# Near-Optimal Algorithm for Directed Expander Decompositions

Aurelio L. Sulser*
ETH Zurich
asulser@ethz.ch

Maximilian Probst Gutenberg*
ETH Zurich
maximilian.probst@inf.ethz.ch

## Abstract

In this work, we present the first algorithm to compute expander decompositions in an $m$-edge *directed* graph with near-optimal time $\tilde{O}(m)$[1]. Further, our algorithm can maintain such a decomposition in a dynamic graph and again obtains near-optimal update times. Our result improves over previous algorithms [BGS20, HKGW23] that only obtained algorithms optimal up to subpolynomial factors.

In order to obtain our new algorithm, we present a new push-pull-relabel flow framework that generalizes the classic push-relabel flow algorithm [GT88] which was later dynamized for computing expander decompositions in *undirected* graphs [HRW20, SW19]. We then show that the flow problems formulated in recent work [HKGW23] to decompose directed graphs can be solved much more efficiently in the push-pull-relabel flow framework.

Recently, our algorithm has already been employed to obtain the currently fastest algorithm to compute min-cost flows [VDBCK$^{+}$24]. We further believe that our algorithm can be used to speed-up and simplify recent breakthroughs in combinatorial graph algorithms towards fast maximum flow algorithms [CK24a, CK24b, BBST24].

*The research leading to these results has received funding from the starting grant "A New Paradigm for Flow and Cut Algorithms" (no. $TMSGI2\_218022$) of the Swiss National Science Foundation.

[1] In this article, we use $\tilde{O}(\cdot)$ notation to suppress factors logarithmic in $m$, i.e. $O(m \log^c m) = \tilde{O}(m)$ for every constant $c > 0$.

# 1 Introduction

Over the past two decades, expanders and expander decompositions have been pivotal in advancing on fundamental algorithmic graph problems. The development and application of the first fast algorithm to compute near-expander decompositions was given in the development of the first near-linear time Laplacian solvers [ST04], a breakthrough in modern graph algorithms. Subsequently, a line of research [HRW20, WN17, NS17, NSWN17] has focused on strengthening this result by developing fast flow-based pruning techniques that refine near-expander decompositions into expander decompositions. This line of research culminated in [SW19] where a new, faster, simpler, and more user-friendly expander decomposition framework was presented. This advancement has catalyzed the widespread use of expander decompositions as a tool in graph algorithms and was instrumental in the recent surge of applications of expander decompositions in both static and dynamic graph settings for various cut, flow, and shortest path problems [ST04, KLOS14, HRW20, WN17, NS17, NSWN17, CK19, BGS20, Liu23, BvdBPG+22, VDBLL+21, Sar21, CDK+21, Li21, CS21, GRST21, Chu21, BGS22, BGS22, KMP22, JS22, VDBCP+23, KMG24, CKL+24, JST24, CK24b, BBST24, VDBCK+24]. In this work, we study the problem of computing and maintaining expander decompositions in *directed* graphs, defined as follows.

**Definition 1.1** (Directed Expander Decomposition)**.** *Given an $m$-edge directed graph $G$, we say a partition $\mathcal{X}$ of the vertex set of $G$ and a subset of edges $E^r \subseteq E$ forms an $(\beta, \phi)$-expander decomposition if*

1. *$\forall X \in \mathcal{X}$, $G[X]$ is a $\phi$-expander meaning for all cuts $(S, \bar{S})$ : $\frac{\min\{e_G(S,\bar{S}), e_G(\bar{S},S)\}}{\min\{\mathrm{vol}_G(S), \mathrm{vol}_G(\bar{S})\}} \geq \phi$, and*
2. *$|E^r| \leq \beta \cdot \phi \cdot m$, and*
3. *the graph $(G \setminus E^r)/\mathcal{X}$, that is the graph $G$ minus the edges in $E^r$ where expander components in $\mathcal{X}$ are contracted into supernodes, is a directed acyclic graph (DAG).*

In our algorithm, we implicitly maintain an ordering of the partition sets in $\mathcal{X}$ and let $E^r$ be the edges that go 'backward' in this ordering of expander components. Note that we can only obtain a meaningful bound on the number of such 'backward' edges since a bound on *all* edges between expander components cannot be achieved as can be seen from any graph $G$ that is acyclic (which implies that $\mathcal{X}$ has to be a collection of singletons by Item 1 forcing all edges to be between components). Directed expanders and expander decompositions have been introduced in [BGS20] in an attempt to derandomize algorithms to maintain strongly connected components and single-source shortest paths in directed graphs undergoing edge deletions. Recently, [HKGW23] gave an alternative algorithm to compute and maintain directed expander decomposition that refines the framework from [BGS20] and heavily improves subpolynomial factors. Besides working on directed graphs, this algorithm also yields additional properties for expander decompositions that cannot be achieved with existing techniques - even in undirected graphs. In this article, we further refine these techniques to obtain an algorithm that is optimal up to logarithmic factors in $m$ - as opposed to subpolynomial factors. Since their invention, directed expander decompositions and the techniques to maintain such decompositions have been pivotal in the design of fast dynamic graph data structures and in ongoing research for fast 'combinatorial' maximum flow algorithms. We discuss in Section 1.2 these recent lines of research and how our algorithm benefits recent breakthrough results. For an in-depth discussion of expander decomposition techniques and applications both in directed and undirected graphs, we refer the interested reader to Appendix A.1.

### 1.1 Our Contribution

In this article, we finally give a simple algorithm that generalizes the algorithm from [SW19] in a clean way. Further, our algorithm is the first to obtain near-optimal runtimes for both static and dynamic expander decompositions in directed graphs. Our result is summarized in the theorem below.

**Theorem 1.2.** *Given a parameter $\phi \leq c/\log^{12} m$ for a fixed constant $c > 0$, and a directed $m$-edge graph $G$ undergoing a sequence of edge deletions, there is a randomized data structure that constructs and maintains a $(O(\log^{19} m), \Omega(\phi/\log^{12} m), O(1/\log^{8} m))$-expander decomposition* [2] *$(\mathcal{X}, E^r)$ of $G$. The initialization of the data structure takes time $O(m \log^{20}(m)/\phi)$ and the amortized time to process each edge deletion is $O(\log^{28}(m)/\phi^2)$.*

Further, our algorithm has the property that it is refining for up to $O(\phi \cdot \psi \cdot m)$ edge deletions meaning that $\mathcal{X}$ is a refinement of its earlier versions (every expander component in $\mathcal{X}$ is a subset of an expander component in any earlier expander decomposition) and the size of $E^r$ does never exceed $\tilde{O}(\phi m)$. Our algorithms are deterministic, however, they rely on calling a fast randomized algorithm to find balanced sparse cuts or certify that no such cut exists (see [KRV09, Lou10]). Our new techniques are much simpler and more accessible than previous work, besides also being much faster. We hope that by giving a simpler algorithm for directed expander decompositions, we can help to make this tool more accessible to other researchers in the field with the hope that this can further accelerate recent advances in dynamic and static graph algorithms.

### 1.2 Applications

Our new algorithms have direct applications to the currently fastest approaches for bipartite matching/ maximum flow/ min-cost flows and the data structures that are employed to obtain these results:

1. Our algorithm is already used in the fastest min-cost flow algorithm that is known to-date [VDBCK+24] which achieves runtime $m \cdot e^{O(\log^{3/4}(m) \log\log(m))}$ yielding the first improvement over the recent breakthrough in [CKL+22] achieving the first near-linear time algorithm for min-cost flows. In the framework of [VDBCK+24], our algorithmic techniques are used to maintain an expander decomposition of an *undirected* graph where it is heavily exploited that our algorithm maintains the expander decomposition such that it refines over time. This guarantee is pivotal in the construction of a fully-dynamic algorithm to maintain dynamic expander hierarchies which is the key data structure in the paper. While [VDBCK+24] could also have relied solely on the techniques [HKGW23] to obtain such a data structure with subpolynomial update and query times, these subpolynomial factors would have been substantially larger and would thus not have yielded a faster min-cost flow algorithm overall.

2. In [BGS20], directed expander decompositions for decremental graphs were used to obtain the first algorithm to maintain $(1+\epsilon)$-approximate Single-Source Shortest-Paths (SSSPs) in a decremental graphs in time $o(mn)$, though only for the special case of dense graphs. This problem is well-motivated as a simple reduction based on the Multiplicative Weights Update (MWU) framework implies that the maximum flow problem can be solved approximately by

[2] Here we use the augmented expander decomposition definition 2.5.

solving approximate decremental SSSP. By standard refinement of flows this yields an exact maximum flow algorithm. Very recently, Chuzhoy and Khanna [CK24a, CK24b] showed that for the update sequence generated by the MWU to solve the bipartite matching problem - a special case of the maximum flow problem - decremental SSSP can be maintained in $n^{2+o(1)}$ time by refining the techniques from [BGWN20, BGS20]. This yields the first near-optimal 'combinatorial'[3] algorithm for the bipartite matching problem for very dense graphs. While the above algorithms are 'combinatorial', they are still very intricate. Most of these complications stem from the maintenance of the directed expander decomposition used internally by the decremental SSSP data structure. We hope that our technique can help to simplify and speed-up these components to yield a simpler algorithm overall.

3. In independent work [BBST24], an alternative 'combinatorial' maximum flow that runs in $n^{2+o(1)}$ time was given. This algorithm cleverly extends push-relabel algorithms to run more efficiently when given an ordering of vertices that roughly aligns with the topological order induced by the acyclic graph formed from the support of an optimal maximum flow solution. To obtain this approximate ordering they compute a static expander hierarchy of the directed input graph. This generalizes the notion of directed expander decompositions further. In their work, they heavily build on the techniques from [HKGW23] to obtain the expander hierarchy. Unfortunately, the algorithm to obtain this hierarchy is very involved. We hope that our new techniques can help to simplify and speed-up their algorithms.

