching iteration.

2.1.3 Finding Bipartitions

It remains to give the implementation of procedure FindBipartition$(M_1, M_2, \ldots, M_{t-1})$ in Algorithm 2. The main idea behind the cut-matching algorithm and its analysis is to consider the lazy random walk suggested by the matchings $M_1, M_2, \ldots, M_t$ where in the i-th step of the random walk, each vertex u keeps half of its mass and sends the other half to its matching partner v in M_i, i.e. $\{u, v\} \in M_i$. Let $\mathbf{I}$ be the $n \times n$-identity matrix and $\mathbf{A}_{M_i}$ be the adjacency matrix of graph M_i. Let $\mathbf{P}_i = (\mathbf{I}/2 + \mathbf{A}_{M_i}/2) \circ (\mathbf{I}/2 + \mathbf{A}_{M_{i-1}}/2) \circ \cdots \circ (\mathbf{I}/2 + \mathbf{A}_{M_1}/2)$ be the i-th step lazy random walk matrix. We make the following observation.

▶ **Observation 12.** *We say that $\mathbf{P}_{t-1}$ is almost-mixing, if for every vertex $u, v \in V$, we have $\mathbf{1}_v^T \mathbf{P}_{t-1} \mathbf{1}_u \geq 1/2n$. If $\mathbf{P}_{t-1}$ is almost-mixing, then $H = M_1 \cup M_2 \cup \ldots \cup M_{t-1}$ is $\frac{1}{2}$-expander.*

Proof. Consider any cut S in H where w.l.o.g. $|S| \leq |V \setminus S|$. From almost-mixing and linearity, we have that $\mathbf{1}_{V \setminus S}^T \mathbf{P}_{t-1} \mathbf{1}_S \geq |V \setminus S|/(2n) \cdot |S| \geq |S|/4$, i.e. at least $|S|/4$ of the probability mass that is initially placed on S is on $V \setminus S$ after the $(t-1)$-step lazy random walk. But since each matrix $(\mathbf{I}/2 + \mathbf{A}_{M_i}/2)$ simulating one step of the random walk is doubly stochastic, we have that no vertex receives mass larger than 1. And since each edge (u, v) in $E_H(S, V \setminus S)$ only occurs in one matching M_i, and is then used to route at most $1/2$ of the mass at u to v, we have that each such edge can transport at most $1/2$ units of flow from S to $V \setminus S$ and thus, we can conclude that $|E_H(S, V \setminus S)| \geq |S|/2$. ◀

Finally, consider a random n-dimensional unit vector $\mathbf{r}$ orthogonal to $\mathbf{1}$ (i.e. $\mathbf{r}$ is a demand). If the walk $\mathbf{P}_{t-1}$ was already perfectly-mixing on V, i.e. $\mathbf{1}_v^T \mathbf{P}_{t-1} \mathbf{1}_u = 1/n$ for all $u, v \in V$, then we have $\mathbf{P}_{t-1}\mathbf{r} = \mathbf{0}$. However, if $\mathbf{P}_{t-1}$ is still far from perfectly-mixing, then intuitively, a lot of positive demand and negative demand should get stuck on two sides of a relatively balanced sparse cut in $M_1 \cup M_2 \cup \ldots \cup M_{t-1}$. This informs the following algorithm. Note that $\mathbf{P}_{t-1}\mathbf{r}$ can be computed in $O(tn)$ time.

Algorithm 3 FindBipartition$(A, M_1, M_2, \ldots, M_{t-1})$.

1 Take $\mathbf{u} = \mathbf{P}_{t-1}\mathbf{r}$ for n-dimensional unit vector $\mathbf{r}$ orthogonal to $\mathbf{1}$.
2 Let S be the set of $n/2$ vertices that receive the smallest value in $\mathbf{u}$ (ties broken arbitrarily).
3 **return** $(S, V \setminus S)$.

2.1.4 Analyzing the Cut-Matching Algorithm

Finally, let us sketch the analysis of the above algorithm (assuming it does not terminate with a sparse cut).

Let us start by defining the potential $\mathcal{P}_t = \sum_{u \in V} \|\mathbf{1}_u^T \mathbf{P}_t - \mathbf{1}/n\|_2^2$ measuring the gap from mixing perfectly. Note that if $\mathbf{P}_t$ is not almost-mixing, then for some $u, v \in V$, we have $|\mathbf{1}_u^T \mathbf{P}_t \mathbf{1}_v - 1/n| > 1/2n$ and thus $\mathcal{P}_t > 1/4n^2$. Crucially, [22] shows that for every t, we have $\mathbb{E}[\mathcal{P}_{t-1} - \mathcal{P}_t] = \Omega(\mathcal{P}_{t-1}/\log n)$ w.p. at least $1 - n^{-10}$. Thus, for some $T = \Theta(\log^2 m)$, we have $\mathcal{P}_T < 1/4n^2$ w.p. at least $1 - n^{-5}$ and thus H is $\frac{1}{2}$-expander w.r.t. sparsity.

