ABSTRACT

Computing routing schemes that support both high throughput and low latency is one of the core challenges of network optimization. Such routes can be formalized as \( h \)-length flows which are defined as flows whose flow paths have length at most \( h \). Many well-studied algorithmic primitives—such as maximal and maximum length-constrained disjoint paths—are special cases of \( h \)-length flows. Likewise the optimal \( h \)-length flow is a fundamental quantity in network optimization, characterizing, up to poly-log factors, how quickly a network can accomplish numerous distributed primitives.

In this work, we give the first efficient algorithms for computing \((1 - \epsilon)\)-approximate \( h \)-length flows that are nearly “as integral as possible.” We give deterministic algorithms that take \( \tilde{O}(\text{poly}(h, \frac{1}{\epsilon})) \) parallel time and \( \tilde{O}(\text{poly}(h, \frac{1}{\epsilon}) \cdot 2^O(\sqrt{\log n})) \) distributed CONGEST time. We also give a CONGEST algorithm that succeeds with high probability and only takes \( \tilde{O}(\text{poly}(h, \frac{1}{\epsilon})) \) time.

Using our \( h \)-length flow algorithms, we give the first efficient deterministic CONGEST algorithms for the maximal disjoint paths problem with length constraints—settling an open question of Chang and Saranurak (FOCS 2020)—as well as essentially-optimal parallel and distributed approximation algorithms for maximum length-constrained disjoint paths. The former greatly simplifies deterministic CONGEST algorithms for computing expander decompositions. We also use our techniques to give the first efficient and deterministic \((1 - \epsilon)\)-approximation algorithms for bipartite \( b \)-matching in CONGEST. Lastly, using our flow algorithms, we give the first algorithms to efficiently compute \( h \)-length cutmatch, an object at the heart of recent advances in length-constrained expander decompositions.

CCS CONCEPTS

- Mathematics of computing → Graph algorithms: Network flows: Approximation algorithms.

KEYWORDS

length-bounded flows, hop-bounded flows, distributed algorithms, parallel algorithms, flow rounding, cycle covers, \( b \)-matching

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

STOC '23, June 20–23, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9913-5/23/06...
$15.00
https://doi.org/10.1145/3564246.3585202

1 INTRODUCTION

Throughput and latency are two of the most fundamental quantities in a communication network. Given node sets \( S \) and \( T \), throughput measures the rate at which bits can be delivered from \( S \) to \( T \) while the worst-case latency measures the maximum time it takes for a bit sent from \( S \) to arrive at \( T \). Thus, a natural question in network optimization is:

\textbf{How can we achieve high throughput while maintaining a low latency?}

If we imagine that each edge in a graph incurs some latency and edges in a graph can only support limited bandwidth, then achieving high throughput subject to a latency constraint reduces to finding a large collection of paths that are both short and non-overlapping. One of the simplest and most well-studied ways of formalizing this is the maximal edge-disjoint paths problems (henceforth we use \( h \)-length to mean length \( at most \) \( h \)).

Maximal Edge-Disjoint Paths: Given graph \( G = (V, E) \), length constraint \( h \geq 1 \) and two disjoint sets \( S, T \subseteq V \), find a collection of \( h \)-length edge-disjoint \( S \) to \( T \) paths \( P \) such that any \( h \)-length \( S \) to \( T \) path shares an edge with at least one path in \( P \).

The simplicity of the maximal edge-disjoint paths problem has made it a crucial primitive in numerous algorithms. For example, algorithms for maximal edge-disjoint paths are used in approximating maximum matchings [35] and computing expander decompositions [16, 41]. While efficient randomized algorithms are known for maximal edge-disjoint paths in the CONGEST model of distributed computation [13, 35], no deterministic CONGEST algorithms are known. Indeed, the existence of such algorithms was stated as an open question by [13].

Of course, a maximal collection of routing paths need not be near-optimal in terms of cardinality and so a natural extension of the above problem is its maximum version.

Maximum Edge-Disjoint Paths: Given graph \( G = (V, E) \), length constraint \( h \geq 1 \) and disjoint sets \( S, T \subseteq V \), find a max cardinality collection of \( h \)-length edge-disjoint \( S \) to \( T \) paths.

While this problem and its variants have received considerable attention [10, 12, 33], it is unfortunately known to suffer from strong...
hardness results: the above problem has an $\Omega(h)$ integrality gap and is $\Omega(h)$-hard to approximate under standard complexity assumptions in the directed case [8, 24]. Indeed, as observed in several works [3, 25, 33], working in the presence of latency bounds in the form of a length constraint can make otherwise tractable problems computationally infeasible and render otherwise structured objects poorly behaved.

The above problems are common primitives because their solutions are special cases of a more general class of routing schemes that are central to distributed computing. Length-constrained flows.

**Maximum Length-Constrained Flow**: Given digraph $D = (V, A)$, length constraint $h \geq 1$ and two disjoint sets $S, T \subseteq V$, find a collection of $h$-length $S$ to $T$ paths $\mathcal{P}$ and a value $f_P \geq 0$ for $P \in \mathcal{P}$ where $\sum_{P \in \mathcal{P}} f_P \leq 1$ for every $a \in A$ and $\sum_P f_P$ is maximized.

In several formal senses, length-constrained flows are the problem that describes how to efficiently communicate in a network. [27] showed that, up to poly-log factors, the maximum length-constrained flow gives the minimum makespan of multiple unicasts in a network, even when (network) coding is allowed. Even stronger, the “best” length-constrained flow gives, up to poly-log factors, the optimal running time of a CONGEST algorithm for numerous distributed optimization problems, including minimum spanning tree (MST), approximate min-cut and approximate shortest paths [28].

Correspondingly, there has been considerable work on centralized, parallel and distributed algorithms for computing length-constrained flows [2, 5–7, 17]. Most notably, [3] gave efficient deterministic algorithms in the distributed ROUTERS model and [2] gave sequential algorithms that take time about $O(m^2)$. The principal downside of the former’s algorithms is that it may produce solutions that are arbitrarily fractional in the sense that they are a convex combination of arbitrarily-many integral solutions. The latter does not do this but does not clearly admit an efficient distributed or parallel implementation. Often, however, there is a need for efficient algorithms that produce (near) integral length-constrained flows; in particular computing many classic integral objects (such as matchings) reduce to length-constrained flows with $h = O(1)$ and so, if we hope to use length-constrained flows for computing such objects, we often require that these flows be (nearly) integral.

Thus, in summary a well-studied class of routing problems aims to capture both latency and throughput concerns. These problems are known to serve as important algorithmic primitives as well as complete characterization of the distributed complexity of many problems. However, the simplest of these problems (maximal edge-disjoint paths) lacks good deterministic CONGEST algorithms while the maximum version of this problem and its generalization, length-constrained flows, lack efficient algorithms with reasonable integrality guarantees.

**Our Contributions.** We give the first efficient algorithms for computing these objects in several models of computation.

**Algorithms for Length-Constrained Flows.** Given a digraph with $n$ nodes and $m$ arcs, our main theorem shows how to deterministically compute $h$-length flows that are $(1 - \varepsilon)$-approximate in $O(\text{poly}(h, \frac{1}{\varepsilon}))$ parallel time with $m$ processors and $O(\text{poly}(h, \frac{1}{\varepsilon})) \cdot 2^O(\sqrt{\log n})$ distributed CONGEST time. We additionally give a randomized CONGEST algorithm that succeeds with high probability and runs in time $\tilde{O}(\text{poly}(h, \frac{1}{\varepsilon}))$. Our distributed algorithms for length-constrained flows algorithms can be contrasted with the best distributed algorithms for (non-length-constrained) flows which run in $(d + \sqrt{n}) \cdot n^{o(1)}$ time [23], nearly matching an $\tilde{O}(d + \sqrt{n})$ lower bound of [42].

Not only do our algorithms work for general arc capacities (i.e. connection bandwidths), general lengths (i.e. connection latencies) and multi-commodity flow variants. Furthermore, they also compute a certifying dual solution; a so-called moving cut [5, 14, 28]. Lastly, and most critically, the flows we compute are nearly “as integral as possible”:

**Optimal Integrality.** For constant $\varepsilon > 0$ they are a convex combinations of $\tilde{O}(h)$ sets of arc-disjoint paths.

No near-optimal $h$-length flow can be a convex combination of $o(h)$ such sets since, by an averaging argument, this would violate the aforementioned $\Omega(h)$ integrality gap.

As an immediate consequence of our parallel algorithms we also get deterministic sequential algorithms running in $\tilde{O}(m \cdot \text{poly}(h, \frac{1}{\varepsilon}))$ which improves upon the aforementioned $O(m^2)$-dependence of [2]. Thus our work can be understood as getting the best of both notable prior works—the (near)-integrality of [2] and the efficiency of [5]—both of which are necessary for our applications. See Section 3 for a formal description.

**Applications of our Length-Constrained Flow Algorithms.** Using the optimal integrality of our solutions, we are able to achieve several new results.

**Maximal and Maximum Edge-Disjoint Paths.** First, as an almost immediate corollary of our length-constrained flow algorithms, we derive the first deterministic CONGEST algorithms for maximal edge-disjoint paths and essentially-optimal parallel and distributed algorithms for the maximum edge-disjoint paths problem as well as for many variants of these problems. The former result settles the open question of [13].