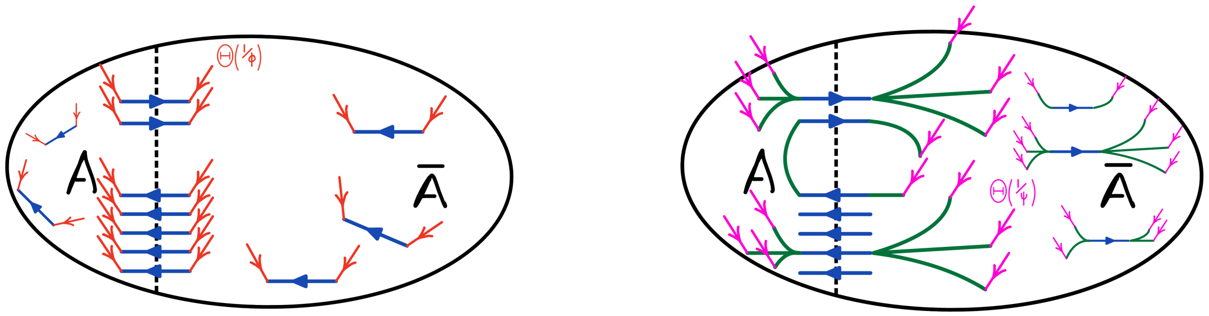
## 1.3 Our Techniques

**High-Level Strategy.** We obtain our result by following the high-level strategy of [SW19] for undirected graphs: we draw on existing literature (specifically [KRV09, Lou10]) for an algorithm that either outputs a balanced sparse cut which allows us to recurse on both sides; or outputs a witness that no such cut exists. This witness can be represented as an expander graph $W$ that embeds into $G \cup F$ with low congestion where $F$ is a set of few *fake* edges. In the second case, we set up a flow problem to extract a large expander (the first algorithm only finds balanced sparse cuts, so many unbalanced sparse cuts might remain) which suffices to again recurse efficiently.

**The (Dynamic) Flow Problem in [SW19].** To outline our algorithm, we first sketch the techniques of [SW19]. In [SW19], the following sequence of flow problems is formulated: initially, we add $\frac{1}{2\phi}$ units of source commodity to each endpoint of an edge in $F$ and then ask to route the commodity in $G$ where each vertex $v$ is a sink of value $\deg_G(v)$ and each edge has capacity $\frac{1}{2\phi}$. It then runs an (approximate) maximum flow algorithm on the flow problem. Whenever the algorithm detects that no feasible flow exists[4], it finds a cut $(A, \overline{A})$ where $A$ is the smaller side of the cut and then poses the same problem for the network $G[\overline{A}]$ where this time the source commodity is assigned for each edge in $E_G(A, \overline{A}) \cup F$. The algorithm terminates once the flow problem can be solved and outputs the final induced graph. In [SW19], it is shown that once a feasible flow exists then the (induced) graph is a $\Omega(\phi)$-expander. Further, it is shown that in the sequence of flow problems, each problem can be warm-started by re-using the flow computed in the previous

[3] We refrain from defining the scope of combinatorial algorithms here and refer the reader to [CK24a, CK24b] for a discussion.

[4] Technically, the algorithm might already output cuts when some cut has capacity less than a constant times the amount of flow that is required to be routed through the cut.



(a) In directed graphs, cuts are asymmetric. While $(A, \bar{A})$ might be a sparse cut, the cut $(\bar{A}, A)$ might contain many edges. A straightforward extension of [SW19] would inject $2/\phi$ units of commodity to each endpoint of $E_G(A, \bar{A}) \cup F$. However, it is not clear with this approach how to bound the total amount of flow injected throughout the algorithm.

(b) We inject $\Theta(1/\psi)$ units of commodity at the end points of any witness embedding path (green) going through an edge of $E_G(A, \bar{A}) \cup F$. We can bound the total amount by $O(\mathrm{vol}(A)/\mathrm{poly}(\psi))$. But the injection might well be in the interior of $A, \bar{A}$ possibly leaving negative excess at the endpoints of the cut edges.

Figure 1: Injection of Commodity due to edges in $E_G(A, \bar{A}) \cup F$

instance to detect a cut induced on the remaining vertex set. This result is obtained by two main insights:

1. if the flow $\boldsymbol{f}$ to find the cut $(A, \bar{A})$ is a *pre-flow*, that is a flow that respects capacities and has no negative excess at any vertex (i.e. it does not route away more flow from a vertex than is inputted by the source), then injecting additional source flow for any edge $E_G(A, \bar{A})$ guarantees that the induced flow $\boldsymbol{f}|_{\bar{A}}$ is a pre-flow in the flow problem formulated for $G[\bar{A}]$. That is because the amount of flow that was routed via such a cut edge is at most $2/\phi$ and thus placing $2/\phi$ new source commodity at the endpoint ensures that no negative excess exists in the induced flow $\boldsymbol{f}|_{\bar{A}}$,

2. the classic push-relabel framework can naturally be extended to warm-start on such a flow $\boldsymbol{f}|_{\bar{A}}$ as it is built to just further refine pre-flows at every step.

This dynamization of the push-relabel framework allows to bound the cost of computation of *all* flow problems linearly in the amount of source commodity which in term is bounded by the number of edges that appear in either $F$ or one of the identified min-cuts. Finally, [SW19] shows that the amount of source commodity remaining in $\bar{A}$ decreases over the sequence of flow problems proportional to the volume of the set $A$ of vertices that are removed at each step. Indeed, they observe that at each vertex $v$ in $A$ at least $\deg(v)$ many commodity units are absorbed and that the total amount of source injected due to the cut edges $E_G(A, \bar{A})$ is bounded by $\frac{1}{2\phi} \cdot e_G(A, \bar{A}) \leq \mathrm{vol}_G(A)/2$. This yields that the final induced graph is still large. Thus, the final graph outputted is a large expander, as desired.

**The (Dynamic) Flow Problem in Directed Graphs.** In directed graphs, while the above flow problem upon becoming feasible also certifies that the remaining graph is a $\Omega(\phi)$-expander, the argument that the sequence of flow problems terminates does not work: the asymmetry of cuts might force us for a small cut $E_G(A, \bar{A})$ to induce on $\bar{A}$ while having many edges in $E_G(\bar{A}, A)$ each of which would add $2/\phi$ source flow to the new flow problem (see Figure 1a). Hence, we might end up injecting up to $\Omega(\mathrm{vol}_G(A)/\phi)$ more flow due to the cut edges. This makes it seemingly impossible to argue that the amount of source commodity in the next flow problem is smaller. To recover the argument that the sequence of flow problems terminates (with the remaining expander graph being large) both [BGS20, HKGW23] suggest setting up the flow problems more carefully such that each cut $(A, \bar{A})$ that is found in this sequence and induced upon is a *sparse cut*. Here, we only describe the less lossy flow problem formulation developed in [HKGW23]. To ensure that each cut $(A, \bar{A})$ that is found is a sparse cut, [HKGW23] proposes a slightly different flow problem: instead of adding source commodity $\frac{1}{2\phi}$ per endpoint of an edge that is fake or not fully contained in the induced graph, it tailors the amount of new source commodity using the witness graph $W$, possibly injecting much less source commodity in the process. Concretely, we have that $W$ is a $\psi$-expander over the same vertex set as $G$ with degrees similar to degrees in $G$ up to a factor of $\Theta(\mathrm{poly}(\psi))$ and an embedding $\Pi$ into $G \cup F$ with congestion $O(\phi/\psi)$, for $\phi = \tilde{\Theta}(1)$. To set-up the flow problem, we inject $\Theta(1/\psi)$ units at the endpoints of any edge $e$ in the witness $W$ whose embedding path $\Pi(e)$ goes through an edge in $E_G(A, \bar{A}) \cup F$. We note that any such witness embedding path $\Pi(e)$ either crosses the sparse cut $(A, \bar{A})$ or has an endpoint in $A$. This allows to bound the additional source injected by $O(\mathrm{vol}_G(A)/\mathrm{poly}(\psi)) = \tilde{O}(\mathrm{vol}_G(A))$ where we use that $\psi = \tilde{\Theta}(1)$. But while correctness and termination of the flow problem sequence are now ensured, this leaves a significant problem: the current flow $\boldsymbol{f}$ that was used to find the cut $(A, \bar{A})$ no longer has the property that $\boldsymbol{f}|_{\bar{A}}$ is a *pre-flow* in the flow problem formulated on network $G[\bar{A}]$ even if $\boldsymbol{f}$ is a pre-flow. While capacity constraints are still enforced, i.e. $\boldsymbol{f}|_{\bar{A}}$ still is a pseudo-flow (see [Hoc08] for reference on pseudo-flow), some vertices might now have negative excess since the additional commodity might be injected in the interior of $\bar{A}$ far from the cut $(A, \bar{A})$ (see Figure 1b). Indeed, for an edge $(u, v)$ in $E_G(A, \bar{A})$ a lot of flow might have been routed through $(u, v)$ but no additional commodity might be injected at $v$ since all witness embedding paths passing through $(u, v)$ might not end in $v$. Thus, dynamizing the push-relabel framework does not appear natural for this sequence of problems as it crucially requires that the maintained flow is a pre-flow at all times. In [HKGW23], an involved batching technique is used instead (based on the technique in [NSWN17]) that does not use dynamic flow problems but instead reduces to few static flow problems, however, at the loss of quality and runtime by subpolynomial factors.

**The Push-Pull-Relabel Framework.** The main technical contribution of this paper is a new framework that refines pseudo-flows as efficiently as the push-relabel framework refines pre-flows. Thus, we give a generalization of the latter widely-used and well-studied framework that we believe might have applications well beyond our problem. Recall that the classic push-relabel framework maintains labels $\boldsymbol{\ell}$ for all vertices and a pre-flow $\boldsymbol{f}$. In each iteration, it 'pushes' positive excess flow at a vertex $v$ to a vertex at a lower label (to be precise to a vertex at level $\boldsymbol{\ell}(v) - 1$); or if no 'push' is possible, it increases the labels of some vertices: it 'relabels'. Using a clever potential-based analysis, one can show that it suffices to only increase the labels to a certain threshold before all flow is settled. In our framework, we allow deleting edges, without compensating by adding source commodity at the endpoints, which might create negative excess leaving $\boldsymbol{f}$ a pseudo-flow (instead of a pre-flow). Now, while our framework applies the same strategy for 'pushes' and 'relabels', we also need a new operation 'pull'. Intuitively, our algorithm tries to 'pull' back the source commodity that now causes the negative excess (this unit of commodity was 'pushed' earlier to some other vertices). To do so, a vertex $v$ with negative excess can 'pull' commodity from vertices at a higher level (again it can only pull from a vertex at level $\boldsymbol{\ell}(v) + 1$). But it is not difficult to construct an example where this strategy does not suffice: therefore, we also need to sometimes decrease the label of a vertex to ensure correctness. However, the latter change to the 'relabel' operation breaks the property that labels are non-decreasing over time. A property that is crucial in the existing efficiency analysis. Instead, we give a much more careful argument to analyze the potentials that bound the number of push, pull, and relabeling operations that deal with the non-monotonicity of the levels over time. The argument is sketched below. Combining this framework with the above-discussed set-up of dynamic flow problems as proposed in [HKGW23] then yields the first near-optimal algorithm to compute an expander decomposition in a directed graph. Further, our technique extends seamlessly to also deal with edge deletions to $G$, yielding an algorithm to prune expander graphs that undergo edge deletions.