Let us sketch the proof that $\mathbb{E}[\mathcal{P}_{t-1} - \mathcal{P}_t] = \Omega(\mathcal{P}_{t-1}/\log n)$. We define $\mathbf{p}_u^{OLD} = \mathbf{1}_u^T \mathbf{P}_{t-1}$ and $\mathbf{p}_u^{NEW} = \mathbf{1}_u^T \mathbf{P}_{t-1}$, the vectors where the v-th component gives the probability that the $t-1$-step walk/ t-step walk starting in v ends in u. Note that for $(u, v) \in M_t$, we have $\mathbf{p}_u^{NEW} = \mathbf{p}_v^{NEW} = \frac{1}{2}(\mathbf{p}_u^{OLD} + \mathbf{p}_v^{OLD})$. Thus, by rather straightforward calculations, one obtains $\mathcal{P}_{t-1} - \mathcal{P}_t = \frac{1}{2}\sum_{(u,v)\in M_t} \|\mathbf{p}_u^{OLD} - \mathbf{p}_v^{OLD}\|_2^2$. We use standard facts about random projections to relate this potential drop to the projected random vector $\mathbf{u}$.

▶ **Theorem 13** (see [37]). *If $\mathbf{y}$ is a vector of length ℓ in $\mathbb{R}^d$, and $\mathbf{r}$ a random unit vector in $\mathbb{R}^d$, then $\mathbb{E}[(\mathbf{y}^T\mathbf{r})^2] = \ell^2/d$ and for $x \le d/16$, we have $\mathbb{P}[(\mathbf{y}^T\mathbf{r})^2 \ge x \cdot \ell^2/d] \le e^{-x/4}$ and $\mathbb{P}[(\mathbf{y}^T\mathbf{r})^2 \le \frac{1}{x} \cdot \ell^2/d] \le e^{-x/4}$.*

We can apply the second property of this theorem with $x = 64 \log m$, to obtain that for every $(u, v) \in M_t$, w.p at least $1 - m^{-11}$ we have $\mathbf{u}_u - \mathbf{u}_v = (\mathbf{p}_u^{OLD} - \mathbf{p}_v^{OLD})^T\mathbf{r} < 64 \log m \cdot \|\mathbf{p}_u^{OLD} - \mathbf{p}_v^{OLD}\|_2^2/(n-1)$ where we use that we work orthogonally to $\mathbf{1}$, i.e. both $\mathbf{r}$ and $(\mathbf{p}_u^{OLD} - \mathbf{p}_v^{OLD})$ are orthogonal to $\mathbf{1}$, and therefore in $(n-1)$-dimensional space. Thus,

$$\mathcal{P}_{t-1} - \mathcal{P}_t = \sum_{(u,v)\in M_t} \|\mathbf{p}_u^{OLD} - \mathbf{p}_v^{OLD}\|_2^2 > \frac{n-1}{64 \log m} \cdot \sum_{(u,v)\in M_t} \|\mathbf{u}_u - \mathbf{u}_v\|_2^2. \tag{1}$$

Again, via simple algebraic manipulations, we obtain $\sum_{(u,v)\in M_t} \|\mathbf{u}_u - \mathbf{u}_v\|_2^2 \ge \sum_{u \in V} \mathbf{u}_u^2$ for M_t being a perfect matching. Finally, the analysis uses the insight that

$$\mathbb{E}\left[\sum_{u \in V} \mathbf{u}_u^2\right] = \sum_{u \in V} \mathbb{E}[((\mathbf{p}_u^{OLD})^T\mathbf{r})^2] = \sum_{u \in V} \mathbb{E}[(\mathbf{p}_u^{OLD} - \mathbf{1}/n)^T\mathbf{r})^2] = \frac{\mathcal{P}_{t-1}}{n-1} \tag{2}$$

where we used $\mathbf{1}^T\mathbf{r} = 0$ in the second equality, and the first fact from Theorem 13 in the last equality. Combining Equations 1 and 2 carefully, we obtain the claimed result.

2.1.5 The Cut-Matching Game with Balanced Cuts

Finally, to obtain Theorem 8, Algorithm 2 is adapted to work on a set of active vertices $A \subseteq V$ and with induced graph $G[A]$. Initially $A = V$. Then, whenever the flow problem in Line 3 of Algorithm 2 is infeasible, the algorithm takes the min-cut X, and removes X from A. If A becomes a balanced cut, we output it as a balanced sparse cut. Otherwise, the

algorithm finds a flow $\mathbf{f}$ that routes most of the demand in the flow problem, and this flow is then used to build a matching M_t as described in the else-case starting in Line 5. Note that M_t might only be a partial matching. Finally, this adaptation necessitates a slightly more intricate implementation for the procedure FindBipartition. We refer the reader to [31, 33] for a detailed description.

2.2 The Trimming Algorithm

Finally, we discuss the algorithm behind Theorem 6. First, consider the following insight.

▶ **Lemma 14.** *Let A be ϕ-expander in G. Consider the flow problem on $G[A]$ where each vertex $v \in A$ has demand $\mathbf{d}_v = |E_G(v, V \setminus A)| \cdot \frac{2}{\phi} - \deg_G(v)$ and all edges have capacity $2/\phi$. If the flow problem has a feasible solution $\mathbf{f}$ then $G[A]$ is $\phi/6$-expander.*

Proof. If $G[A]$ is not a $\phi/6$-expander, then there is some cut $\emptyset \subsetneq S' \subsetneq A$ such that $\Phi_{G[A]}(S') < \phi/6$. But since A is ϕ-expander in G, this implies that $|E_G(S', S)| > \frac{5}{6} \cdot \phi \mathrm{vol}_G(S')$.