**Simpler Distributed Expander Decompositions Deterministically.** As a consequence of our maximal edge-disjoint paths algorithms, we are able to greatly simplify known distributed algorithms for deterministically computing expander decompositions.

We refer the reader to [13] for a more thorough overview of the area, but provide a brief synopsis here. An $(\varepsilon, \phi)$ expander decomposition removes an $\varepsilon$ fraction of edges from a graph so as to ensure that each remaining connected component has conductance at least $\phi$. Expander decompositions have led to many recent exciting breakthroughs, including in linear systems [43], unique games [4, 40, 44], minimum cut [32], and dynamic algorithms [37]. [13] gave the first deterministic CONGEST algorithms for constructing expander decompositions. However, most existing ways of computing expander decompositions repeatedly find maximal disjoint paths. As a result of the lack of such algorithms, the authors employ significant technical work-arounds, observing:

\[ \text{We use } \tilde{O} \text{ notation to suppress dependence on } \text{poly}(\log n) \text{ factors, "with high probability" to mean with probability at least } 1 - \frac{1}{n^{\Omega(1)}} \text{ and } d \text{ for graph diameter.} \]
In the deterministic setting, we are not aware of an algorithm that can [efficiently] solve [maximal disjoint paths]... [A solution to this problem would] simplify our deterministic expander decomposition and routing quite a bit. [13]

Our deterministic CONGEST algorithms for this problem when plugged into [13] provide a conceptual simplification of deterministic distributed algorithms by bringing them in line with known paradigms. Additionally, we note that the algorithm of [13] incurs a $2^\Omega(\sqrt{\log n})$ overhead regardless of the maximal disjoint paths algorithm used so further improvement requires a fundamentally different approach.

Bipartite $b$-Matching. Using our length-constrained flow algorithms, we give the first efficient and deterministic $(1-\epsilon)$ approximations for bipartite $b$-matching in CONGEST. $b$-matching is a classical problem in combinatorial optimization which generalizes matching where we are given a graph $G = (V,E)$ and a function $b : V \to \mathbb{Z}_{\geq 0}$. Our goal is to assign integer values to edges so that each vertex $v$ has at most $b(v)$ assigned value across its incident edges. $b$-matching and its variants have been extensively studied in distributed settings [1, 9, 11, 18–20, 29, 34]. A standard folklore reduction which replaces vertex $v$ with $b(v)$-non-adjacent copies and edge $e = \{u,v\}$ with a bipartite clique between the copies of $u$ and $v$ reduces $b$-matching to matching but requires overhead $\max_{(u,v) \in E} b(u) \cdot b(v)$ to run in CONGEST. Thus, the non-trivial goal here is a CONGEST algorithm whose running time does not depend on $b$. While $b$-matching has been extensively studied in distributed settings, currently all that is known is either deterministic algorithms which give $(1/2-\epsilon)$-approximations in $\tilde{O}(\text{poly}(\log n))$ time [19] or randomized $(1-\epsilon)$-approximations in $\tilde{O}(\text{poly}(1/\epsilon))$ time but which only allow for each edge to be chosen at most once [29].

Similarly to classical matching, it is easy to reduce bipartite $b$-matching to $O(1)$-length flow. Thus, applying our flow algorithms and our flow rounding techniques allows us to give the first $(1-\epsilon)$-approximation for $b$-matching in bipartite graphs running in CONGEST time $\tilde{O}(\text{poly}(1/\epsilon) \cdot 2^\Omega(\sqrt{\log n}))$. Our algorithms are deterministic and work for the more general problem where each edge has some capacity indicating the number of times it may be chosen. See Section 12.

Length-Constrained Cutmatches. Our results allow us to give the first efficient constructions of length-constrained cutmatches. Informally, an $h$-length cutmatch with congestion $\gamma$ is a collection of $h$-length $\gamma$-congestion paths between two vertex subsets along with a moving cut that shows that adding any more $h$-length paths to this set would incur congestion greater than $\gamma$.

A recent work [26] uses our length-constrained cutmatches algorithms to give the first efficient constructions of length-constrained expander decompositions. This work uses these constructions to give CONGEST algorithms for problems, including MST, $(1+\epsilon)$-min-cut and $(1+\epsilon)$-shortest paths, that are guaranteed to run in sub-linear rounds if such algorithms exist on the network.

2 NOTATION AND CONVENTIONS

Before moving on to a formal statement of length-constrained flows, moving cuts and our results we introduce some notation and conventions. Suppose we are given a digraph $D = (V,A)$.

Digraph Notation. We will associate three functions with the arcs of $D$. We clarify these here.

(1) Lengths: We will let $\ell = \{\ell_a\}_a$ be the lengths of arcs in $A$. These lengths will be input to our problem and determine the lengths of paths when we are computing length-constrained flows. Throughout this work we imagine each $\ell_a$ is in $\mathbb{Z}_{\geq 0}$. Informally, one may think of $\ell$ as giving link latencies. We assume $\ell_a$ is poly(n).

(2) Capacities: We will let $U = \{U_a\}_a$ be the capacities of arcs in $A$. These capacities will specify a maximum amount of flow (either length-constrained or not) that is allowed over each arc. Throughout this work we imagine each $U_a$ is in $\mathbb{Z}_{\geq 0}$ and we let $U_{\text{max}}$ give max$_a U_a$. We assume $U_{\text{max}}$ is poly(n). Informally, one may think of $U$ as link bandwidths.

(3) Weights: We will let $w = \{w_w\}_w$ stand for the weights of arcs in $A$. These weights will be given by our moving cut solutions. Throughout this work each $w_w$ will be in $\mathbb{R}_{>0}$.

In general we will treat a path $P = \{(v_1, v_2), (v_2, v_3), \ldots\}$ as series of consecutive arcs in $A$ (all oriented consistently towards one endpoint). For any one of these weighting functions $\phi \in \{\ell, U, w\}$, we will let $d_{\phi}(u,v)$ give the minimum value of a path in $D$ that connects $u$ and $v$ where the value of a path $P$ is $\phi(P) := \sum_{a \in P} \phi(a)$. That is, we think of $d_{\phi}(u,v)$ as the distance from $u$ to $v$ with respect to $\phi$. We will refer to paths which minimize $w$ as lightest paths (so as to distinguish them from e.g. shortest paths with respect to $\ell$).

We let $d^\ell(u,v) := \{a : a = (u,v)\}$ and $N^\ell(u,v) := \{a : (u,v) \in A\}$ give the out arcs and out neighborhoods of vertex $v$. $d^\ell(v) := \{a : a = (u,v)\}$ and $N^\ell(v) := \{a : (u,v) \in A\}$ are defined symmetrically. We let $P^\ell(u,v)$ be all simple paths between $u$ and $v$ and for $W, W' \subseteq V$, we let $P^\ell(W, W') := \bigcup_{w \in W, w' \in W'} P^\ell(w, w')$ give all paths between vertex subsets $W$ and $W'$.

Given sources $S \subseteq V$ and sinks $T \subseteq V$, we say that $D$ is an $S$-$T$ DAG if $d^\ell(v) = \emptyset$ iff $v \in S$ and $d^\ell(u) = \emptyset$ iff $v \in T$. We say that such an $S$-$T$ DAG is an $h$-layer DAG if the vertex set $V$ can be partitioned into $h+1$ layers $S = V_1 \cup V_2 \cup \ldots \cup V_{h+1} = T$ where any arc $a = (u,v)$ is such that $u \in V_i$ and $v \in V_{i+j}$ for some $i$ and $j > i$. We say that $D$ has diameter at most $d$ if in the graph where we forget arc directions every pair of vertices is connected by a path of at most $d$ edges. Notice that the diameter of an $h$-layer $S$-$T$ DAG might be much larger than $h$.

For a (di)graph $D = (V,A)$ and a collection of subgraphs $\mathcal{H}$ of $D$, we let $D[\mathcal{H}]$ be the graph induced by the union of elements of $\mathcal{H}$. $A[\mathcal{H}]$ is defined as all arcs contained in some element of $\mathcal{H}$.

(Non-Length Constrained) Flow Notation and Conventions. We will make extensive use of non-length constrained flows and so clarify our notation for such flows here.
Given a DAG $D = (V, A)$ with capacities $U$ we will let a flow $f$ be any assignment of non-negative values to arcs in $a$ where $f_a$ gives the value that $f$ assigns to $a$ and $f_a \leq U_a$ for every $a$. If it is ever the case that $f_a > U_a$ for some $a$, we will explicitly state that this "flow" does not respect capacities. We say that $f$ is an integral flow if it assigns an integer value to each arc. We let $f(A') := \sum_{a \in A} f_a$ for any $A' \subseteq A$. We define the deficit of a vertex $v$ as $\text{deficit}(f, v) := \sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a$. We will let $\text{supp}(f) := \{a : f_a > 0\}$ give the support of flow $f$.