**A Sketch of the Runtime of Push-Pull-Relabel.** The run-time analysis in standard push-relabel considers the run-time contribution of the push and relabel operations separately. Using a potential argument, one can then relate the contribution of the push operations to the relabeling operations. A similar argument albeit much more delicate also allows to relate the contribution of the push and the pull operations to the contribution of the relabeling operations in the extended push-pull-relabel algorithm. What might seem more daunting is to bound the run-time contribution of the relabelings given that the level function $\boldsymbol{\ell}$ is no longer point-wise non-decreasing. Let us revisit the argument to bound the run-time contribution of the relabelings in the push-relabel of [SW19]. Any relabeling of $v$, incurs a cost of $O(\deg(v))$ as each incident edge has to be checked before a relabeling. At such a relabeling of $v$, the sink of $v$, which has by choice capacity $\deg_G(v)$, must be full and any commodity unit in it remains there till termination. Since the label of $v$ does not decrease and is bounded by $h$, we may thus charge for the run-time contribution of the relabelings of $v$ each unit in the sink of $v$ exactly $h$. Overall vertices, we conclude that any commodity unit was charged at most $h$ units. In the push-pull-relabel algorithm, a commodity unit might end up in various sinks. So a more clever charging argument needs to be devised. To guide intuition an analogy to flows of protons and electrons is useful. The protons correspond to commodity units and the electrons to the lack of commodity units. We are now moving protons and electrons around in the network and whenever a proton and an electron meet at a vertex they form a neutron which stays put at the vertex indefinitely. The neutrons will provide a mean to charge work in the analysis. The sink of each vertex $v$ can absorb $\deg(v)$ commodity units or put differently $\deg(v)$ protons. Electrons only form when we delete an edge $(u, v)$, i.e. exactly $\boldsymbol{f}^+(u, v)$ new electrons form at $v$. By choice, we now say that if we perform a push from $u$ to $v$ of one unit then exactly one proton moves from $u$ to $v$, while if we perform a pull from $u$ to $v$ then exactly one electron moves from $v$ to $u$. Since we only perform a push away from $v$ if there are more than $\deg_G(v)$ free protons (not part of a neutron) at $v$ and a pull towards $v$ if there are free electrons at $v$, we find that $\min(\mathbf{n}(v) + \deg_G(v)/2, \mathbf{p}(v))$, where $\mathbf{p}(v)$ denotes the number of protons at $v$ (including the ones in a neutron) and $\mathbf{n}(v)$ denotes the number of electrons at $v$, is non-decreasing. This fact allows us to conclude that whenever we change from increasing $\boldsymbol{\ell}(v)$ to decreasing $\boldsymbol{\ell}(v)$ at least $\deg(v)/2$ new neutrons have formed at $v$. Since any neutron at $v$ stays put indefinitely, we can charge the run-time contributions of the relabelings of $v$ to the number of neutrons at $v$.

**Roadmap.** In the remainder of the article, we first give preliminaries in Section 2, then present our new push-pull-relabel framework in Section 3 and finally show how to obtain our result in Theorem 1.2 using the new framework in Section 4.

## 2 Preliminaries

**Graphs.** We let $\deg_G \in \mathbb{R}^{V(G)}$ denote the degree vector of graph $G$. For all vertex $v \in V$, we have $\deg_G(v)$ equal to the number of edges incident to $v$ (both incoming and outgoing are counted). Moreover, for any subset of edges $D \subseteq E(G)$ we denote by $G_D$ the graph with vertex set $V(G)$ and edge set $D$ and by $\deg_D$ the degree vector of the subgraph $G_D$. We denote by $\mathrm{vol}_G(S)$ for any $S \subseteq V$, the sum of degrees of vertices in $S$. We denote by $E_G(A, B)$ for any $A, B \subseteq V$ the set of directed edges in $E(G)$ with tail in $A$ and head in $B$. We define $e(G) = |E(G)|$ and $e_G(A, B) = |E_G(A, B)|$. For any partition $\mathcal{P}$ of $V(G)$, we denote the graph obtained by contracting each partition class to a single vertex by $G/\mathcal{P}$. Two vertices in $G/\mathcal{P}$ are adjacent if there is an edge between the corresponding partition classes in $G$. Moreover, for any vertex $v \in V(G)$ we denote by $\mathbb{1}_v \in \mathbb{R}^{V(G)}$ the vector with all entries equal to zero apart from the entry at $v$ equaling one.

**Flows.** We call a tuple $(G, \boldsymbol{c}, \Delta, \nabla)$ a flow problem, if $G$ is a directed graph, the capacity function $\boldsymbol{c} : V(G) \times V(G) \to \mathbb{R}^{\geq 0}$ is such that for all $(u, v) \notin E$ we have $\boldsymbol{c}(u, v) = 0$, and $\Delta, \nabla : V(G) \to \mathbb{R}^{\geq 0}$ denote the source and the sink capacities. We denote flows on $G$ as functions $\boldsymbol{f} : V(G) \times V(G) \to \mathbb{R}$ such that $\boldsymbol{f}$ is anti-symmetric, i.e. $\boldsymbol{f}(u, v) = -\boldsymbol{f}(v, u)$. Given a vertex $v \in V(G)$ we introduce the notation $\boldsymbol{f}(v) = \sum_u \boldsymbol{f}(v, u)$ and likewise $\boldsymbol{c}(v) = \sum_u \boldsymbol{c}(v, u) + \boldsymbol{c}(u, v)$. Moreover, we write $\boldsymbol{f}^+(u, v) = \max(\boldsymbol{f}(u, v), 0)$. Given a flow $\boldsymbol{f}$, we say a vertex $v \in V(G)$ has $\Gamma(v) = \Delta(v) - \boldsymbol{f}(v)$ excess. We say that it has positive excess if $\Gamma(v) > \nabla(v)$ and $v$ has negative excess if $\Gamma(v) < \nabla(v)/2$. For a subset $\tilde{V} \subseteq V(G)$, we induce the flow $\boldsymbol{f}$, and the sink $\nabla$, source $\Delta$ and edge capacities $\boldsymbol{c}$ onto $G[\tilde{V}]$ in a function sense and write $\boldsymbol{f}|_{\tilde{V}}, \nabla|_{\tilde{V}}, \Delta|_{\tilde{V}}, \boldsymbol{c}|_{\tilde{V}}$. Moreover, for any subset $E \subseteq E(G)$, we write $\boldsymbol{f}|_E$ for the flow induced by $\boldsymbol{f}$ onto the subgraph of $G$ consisting only of the edges $E$. We say that a flow $\boldsymbol{f}$ is a *pseudo-flow* if it satisfies the capacity constraints:

$$\forall (u, v) \in V(G) \times V(G) : -\boldsymbol{c}(v, u) \leq \boldsymbol{f}(u, v) \leq \boldsymbol{c}(u, v).$$

We say $\boldsymbol{f}$ is a *pre-flow* if $\boldsymbol{f}$ is a pseudo-flow and has no negative excess at any vertex. We say a flow $\boldsymbol{f}$ is *feasible* if it is a pre-flow and additionally no vertex has positive excess. Moreover, for any subset $S \subseteq V$ we denote by $\Delta(S), \nabla(S)$ the sum $\sum_{v \in S} \Delta(v), \sum_{v \in S} \nabla(v)$ respectively and for any subset $D \subseteq E(G)$ we denote by $\boldsymbol{c}(D)$ the sum $\sum_{d \in D} \boldsymbol{c}(d)$.

**Expanders.** Given graph $G = (V, E)$, we say a cut $(S, \bar{S})$ is $\phi$-out sparse if $e_G(S) \leq e(G)$ and $e_G(S, V \setminus S) < \phi \cdot \mathrm{vol}_G(S)$. We say $G$ is a $\phi$-out expander if it has no $\phi$-out-sparse cut. We say $G$ is a $\phi$-expander if $G$ and $G^{rev}$, the graph where all edges of $G$ are reversed, are both $\phi$-out expander. The next lemma that is folklore and crucial in our expander pruning argument.

**Lemma 2.1.** *Given a $\phi$-expander $G = (V, E)$, then take $S \subseteq V$ and a set of edge deletions $D$. We have that $e_{G \setminus D}(S, V \setminus S) < \frac{\phi}{4} \cdot \mathrm{vol}_G(S)$ implies $\min(\mathrm{vol}_G(S), \mathrm{vol}_G(V \setminus S)) < \frac{4 \cdot |D|}{3\phi}$.*

*Proof.* TOPROVE 0 □

**Graph Embeddings.** Given two graphs $H$ and $G$ over the same vertex set $V$, we say that $\Pi$ is an embedding of $H$ into $G$ if for every edge $e = (u, v) \in E(H)$, $\Pi(e)$ is a simple $uv$-path in $G$. We define the congestion of an edge $e \in E(G)$ induced by the embedding $\Pi$ to be the maximum number of paths in the image of $\Pi$ that contain $e$. We define the congestion of $\Pi$ to be the maximum congestion achieved by any such edge $e \in E(G)$. Moreover, for any edge $e \in E(G)$ we will denote by $\Pi^{-1}(e)$ the set of edges $f \in E(H)$ such that $e$ is an edge on the path $\Pi(f)$. Given an entire set of edges $D \subseteq E(G)$, we denote by $\Pi^{-1}(D)$ the set of edges $f \in E(H)$ such that some edge of the path $\Pi(f)$ is in $D$. Given two graphs $H_1, H_2$ on the same vertex set and embeddings $\Pi_1 : H_1 \to G, \Pi_2 : H_2 \to G$ then we denote by $\Pi_1 \cup \Pi_2 : H_1 \cup H_2 \to G$ the embedding of the graph $H_1 \cup H_2 = (V(H_1), E(H_1) \cup E(H_2))$ .