Thus, we have that $\sum_{v \in S'} \mathbf{d}_v \geq \frac{5}{6} \cdot \phi \mathrm{vol}_G(S') \cdot \frac{2}{\phi} - \mathrm{vol}_G(S') > \frac{4}{6} \mathrm{vol}_G(S')$. But from existence of $\mathbf{f}$, $\mathbf{d}$ is routable and thus $\frac{4}{6}\mathrm{vol}_G(S')$ units of flow are routed through the cut $E_{G[A]}(S', A \setminus S')$. But since each edge has capacity $2/\phi$, we have that this cut must contain at least $\frac{2}{6}\phi \cdot \mathrm{vol}_G(S')$ edges, contradicting that S' is $\phi/6$-sparse in $G[A]$. ◀

While this gives a concrete way to certify that $G[A]$ is ϕ-expander, we also need a strategy to proceed when the flow is not routable. In this case, we find a maximum flow $\mathbf{f}$ and a min-cut $(X, A \setminus X)$. Now Lemma 14 shows that $G[A \setminus X]$ is a $\phi/6$-expander. That is because the flow problem in Lemma 14 on $A' = A \setminus X$ asks to route $2/\phi$ units of flow into $G[A']$ via every edge from outside A'. But such a flow is given by $\mathbf{f}[A']$, i.e. by inducing $\mathbf{f}$ on A' as each edge in the min-cut is saturated and thus carries exactly $2/\phi$ flow while each edge in $(A \setminus X, X)$ carries no flow.

Let us finalize a proof sketch of Theorem 6. The case where $G[A]$ is $\phi/6$-expander is trivial. Let us, therefore, analyze the second scenario where $S' = S \cup X$. From the max-flow min-cut theorem, we have that no vertex $x \in X$ has sink capacity left, i.e. $\mathbf{d}_v - \sum_{(v,x) \in E} \mathbf{f}_{(v,x)} \geq 0$ (where we define $\mathbf{f}_{(v,x)} = -\mathbf{f}_{(x,v)}$). But this implies that $\mathrm{vol}_G(X) \leq |E_G(A, V \setminus A)| \cdot \frac{2}{\phi}$ since on any larger set the sum of demands is negative. From the guarantees in Theorem 5, it follows that $\mathrm{vol}_G(S') = \mathrm{vol}_G(S) + \mathrm{vol}_G(X) \leq m$. Further, we have that $|E_G(S', V \setminus S')| = |E_G(S, V \setminus S')| + |E_G(X, V \setminus S')| \leq |E_G(S, V \setminus S)|$ since every edge in $E_G(X, V \setminus S')$ is saturated in the above flow problem, but there is only $|E_G(A, V \setminus A)| \cdot \frac{2}{\phi}$ source overall.

Finally, we note that again, [33] does not run a max-flow algorithm, but Unit-Flow with $h = 40 \log(2m)\phi^{-1}$. While this forces them to recurse multiple times and thus shrink A over time, they recover the runtime by observing that each iteration can be *warm-started* with the flow and labeling from the previous iteration which allows them to obtain total runtime bound of $O(m \log m/\phi)$.

2.3 Arvestad's Implementation

The implementation of Arvestad [6] largely follows the expander decomposition procedure outlined by Saranurak and Wang [33] directly, but makes two improvements. First, they modified the bipartitioning procedure of [31] to achieve faster potential reduction in order to reduce the number of cut-matching iterations and thus alleviate issues with imprecision in the edge capacities. They report that this does not improve running times. Additionally, in case a sparse cut is found, they keep running cut-matching to improve its balance, which leads to better recursion depth.

3 Practical Optimizations

In this section, we propose five optimizations that improve performance in practice, namely sparse cut heuristics, a dynamic stopping criterion for cut-matching, fast few-hop flow routing, partial flow routing and trimming with maximum flow instead of Unit-Flow. In particular, the sparse cut heuristics lead to huge savings. An important design goal is to guarantee the same theoretical time complexity and approximation as Saranurak and Wang.

3.1 Sparse Cut Heuristics

Our most simple and most effective technique is to try sparse cut heuristics that have a much worse approximation guarantee (or none at all) but are much faster to compute than the cut-matching game. If the cut S resulting from a heuristic happens to satisfy the required approximation guarantee $\Phi_G(S) \leq \log^2(m) \cdot \phi$, we simply take this cut and circumvent the cut-matching game completely, resulting in substantial running time savings. If the approximation guarantee is not satisfied, we proceed to run cut-matching, having wasted some computation time on the heuristics. To minimize runtime losses in this case, we choose three heuristics that are extremely fast in practice with near-linear time complexity: Metis [19], a balanced graph partitioning algorithm, PageRank-Nibble [4], and a local search for conductance that explores greedy vertex moves. In real-world graphs one has to make several sparse cuts until arriving at a small enough component that could reasonably be an expander. Moreover, most of the time is spent on these early sparse cuts when the graphs are large, whereas proving expansion for small graphs should be much faster, rendering this an effective technique.

3.2 Dynamic Stopping Criterion

In Algorithm 2, we perform up to $T = O(\log^2 n)$ iterations of cut-matching. In previous work, T was determined analytically/empirically. However, from Observation 12, we know that Algorithm 2 can be terminated after the t-th iteration if the potential $\mathcal{P}_t$ dropped below $1/4n^2$. However, evaluating the potential $\mathcal{P}_t$ explicitly is costly as it requires evaluating a matrix-vector product of a dense matrix and thus takes $O(n^2)$ time.