Given desired sources $S$ and sinks $T$, let

$$\text{deficit}(f) := \sum_{a \in S \cup T} \text{deficit}(f, a)$$

be the total amount of flow produced but not at $S$ plus the amount of flow consumed but not at $T$; likewise, we say that a flow $f$ is an $S$-$T$ flow if $\text{deficit}(f) = 0$. We let $\text{val}(f) = \bigcup_{s \in S} f(\delta^+(s))$ be the amount of flow delivered by an $S$-$T$ flow $f$ and we say that $f$ is $\alpha$-approximate if $\text{val}(f) \geq \alpha \cdot \text{val}(f^*)$ where $f^*$ is the $S$-$T$ flow that maximizes $\text{val}$. We say that $f$ is $\alpha$-blocking for $\alpha \in [0, 1]$ if for every path from $S$ to $T$ there is some $a \in P$ where $f_a \geq \alpha \cdot U_a$. We say that a 1-blocking flow is blocking. We say that flow $f'$ is a subflow of $f$ if $f'_a \leq f_a$ for every $a$.

Given a maximum capacity of $U_{\text{max}}$, we may assume that every flow $f$ is of the form $f = \sum_i f^{(i)}$ where $f^{(i)}$ is an $(S_i, T_i)$ flow for every $a$ and $t$; that is, a given flow can always be decomposed into its values on each bit. We call $f^{(i)}$ the $i$th bit flow of $f$ and call the decomposition of $f$ into these bits the bitwise decomposition of $f$.

**Length-Constrained Notation.** Given a length function $\ell$, vertices $u, v \in V$ and length constraint $h \geq 1$, we let $P_h(u, v) := \{P \in P(u, v) : \ell(P) \leq h\}$ be all paths between $u$ and $v$ which have length at most $h$. For vertex sets $W$ and $W'$, we let $P_h(W, W') := \{P \in P(W, W') : \ell(P) \leq h\}$. If $G$ also has weights $w$ then we let $d^h_w(u, v) := \min_{P \in P_h(u, v)} w(P)$ give the minimum weight of a path at most $h$ path connecting $u$ and $v$. For vertex sets $W$ and $W'$ we define $d_w^h(W, W') := \min_{P \in P_h(W, W')} w(P)$ analogously. As mentioned an $h$-length path is a path at length at most $h$.

**Parallel and Distributed Models.** Throughout this work the parallel model of computation we will use is the EREW PRAM model [31]. Here we are given some processors and shared random access memory; every memory cell can be read or written to by one processor at a time.

The distributed model we will make use of is the CONGEST model, defined as follows [39]. The network is modeled as a graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. Communication is conducted over discrete, synchronous rounds. During each round each node can send an $O(\log n)$-bit message along each of its incident edges. Every node has an arbitrary and unique ID of $O(\log n)$ bits, first only known to itself. The running time of a CONGEST algorithm is the number of rounds it uses. We will slightly abuse terminology and talk about running a CONGEST algorithm in digraph $D$; when we do so we mean that the algorithm runs in the (undirected) graph $G$ which is identical to $D$ but where we forget the directions of arcs. In this work, we will assume that if an arc $a$ has capacity $U_a$ then we allow nodes to send $O(U_a \cdot \log n)$ bits over the corresponding edge, though none of our applications rely on this assumption.

### 3 Length-Constrained Flows, Moving Cuts and Main Result

We proceed to more formally define a length-constrained flow, moving cuts and our main result which computes them. While we have defined length-constrained flows in Section 1 for unit capacities, it will be convenient for us to formally define length-constrained flows for general lengths and capacities in terms of a relevant linear program (LP). We do so now.

Suppose we are given a digraph $D = (V, A)$ with arc capacities $U$, lengths $\ell$ and specified source and sink vertices $S$ and $T$. A maximum $S$ to $T$ flow in $D$ in the classic sense can be defined as a collection of paths between $S$ and $T$ where each path receives some value and the total value incident to an edge does not exceed its capacity. This definition naturally extends to the length-constrained setting where we imagine we are given some length constraint $h \geq 1$ and define a length-constrained flow as a collection of $S$ to $T$ paths each of length at most $h$ where each such path receives some value $f_P$. Additionally, these values must respect the capacities of arcs. More precisely, we have the following LP with a variable $f_P$ for each path $P \in P_h(S, T)$.

$$\max \sum_{P \in P_h(S, T)} f_P \quad \text{s.t.} \quad (\text{Length-Constrained Flow LP})$$

$$\sum_{P \in P_h(S, T)} f_P \leq U_a \quad \forall a \in A$$

$$0 \leq f_P \quad \forall P \in P_h(S, T)$$

For a length-constrained flow $f$, we will use the shorthand $f(a) := \sum_{P \in P_h(S, T)} f_P$ and $\text{supp}(f) := \{P : f_P > 0\}$ to give the support of $f$. We will let $\text{val}(f) := \sum_{P \in P_h(S, T)} f_P$ give the value of $f$. An $h$-length flow, then, is simply a feasible solution to this LP.

**Definition 3.1 (h-Length Flow).** Given digraph $D = (V, A)$ with lengths $\ell$, capacities $U$ and vertices $S, T \subseteq V$, an $h$-length $S$-$T$ flow is any feasible solution to **Length-Constrained Flow LP**.

With the above definition of length-constrained flows we can now define moving cuts as the dual of length-constrained flows with the following moving cut LP with a variable $w_a$ for each $a \in A$.

$$\min \sum_{a \in A} U_a \cdot w_a \quad \text{s.t.} \quad (\text{Moving Cut LP})$$

$$\sum_{a \in P} w_a \geq 1 \quad \forall P \in P_h(S, T)$$

$$0 \leq w_a \quad \forall a \in A$$

An $h$-length moving cut is simply a feasible solution to this LP.

**Definition 3.2 (h-Length Moving Cut).** Given digraph $D = (V, A)$ with lengths $\ell$, capacities $U$ and vertices $S, T \subseteq V$, an $h$-length moving cut is any feasible solution to **Moving Cut LP**.

We will use $f$ and $w$ to stand for solutions to **Length-Constrained Flow LP** and **Moving Cut LP** respectively. We say that $(f, w)$ is a

---

3 We only make use of this assumption once and only make use of it in our deterministic algorithms. Furthermore, we do not require this assumption if the underlying digraph is a DAG.
feasible pair if both $f$ and $w$ are feasible for their respective LPs and that $(f, w)$ is $(1 + \varepsilon)$-approximate for $\varepsilon \geq 0$ if the moving cut certifies the value of the length-constrained flow up to a $(1 + \varepsilon)$; i.e., if $(1 + \varepsilon) \sum_u U_u \cdot w_u \leq \sum_P f_P$.

We clarify what it means to compute $(f, w)$ in CONGEST. When we are working in CONGEST we will say that $f$ is computed if each vertex $v$ stores the value $f_v(h) = \sum_{p \in \mathcal{P}_h(v, a, t)} f_p$ for every $a$ incident to $v$ and $h' \leq h$. Here, we let $\mathcal{P}_h(s, t)$ be all paths in $\mathcal{P}_h(S, T)$ of the form $P' = (a_1, a_2, \ldots, a_t)$ where the path $(a_1, b_1, b_2, \ldots)$ has length exactly $h'$ according to $t$. We say moving cut $w$ is computed if each vertex $v$ knows the value of $w_v$ for its incident arcs. Likewise, we imagine that each node initially knows the capacities and lengths of its incident arcs.

With the above notions, we can now state our main result. In the following we say $f$ is integral if $f_P$ is an integer for every path in $\mathcal{P}_h(S, T)$. The notable aspect of our results is the polynomial dependence on $h$ and $\frac{1}{\varepsilon}$; the polynomials could be optimized to be much smaller.

**Theorem 3.1.** Given a digraph $D = (V, A)$ with capacities $U$, lengths $\ell$, length constraint $h \geq 1$, $r = 0$ and source and sink vertices $S, T \subseteq V$, one can compute a feasible $h$-length flow, moving cut pair $(f, w)$ that is $(1 + \varepsilon)$-approximate in:

1. Deterministic parallel: $O\left(\frac{1}{\varepsilon^2} \cdot h^{17}\right)$ with $m$ processors;
2. Randomized CONGEST w.h.p.: $O\left(\frac{1}{\varepsilon^2} \cdot h^{17}\right)$;
3. Deterministic CONGEST: $O\left(\frac{1}{\varepsilon^2} \cdot h^{17} + \frac{1}{\varepsilon} \cdot h^{16} \cdot \rho_{CC}\right)^10$.

Also, $f = \eta \cdot \sum_{j=1}^k f_j$ where $\eta = \tilde{O}(\varepsilon^2)$, $k = \tilde{O}\left(\frac{h}{\varepsilon}\right)$ and each $f_j$ is an integral $h$-length $S$-$T$ flow.

All of our algorithms compute and separately store each $f_j$. The above result immediately gives the deterministic parallel and randomized CONGEST algorithms running in time $O\left(poly(h, \frac{1}{\varepsilon})\right)$ mentioned in Section 1. For our deterministic CONGEST algorithms, $\rho_{CC}$ in the above gives the quality of the optimal deterministic CONGEST cycle cover algorithm. We formally define this parameter in Section 5 but for now we simply note that $\rho_{CC} \leq 2^{O(\sqrt{\log n})}$ by known results [30, 38]. Applying this bound on $\rho_{CC}$ gives deterministic CONGEST algorithms running in time $O(poly(h, \frac{1}{\varepsilon}) \cdot 2^{O(\sqrt{\log n})})$. If $\rho_{CC}$ is shown to be poly($\log n$), we immediately would get an $O(poly(h, \frac{1}{\varepsilon}))$ time deterministic algorithm for solving $(1 + \varepsilon)$-approximate $h$-length flow in CONGEST. As mentioned in Section 1, $k$ in the above result is optimal up to $O(1)$ factors [8, 24].