**Expander Decompositions with Witnesses.** For the rest of the article, we use a definition of expander decompositions that encodes much more structure than given in Definition 1.1. In particular, the definition below incorporates the use of witness graphs which are instrumental to our algorithm.

**Definition 2.2.** *Given a directed graph $G$, we say $(W, \Pi)$ is a $(\phi, \psi)$-out-witness for $G$ if 1) $W$ is a $\psi$-out-expander, and 2) $\Pi$ embeds $W$ into $G$ with congestion at most $\frac{\psi}{\phi}$, and 3) $\forall v \in V(W)$ : $\deg_G(v) \leq \deg_W(v) \leq \frac{\deg_G(v)}{\psi}$. If $(W, \Pi)$ is a $(\phi, \psi)$-out-witness for $G$ and for $G^{rev}$, then we say that $(W, \Pi)$ is a $(\phi, \psi)$-witness for $G$.*

**Fact 2.3.** *If $(W_1, \Pi_1)$ is a $(\phi, \psi)$-out-witness for $G$ and $(W_2, \Pi_2)$ is a $(\phi, \psi)$-out-witness for $G^{rev}$, then $(W_1 \cup W_2, \Pi_1 \cup \Pi_2)$ is a $(\psi\phi/4, \psi^2/2)$-witness for $G$.*

*Proof.* TOPROVE 1 □

The next fact establishes that a $(\phi, \psi)$-(out-)witness for $G$ certifies that $G$ is a $\phi$-(out-)expander, justifying the name witness.

**Fact 2.4** (see [HKGW23], Claim 2.1)**.** *If $(W, \Pi)$ is a $(\phi, \psi)$-witness for $G$, then $G$ is a $\phi$-expander.*

*Proof.* TOPROVE 2 □

**Definition 2.5** (Augmented Expander Decomposition)**.** *We call a collection $\mathcal{X}$ and a subset $E^r \subseteq E$ a $(\beta, \phi, \psi)$-expander decomposition of a graph $G$, if*

1. *$\forall (X, W, \Pi) \in \mathcal{X}$, $G[X]$ has a $(\phi, \psi)$-witness $(W, \Pi)$,*
2. *$|E^r| \leq \beta \cdot \phi \cdot e(G)$,*
3. *$(G \setminus E^r)/\mathcal{P}$ is a DAG, where $\mathcal{P} = \{X \mid (X, W, \Pi) \in \mathcal{X}\}$.*

*Given two expander decompositions $(\mathcal{X}_1, E_1^r)$ of the graph $G$ and $(\mathcal{X}_2, E_2^r)$ of the graph $G \setminus \mathcal{D}$, where $\mathcal{D} \subseteq E(G)$, we say $(\mathcal{X}_2, E_2^r)$ refines $(\mathcal{X}_1, E_1^r)$ if 1) for all partition classes $X_2$, where $(X_2, W_2, \Pi_2) \in \mathcal{X}_2$, there is a class $X_1$, where $(X_1, W_1, \Pi_1) \in \mathcal{X}_1$, such that $X_2 \subseteq X_1$ and 2) $E_1^r \subseteq E_2^r \cup \mathcal{D}$.*

# 3 The Push-Pull-Relabel Framework

In the push-relabel framework as presented in [GT88], we are trying to compute a feasible flow for a flow problem $(G, \boldsymbol{c}, \Delta, \nabla)$ by maintaining a pre-flow $\boldsymbol{f}$ together with a level function $\boldsymbol{\ell}$. The algorithm then runs in iterations terminating once $\boldsymbol{f}$ has no positive excess at any vertex. In each iteration of the algorithm, the algorithm identifies a vertex $v$ that still has positive excess at a vertex $v$. This positive excess is then pushed to neighbors on lower levels such that the capacity constraint is still enforced. If this is not possible $v$ is relabeled meaning that its label $\boldsymbol{\ell}(v)$ is increased. In this section, we are extending the push-relabel framework to the dynamic setting, where we allow for increasing the source function $\Delta$ and the deletion of edges from $G$. To do so, we need to introduce the notion of negative excess. Because for a flow $\boldsymbol{f}$ of the flow problem, it might happen that once we delete an edge $(u, v)$ there is more flow leaving the vertex $v$ than is entering or sourced, i.e. $\Gamma_t(v) < 0$. Hence, we need to extend the discussion to include negative excess in the dynamic version. Similar to the standard push-relabel algorithm, we maintain a pseudo-flow $\boldsymbol{f}$ and a vertex labeling $\boldsymbol{\ell}$ in so-called valid states $(\boldsymbol{f}, \boldsymbol{\ell})$. The only difference is that in the standard push-relabel algorithm, $\boldsymbol{f}$ is a pre-flow (not only a pseudo-flow).

**Definition 3.1.** *Given a level function $\boldsymbol{\ell} : V(G) \to [h]$ and a pseudo-flow $\boldsymbol{f}$, we call a tuple $(\boldsymbol{f}, \boldsymbol{\ell})$ a state for $(G, \boldsymbol{c}, \Delta, \nabla)$ if for all edges $e = (u, v)$ having $\boldsymbol{\ell}(u) > \boldsymbol{\ell}(v) + 1$ implies $\boldsymbol{f}(e) = \boldsymbol{c}(e)$.*
*We call the state $(\boldsymbol{f}, \boldsymbol{\ell})$ valid if for all vertices $v \in V(G)$ : 1) $\Gamma(v) < \nabla(v)/2$ implies $\boldsymbol{\ell}(v) = 0$, 2) $\nabla(v) < \Gamma(v)$ implies $\boldsymbol{\ell}(v) = h$.*

For the remainder of the paper, we have $h = O\left(\frac{\log(n)}{\phi}\right)$. To provide the reader with some intuition about the definitions, we remark that the pseudo-flow of a valid state is not feasible in the usual sense. Indeed, some vertices might have positive or negative excess. But these vertices are guaranteed to either be at level $h$ or at level 0. Moreover, we point out that if a valid state $(\boldsymbol{f}, \boldsymbol{\ell})$ has no vertices at level $h$, then $\boldsymbol{f}$ might still not be a feasible flow since there might be a vertex $v$ with $\Gamma_t(v) < 0$. But it is straightforward to obtain from $\boldsymbol{f}$ a feasible flow: extract from $\boldsymbol{f}$ at each vertex $v$ exactly $\Delta(v)$ unit flow paths (possibly empty paths starting and ending in $v$). Let us briefly outline how we will use Lemma 3.2 in the directed expander pruning algorithm 3. In the directed expander pruning algorithm, we start with an out-expander and an adversary performs a number of vertex or edge deletions. We aim to find a small pruning set of vertices such that if we prune away this small pruning set from the remaining graph, then we can certify that the obtained graph is still an expander. The classical way (see [SW19]) to either certify expansion of the remaining graph or to find a pruning set is by setting up a flow problem, where we inject for any deletion some additional source flow and where each vertex has sink capacity equal to its degree (this is the reason for the condition $\nabla \geq \deg$ in Lemma 3.2). We will solve this flow problem using the flow provided by our ValidState algorithm (see Algorithm 1). We maintain this flow problem using the interface functions IncreaseSource, RemoveEdges under both adversarial deletions and under necessary pruning deletions. If the valid state computed after any such update has an additional vertex on level $h$, this will indicate that more vertices need to be pruned away. If on the other hand all vertices are on levels strictly below $h$, we will certify that the remaining graph is an expander.

**Lemma 3.2.** *Given a flow problem $(G = (V, E), \boldsymbol{c}, \Delta, \nabla)$, where $\nabla \geq \deg$. Then, there is a deterministic data structure* ValidState$(G, \boldsymbol{c}, \Delta, \nabla)$ *(see Algorithm 1) that initially computes a valid state $(\boldsymbol{f}, \boldsymbol{\ell})$ and after every update of the form*

- IncreaseSource($\delta$) : *where* $\delta \in \mathbb{N}^n_{\geq 0}$*, we set* $\Delta$ *to* $\Delta + \delta$*,*
- RemoveEdges($D$)*: where* $D \subseteq E(V) \setminus \mathcal{D}$*, sets* $\mathcal{D}$ *to* $\mathcal{D} \cup D$ *(initially* $\mathcal{D} = \emptyset$*),*

*the algorithm explicitly updates the tuple* $(\boldsymbol{f}, \boldsymbol{\ell})$ *such that thereafter* $(\boldsymbol{f}, \boldsymbol{\ell})$ *is a valid state for the current flow instance* $(G \backslash \mathcal{D}, \boldsymbol{c}|_{E\backslash\mathcal{D}}, \Delta|_{E\backslash\mathcal{D}}, \nabla|_{E\backslash\mathcal{D}})$*. The run-time is* $O\left(h \cdot \left(\|\Delta\|_1 + \sum_{d \in \mathcal{D}}(1 + |\boldsymbol{f}_{t_d}(d)|)\right)\right)$*, where* $\boldsymbol{f}_{t_d}(d)$ *is equal to* $\boldsymbol{f}(d)$ *at the time d is deleted and* $\Delta$ *is the variable at the end of the algorithm.*

The user interface of the data structure ValidState are the functions Init, IncreaseSource and RemoveEdges. These functions can be used to initialize or update the flow problem. After any such update the internal state $(\boldsymbol{f}, \boldsymbol{\ell})$ has to be updated to remain valid for the new flow problem. To facilitate these necessary updates to $(\boldsymbol{f}, \boldsymbol{\ell})$ these functions make calls to the internal functions PushRelabel and PullRelabel. As described before, the function PushRelabel performs push and relabel operations to handle any positive excess, while the function PullRelabel performs pull and relabel operations to handle any negative excess. Let us take a closer look at the individual functions: using Init we set up the flow problem. It makes an internal call to PushRelabel to compute the first valid state using the standard push-relabel algorithm. Note that at this point no negative excess has been introduced and thus we do not need to resort to any pull operations. The function IncreaseSource can be used to increase the source of the current flow problem and again this does not introduce any negative excess and hence we can again update the valid state using the standard push-relabel algorithm. Using the function RemoveEdges, we can restrict the flow problem to a subgraph. The function induces the capacities $\boldsymbol{c}, \Delta, \nabla$ and the flow $\boldsymbol{f}$ to the subgraph and then makes calls to both PushRelabel and PullRelabel to update the state. The function PushRelabel is just the standard push-relabel algorithm. We look for a vertex $v$ with positive excess $\Gamma(v) > \nabla(v)$ at level below $h$. We pick a vertex on the lowest level possible. We try to push flow along some unsaturated outgoing edge to a vertex on a lower level. If it is not possible to push flow, we increase the label of the vertex and repeat. The function PullRelabel is the analog of the push-relabel algorithm for negative excess. Here, we look for a vertex $v$ with negative excess $\Gamma(v) < \nabla(v)/2$ at level above 0. We pick a vertex on the highest level possible. We try to pull flow along some unsaturated incoming edge from a vertex on a higher level. If it is not possible to pull flow then all edges $(u, v)$ where $\boldsymbol{\ell}(u) > \boldsymbol{\ell}(v)$ must be saturated and it is safe to decrease the label of the vertex and repeat.