In our algorithm, we exploit that the random projected vector $\mathbf{u} = \mathbf{P}_{t-1}\mathbf{r}$ computed at the beginning of the t-th iteration of the algorithm is an unbiased estimator of $\mathcal{P}_{t-1}$, i.e. $\mathbb{E}\left[\|\mathbf{u}\|_2^2\right] = \frac{\mathcal{P}_{t-1}}{n-1}$ (see Equation 2). Moreover, $\|\mathbf{u}\|_2^2$ is extremely concentrated around the mean. With probability at least $1 - 2/e$, we have that the length of $\mathbf{u}$ is within a factor 4 of the mean (and probability $1 - n^{-10}$ that is within factor $O(\log n)$). Further, $\mathbf{u}$ is cheap to compute.

Note that if $\|\mathbf{u}\|_2^2$ is within a factor of 4 of its expectation, then we can terminate the cut matching game if $\|\mathbf{u}\|_2^2 < 1/(16n^3)$ as this ensures $\mathcal{P}_{t-1} < 1/(4n^2)$ which is our termination criterion (see Section 2.1.4). But note that if $\|\mathbf{u}\|_2^2$ is not within a factor of 4 of its mean, we risk falsely certifying expansion. To remedy this situation, we sample $\rho = 20$ independent vectors $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_\rho$, and use the largest length as our estimate for $\mathcal{P}_{t-1}$. Effectively, this is probability boosting to increase confidence that $\|\mathbf{u}\|_2^2$ is truly within a factor 4 of $\frac{\mathcal{P}_{t-1}}{n-1}$ from $1 - 2/e$ to $1 - (2/e)^\rho$. Moreover, the number of iterations needed to converge to $\|\mathbf{u}\|_2^2 < 1/(16n^3)$ is typically reduced by about half, compared to the constant T.

Further, if our algorithm determines that it has to take another cut-matching step t, it can use as vector $\mathbf{u}$ the vector $\mathbf{u}_i$ of largest lengths. This ensures that the decrease in potential in this step is maximized as the decrease is lower bounded by $\|\mathbf{u}\|_2^2/(64 \log m)$ (see Equations 1 and 2).

3.3 Fast Few-Hop Flow Routing

As we show later in Figure 1 in Section 4.2, flow computation with Unit-Flow is the main bottleneck, constituting around 80% of the total running time. Due to the way the flow problem is set up, with roughly half of the subdivision vertices as sources and half as sinks, most of the flow can be routed along paths with few hops. This is particularly the case when we are able to route all demands, i.e., the network is not heavily congested, in which case a very large fraction of the flow can be routed in few hops. This is even exacerbated by the use of sparse cut heuristics, where cut-matching is only run if we suspect there is no sparse cut. We accelerate the flow computation by performing two iterations of the blocking flow computation in Dinitz [14] algorithm, which constitutes pushing flow along paths with two and (typically) four hops, respectively.

This optimization eliminates most of the push work from Unit-Flow push-relabel, however Unit-Flow still has to perform its excruciatingly slow incremental relabeling. Therefore, we complement our optimization with a run of global relabeling, before we route the small remainder of the flow with Unit-Flow.

3.4 Partial Supply Routing

In addition to speeding up flow routing, we can also route a large subset but not all of the demands in every iteration. This is helpful when the network is heavily congested and a lot of flow is being pushed back and forth near the end of the computation. With our *Partial Supply Routing* optimization, we only route a $1 - \frac{1}{4T \cdot C \log^2 n}$ fraction of the injected flow in each such round. This means we are missing roughly a $\frac{1}{4T \cdot C \log^2 n}$ fraction of edges in each matching, which we fix by adding arbitrary *fake matching edges* between unmatched subdivision vertices. After cut-matching has converged and no sparse cut was found, we fix the error introduced by these fake matches by running a trimming flow problem. In our experiments, we see that this strategy yields some improvement especially on the largest graph instances.

We give a formal proof that our trimming procedure recovers the necessary guarantees using a refinement of techniques presented in Section 2.2 as suggested in [16]. Here, we let H be a $\frac{1}{2}$-expander w.r.t. sparsity found in a cut-matching algorithm and let $F \subseteq E(H)$ be the fake edges added throughout the process. Let $\mathcal{E}_{H \setminus F \mapsto G}$ be the embedding of the non-fake edges of G into H. Define $\psi = 1/2cong(\mathcal{E}_{H \setminus F \mapsto G})$.