## 4 TECHNICAL HIGHLIGHTS, INTUITION AND OVERVIEW OF APPROACH

Before moving on, we give an overview of our strategy for length-constrained flows. In the interest of highlighting what is new in this work we begin by summarizing three key technical contributions. To our knowledge these ideas are new in our work. We will then proceed to provide more detail on how these ideas fit together. For simplicity, we assume the capacity $U_u = 1$ for all $a$ in this section.

1. **Batched Multiplicative Weights:** First, the core idea of our algorithm is an "batched" version of the "multiplicative weights" framework. In particular, we will use what we call "near-lightest path blockers" to perform many independent multiplicative weight updates in parallel. Both this batched approach to multiplicative weights and our analysis showing that it converges to a near-optimal solution quickly are new to our work.

2. **Length-Weight Expanded DAG:** Second, we provide a new approximate representation of all near-lightest $h$-length paths by a "length-weight expanded DAG." This representation can be efficiently simulated in a distributed setting and serves as a provably good proxy for flows on all near-lightest $h$-length paths by a DAG. It is a priori not clear such a DAG exists since lightest $h$-length paths do not even induce a DAG. Even harder, this representation has to summarize three arc values at once: lengths, weights and capacities.

3. **Deterministic Integral Blocking Flows:** Third, we give the first efficient distributed deterministic algorithms for computing so-called integral blocking flows. In particular, we show how to use round flow rounding techniques to derandomize an approach of [35]: previous works noted that this approach seems inherently randomized [13]. Our flow rounding techniques are, in turn, built around a novel application of the recently introduced idea of "cycle covers." In particular, we will make use of a slight variant of cycle covers and show how to use them to efficiently round flows in a distributed setting.

### 4.1 Using Lightest Path Blockers for Multiplicative Weights

Computing a length-constrained flow, moving cut pair is naturally suggestive of the following multiplicative-weights-type approach. We initialize our moving cut value $w_a$ to some very small value for every $a$. Then, we find a lightest $h$-length path from $S$ to $T$ according to $w$, send some small $(\approx \varepsilon)$ amount of flow along this path and multiplicatively increase the value of $w$ on all arcs in this path by $(1 + \varepsilon)$. We repeat this until $S$ and $T$ are at least 1 apart according to $d_w^h$, where $d_w^h$ (where $d_w^h(u, v)$ gives the lightest according to $w$ path from $u$ to $v$ with length at most $h$). This general idea is an adaptation of ideas of [22].

The principle shortcoming of using such an algorithm for our setting is that it is easy to construct examples where there are polynomially-many arc-disjoint $h$-length paths between $S$ and $T$ and so we would clearly have to repeat the above process at least polynomially-many times until $S$ and $T$ are at least 1 apart according to $d_w^h$. This is not consistent with our goal of poly($h$) complexities since $h$ may be much smaller than $n$. To solve this issue, we use an algorithm similar to the above but instead of sending flow along one path, we send it along a large batch of arc-disjoint paths.

What can we hope to say about how long such an algorithm takes to make $S$ and $T$ at least 1 apart according to $d_w^h$? If it were the case that every lightest (according to $w$) $h$-length path from $S$ to $T$ shared an arc with some path in our batch of paths then after each batch we would know that we increased $d_w^h(S, T)$ by some small amount; in principle we might only increase $d_w^h(S, T)$ by some zero amount. However, there is no way to lower bound this amount; in principle we might only increase $d_w^h(S, T)$ by some small $\varepsilon > 0$. To solve this issue we find a batch of arc-disjoint paths which have weight essentially $d_w^h(S, T)$ but which share an arc
with every \( h \)-length path with weight at most \((1 + \epsilon) \cdot d^h_w(S, T)\). Thus, when we increment weights in a batch we know that all near-lightest \( h \)-length paths have their weights increased so we can lower bound the rate at which \( d^h_w(S, T) \) increases, meaning our algorithm completes quickly.

Thus, in summary we repeatedly find a batch of arc-disjoint \( h \)-length paths between \( S \) and \( T \) which have weight about \( d^h_w(S, T) \); these paths satisfy the property that every \( h \)-length path from \( S \) to \( T \) with weight at most \((1 + \epsilon) \cdot d^h_w(S, T)\) shares an edge with at least one of these paths; we call such a collection an \( h \)-length \((1 + \epsilon)\)-lightest path blocker. We then send a small amount of flow along these paths and multiplicatively increase the weight of all incident edges, appreciably increasing \( d^h_w(S, T) \). We repeat this until our weights form a feasible moving cut. See Figure 1.

![Figure 1: An illustration of the first two iterations of our multiplicative-weights-type algorithm where \( h = 5 \), \( S = \{s\} \) and \( T = \{t\} \) and capacities are all 1. Each arc is labelled with the value we multiply its initial weight by (initialized to \( w^0 = 1 + \epsilon \)) then length then flow. Our \( h \)-length shortest path blockers are in blue.](image1)

### 4.2 Length-Weight Expanded DAG to Approximate \( h \)-Length Lightest Paths

The above strategy relies on the computation of \( h \)-length lightest path blockers. Without the presence of a weight constraint computing such an object easily reduces to computing an integral blocking \( S-T \) flow on an \( h \)-layer \( S-T \) DAG. Specifically, consider the problem of computing a collection of paths from \( S \) to \( T \) so that every lightest \( S \) to \( T \) path shares an arc with one path in this collection. It is easy to see that all lightest paths between \( S \) and \( T \) induce an \( h \)-layer \( S-T \) DAG where \( h' \) is the minimum weight of a path between \( S \) and \( T \). One can then consider this DAG and compute an integral blocking \( S-T \) flow in it—i.e. a maximal arc-disjoint collection of \( h' \)-length \( S-T \) paths. By maximality of the flow, the paths corresponding to this flow will guarantee that every \( h' \)-length \( S \) to \( T \) path shares an arc with one path in this collection.

However, the presence of a length constraint and a weight constraint make such an object much trickier. Indeed, lightest paths subject to length constraints are known to be notoriously poorly behaved; not only do lightest paths subject to a length constraint not induce a metric but they are also arbitrarily far from any metric [5, 25]. As such, all \( S \) to \( T \) lightest paths subject to a length constraint do not induce a DAG, much less an \( h \)-layer \( S \) to \( T \) DAG; e.g. see Figure 2.

Our solution is to observe that, if we are allowed to duplicate vertices, then we can construct an \( S-T \) DAG with about \( h^2 \) layers that approximately captures the structure of all \( h \)-length \((1 + \epsilon)\)-lightest paths. Specifically, we discretize weights and then make a small number of copies of each vertex to compute a DAG \( D^{(h, \lambda)} \) which we call the length-weight expanded DAG. \( D^{(h, \lambda)} \) will satisfy the property that if we compute an integral blocking flow and then project this back into \( D \) as a set of paths \( \mathcal{P} \), then \( \mathcal{P} \) is almost a \((1 + \epsilon)\)-lightest path blocker. In particular, \( \mathcal{P} \) will guarantee that some arc of any \( h \)-length path with weight at most \((1 + \epsilon) \cdot d^h_w(S, T)\) is used by some path in \( \mathcal{P} \); however, the paths of \( \mathcal{P} \) may not be arc-disjoint as required of a lightest path blockers. Nonetheless, by carefully setting capacities in \( D^{(h, \lambda)} \), we will be able to argue that \( \mathcal{P} \) is nearly arc-disjoint and these violations of arc-disjointness can be repaired with small loss by a “decongesting” procedure. It remains to understand how to compute integral blocking flows in layered \( S-T \) DAGs.

### 4.3 Deterministic Integral Blocking Flows Paths via Flow Rounding

Lastly, we describe how we compute integral blocking flows in layered \( S-T \) DAGs.

A somewhat straightforward adaptation of a randomized algorithms of [35] solves this problem in \( O(\text{poly}(h)) \) time both in parallel and in CONGEST. This algorithm samples an integral \( S-T \) flow in \( D \) (i.e. a collection of arc-disjoint \( S \) to \( T \) paths) according to a carefully chosen distribution based on "path counts", deletes these paths and repeats. The returned solution is the flow induced by all paths that were ever deleted. Unfortunately [35]'s algorithm seems inherently randomized and our goal is to solve this problem deterministically.

We derandomize the algorithm of [35] in the following way. Rather than integrally sampling according to [35]'s distribution and then deleting arcs that appear in sampled paths, we instead calculate the probability that an arc is in a path in this distribution and then
“fractionally delete” it to this extent. We repeat this until every path between \( S \) and \( T \) has some arc which has been fully deleted. In other words, we run a smoothed version of [35] which behaves (deterministically) like the algorithm of [35] does in expectation. The fractional deletion values of arcs at the end of this process induce a blocking \( S-T \) flow but a blocking flow that may be fractional. We call this flow the “iterated path count flow.”