*Proof.* TOPROVE 3 □

## 4 Directed Expander Decompositions via the Push-Pull-Relabel Framework

In this section, we discuss our main technical result that states that if $G$ is initially a $\phi$-expander, then an expander decomposition can be maintained efficiently. The algorithm behind this theorem heavily relies on the new push-pull-relabel algorithm presented in the previous section.

**Theorem 4.1.** *Given a* $\phi$*-expander* $G = (V, E)$ *with* $(\phi, \psi)$*-witness* $(W, \Pi)$*, there is a deterministic data structure* BidirectedExpanderPruning$(G, W, \Pi)$ *(see Algorithm 2). After every update of the form*

- RemoveEdges($D$)*: where* $D \subseteq E$*, sets* $\mathcal{D}$ *to* $\mathcal{D} \cup D$ *(initially* $\mathcal{D} = \emptyset$*),* *the algorithm explicitly updates $\tilde{V} \subseteq V$ and the tuple $(E^r, \mathcal{P})$, where $E^r \subseteq E(G)$ and $\mathcal{P}$ is a partition of $V \setminus \tilde{V}$ (initially $E^r, \mathcal{P} = \emptyset$), such that both $E^r$ and $\mathcal{P}$ are non-decreasing and such that after the update*

1. *the graph $\tilde{G} = (G \setminus \mathcal{D}) \left[\tilde{V}\right]$ has a $\left(\frac{\phi \cdot \psi^6}{32000}, \frac{\psi^4}{800}\right)$-witness $(\tilde{W}, \Pi)$,*
2. *$|E^r| \leq \frac{\phi}{4} \cdot \sum_{P \in \mathcal{P}} \mathrm{vol}_G(P)$*
3. *$\sum_{P \in \mathcal{P}} \mathrm{vol}_W(P) \leq \frac{8}{\psi} \cdot |\Pi^{-1}(\mathcal{D})|$*
4. *$(G \setminus (\mathcal{D} \cup E^r))/(\mathcal{P} \cup \{\tilde{V}\})$ is a directed acyclic graph (DAG),*

*provided $\frac{200}{\psi^2} \cdot |\Pi^{-1}(\mathcal{D})| < e(G)$. The run-time is $O\left(\frac{h}{\psi^2} \cdot |\Pi^{-1}(\mathcal{D})| + h \cdot \sum_{d \in \mathcal{D} \cup E_{\mathcal{P}}} (1 + |\boldsymbol{f}_{t_d}(d)|)\right) = O\left(h \cdot \frac{|\mathcal{D}|}{\psi^2 \phi}\right)$, where $\mathcal{D}$ denotes the variable at the end of all updates and $E_{\mathcal{P}}$ denotes all intercluster edges of the partition $\mathcal{P} \cup \{\tilde{V}\}$.*

Following the high-level approach of [SW19], we obtain our main result Theorem 1.2 for the special case where $G$ is static by running a generalization of the cut-matching game to directed graphs [KRV09, Lou10] and then apply Theorem 4.1 to handle unbalanced cuts. Combining this result again with Theorem 4.1, we obtain our main result Theorem 1.2 in full generality. Both of these reductions have been known from previous work [BGS20, HKGW23]. We defer the former reduction to Appendix A.5 and the latter to Appendix A.6.

## 4.1 Reduction to Out-Expanders

In this subsection, we show that the task of maintaining an expander decomposition as described in Theorem 4.1 can be reduced to a simpler problem that only requires maintaining out-expanders (however under edge and vertex deletions). In particular, we reduce to the following statement whose proof is deferred to Appendix A.4.

**Lemma 4.2.** *For every $\phi$-out-expander $G = (V, E)$ with $(\phi, \psi)$-witness $(W, \Pi)$, there is a deterministic data structure* DirectedExpanderPruning$(G, W, \Pi)$ *(see Algorithm 3). After every update of the form (initially $\tilde{V} = V, \mathcal{S} = \emptyset, E_{\mathcal{S}} = \emptyset, E^+_{\mathcal{S}} = \emptyset, E^-_{\mathcal{S}} = \emptyset, \mathcal{D} = \emptyset$)*

- RemoveEdges$(D)$*: where $D \subseteq E(\tilde{V}) \setminus \mathcal{D}$, sets $\mathcal{D}$ to $\mathcal{D} \cup D$,*
- RemoveVertices$(S)$*: where $S \subseteq \tilde{V}$, sets $\tilde{V}$ to $\tilde{V} \setminus S$, $\mathcal{S}$ to $\mathcal{S} \cup S$, $E^+_{\mathcal{S}}$ to $E^+_{\mathcal{S}} \cup E_G(S, \tilde{V} \setminus S)$, $E^-_{\mathcal{S}}$ to $E^-_{\mathcal{S}} \cup E_G(\tilde{V} \setminus S, S)$ and $E_{\mathcal{S}}$ to $E^+_{\mathcal{S}} \cup E^-_{\mathcal{S}}$*

*the algorithm explicitly updates $\tilde{V} \subseteq V$ and the tuple $(E^r, \mathcal{P})$, where $E^r \subseteq E$ and $\mathcal{P}$ is a partition of $V \setminus \left(\mathcal{S} \cup \tilde{V}\right)$ (initially $E^r = \emptyset, \mathcal{P} = \emptyset$), such that both $E^r$ and $\mathcal{P}$ are non-decreasing and such that after the update*

1. *the graph $(G \setminus \mathcal{D}) \left[\tilde{V}\right]$ has a $\left(\frac{\phi \cdot \psi^4}{400}, \frac{\psi^2}{20}\right)$-out-witness $(\tilde{W}, \tilde{\Pi})$,*
2. *$|E^r| \leq \frac{\phi}{4} \cdot \sum_{P \in \mathcal{P}} \mathrm{vol}_G(P)$,*
3. *$\sum_{P \in \mathcal{P}} \mathrm{vol}_W(P) \leq \frac{4}{3\psi} \cdot \left|\Pi^{-1}(\mathcal{D} \cup E^-_{\mathcal{S}})\right|$,*

*4.* $E^r = \bigcup_P E_{G \setminus (\mathcal{D} \cup E_{\mathcal{S}}^-)}(P, V_P)$*, where* $V_P$ *is equal to* $\tilde{V}$ *at the time* $P$ *is added to* $\mathcal{P}$*,*

*provided* $\frac{20}{\psi} \cdot |\Pi^{-1}(\mathcal{D} \cup E_{\mathcal{S}} \cup E_{\mathcal{P}})| < e(G)$*, where* $\mathcal{D}$ *denotes the variable at the end of all updates and* $E_{\mathcal{P}}$ *denotes all intercluster edges of the partition* $\mathcal{P} \cup \{\tilde{V}\}$*. The algorithm runs in total time* $O\left(\frac{h \cdot |\Pi^{-1}(\mathcal{D} \cup E_{\mathcal{S}} \cup E_{\mathcal{P}})|}{\psi^2} + h \cdot \sum_{d \in \mathcal{D} \cup E_{\mathcal{S}} \cup E_{\mathcal{P}}} (1 + |\boldsymbol{f}_{t_d}(d)|)\right)$*.*

We present BiDirectedExpanderPruning (see Algorithm 2), which implements this reduction. The Init function initializes two out-expander pruning data structure. One for the graph $G$ with witness $(W, \Pi)$ with variables $\tilde{V}_1, \mathcal{P}_1, E_1^r$ and one for the graph $G^{rev}$ with witness $(W, \Pi)$ with variables $\tilde{V}_2, \mathcal{P}_2, E_2^r$. From the guarantees of the one directional DirectedExpanderPruning algorithm, we will have that after each update $G[\tilde{V}_1]$ and $G^{rev}[\tilde{V}_2]$ are out-expanders. To conclude that the $G[\tilde{V}]$ is in fact an expander in both directions, we will ensure using the function AdjustPartition that $\tilde{V}_1, \tilde{V}_2$ agree. If they do not agree, we will remove the vertices $\tilde{V}_1 \setminus \tilde{V}_2$ from $\tilde{V}_1$ using the function RemoveVertices of the algorithm DirectedExpanderPruning (see Algorithm 3). Using the function RemoveEdges of BiDirectedExpanderPruning, we can remove edges from $G[\tilde{V}]$. This is again accomplish by calling the analogous function RemoveEdges of DirectedExpanderPruning for both directions. Again we have to adjust the partition such that $\tilde{V}_1, \tilde{V}_2$ agree afterwards. The algorithm and the proof are defered to Appendix A.3.