▶ **Lemma 15.** *Let H, G and $\mathcal{E}_{H \setminus F \mapsto G}$ be as above. Consider the flow problem on G where each vertex $v \in V$ has demand $\mathbf{d}_v = 4 \cdot \deg_F(v) - 1$ and all edges receive capacity $3/\psi$. If the flow problem has a feasible solution $\mathbf{f}$ then G is a $\psi/6$-expander.*

Proof. Consider any cut $S \subseteq V$ with $|S| \leq |V \setminus S|$ (by symmetry it suffices only to consider such cuts). If $|E_F(S, V \setminus S)| < \frac{5}{6}|E_H(S, V \setminus S)|$, then the proof follows the line of reasoning given in Theorem 11. Since H is $\frac{1}{2}$-expander, this implies $\sum_{v \in S} \mathbf{d}_v \geq \frac{20}{12}|S| - |S| > |S|/2$. Thus, if there is a feasible flow $\mathbf{f}$, then the demand is routed through the cut $(S, V \setminus S)$ and thus we have at least $\psi|S|/6$ edges in $E_G(S, V \setminus S)$, as desired. ◀

Finally, note that if no feasible flow $\mathbf{f}$ can be found, then we can find a min-cut $(X, V \setminus X)$ that certifies infeasibility. But note that the demand $\sum_{v \in X} \mathbf{d}_v < 4T|X|$ because H has every vertex of degree T (each perfect matching M_t increases the degree by 1). But this implies that the number of edges in $(X, V \setminus X)$ is at most $12\psi T|X|$ and thus the cut is $O(T\psi)$-sparse w.r.t. sparsity (if $|X| \leq |V \setminus X|$). Since at most $\frac{1}{4T \cdot C \log^2 n}$-fraction of edges in H are fake

edges and the degree of every vertex is T, we have that fake edges add at most $n/(C \log^2 n)$ units of flow in the above flow problem, and thus X is of size at most $n/(C \log^2 n)$. We can therefore output X as scenario 1 in Theorem 8.

Finally, we prove that $G[V \setminus X]$ is a $\phi/8$-expander (we thus decrease the quality slightly). We can construct a similar flow problem as above where demands are $d_v = 4 \cdot (|E_H(v, X)| + \deg_F(v)) - 1$ for $v \in V \setminus X$ with capacities $8/\psi$. It is not hard to adapt Lemma 15 to this case. However, this time, there exists a feasible flow $\mathbf{f}$. To see this, observe that for every edge in $e \in E_H(v, X)$, we can route the 4 units of demand caused by it along the embedding path $\mathcal{E}_{H \setminus F \mapsto G}(e) \cap G[V \setminus X]$. This routes it to one of the edges in the cut $(X, V \setminus X)$, and none of these edges gets more than $4/\psi$ units flow as congestion of $\mathcal{E}_{H \setminus F \mapsto G} \leq \psi$. But this implies that the flow from the first flow problem instance (the one that produced $(X, V \setminus X)$) can be used to route all of this flow to sinks with capacity $\frac{4}{3} \cdot 3/\psi = 4/\psi$. All additional demand was routed by the former flow as well. Thus, we have total congestion $8/\psi$, half from the flow along embedding paths, and half from the old flow slightly scaled up (by $\frac{4}{3}$).

3.5 Trimming with Max Flow

As noted in Section 2.2, the trimming step could also be implemented with a single invocation of maximum flow instead of repeatedly pruning with height-constrained min-label push-relabel, to the detriment of theoretical efficiency. Using push-relabel with FIFO selection rule and global relabeling, we observe that in practice, the opposite is the case. On our largest benchmark instance we were able to reduce the running time for the trimming step from 26.5 hours to 0.7s.

Note that in most cases trimming is not needed, as the $\Omega(1/\log^2 m)$ balance enforced is small in practice, but on the instances where it is needed, trimming is extremely slow. This is due to the large height value – for example $h = 40\phi^{-1} \ln 2m = 722560$ for our largest benchmark instance. The theoretically near-linear runtime of $\tilde{O}(mh)$ is thus highly impractical. We configure trimming to run FIFO push-relabel until the amount of work performed exceeds the work that Unit-Flow would need. Then we switch to Unit-Flow, thus increasing the theoretical work by at most a factor two. If the flow is maximal before we need to switch, we are finished with trimming. Otherwise, Unit-Flow continues with the current flow assignment, as push-relabel can be warm-started.

4 Experiments

In this section, we present our experimental evaluation. First, we analyze the running time of the different components of the expander decomposition algorithms. Then, we enable our optimizations step by step to demonstrate how each of them improves the running time.

4.1 Experimental Setup

All experiments are performed on a 2x8-core Intel Xeon Gold 6144 Skylake CPU, clocked at 3.5GHz with 24.75MB L3 cache and 192GB DDR4 RAM (2666 MHz). Each run is single-threaded. We run different graphs and configurations simultaneously to save machine time. The graph benchmark set is split into two parts. First, we use the same 64 graphs as Arvestad [6] (which are small with $m \leq 10^5$), referred to as set A. These are obtained from the SuiteSparse Matrix Collection [23, 13]. Second, we use 10 larger graphs, but still considered medium-size ($m \leq 3.5 \cdot 10^7$), which are referred to as set B. Set B is obtained from [25], except for RMAT which is generated using the R-MAT model from [7]. Table 1 lists the graphs of

Table 1 Benchmark set B.