However, recall that our goal is to compute an integral blocking flow in an \( S-T \) DAG. Thus, we may naturally hope to round the iterated path count flow. Indeed, drawing on some flow rounding techniques of [17], doing so is not too difficult in parallel. Unfortunately, it is less clear how to do so in CONGEST. Indeed, [13] state:

...Cohen’s algorithm that rounds a fractional flow into an integral flow does not seem to have an efficient implementation in CONGEST...

Roughly, Cohen’s technique relies on partitioning edges in a graph into cycles and paths and then rounding each cycle and path independently. The reason this seems infeasible in CONGEST is that the cycles and paths that Cohen’s algorithm relies on can have unbounded diameter and so communicating within one of these cycles or paths is prohibitively slow. To get around this, we argue that, in fact, one may assume that these cycles and paths have low diameter if we allow ourselves to discard some small number of arcs. This, in turn allows us to orient these cycles and paths and use them in rounding flows. We formalize such a decomposition with the idea of a \((1 - \varepsilon)\)-near Eulerian partition. Arguing that discarding these arcs does bounded damage to our rounding then allows us to make use of Cohen-type rounding to deterministically round the path count flow, ultimately allowing us to compute \( h \)-length \((1 + \varepsilon)\)-lightest path blockers.

### 4.4 Summary of Our Algorithm

We now summarize the above discussion with a bottom-up sketch of our algorithm and highlight where each of these components appear in our paper.

The most basic primitive that we provide is an algorithm for efficiently computing blocking integral flows in \( h \)-layer \( S-T \) DAGs which, more or less, are maximal and integral \( S \) to \( T \) flows. To do so we make use of path count flows (formally defined in Section 6). In Section 7 we observe that, essentially by the ideas of [35], sampling paths proportional to the path count flows gives an efficient randomized algorithm for blocking integral flows in such DAGs. In Section 9 we give a deterministic algorithm for computing such flows. This algorithm relies on the idea of near Eulerian partitions (Section 8) which is an adaptation of recent ideas in cycles covers for our purposes. Our deterministic algorithm takes the expected result of [35] and deterministically rounds it by “turning” flow along the components of a near Eulerian partition and then repairs the resulting solution into a true flow by discarding flows not from \( S \) to \( T \). More generally, we show how to efficiently round any fractional flow on such a DAG with only a small loss in flow value.

Next, in Section 10 we use our algorithms for blocking integral flows in \( h \)-layer \( S-T \) DAGs to show how to compute \((1 + \varepsilon)\)-lightest path blockers which, informally, are a collection of paths that share an edge with every \( h \)-length near-lightest path. We do this by constructing the length-weight expanded DAG, a DAG that approximates the structure of \( h \)-length near-lightest paths. We then apply our blocking flow algorithms on this DAG, project the result back into our original graph and then “decongest” the result by finding an appropriate subflow that respects capacities.

Lastly, in Section 11 we plug our \((1 + \varepsilon)\)-lightest path blocker algorithm into a multiplicative-weights-type framework. In particular, we repeatedly compute a lightest path blocker, send some small amount of flow along the paths of this blocker and then update the weight of all edges that have flow sent along them by a multiplicative \((1 + \varepsilon)\).

Lastly, in Section 12 we give our new algorithms for bipartite \( b \)-matching based on our flow algorithms. Due to space constraints, we omit the majority of our proofs, further applications and extensions of our flow algorithms (as summarized in Section 1). See the full version of our work for these proofs, applications and extensions.

### 5 PRELIMINARY: CYCLE COVERS

Before moving on to our own technical content, we briefly review the idea of cycle covers some algorithmic tools.

Our flow rounding algorithm will make use of low diameter cycles. Thus, it will be useful for us to make use of some recent insights into distributively and deterministically decomposing graphs into low diameter cycles. We define the diameter of a cycle \( C \) as \(|C|\) and the diameter of a collection of cycles \( C \) as the maximum diameter of any cycle in it. Likewise the congestion of \( C \) is \( \max_{e \in E} |C : e \in C| \).

The idea of covering a graph with low congestion cycles is well-studied [15, 30, 38] and formalized by the idea of a cycle cover.

#### Definition 5.1 (Cycle Cover).

Given a simple graph \( G = (V, E) \) where \( E_0 \) is the set of all non-bridge edges\(^3\) of \( G \), a \((d, c)\) cycle cover is a collection of (simple) cycles \( C \) in \( G \) such that:

1. **Covering:** Every \( e \in E_0 \) is contained in some cycle of \( C \);
2. **Low Diameter:** \( \max_{C \in C} |C| \leq d \);
3. **Low Congestion:** \( \max_{e \in E} |C : e \in C| \leq c \).

We now formally define the parameter \( \rho_{CC} \); recall that this parameter appears in the running time of our deterministic CONGEST algorithm in our main theorem (Theorem 3.1).

#### Definition 5.2 (\( \rho_{CC} \)).

Given a deterministic CONGEST algorithm that constructs a \((d, c)\) cycle cover in worst-case time \( T \) in graphs of diameter \( D \), we say that the quality of this algorithm is \( \max\left(\frac{d}{2}, c, \frac{T}{2}\right) \).

We let \( \rho_{CC} \) be the smallest quality of any deterministic CONGEST algorithm for constructing cycle covers.

The following summarizes the current state-of-the-art in deterministic cycle cover computation in CONGEST.

#### Theorem 5.3 ([30, 38]).

There is a deterministic CONGEST algorithm that given a graph \( G \) with diameter \( D \) computes a \((d, c)\) cycle cover with quality \( D \left(\frac{2}{3}, 1, \frac{4}{3}\right) \).

---

\(^{3}\)Somewhat similar to our approach, [15] showed how to partition all but \( O(n \log n) \) edges of a graph into short disjoint cycles. However, these guarantees are unsuitable for our uses on e.g. graphs with \( O(n) \) edges since we may only discard a small fraction of all edges.

\(^{3}\)Recall that a bridge edge of a graph is one whose removal increases the number of connected components in the graph.
cycle cover with \( d = 2^O(\sqrt{\log n}) \cdot D \) and \( c = 2^O(\sqrt{\log n}) \) in time \( 2^O(\sqrt{\log n}) \cdot D \). In other words, \( \rho_{\text{CC}} \leq 2^O(\sqrt{\log n}) \).

6 PATH COUNTS FOR h-LAYER S-T DAGS

We begin by recounting the notion of path counts which we will use for our randomized algorithm to sample flows and for our deterministic algorithms to compute the iterated path count flow. This idea has been used in several prior works [13, 17, 35]. Suppose we are given an \( h \)-layer \( S-T \) DAG \( D \) with capacities \( U \). We define these path counts as follows. We define the capacity of a path as the product of its edge capacities, namely given a path \( P \) we let \( U(P) := \prod_{a \in P} U_a \). Recall that we use \( \mathcal{P}(S, T) \) to stand for all paths between \( S \) and \( T \). We will slightly abuse notation and let \( \mathcal{P}(u, T) = \mathcal{P}((u), T) \) and \( \mathcal{P}(S, u) = \mathcal{P}(S, \{u\}) \). For vertex \( v \) we let \( n^v_0 \) be the number of paths from \( v \) to \( T \), weighted by \( U \), namely \( n^v_0 := \sum_{P \in \mathcal{P}(S, v) \setminus U(P)} \). Symmetrically, we let \( n^u_a := \sum_{P \in \mathcal{P}(S, u) \setminus U(P)} \). For any arc \( a = (u, v) \), we define \( n_a \) as

\[
    n_a := n^u_a \cdot U_a \cdot n^v_a.
\]

Equivalently, we have that \( n_a \) is the number of paths in \( \mathcal{P}(S, T) \) that use \( a \) weighted by capacities:

\[
    n_a = \sum_{P \in \mathcal{P}(S, T), a \in P} U(P).
\]

It may be useful to notice that if we replace each arc \( a \) with \( U_a \)-many parallel arcs then \( n_a \) exactly counts the number of unique paths from \( S \) to \( T \) that use \( a \) in the resulting (multi) digraph. A simple dynamic-programming type algorithms that does a “sweep” from \( T \) to \( S \) and \( S \) to \( T \) shows how to efficiently compute path counts.

**Lemma 6.1.** Let \( D \) be a capacitated \( h \)-layer \( S-T \) DAG. Then one can compute \( n^v_0 \) and \( n^u_a \) for every vertex \( v \) and \( n_a \) for every arc \( a \) in:

1. **Parallel time** \( O(h) \) with \( m \) processors;
2. **CONGEST time** \( O(h^2) \).

7 RANDOMIZED BLOCKING INTEGRAL FLOWS IN h-LAYER DAGS

We now describe how to compute blocking integral flows in \( h \)-layer DAGs with high probability by using the path counts of the previous section. This is the general capacities version of the problem described in Section 4.3. More or less, the algorithm we use is one of [13] adapted to the general capacities case; the algorithm of [13] is itself an adaptation of an algorithm of [35]. We mostly include these results for the sake of completeness.

Our randomized algorithm will repeatedly sample an integral flow proportional to the path counts of Section 6, add this to our existing flow, reduce capacities and then repeat. We will argue that we need only iterate this process a small number of times until we get a blocking integral flow by appealing to the fact that “high degree” paths have their capacities reduced with decent probability.