## References

[BBST24] Aaron Bernstein, Joakim Blikstad, Thatchaphol Saranurak, and Ta-Wei Tu. Maximum flow by augmenting paths in $n^{2+o(1)}$ time. *to appear at FOCS'24*, 2024. , 1, 3

[BGS20] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1123–1134. IEEE, 2020. , 1, 2, 3, 5, 11, 18

[BGS22] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1000–1008. IEEE, 2022. 1, 17, 18

[BGWN20] Aaron Bernstein, Maximilian Probst Gutenberg, and Christian Wulff-Nilsen. Near-optimal decremental sssp in dense weighted digraphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1112–1122. IEEE, 2020. 3

[BNWN22] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 600–611. IEEE, 2022. 17

[BPWN19] Aaron Bernstein, Maximilian Probst, and Christian Wulff-Nilsen. Decremental strongly-connected components and single-source reachability in near-linear time. In

*Proceedings of the 51st Annual ACM SIGACT Symposium on theory of computing*, pages 365–376, 2019. 18

[BvdBPG+22] Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022. 1, 18

[CDK+21] Parinya Chalermsook, Syamantak Das, Yunbum Kook, Bundit Laekhanukit, Yang P Liu, Richard Peng, Mark Sellke, and Daniel Vaz. Vertex sparsification for edge connectivity. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1206–1225. SIAM, 2021. 1, 17

[CGL+20] 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. 18, 22

[CGP+20] 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*, (0):FOCS18–85, 2020. 17

[Chu21] Julia Chuzhoy. Decremental all-pairs shortest paths in deterministic near-linear time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 626–639, 2021. 1, 18

[CK19] Julia Chuzhoy and Sanjeev Khanna. A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 389–400, 2019. 1, 18

[CK24a] Julia Chuzhoy and Sanjeev Khanna. A faster combinatorial algorithm for maximum bipartite matching. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2185–2235. SIAM, 2024. , 3, 17

[CK24b] Julia Chuzhoy and Sanjeev Khanna. Maximum bipartite matching in $n^{2+o(1)}$ time via a combinatorial algorithm. *to appear at STOC'24*, 2024. , 1, 3

[CKL+22] 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. 2, 17, 18

[CKL+24] 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. 1, 17

[CS21] Julia Chuzhoy and Thatchaphol Saranurak. Deterministic algorithms for decremental shortest paths via layered core decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2478–2496. SIAM, 2021. 1, 18

[GRST21] Gramoz Goranci, Harald Räcke, Thatchaphol Saranurak, and Zihan Tan. The expander hierarchy and its applications to dynamic graph algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2212–2228. SIAM, 2021. 1, 18

[GT88] Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988. , 9

[HKGW23] 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. , 1, 2, 3, 5, 6, 8, 11

[Hoc08] Dorit S Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations research*, 56(4):992–1009, 2008. 5

[HRW20] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020. , 1

[JS22] Wenyu Jin and Xiaorui Sun. Fully dynamic st edge connectivity in subpolynomial time. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 861–872. IEEE, 2022. 1, 18

[JST24] Wenyu Jin, Xiaorui Sun, and Mikkel Thorup. Fully dynamic min-cut of superconstant size in subpolynomial time. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2999–3026. SIAM, 2024. 1, 18

[KLOS14] 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. 1, 17

[KMG24] Rasmus Kyng, Simon Meierhans, and Maximilian Probst Gutenberg. A dynamic shortest paths toolbox: Low-congestion vertex sparsifiers and their applications. *to appear at STOC'24*, 2024. 1, 17

[KMP22] 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. 1, 17

[KRV09] Rohit Khandekar, Satish Rao, and Umesh Vazirani. Graph partitioning using single commodity flows. *Journal of the ACM (JACM)*, 56(4):1–15, 2009. 2, 3, 11, 22, 23

[KT18] Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic edge connectivity in near-linear time. *Journal of the ACM (JACM)*, 66(1):1–50, 2018. 17

[Li21] 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. 1, 17

[Liu23] Yang P Liu. Vertex sparsification for edge connectivity in polynomial time. In *14th Innovations in Theoretical Computer Science Conference (ITCS 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023. 1, 17

[Lou10] Anand Louis. Cut-matching games on directed graphs. *arXiv preprint arXiv:1010.1047*, 2010. 2, 3, 11, 22, 23

[LSY19] Yang P Liu, Sushant Sachdeva, and Zejun Yu. Short cycles via low-diameter decompositions. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2602–2615. SIAM, 2019. 17

[NS17] Danupon Nanongkai and Thatchaphol Saranurak. Dynamic spanning forest with worst-case update time: adaptive, las vegas, and o (n1/2-$\varepsilon$)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1122–1129, 2017. 1, 18

[NSWN17] 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. 1, 5, 18

[PY19] Merav Parter and Eylon Yogev. Optimal short cycle decomposition in almost linear time. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. 17

[Sar21] Thatchaphol Saranurak. A simple deterministic algorithm for edge connectivity. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 80–85. SIAM, 2021. 1, 17

[ST04] 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. 1, 17

[SW19] Thatchaphol Saranurak and Di Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2616–2635. SIAM, 2019. , 1, 2, 3, 4, 6, 9, 11, 17, 20

[VDBCK$^+$24] Jan Van Den Brand, Li Chen, Rasmus Kyng, Yang P Liu, Simon Meierhans, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-linear time algorithms for decremental graphs: Min-cost flow and more via duality. *to appear at FOCS'24*, 2024. , 1, 2

[VDBCP$^+$23] 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. 1, 17

[VDBLL+21] Jan Van Den Brand, Yin Tat Lee, Yang P Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and l1-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 859–869, 2021. 1, 17

[vdBLN+20] Jan van den Brand, Yin-Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 919–930. IEEE, 2020. 17

[WN17] 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. 1, 18

# A Appendix

## A.1 Previous Work

**Expander Decompositions for Static Flow Problems.** In static graph settings, expander decompositions have been employed in many recent algorithms for electrical, maximum flow and min-cost flow problems. As mentioned, they were instrumental in the first Laplacian solver [ST04] that computes electrical flows, and still are used in recent Laplacian solvers, for example in the recent first almost-linear deterministic Laplacian solver for directed graphs [KMP22].
For the maximum and min-cost flow problems, expander decompositions have been crucial, as seen in [KLOS14, vdBLN+20, VDBLL+21, BGS22] and the recent development of an almost-linear time algorithm for max flow and min-cost flow in directed graphs [CKL+22]. In [CKL+22], the static min-cost flow problem in a directed graph is transformed via advanced convex optimization methods into a dynamic problem in an undirected graph. This dynamic problem is then solved efficiently by a data structure that uses the undirected expander decomposition algorithm from [SW19] internally. By simple reductions, the result in [CKL+22] also gave the first almost-linear time algorithms for the problems of negative Single Source Shortest Path (SSSP) and bipartite matching, but also to compute expanders in directed graphs.
Since the breakthrough result in [CKL+22] (and follow-up work [VDBCP+23, KMG24, CKL+24]), a natural new research initiative has emerged: can we solve the min-cost flow problem without relying on advanced convex optimization methods, or put differently, can it be solved with purely *combinatorial* methods? This question aims to further our understanding of the min-cost flow problem by developing a radically different (possibly more accessible) perspective but is also motivated by the quest to find a simpler and more practical algorithm. This initiative has already led to significant achievements, including a near-linear time algorithm for Negative SSSP [BNWN22] and a purely combinatorial approach to bipartite matching [CK24a] that improves current combinatorial approaches in dense graphs, a barrier that stood since the 80s.
Directed expander decomposition has emerged as a critical tool in this landscape, exemplified by the work of [BNWN22], who utilized directed low-diameter decompositions akin to directed expander decompositions, and [CK24a], who directly applied directed expander decompositions. The aim of our new accessible directed expander decomposition framework is to further accelerate this essential research initiative, contributing significantly to the field of graph algorithms.

**Expander Decompositions for Graph Problems beyond Flows.** In undirected graphs, expander decompositions have also been crucial in all deterministic almost-linear time global min-cut algorithms for undirected graphs [KT18, Sar21, Li21] in computing short-cycle decompositions [CGP+20, PY19, LSY19], and in finding min-cut preserving vertex sparsifiers [CDK+21, Liu23].
It is noteworthy that the above achievements pertain exclusively to undirected graphs. Directed graphs have yet to benefit from the application of expander decompositions. In the directed setting only considerably less efficient algorithms are known. We hope that our directed expander decomposition framework will facilitate adapting the existing methodologies used in undirected graphs to the directed context, or will inspire novel strategies to address these algorithmic challenges.

**Expander Decompositions in Dynamic Graphs.** In dynamic graphs, which are characterized by ongoing edge insertions and deletions, expanders have played a significant role in the undirected setting. They have been fundamental in achieving new worst-case update time and derandomization results in dynamic connectivity [WN17, NS17, NSWN17, CGL+20], in single-source shortest paths [CK19, BGS20, CS21, Chu21, BGS22], in approximate $(s,t)$-max-flow and $(s,t)$-min-cut algorithms [GRST21], and in developing sparsifiers resistant to adaptive adversaries [BvdBPG+22, CKL+22]. Notably, they were also a key component in the first subpolynomial update time c-edge connectivity algorithm [JS22] and bounded-value min-cut algorithm [JST24].

In the context of directed graphs, there remains a significant gap in our understanding. A notable challenge is the absence of near-linear time solutions for many problems, such as decremental Single-Source Shortest Paths (SSSP). Where solutions do exist, such as for decremental Strongly Connected Components (SCC), they are typically effective only against oblivious adversaries, as highlighted in [BPWN19]. Furthermore, in scenarios where algorithms are devised to tackle adaptive adversaries, the trade-off is often a drastic reduction in speed, a fact exemplified in [BGS20]. However, the use of directed expander decompositions in these algorithms suggests that enhancing these decompositions could be key to developing faster and more robust algorithms for directed dynamic graphs.