Graph	n	m	Graph	n	m
soc-Epinions1	75,879	405,740	loc-gowalla	196,591	950,327
soc-Slashdot0902	82,168	504,230	com-youtube	1,157,828	2,987,624
coAuthorsCiteseer	227,320	814,134	amazon-2008	735,323	3,523,472
web-NotreDame	325,729	1,090,108	as-skitter	1,696,415	11,095,298
RMAT	65,484	8,388,607	com-livejournal	4,036,538	34,681,189

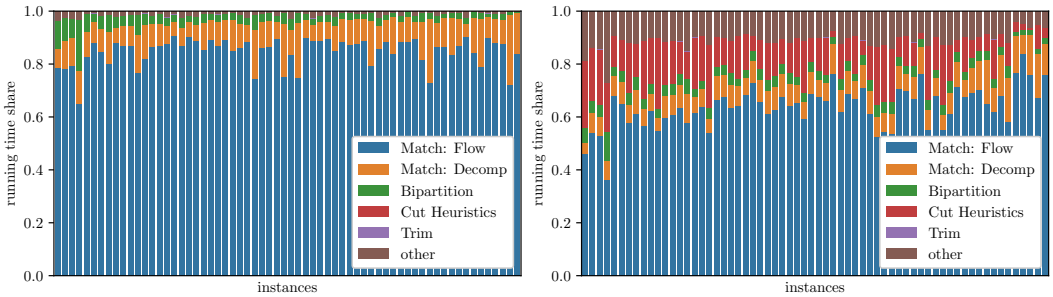

Figure 1 Relative running time of each algorithmic component on benchmark set A, where the instances are sorted by total running time; `Arv` on the left, `+Cut` on the right.

set B. All experiments use $\phi = 0.001$. Our implementation extends the C++ implementation of Arvestad (`https://github.com/isaacarvestad/edc`) with the optimizations presented in Section 3. Our implementation is available at `https://github.com/larsgottesbueren/practical-expander-decomposition`. The configuration of existing parameters was left as tuned in their work. We denote the baseline configuration by `Arv`, adding our sparse cut heuristics by `+Cut`, the dynamic stopping criterion by `+Dyn`, using max-flow in trimming by `+MFT`, few-hop routing by `+FHR` and partial supply routing by `+PSR`.

4.2 Breaking Down Running Times

In Figure 1 we break the running time of expander decomposition on the instances of the benchmark set A down into its individual components: Bipartition (Algorithm 3), Trim (Theorem 6), cut heuristics, and matching, which we separate into flow computation and flow decomposition. The part of the total running time not accounted for by these components is shown as "other", which includes parts such as computing connected components to identify recursive sub-problems, resetting data structures, reorganizing data in different formats and logging statistics. For each instance of benchmark set A (x-axis) we show the fraction of the total running time that each component takes in a stacked bar plot. The instances are sorted by total running time.

The left plot shows the breakdown for the baseline configuration `Arv`. Flow computation for matching constitutes around 80% of the total running time, and flow decomposition another 10%. Notably, expander trimming is never used on the graphs in benchmark set A (it is used on set B). These results indicate that the best avenue for optimization are to speed up the flow computation or to bypass it when possible. Our `+Cut` optimization targets the latter. Its runtime breakdown shown on the right of Figure 1 shows that the sparse cut heuristics only make up a small fraction of the running time (10 - 20%). Thus, in the cases where they fail to produce a sufficiently sparse cut, the penalty on the overall running time is small.

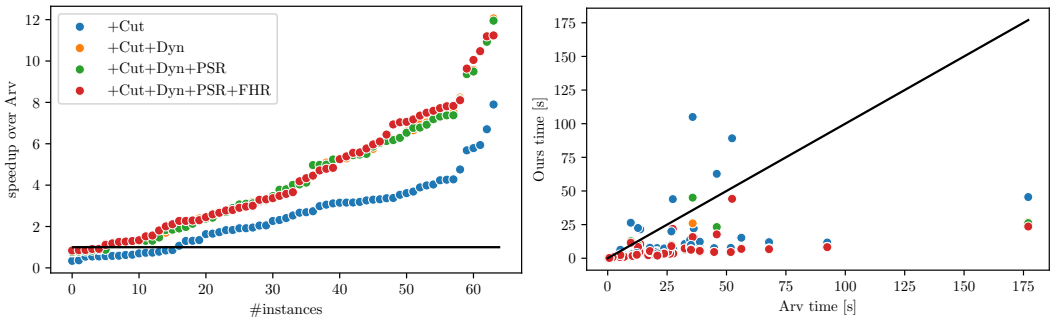

Figure 2 Left: speedup of our configurations over `Arv` (median of 10 runs) with $\rho = 20$ on benchmark set A, where the instances are sorted by speed up. Right: absolute per-instance running times (in seconds, median of 10 runs).

4.3 Evaluating Optimizations on Small Graphs

Next, we test how our optimizations improve the running time on benchmark set A. We perform 10 repetitions with different random seeds and pick the median running time. Recall that we use ρ to denote the number of projected flow vectors used in cut-matching. We use $\rho = 20$ for `Arv` and our configurations, but we note that on these instances `Arv` with $\rho = 1$ has similar running time. The scatter plot in Figure 2 (left) shows the speedup (y-axis) of our versions over `Arv`, sorted by increasing speedup on the different instances (x-axis). We observe that `+Cut` and `+Dyn` in combination offer substantial speedups of up to a factor of 12 and a median speedup of a factor of 3.81. On 59/64 instances `+Cut+Dyn` is faster than `Arv`, with the smallest speedup being 0.75. On its own, `+Cut` is faster on 48/64 instances, with the smallest speedup being 0.34. Notably, the other optimizations do not have an influence on running time here. For `+MFT` this is expected since none of these instances require trimming. Furthermore, for `+PSR` the number of fake matches we can allow is too small on benchmark set A. This optimization becomes important on larger instances as we show later. To also give an impression of the scale, Figure 2 (right) shows absolute running times, with `Arv` on the x-axis and our optimizations on the y-axis.