One can see this as essentially running the randomized MIS algorithm of [36] but with two caveats: (1) the underlying graph in which we compute an MIS has a node for every path between \( S \) and \( T \) and so has up to \( O(n^h) \)-many nodes; as such we cannot explicitly construct this graph but rather we implicitly run Luby’s algorithm on it; (2) Luby’s analysis assumes nodes attempt to enter the MIS independently but our sampling will have some dependencies between nodes (i.e. paths) entering the MIS which must be addressed in our analysis.

More formally, suppose we are given a capacitated \( S-T \) DAG \( D \). For a given path \( P \in \mathcal{P}(S, T) \) we let \( \Delta_P \) be \( \sum_{a \in P} U_a \) be the “degree” of path \( P \) where the sum over \( P' \) ranges over all \( P' \) that share at least one arc with \( P \) and are in \( \mathcal{P}(S, T) \). We let \( \Delta = \max_{P \in \mathcal{P}(S, T)} \Delta_P \) be the maximum degree. Similarly, we let \( \mathcal{P}_{\text{max}} := \{ P : \Delta_P \geq \frac{\Delta}{2} \} \) be all paths with near-maximum degree.

The following summarizes the flow we repeatedly compute.

**Lemma 7.1.** Given a \( h \)-layer \( S-T \) DAG \( D \) with capacities \( U \) and \( \Delta \) satisfying \( \frac{\Delta}{2} \leq \Delta \leq \Lambda \), one can sample an integral \( S-T \) flow \( f \) where for each \( P \in \mathcal{P}_{\text{max}} \) we have \( \sum_{a \in P} U_a - f_a \leq \frac{2m^2}{n^2} \cdot U(P) \) with probability at least \( O(1) \). This can be done in:

1. **Parallel time** \( O(h) \) with \( m \) processors;
2. **CONGEST time** \( O(h^2) \) with high probability.

Repeatedly applying the above lemma gives our randomized algorithm for blocking integral \( S-T \) flows.

**Lemma 7.2.** There is an algorithm which, given an \( h \)-layer \( S-T \) DAG \( D \) with capacities \( U \), computes an integral \( S-T \) flow that is blocking in:

1. **Parallel time** \( O(h^3) \) with \( m \) processors with high probability;
2. **CONGEST time** \( O(h^4) \) with high probability.

8 DETERMINISTIC AND DISTRIBUTED NEAR EULERIAN PARTITIONS

In the previous section we described how to efficiently compute blocking integral flows in \( h \)-layer DAGs with high probability. In this section, we introduce the key idea we make use of in doing so deterministically, a near Eulerian partition.

Informally, a near Eulerian partition will discard a small number of edges and then partition the remaining edges into cycles and paths. Because these cycles and paths will have small diameter in our construction, we will be able to efficiently orient them in CONGEST. In Section 9 we will see how to use these oriented cycles and paths to efficiently round flows in a distributed fashion in order to compute a blocking integral flow in \( h \)-layer DAGs.

We now formalize the idea of a \((1 - \epsilon)\)-near Eulerian partition.

**Definition 8.1** \(((1 - \epsilon)\text{-Near Eulerian Partition})\). Let \( G = (V, E) \) be an undirected graph and \( \epsilon \geq 0 \). A \((1 - \epsilon)\text{-near Eulerian partition} \( \mathcal{H} \) is a collection of edge-disjoint cycles and paths in \( G \), where

1. \((1 - \epsilon)\text{-Near Covering: The number of edges in } E[\mathcal{H}] \text{ is at least } (1 - \epsilon) \cdot |E|; \)
2. Eulerian Partition: Each vertex is the endpoint of at most one path in \( \mathcal{H} \).

The following is the main result of this section and summarizes our algorithms for construction \((1 - \epsilon)\text{-near Eulerian partitions}. In what follows we say that a cycle is oriented if every edge is directed so that every vertex in the cycle has in and out degree \( 1 \); a path \( P \) is oriented if it has some designated source and sink \( s_p \) and \( t_p \). We say that a collection of paths and cycles \( \mathcal{H} \) is oriented if each element of \( \mathcal{H} \) is oriented. In CONGEST we will imagine that a cycle is oriented if each vertex knows the orientation of its incident arcs and a path is oriented if every vertex knows which of its neighbors are closer to \( s_p \).
We argue that any such flow is also integrally flow while preserving the value of the flow up to a small deficit. This deficit is small and so after deleting all flow that originates in the above way can increase the deficit of our flow. However, by Lemma 7.1, as the flow we compute is the expected flow of the aforementioned sampling, this process is deterministic. The result of this is a \( \Omega(\frac{1}{\epsilon}) \)-approximate flow, which can only repeat this about \( h^2 \) time (otherwise we would end up with a flow of value greater than that of the optimal flow).

9 DETERMINISTIC BLOCKING INTEGRAL FLOWS IN \( h \)-LAYER DAGS

In Section 7 we discussed how to efficiently compute blocking integral flows in \( h \)-layer DAGs with high probability. In this section, we discuss how to do so deterministically by making use of the near Eulerian partitions of Section 8. Specifically, the main result of this section is the following.

Lemma 9.1. There is a deterministic algorithm which, given a capacitated \( h \)-layer \( S,T \) DAG \( D \), computes an integral \( S,T \) flow that is blocking in:

1. Parallel time \( \tilde{O}(h) \) with \( m \) processors;
2. CONGEST time \( \tilde{O}(h \cdot (\rho_{CC})^{10}) \).

The above parallel algorithm is more or less implied by the work of [17]. However, the key technical challenge we solve in this section is a distributed implementation of the above.

Our strategy for showing the above lemma has two ingredients.

Iterated Path Count Flow. First, we construct the iterated path count flow. This corresponds to repeatedly taking the expected flow induced by the sampling of our randomized algorithm (as given by Lemma 7.1). As the flow we compute is the expected flow of a path count flow, this process is deterministic.

Flow Rounding. Next, we provide a generic way of rounding a fractional flow to be in integral in an \( h \)-layer DAG while approximately preserving its value. Here, the main challenge is implementing such a rounding in CONGEST; the key idea we use is that of a \( (1 - \epsilon) \)-near Eulerian partition from Section 8 which discards a small number of edges and then partitions the remaining graph into cycles and paths.

These partitions enables us to implement a rounding in the style of [17]. In particular, we start with the least significant bit of our flow, compute a \( (1 - \epsilon) \)-near Eulerian partition of the graph induced by all arcs which set this bit to 1 and then use this partition to round all these bits to 0. Working our way from least to most significant bit results in an integral flow. The last major hurdle to this strategy is showing that discarding a small number of edges does not damage our resulting integral flow too much; in particular discarding edges in the above way can increase the deficit of our flow. However, by always discarding an appropriately small number of edges we show that this deficit is small and so after deleting all flow that originates or ends at vertices not in \( S \) or \( T \), we are left with a flow of essentially the same value of the input fraction flow. The end result of this is a rounding procedure which rounds the input fractional flow to an integral flow while preserving the value of the flow up to a constant. The following summarizes our rounding theorem.

Lemma 9.2. There is a deterministic algorithm which, given a capacitated \( h \)-layer \( S,T \) DAG \( D \), \( \epsilon = \Omega(\frac{1}{poly(n)}) \) and (possibly fractional) flow \( f \), computes an integral \( S,T \) flow \( \hat{f} \) in:

1. Parallel time \( \tilde{O}(h) \) with \( m \) processors;
2. CONGEST time \( \tilde{O}(h^2 \cdot (\rho_{CC})^{10}) \).

Furthermore, \( val(\hat{f}) \geq (1 - \epsilon) \cdot val(f) \).

Parts of the above parallel result are implied by the work of [17] while the CONGEST result is entirely new.

Our algorithm to compute blocking integral flows in \( h \)-layer DAGs deterministically combines the above two tools. Specifically, we repeatedly compute the iterated path count flow, round it to be integral and add the resulting flow to our output. As the iterated path count flow is \( \Omega(\frac{1}{\epsilon}) \)-approximate, we can only repeat this about \( h^2 \) times (otherwise we would end up with a flow of value greater than that of the optimal flow).

10 \( h \)-LENGTH \((1 + \epsilon)\)-LIGHTEST PATH BLOCKERS

In this section we discuss how to efficiently compute the main subroutine for our multiplicative-weights-type algorithm: what we call \( h \)-length \((1 + \epsilon)\)-lightest path blockers. We will use the blocking integral flow primitives of Section 7 for our randomized algorithm and that of Section 9 for our deterministic algorithm.

Our \((1 + \epsilon)\)-lightest path blockers are defined below. In what follows, \( \lambda \) is intuitively a guess of a \( d_w^{(h)}(S,T) \) path then assigns flow values to entire paths (rather than just arcs as a non-length-constrained flow does). As such the support of \( f \), supp\((f)\), is a collection of paths. However, as mentioned earlier, for an \( h \)-length flow \( f \), we will use \( f(a) \) as shorthand for \( \sum_{\rho \ni a} fp \).