### A.2 Push-Pull-Relabel Algorithm

**Algorithm 1** ValidState($G = (V, E), \boldsymbol{c}, \Delta, \nabla, h$)

```
def Init
    $\tilde{E} \leftarrow E(G), (\boldsymbol{c}, \Delta, \nabla) \leftarrow (\boldsymbol{c}, \Delta, \nabla), (\boldsymbol{f}, \boldsymbol{\ell}) \leftarrow (\mathbf{0}, \mathbf{0})$
    PushRelabel()

def IncreaseSource($\delta$)
    $\Delta \leftarrow \Delta + \delta$
    PushRelabel()

def RemoveEdges($D$)
    $\tilde{E} \leftarrow \tilde{E} \setminus D, (\boldsymbol{f}, \boldsymbol{\ell}) \leftarrow (\boldsymbol{f}|_{\tilde{E}}, \boldsymbol{\ell}|_{\tilde{V}})$
    PullRelabel()
    PushRelabel()

def PushRelabel()
    while $\exists v$ where $\boldsymbol{\ell}(v) < h$ and $\Gamma(v) > \nabla(v)$ do
        Let $v$ be a vertex minimizing $\boldsymbol{\ell}(v)$.
        if $\exists(v, u)$ such that $\boldsymbol{c}_f(v, u) > 0, \boldsymbol{\ell}(v) = \boldsymbol{\ell}(u) + 1$ then
            $\boldsymbol{f}(v, u) \leftarrow \boldsymbol{f}(v, u) + \frac{1}{2}, \boldsymbol{f}(u, v) \leftarrow -\boldsymbol{f}(v, u)$ // Sends $\frac{1}{2}$ units of pos. excess from $v$ to $u$
        else
            $\boldsymbol{\ell}(v) \leftarrow \boldsymbol{\ell}(v) + 1$
        end if
    end while

def PullRelabel()
    while $\exists v$ where $\boldsymbol{\ell}(v) > 0$ and $\Gamma(v) < \nabla(v)/2$ do
        Let $v$ be a vertex maximizing $\boldsymbol{\ell}(v)$
        if $\exists(u, v)$ such that $\boldsymbol{c}_f(u, v) > 0, \boldsymbol{\ell}(u) = \boldsymbol{\ell}(v) + 1$ then
            $\boldsymbol{f}(v, u) \leftarrow \boldsymbol{f}(v, u) - \frac{1}{2}, \boldsymbol{f}(u, v) \leftarrow -\boldsymbol{f}(v, u)$ // Sends $\frac{1}{2}$ units of neg. excess from $v$ to $u$
        else
            $\boldsymbol{\ell}(v) \leftarrow \boldsymbol{\ell}(v) - 1$
        end if
    end while
```

### A.3 Reduction to Out-Expanders

**Algorithm 2** BiDirectedExpanderPruning$(G, W, \Pi)$

```
def Init
    $(\tilde{V}_1, \mathcal{P}_1, E_1^r) \leftarrow$ DirectedExpanderPruning$(G, W, \Pi)$
    $(\tilde{V}_2, \mathcal{P}_2, E_2^r) \leftarrow$ DirectedExpanderPruning$(G^{rev}, W, \Pi)$
    $\tilde{V} \leftarrow \tilde{V}_1 \cap \tilde{V}_2, \mathcal{P} \leftarrow \mathcal{P}_1 \cup \mathcal{P}_2, E^r \leftarrow E_1^r \cup E_2^r$ // dynamically updated

def RemoveEdges$(D)$
    $(\tilde{V}_1, \mathcal{P}_1, E_1^r)$.RemoveEdges$(D)$
    $(\tilde{V}_2, \mathcal{P}_2, E_2^r)$.RemoveVertices$(\tilde{V}_2 \setminus \tilde{V}_1)$
    $(\tilde{V}_2, \mathcal{P}_2, E_2^r)$.RemoveEdges$(D)$
    AdjustPartition

def AdjustPartition()
    while $\tilde{V}_1 \neq \tilde{V}_2$ do
        $(\tilde{V}_1, \mathcal{P}_1, E_1^r)$.RemoveVertices$(\tilde{V}_1 \setminus \tilde{V}_2)$
        $(\tilde{V}_2, \mathcal{P}_2, E_2^r)$.RemoveVertices$(\tilde{V}_2 \setminus \tilde{V}_1)$
    end while
```

*Proof.* TOPROVE 4 □

### A.4 Maintaining Out-Expanders

It remains to prove Lemma 4.2. Algorithm 3 gives an implementation of DirectedExpanderPruning. The Init function initializes for an out-expander $(G, W, \Pi)$ the expander pruning variables $\tilde{V}, \mathcal{P}, E^r, \mathcal{D}$ as well as a first valid state $(\boldsymbol{f}, \boldsymbol{\ell})$ of the ValidState algorithm (see Algorithm 1). This valid state $(\boldsymbol{f}, \boldsymbol{\ell})$, we will use to find the pruning cuts in function Prune. Using RemoveEdges the user can remove edges $D$ from $G[\tilde{V}]$. Removing edges might force us to prune away some part of the $G[\tilde{V}]$ and add the pruned part to the collection of pruning sets $\mathcal{P}$. To find the pruning set $P$, we adopt a similar strategy as in [SW19]. We inject additional source flow into the flow problem: for any witness edge $e = (u, v) \in E(W)$, where an edge on the embedding path $\Pi(e)$ is in $D$, we increase the sources $\Delta(u), \Delta(v)$ by $\frac{4}{\psi}$. Since we updated the flow problem by increasing the source capacities and removing edges, we need to update the valid state $(\boldsymbol{f}, \boldsymbol{\ell})$. This new valid state $(\boldsymbol{f}, \boldsymbol{\ell})$, then allows us to locate the pruning set in the function AdjustPartition. In AdjustPartition, we check whether there is a vertex in the remaining graph $G[\tilde{V}]$ on level $h$. If there is no such vertex, it will certify that in fact $G[\tilde{V}]$ has already a good out-witness and we don't need to prune away any subgraph. If on the other hand, there is a vertex on level $h$ we will find a pruning set $P$ using the function Prune. This set $P$ is then added to the collection of pruning sets $\mathcal{P}$ and the vertices in $P$ are removed from $\tilde{V}$. The cut edges $E_{G \setminus \mathcal{D}}(P, \tilde{V})$ are added to the remove edges $E^r$. Since $\tilde{V}$ has become smaller, we need to update the valid state $(\boldsymbol{f}, \boldsymbol{\ell})$ again. We do this once more by injecting additional source at the endpoints of any witness edge $e \in E(W)$, where some edge on the path $\Pi(e)$ is in $E_{G \setminus \mathcal{D}}(\tilde{V}, P)$. Thereafter we again check whether in the new valid state $(\boldsymbol{f}, \boldsymbol{\ell})$ there is a vertex in $\tilde{V}$ on level $h$. We keep on doing this procedure until there no longer is a vertex on level $h$. We will prove that the volume of the pruning sets can be related to the size of the set of edges $D$ initially removed by the user. So far we have not explained how to find the pruning set in function Prune. This is accomplished by a standard level cutting procedure. We start with $P$ being all vertices on level $h$ and then check whether the vertices on the next level have volume at least $\phi \cdot \mathrm{vol}_G(P)$. If it is the case, we add vertices on the next level to $P$ and otherwise return $P$.

Through the function RemoveVertices the user can remove vertices $S$ from $\tilde{V}$. Similar to the function RemoveEdges, we will have to inject additional source at the endpoints of the witness edges $e \in E(W)$, where some edge of $\Pi(e)$ is in $E_{G \setminus \mathcal{D}}(\tilde{V}, S)$, and update the valid state $(\boldsymbol{f}, \boldsymbol{\ell})$ accordingly. This might again leave some vertices on level $h$ and will again force us to prune some vertices. This is again accomplished by a call to AdjustPartition. And similar to RemoveEdges, we will again prove that the volume of the pruning sets can be related to the volume of the set $S$ initially removed by the user.

**Algorithm 3** DirectedExpanderPruning$(G, W, \Pi)$

```
def Init
    $\tilde{V} \leftarrow V, \mathcal{P} \leftarrow \emptyset, E^+_{\mathcal{S}} \leftarrow \emptyset, E^-_{\mathcal{S}} \leftarrow \emptyset, E^r \leftarrow \emptyset, \mathcal{D} \leftarrow \emptyset$
    $(\boldsymbol{f}, \boldsymbol{\ell}) \leftarrow \textsc{ValidState}(G, \frac{18}{\phi\psi^2} \cdot \mathbf{1}, \mathbf{0}, \deg_W, h)$

def RemoveEdges$(D)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup D$
    $(\boldsymbol{f}, \boldsymbol{\ell}).\textsc{RemoveEdges}(D)$
    $(\boldsymbol{f}, \boldsymbol{\ell}).\textsc{IncreaseSource}(\frac{4}{\psi} \deg_{\Pi^{-1}(D)})$
    AdjustPartition()

def RemoveVertices$(S)$
    $\tilde{V} \leftarrow \tilde{V} \setminus S, E^+_{\mathcal{S}} \leftarrow E^+_{\mathcal{S}} \cup E_{G \setminus \mathcal{D}}(S, \tilde{V}), E^-_{\mathcal{S}} \leftarrow E^-_{\mathcal{S}} \cup E_{G \setminus \mathcal{D}}(\tilde{V}, S)$
    $(\boldsymbol{f}, \boldsymbol{\ell}).\textsc{RemoveEdges}\left(E_{G \setminus \mathcal{D}}(\tilde{V}, S) \cup E_{G \setminus \mathcal{D}}(S, \tilde{V})\right)$
    $(\boldsymbol{f}, \boldsymbol{\ell}).\textsc{IncreaseSource}(\frac{4}{\psi} \deg_{\Pi^{-1}(E_{G \setminus \mathcal{D}}(\tilde{V}, S) \cup E_{G \setminus \mathcal{D}}(S, \tilde{V}))})$
    AdjustPartition()

def AdjustPartition()
    while $\exists v \in \tilde{V}$ with $\boldsymbol{\ell}(v) = h$ do
        $P \leftarrow \textsc{Prune}(G \setminus \mathcal{D}, \boldsymbol{\ell})$
        $\mathcal{P} \leftarrow \mathcal{P} \cup \{P\}, \tilde{V} \leftarrow \tilde{V} \setminus P, E^r \leftarrow E^r \cup E_{G \setminus \mathcal{D}}(P, \tilde{V})$
        $(\boldsymbol{f}, \boldsymbol{\ell}).\textsc{RemoveEdges}(E_{G \setminus \mathcal{D}}(P, \tilde{V}) \cup E_{G \setminus \mathcal{D}}(\tilde{V}, P))$
        $(\boldsymbol{f}, \boldsymbol{\ell}).\textsc{IncreaseSource}(\frac{4}{\psi} \deg_{\Pi^{-1}\left(E_{G \setminus \mathcal{D}}(P, \tilde{V}) \cup E_{G \setminus \mathcal{D}}(\tilde{V}, P)\right)})$
    end while

def Prune$(G, \boldsymbol{\ell})$
    $S \leftarrow \emptyset, i \leftarrow h$
    repeat
        $S \leftarrow S \cup \left\{v \in \tilde{V} \mid \boldsymbol{\ell}(v) = i\right\}$
        $i \leftarrow i - 1$
    until $\mathrm{vol}_{G \setminus (\mathcal{D} \cup E^-_{\mathcal{S}})}\left(\{v \in \tilde{V} \mid \boldsymbol{\ell}(v) \leq i\}\right) < (1 + \frac{\phi}{72}) \cdot \mathrm{vol}_{G \setminus (\mathcal{D} \cup E^-_{\mathcal{S}})}(S)$
    return the cut $S$
```