4.4 Evaluating Optimizations on Medium-Sized Graphs

Table 2 Running times on the benchmark B with optimizations enabled one after another. The best result in each row is marked in bold. Runs exceeding the two day time-limit are marked with ✻.

	time [s]						
Graph	`Arv` $\rho = 1$	$\rho = 20$	`+Cut`	`+Dyn`	`+MFT`	`+FHR`	`+PSR`
soc-Epinions1	745	1180	1220	679	660	**465**	576
soc-Slashdot0902	760	1114	1030	544	547	**330**	522
coAuthorsCiteseer	$2.77 \cdot 10^4$	1800	150	109	110	**105**	108
web-NotreDame	3530	2450	853	565	555	416	**409**
RMAT	9900	$2.32 \cdot 10^4$	$2.23 \cdot 10^4$	6530	6580	**2240**	2680
loc-gowalla	$4.59 \cdot 10^4$	6110	3960	2430	2460	**2170**	2480
com-youtube	✻	$9.04 \cdot 10^4$	$2.93 \cdot 10^4$	$1.93 \cdot 10^4$	$\mathbf{1.9 \cdot 10^4}$	$2.21 \cdot 10^4$	$1.91 \cdot 10^4$
amazon-2008	✻	$1.21 \cdot 10^4$	1360	943	925	**851**	912
as-skitter	✻	✻	$1.24 \cdot 10^5$	$8.14 \cdot 10^4$	$8.19 \cdot 10^4$	$7.44 \cdot 10^4$	$\mathbf{6.62 \cdot 10^4}$
com-livejournal	✻	✻	✻	✻(2.8d)	$1.5 \cdot 10^5$	$1.22 \cdot 10^5$	$\mathbf{1.02 \cdot 10^5}$

Table 3 Edge cut sizes on benchmark set B. Timeouts are marked with ✻. Note that the cut-sizes for `Arv` with $\rho = 1$ are not representative, as this version falsely claims expansion too early in the recursion. We see that the cut sizes for the other configurations are largely similar. Two exceptions are `com-youtube` and `amazon-2008`; however running additional seeds (not reported here) reveals that `Arv` with $\rho = 20$ sometimes also finds cuts in the order of 450000 and 600000, respectively.

	edge cut						
Graph	`Arv` $\rho = 1$	$\rho = 20$	`+Cut`	`+Dyn`	`+MFT`	`+FHR`	`+PSR`
soc-Epinions1	181	206	216	216	216	204	220
soc-Slashdot0902	30	48	48	48	48	39	47
coAuthorsCiteseer	15855	101427	115691	115161	115161	115019	115306
web-NotreDame	14710	54768	73670	72882	72882	73614	72898
RMAT	0	0	0	0	0	0	0
loc-gowalla	1275	39362	37879	37903	37903	37867	37886
com-youtube	✻	332638	459205	459118	459118	460491	455528
amazon-2008	✻	483325	639674	644054	640054	634426	636118
as-skitter	✻	✻	2229142	2205262	2205262	2210590	2210912
com-livejournal	✻	✻	✻	✻(8593577)	8593577	8619813	8611120

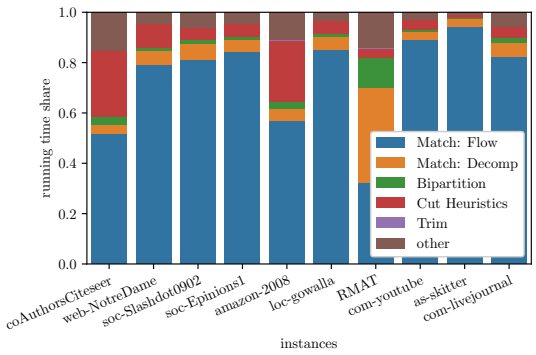

Figure 3 Relative fractional running time of algorithmic components with all optimizations on benchmark set B.

After optimizing running time on the small graphs, we want to push the envelope and compute expander decompositions on the million-scale graphs of benchmark set B. Due to high running times, we run each configuration with one seed only. As opposed to the previous section, we show results for Arv with $\rho = 1$ and $\rho = 20$, as we consider $\rho = 1$ the baseline. Note that `Arv` with $\rho = 1$ tends to find too small cuts as it falsely certifies expansion due to mixing flow vectors too quickly, see Table 3. We fix this by using $\rho = 20$ which acts as success probability boosting, as discussed in Section 3.2. The results for running times are shown in Table 2. We observe that our optimizations again consistently achieve good speedups, with the largest speedup being 263.8 over `Arv` with $\rho = 1$ on `coAuthorsCiteseer` and > 203 on `amazon-2008`. Moreover, `Arv` with $\rho = 1$ times out on the four largest instances, and $\rho = 20$ still times out on the two largest graphs `as-skitter` and `com-livejournal`, whereas our fastest configuration takes around one day. We also ran `Arv` on `com-livejournal` for three weeks, at which point not a single cluster was finalized.

On `com-livejournal` 94.3% of runtime spent in cut-matching went towards finding sparse cuts, not expanders. On the other hand, the fraction of the total runtime attributed to sparse cut heuristics is only around 5%. Moreover, as shown in Figure 3 cut-matching is

still the biggest bottleneck, even more so on large graphs. A good next direction is thus to expand the portfolio with more aggressive and potentially slower cut-heuristics, that will let us bypass cut-matching more often.