Definition 10.1 \((h \text{-length } (1 + \epsilon) \text{-Lightest Path Blockers})\). Let \( G = (V,E) \) be a graph with lengths \( t \), weights \( w \) and capacities \( U \). Fix \( \epsilon > 0 \), \( h \geq 1 \), \( \lambda \leq d_w^{(h)}(S,T) \) and \( S,T \subseteq V \). Let \( f \) be an \( h \)-length integral \( S,T \)-flow. \( f \) is an \( h \)-length \((1 + \epsilon)\)-lightest path blocker if:

1. \( \text{Near-Lightest: } P \in \text{supp}(f) \) has weight at most \((1 + 2\epsilon) \cdot \lambda \);
2. \( \text{Near-Lightest Path Blocking: If } P' \in \text{ supp}(S,T) \) has weight at most \((1 + \epsilon) \cdot \lambda \) then there is some \( a \in P' \) where \( f(a) = U_a \).

Our main theorem in of this section shows how to compute \((1 + \epsilon)\)-lightest path blockers efficiently.

Theorem 10.1. Given digraph \( D = (V,A) \) with lengths \( t \), weights \( w \), capacities \( U \), length constraint \( h \geq 1 \), \( \epsilon > 0 \), \( S,T \subseteq V \) and \( \lambda \leq d_w^{(h)}(S,T) \), one can compute an \( h \)-length \((1 + \epsilon)\)-lightest path blocker in:

1. Deterministic parallel: \( \tilde{O}(\frac{1}{\epsilon} \cdot h^{16}) \) with \( m \) processors;
2. Randomized CONGEST w.h.p.: \( \tilde{O}(\frac{1}{\epsilon} \cdot h^{16}) \);.
3. Deterministic CONGEST: \( \tilde{O}(\frac{1}{\epsilon} \cdot h^{16} + \frac{1}{\epsilon^2} \cdot h^{15} \cdot (\rho_{CC})^{10}) \).

The main idea for computing these objects is to reduce finding them to computing a series of blocking flows in a carefully constructed "length-weight expanded DAG." In particular, by rounding arc weights up to multiples of \( \frac{1}{h} \lambda \) we can essentially discretize the space of weights. Since each path has at most \( h \) arcs, it follows...
that this increases the weight of a path by at most only $\lambda \epsilon$. This discretization allows us to construct DAGs from which we may extract blocking flows which we then project back into $D$ and then “decongest” so as to ensure they are feasible flows.

## 11 Computing Length-Constrained Flows and Moving Cuts

Having discussed how to compute an $h$-length $(1 + \epsilon)$-lightest path blocker, we now use a series of these as batches to which we apply multiplicative-weights-type updates. The result is our algorithm which returns both a length-constrained flow and a (nearly) certifying moving cut.

### Algorithm 1 Length-Constrained Flows and Moving Cuts

**Input:** digraph $D = (V, A)$ with lengths $\ell$, capacities $U$, $h \geq 1$, $S, T \subseteq V$ and $\epsilon \in (0, 1)$.

**Output:** $(1 + \epsilon)$-approximate $h$-length flow $f$ and moving cut $w$.

Let $\epsilon_0 = \frac{\epsilon}{2} + 1$ and let $\eta = \epsilon_0 \frac{1}{(1 + \epsilon_0) \ell} \cdot \frac{1}{\log m}$.

Initialize $w_a \leftarrow \frac{\lambda}{m}$ for all $a \in A$.

Initialize $\lambda \leftarrow \left( \frac{1}{\epsilon} \right)$.

Initialize $f_a \leftarrow 0$ for all $P \in \mathcal{P}_h(S, T)$.\[\text{while } \lambda < 1 \text{ do:}\]

  \[\text{for } \Theta \left( \frac{h \log_{1+\epsilon_0} n}{\epsilon_0} \right) \text{ iterations: do}\]

  Compute $h$-length $(1 + \epsilon_0)$-lightest path blocker $\hat{f}$.

  $h$-Length Flow Update: $f \leftarrow f + \eta \cdot \hat{f}$.

  Moving Cut Update: $w_a \leftarrow (1 + \epsilon_0) f(a)/U_a \cdot w_a \forall a \in A$.

  $\lambda \leftarrow (1 + \epsilon_0) \cdot \lambda$.

**return** $(f, w)$.

As a reminder for an $h$-length flow $f$, we let $f(a) := \sum_{P \ni a} f_P$. Throughout our analysis we will refer to the innermost loop of Algorithm 1 as one "iteration." We begin by observing that $\lambda$ always lower bounds $d_w^{(h)}(S, T)$ in our algorithm.

### Lemma 11.1.

At the beginning of each iteration of Algorithm 1 we have $\lambda \leq d_{w}^{(h)}(S, T)$.

**Proof.** Our proof is by induction. The statement trivially holds at the beginning of our algorithm.

Let $\lambda_i$ be the value of $\lambda$ at the beginning of the $i$th iteration. We argue that if $d_w^{(h)}(S, T) = \lambda_i$ then after $\Theta \left( \frac{h \log_{1+\epsilon_0} n}{\epsilon_0} \right)$ additional iterations we must have $d_w^{(h)}(S, T) \geq (1 + \epsilon_0) \cdot \lambda_i$. Let $\lambda'_i = (1 + \epsilon_0) \cdot \lambda_i$ be $\lambda$ after these iterations. Let $\hat{f}_i$ be our lightest path blocker in the $i$th iteration.

Assume for the sake of contradiction that $d_w^{(h)}(S, T) < \lambda'_i$ after $i + \Theta \left( \frac{h \log_{1+\epsilon_0} n}{\epsilon_0} \right)$ iterations. It follows that there is some path $P \in \mathcal{P}_h(S, T)$ with weight at most $\lambda'_i$ after $i + \Theta \left( \frac{h \log_{1+\epsilon_0} n}{\epsilon_0} \right)$ many iterations. However, notice that by definition of an $h$-length $(1 + \epsilon_0)$-lightest path blocker (Definition 10.1), we know that for every $j \in [i, i + \Theta \left( \frac{h \log_{1+\epsilon_0} n}{\epsilon_0} \right)]$ there is some $a \in P$ for which $\hat{f}_j(a) = U_a$. By averaging, it follows that there is some single arc $a \in P$ for which $\hat{f}_j(a) = U_a$ for at least $\Theta \left( \frac{\log_{1+\epsilon_0} n}{\epsilon_0} \right)$ of these $j \in [i, i + \Theta \left( \frac{h \log_{1+\epsilon_0} n}{\epsilon_0} \right)]$. Since every such arc starts with dual value $(\frac{1}{m}) \epsilon$ and multiplicatively increases by a $(1 + \epsilon_0)$ factor in each of these updates, such an arc after $i + \Theta \left( \frac{h \log_{1+\epsilon_0} n}{\epsilon_0} \right)$ iterations must have $w_a$ value at least $(\frac{1}{m}) \epsilon \cdot (1 + \epsilon_0) \Theta \left( \frac{\log_{1+\epsilon_0} n}{\epsilon_0} \right) \geq n^2$ for an appropriately large hidden constant in our $\Theta$. However, by assumption, the weight of $P$ is at most $\lambda'_i$ after $i + \Theta \left( \frac{h \log_{1+\epsilon_0} n}{\epsilon_0} \right)$ iterations and this is at most 2 since $\lambda_1 < 1$ since otherwise our algorithm would have halted. But $2 < n^2$ and so we have arrived at a contradiction.

Repeatedly applying the fact that $\lambda'_i = (1 + \epsilon_0) \lambda_i$ gives that $\lambda$ is always a lower bound on $d_w^{(h)}(S, T)$.

We next prove the feasibility of our solution.

### Lemma 11.2.

The pair $(f, w)$ returned by Algorithm 1 is feasible for Length-Constrained Flow LP and Moving Cut LP respectively.

**Proof.** First, observe that by Lemma 11.1 we know that $\lambda$ is always a lower bound on $d_w^{(h)}(S, T)$ and so since we only return once $\lambda > 1$, the $w$ we return is always feasible.

To see that $f$ is feasible it suffices to argue that for each arc $a$, the number of times a path containing $a$ has its primal value increased is at most $\frac{U_a}{\eta}$. Notice that each time we increase the primal value on a path containing arc $a$ by $\eta$ we increase the dual value of this edge by a multiplicative $(1 + \epsilon_0)^1/U_a$. Since the weight of our arcs according to $w$ start at $(\frac{1}{m}) \epsilon$, it follows that if we increase the primal value of $k$ paths incident to arc $a$ then $w_a = (1 + \epsilon_0)^k U_a \cdot (\frac{1}{m}) \epsilon$. On the other hand, by assumption when we increase the dual value of an arc $a$ it must be the case that $w_a < 1$ since otherwise $d_w^{(h)}(S, T) \geq 1$, contradicting the fact that $\lambda$ always lower bounds $d_w^{(h)}(S, T)$. It follows that $(1 + \epsilon_0)^k U_a \cdot (\frac{1}{m}) \epsilon \leq 1$ and so applying the fact that $\ln(1 + \epsilon_0) \geq \frac{\epsilon_0}{1 + \epsilon_0} > -1$ and our definition of $\epsilon$ and $\eta$ we get

$$k \leq \frac{\epsilon}{(1 + \epsilon_0)} \cdot \frac{U_a}{\eta} \leq \frac{U_a \log m}{\eta}$$

as desired.

We next prove the near-optimality of our solution.

### Lemma 11.3.

The pair $(f, w)$ returned by Algorithm 1 satisfies $(1 - \epsilon) \sum_a w_a \leq \sum_p f_P$.