*Proof.* TOPROVE 5 □

### A.5 Static Expander Decomposition

In this section, we discuss how we can use the algorithm BiDirectedExpanderPruning of Theorem 4.1 as a subroutine for a static expander decomposition. But before we turn to the outline of the algorithm, we recall the directed version of the cut-matching game. We point out that in the theorem statements of [KRV09, Lou10] there is no mention of **fake** edges as used below but these can be gleaned off the algorithmic description as first observed in [CGL+20].

**Theorem A.1** ([KRV09, Lou10])**.** *Given a directed graph $G = (V, E)$ of $m$ edges and a parameter $\phi$, the cut-matching game takes $O((m \log^3 m)/\phi)$ time and either returns*

1. *a $O(1/\log^2(m))$-witness $(W, \Pi)$ certifying that $G \cup \mathcal{F}$ is a $\phi$-expander for some set of* ***fake*** *edges $\mathcal{F}$ where $|\Pi^{-1}(\mathcal{F})| \leq c \cdot \frac{m}{\log^4(m)}$, where $c > 0$, or*
2. *a balanced sparse cut $(A, \bar{A})$ in $G$: $e_G(A, \bar{A}) \leq O\left(\phi \cdot \log^2(m) \cdot \min(\mathrm{vol}_G(A), \mathrm{vol}_G(\bar{A}))\right)$ such that $\mathrm{vol}_G(A), \mathrm{vol}_G(\bar{A}) = \Omega(c \cdot m/\log^6 m)$.*

The statement above slightly deviates from well-known cut-matching game formulations. It is more common that the cut-matching game either certifies that $G$ is a $\phi$-expander or provides a cut that might be unbalanced. But it is straightforward to obtain the formulation in Theorem A.1. Recall that the cut-matching algorithm attempts to embed a witness using $O(\log^2(m))$ single commodity flows. A cut $(A, \bar{A})$ is provided if the algorithm fails to route one of these network flows. If one uses the push-relabel algorithm for routing these single commodity flows, it is easy to see that one obtains a pre-flow $\boldsymbol{f}$ such that all positive excess flow is stuck on the smaller side of the cut and the total amount is at most $\min(\mathrm{vol}_G(A), \mathrm{vol}_G(\bar{A}))$. Thus, one can readily find a source-sink pair matching $F$ of size at most $\min(\mathrm{vol}_G(A), \mathrm{vol}_G(\bar{A}))$ and extend the pre-flow to an actual routing in $G \cup F$. Indeed if the cut $(A, \bar{A})$ is unbalanced, the algorithm picks such a set of fake edges $F$ and routes the remaining excess flow along these edges. It then continues with routing the next single-commodity flow in $G$. If the algorithm eventually manages to embed a witness, the witness will actually be embedded into $G \cup \mathcal{F}$, where $\mathcal{F}$ is the union of all the fake edge sets $F$ added over all rounds.

In Algorithm 4, we then use BiDirectedExpanderPruning to remove the set $\mathcal{F}$ of fake edges from $G \cup \mathcal{F}$ making only marginal adjustments to the witness embedding. In the first line of the algorithm, we check if the CutMatching game provides a cut or a witness embedding. If it provides a cut, we recurse on the two sides of the cut. If it provides a witness embedding, we remove the fake edges using the function RemoveEdges. This subroutine initializes an instance of BiDirectedExpanderPruning for the graph and computes a pruning set for the deleted edges $\mathcal{F}$. In the end, the subroutine computes expander decompositions for the pruning sets and combines those into an expander decompositon of the entire graph.

**Theorem A.2.** *Given a directed graph $G$, we can compute a $\left(O(\log^{19} m), \lambda, O\left(\frac{1}{\log^4 m}\right)\right)$-expander decomposition $(\mathcal{X}, E_r)$ in run time $O(m \log^{20}(m)/\phi)$ provided $\lambda \leq O(1/\log^{12}(m))$.*

*Proof.* TOPROVE 6 □

**Algorithm 4** ExpanderDecomposition($G$)

```
if CutMatching($G$) provides a cut $(A, \bar{A})$ then
    $(\mathcal{X}_1, E_1^r) \leftarrow$ ExpanderDecomposition($G[A]$)
    $(\mathcal{X}_2, E_2^r) \leftarrow$ ExpanderDecomposition($G[\bar{A}]$)
    return $(\mathcal{X}_1 \cup \mathcal{X}_2, E_1^r \cup E_2^r \cup E_G(A, \bar{A}))$
else
    CutMatching($G$) provides $(W, \Pi)$ embedded into $G \cup \mathcal{F}$
    return RemoveEdges($G, W, \Pi, \mathcal{F}$)
end if

def RemoveEdges($G, W, \Pi, \mathcal{D}$)
    $((\tilde{V}, \tilde{W}, \tilde{\Pi}), \mathcal{P}, E_0^r) \leftarrow$ BiDirectedExpanderPruning($G, W, \Pi$)
    $((\tilde{V}, \tilde{W}, \tilde{\Pi}), \mathcal{P}, E_0^r)$.RemoveEdges($\mathcal{D}$)
    $\mathcal{X} \leftarrow \{(\tilde{V}, \tilde{W}, \tilde{\Pi})\}, E^r \leftarrow E_0^r$
    for $P \in \mathcal{P}$ do
        $(\mathcal{X}_1, E_1^r) \leftarrow$ ExpanderDecomposition($P$)
        $\mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{X}_1$
        $E^r \leftarrow E^r \cup E_1^r$
    end for
    return $(\mathcal{X}, E^r)$
```

## A.6 Dynamic Expander Decomposition

In this section, we discuss how we can use the algorithm BiDirectedExpanderPruning and the algorithm ExpanderDecomposition as a subroutines for a dynamic expander decomposition. The algorithm is given in DynamicExpanderDecomposition (see Algorithm 5). We initialize the data structure with an expander decomposition $(\mathcal{X}_0, E_0^r)$. The data structure then initializes in the lines 3-5, for each component of the expander decomposition an instance of BiDirectedExpanderPruning. For any edge deletion $d$, the data structure envokes the function RemoveEdges. This algorithm, first finds the component $X$ such that $d \in E(X)$ and then deletes the edge from that expander component using the functions of BiDirectedExpanderPruning. This might potentially require to prune away additional subgraphs of the expander $X$. We replace $X$ in the expander decomposition $\mathcal{X}$ by the remainder of $X$ and add the cut edges of the pruning cuts to $E^r$. For any of these pruned subgraphs, we run the static expander decomposition to obtain an expander decomposition $(\mathcal{X}_1, E_1^r)$ and add the components of $\mathcal{X}_1$ to $\mathcal{X}$ and the edges of $E_1^r$ to $E^r$.

**Theorem A.3.** *For every $(\beta, \phi, \psi)$-expander decomposition $(\mathcal{X}, E_r)$ of a directed graph $G$, there is a randomized data structure* DynamicExpanderDecomposition($G$) *(see Algorithm 5). For up to $c \cdot \phi \cdot \psi \cdot e(G)$ calls, where $c$ is a fixed constant, of the form*

- RemoveEdge($d$)*: where $d \in E(V)$, adds $d$ to $\mathcal{D}$ (initially $\mathcal{D} = \emptyset$)*

*the algorithm explicitly updates the tuple $(\mathcal{X}, E^r)$ after each call, such that thereafter $(\mathcal{X}, E^r)$ is a $\left(4 \cdot \beta, \frac{\phi\psi^6}{32000}, \frac{\psi^4}{800}\right)$-expander-decomposition for $G \setminus \mathcal{D}$ refining the previous. The run-time is bounded by $O\left(\frac{|\mathcal{D}|}{\phi^2} \cdot \log e(G) \cdot \max\left(\log^{19} |\mathcal{D}|, \frac{1}{\psi^2}\right)\right)$.*

**Algorithm 5** DynamicExpanderDecomposition($\mathcal{X}_0, E_0^r$)

```
def Init
    $(\mathcal{X}, E^r) \leftarrow (\mathcal{X}_0, E_0^r)$
    for $X \in \mathcal{X}$ do
        $(V_X, \mathcal{P}_X, E_X^r) \leftarrow$ BiDirectedExpanderPruning$(X, W_X, \Pi_X)$
    end for

def RemoveEdge($d$)
    Find $(X, W, \Pi) \in \mathcal{X}$ such that $d \in E(X)$.
    $((V_X, W_X, \Pi_X), \mathcal{P}_X, E_X^r)$.RemoveEdges($d$)
    Replace $(X, W, \Pi)$ by $(V_X, W_X, \Pi_X)$
    for $P$ new in $\mathcal{P}_X$ do
        $(\mathcal{X}_1, E_1^r) \leftarrow$ ExpanderDecomposition$(P)$
        $\mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{X}_1$
        $E^r \leftarrow E^r \cup E_1^r$
    end for
```

*Proof.* TOPROVE 7 □

To obtain Theorem 1.2, we combine Theorem A.2 and Theorem A.3. After $O(\phi \cdot \psi \cdot e(G))$ deletions, we restart the maintenance of the expander decomposition with Theorem A.2.