5 Conclusion

In this paper we explored the algorithm engineering space of algorithms for computing the expander decomposition of graphs. We proposed practical optimizations that speed up the Saranurak-Wang framework by up to two orders of magnitude, and that allow processing graphs with tens of millions of edges within reasonable time. We believe that a fast practical expander decomposition algorithm is an important milestone in making recent algorithmic breakthroughs for several fundamental problems efficient and scalable in practice.

Future directions in this line of work include both implementing the algorithms that use expander decomposition as a core component (e.g., max flow, min-cost flow, global min-cut, etc), as well as further speeding up the computation of expander decomposition. Regarding the latter, apart from parallelization, our work indicates that exploring stronger cut heuristics and further exploiting the flow problem structure is very promising to obtain further significant speedups.

References

1 Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Apmf< apsp? gomory-hu tree for unweighted graphs in almost-quadratic time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1135–1146. IEEE, 2022.

2 Amir Abboud, Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. All-pairs max-flow is no harder than single-pair max-flow: Gomory-hu trees in almost-linear time. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2204–2212. IEEE, 2023.

3 Daniel Agassy, Dani Dorfman, and Haim Kaplan. Expander decomposition with fewer inter-cluster edges using a spectral cut player. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany*, volume 261 of *LIPIcs*, pages 9:1–9:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ICALP.2023.9.

4 Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486. IEEE, 2006.

5 Alexandr Andoni, Jiecao Chen, Robert Krauthgamer, Bo Qin, David P Woodruff, and Qin Zhang. On sketching quadratic forms. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 311–319, 2016.

6 Isaac Arvestad. Near-linear Time Expander Decomposition in Practice. Master's thesis, KTH Royal Institute of Technology, 2022.

7 Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

8 Li Chen, Rasmus Kyng, Yang P Liu, Simon Meierhans, and Maximilian Probst Gutenberg. Almost-linear time algorithms for incremental graphs: Cycle detection, sccs, *s-t* shortest path, and minimum-cost flow. *to appear at STOC'24*, 2024.

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

10 Timothy Chu, Yu Gao, Richard Peng, Sushant Sachdeva, Saurabh Sawlani, and Junxing Wang. Graph sparsification, spectral sketches, and faster resistance computation via short cycle decompositions. *SIAM Journal on Computing*, pages FOCS18–85, 2020.

11 Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1158–1167. IEEE, 2020.

12 Michael B Cohen, Jonathan Kelner, John Peebles, Richard Peng, Anup B Rao, Aaron Sidford, and Adrian Vladu. Almost-linear-time algorithms for markov chains and new spectral primitives for directed graphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 410–419, 2017.

13 Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

14 Yefim Dinitz. Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Mathematics-Doklady*, 11(5):1277–1280, September 1970.

15 Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020.

16 Yiding Hua, Rasmus Kyng, Maximilian Probst Gutenberg, and Zihang Wu. Maintaining expander decompositions via sparse cuts. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 48–69. SIAM, 2023.

17 Arun Jambulapati and Aaron Sidford. Efficient $\tilde{O}(n/\varepsilon)$ spectral sketches for the laplacian and its pseudoinverse. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2487–2503. SIAM, 2018.

18 Ravi Kannan, Santosh Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM (JACM)*, 51(3):497–515, 2004.

19 G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

20 Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *Journal of the ACM (JACM)*, 66(1):1–50, 2018.

21 Jonathan A Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 217–226. SIAM, 2014.

22 Rohit Khandekar, Satish Rao, and Umesh Vazirani. Graph partitioning using single commodity flows. *Journal of the ACM (JACM)*, 56(4):1–15, 2009.

23 Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

24 Rasmus Kyng, Simon Meierhans, and Maximilian Probst. Derandomizing directed random walks in almost-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 407–418. IEEE, 2022.

25 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

26 Jason Li. Deterministic mincut in almost-linear time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 384–395, 2021.

27 Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 317–329, 2021.

28 Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. A nearly optimal all-pairs mincuts algorithm in simple graphs. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1124–1134. IEEE, 2022.

29 Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o (n1/2-ε)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1122–1129, 2017.

30 Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 950–961. IEEE, 2017.

31 Harald Räcke, Chintan Shah, and Hanjo Täubig. Computing cut-based hierarchical decompositions in almost linear time. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 227–238. SIAM, 2014.

32 Thatchaphol Saranurak. A simple deterministic algorithm for edge connectivity. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 80–85. SIAM, 2021.

33 Thatchaphol Saranurak and Di Wang. Expander Decomposition and Pruning: Faster, Stronger, and Simpler. In *SODA 2019*, pages 2616–2635. SIAM, 2019. doi:10.1137/1.9781611975482.162.

34 Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90, 2004.

35 Aurelio L Sulser and Maximilian Probst Gutenberg. A simple and near-optimal algorithm for directed expander decompositions. *arXiv preprint arXiv:2403.04542*, 2024.

36 Jan Van Den Brand, Li Chen, Richard Peng, Rasmus Kyng, Yang P Liu, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 503–514. IEEE, 2023.

37 Santosh S Vempala. *The random projection method*, volume 65. American Mathematical Soc., 2005.

38 Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1130–1143, 2017.