**Proof.** Fix an iteration $i$ of the above while loop and let $\hat{f}$ be our lightest path blocker in this iteration. Let $k_i$ be val($\hat{f}$), let $\lambda_i$ be $\lambda$ at the start of this iteration and let $D_i := \sum_a w_a$ be our total dual value at the start of this iteration. Notice that $\frac{1}{m} \cdot w$ is dual feasible and has cost $\frac{D_i}{\lambda_i}$ by Lemma 11.1. If $\beta$ is the optimal dual value then by optimality it follows that $\beta \leq \frac{D_i}{\lambda_i}$ giving us the upper
bound on \( \lambda_i \) of \( \frac{D_i}{\beta} \). By how we update our dual, our bound on \( \lambda_i \) and \((1 + x)^r \leq 1 + xr \) for any \( x \geq 0 \) and \( r \in (0, 1) \) we have that

\[
D_{i+1} = \sum_a \left( 1 + e_0 \right) \hat{f}(a) / U_a \cdot w_a \cdot U_a \leq \sum_a \left( 1 + e_0 \right) \hat{f}(a) / U_a \cdot w_a \cdot U_a = D_i + e_0 \sum_a \hat{f}(a) w_a \leq D_i + e_0 (1 + 2r_0) k_i \lambda_i \leq D_i + \left( 1 + \frac{2e_0 r_0}{\beta} k_i \right) \leq D_i \cdot \exp \left( \frac{1 + 2e_0 r_0}{\beta} k_i \right) .
\]

Let \( T = 1 \) be the index of the last iteration of our algorithm; notice that \( D_T \) is the value of \( w \) in our returned solution. Let \( K := \sum_i k_i \). Then, repeatedly applying this recurrence gives us

\[
D_T \leq D_0 \cdot \exp \left( \frac{1 + 2e_0 r_0}{\beta} K \right) = \left( \frac{1}{m} \right)^{1 - \frac{1}{\log m}} \exp \left( \frac{1 + 2e_0 r_0}{\beta} K \right) .
\]

On the other hand, we know that \( w \) is dual feasible when we return it, so it must be the case that \( D_T \geq 1 \). Combining this with the above upper bound on \( D_T \) gives us

\[
1 \leq \left( \frac{1}{m} \right)^{1 - \frac{1}{\log m}} \exp \left( \frac{1 + 2e_0 r_0}{\beta} K \right) .
\]

Solving for \( K \) and using our definition of \( \xi \) gives us

\[
\beta \log m \cdot \frac{\zeta - 1}{1 + 2e_0 r_0} \leq K .
\]

However, notice that \( K \eta \) is the primal value of our solution so using our choice of \( \eta \) and rewriting this inequality in terms of \( K \eta \) by multiplying by \( \eta = \frac{1}{m} \frac{1}{(1 + e_0) \xi} \) and applying our definition of \( \xi = \frac{1 + 2e_0}{e_0} + 1 \) gives us

\[
\beta \log m \cdot \frac{\zeta - 1}{(1 + 2e_0) e_0} \leq K \eta \leq \beta \log m \cdot \frac{1}{e_0} \leq K \eta .
\]

Moreover, by our choice of \( e_0 = \frac{\epsilon}{6} \) and the fact that \( \frac{1}{1 + x \times x^2} \geq 1 - x \) for \( x \in (0, 1) \) we get

\[
1 - \epsilon \leq \frac{1}{1 + \epsilon + \epsilon^2} \leq \frac{1}{(1 + \frac{1}{2} \epsilon)^2} \leq \frac{1}{(1 + 3\epsilon^2)^2} \leq \frac{1}{(1 + e_0)(1 + 3e_0)} .
\]

Combining Equation (1) and Equation (2) we conclude that

\[
(1 - \epsilon) \cdot \beta \leq K \eta .
\]

We conclude with our main theorem by proving that we need only iterate our algorithm \( \tilde{O} \left( \frac{n^{1+\epsilon}}{\epsilon^2} \right) \) times.

**Theorem 3.1.** Given a digraph \( D = (V, A) \) with capacities \( U \), lengths \( t \), length constraint \( h \geq 1 \), \( \epsilon > 0 \) and source and sink vertices \( S, T \subseteq V \), one can compute a feasible h-length flow, moving cut pair \( (f, w) \) that is \((1 + \epsilon)\)-approximate in:

1. Deterministic parallel: \( \tilde{O}(\frac{\log n}{\epsilon^2} \cdot h^{17}) \) with \( m \) processors;
2. Randomized CONGEST w.h.p.: \( \tilde{O}(\frac{\log n}{\epsilon^2} \cdot h^{17}) \);
3. Deterministic CONGEST: \( \tilde{O}(\frac{\log n}{\epsilon^2} \cdot h^{17} + \frac{\log n}{\epsilon^2} \cdot h^{16} \cdot (\rho_{CC})^{10}) \).

Also, \( f = \eta \cdot \sum_{j=1}^{k} f_j \) where \( \eta = \tilde{O}(\epsilon^2) \), \( k = \tilde{O}(\frac{h}{\epsilon^4}) \) and each \( f_j \) is an integer h-length \( S-T \) flow.

**Proof.** We use Algorithm 1. By Lemma 11.2 and Lemma 11.3 we know that our solution is feasible and \((1 + \epsilon)\)-optimal so it only remains to argue the runtime of our algorithm and that the returned flow decomposes in the stated way.

We argue that we must only run for \( O \left( \frac{h \log^2 n}{\epsilon^2} \right) \) total iterations. Since \( \lambda \) increases by a multiplicative \((1 + e_0)\) after every \( \Theta \left( \frac{h \log n}{e_0} \right) \) iterations and starts at at least \( \left( \frac{m}{e_0} \right)^{\Theta(1/n)} \), it follows by Lemma 11.1 that after \( y \cdot \Theta \left( \frac{h \log n}{e_0} \right) \) total iterations the h-length distance between \( S \) and \( T \) is at least \( (1 + e_0)^y \cdot \left( \frac{m}{e_0} \right)^{\Theta(1/n)} \). Thus, for \( y \geq \Omega \left( \frac{\log m \cdot m}{e_0} \right) = \Omega \left( \frac{\log n}{e_0} \right) \) we have that \( S \) and \( T \) are at least 1 apart in h-length distance. Consequently, our algorithm must run for at most \( O \left( \frac{h \log^2 n}{e_0} \right) = O \left( \frac{h \log^2 n}{\epsilon^2} \right) \) many iterations.

Our running time is immediate from the the bound of \( O \left( \frac{h \log^2 n}{\epsilon^2} \right) \) on the number of iterations of the while loop and the running times given in Theorem 10.1 for computing our h-length \((1 + e_0)\)-lightest path blocker.

Lastly, the flow decomposes in the stated way because we have at most \( O \left( \frac{h \log^2 n}{\epsilon^2} \right) \) iterations and each \( f_j \) is an integer \( S-T \) flow by Theorem 10.1. Thus, our solution is \( \eta \cdot \sum_{j=1}^{k} f_j \) and \( k = \tilde{O}(\frac{n^{1+\epsilon}}{\epsilon^2}) \). \( \square \)

**12 APPLICATION:** \((1 - \epsilon)\)-APPROXIMATE DISTRIBUTED BIPARTITE b-MATCHING

In this section we give the first efficient \((1 - \epsilon)\)-approximate CONGEST algorithms for maximum cardinality bipartite b-matching. In fact, our results are for the slightly more general edge-capacitated maximum bipartite b-matching problem, defined as follow.

**Edge-Capacitated Max. Bipartite b-Matching:**
Given bipartite graph \( G = (V, E) \), edge capacities \( U \) and function \( b : V \rightarrow \Sigma_{e \in E} \) compute an integer \( x_e \in [0, U_e] \) for each \( e \in E \) maximizing \( \sum_{e} x_e \) so that for each \( v \in V \) we have \( \sum_{e \in \partial(v)} x_e \leq b(v) \). Notice that the case where \( b(v) = 1 \) for every \( v \) is just the classical maximum cardinality matching problem. "b-matching" seems
to refer to two different problems in the literature depending on whether edges can be chosen with multiplicity: either it is the above problem where \( U_e = 1 \) for every \( e \in E \) or it is the above problem where \( U_e = \max b_e b_f \) for each \( e \in E \). Our algorithms will work for both of these variants since they solve the above problem which generalizes both of these problems.

The following theorem summarizes our main result for bipartite \( b \)-matching in CONGEST. Again, recall that \( p_{CC} \) is defined in Definition 5.2 and is known to be at most \( 2^\Omega(\sqrt{\log n}) \).

**Theorem 12.1.** There is a deterministic \((1 - \varepsilon)\)-approximation for edge-capacitated maximum bipartite \( b \)-matching running in CONGEST time \( \tilde{O} \left( \frac{1}{\varepsilon} + \frac{1}{\varepsilon^2} \cdot (p_{CC})^{10} \right) \).

**Proof.** Our algorithm works in two steps. First, we reduce edge-capacitated \( b \)-matching to length-constrained flow and use our length constrained flow algorithm to efficiently compute a fractional flow. Then, we apply the flow rounding technology we developed (Lemma 9.2) to round this flow to an integral flow which, in turn, corresponds to an integral \( b \)-matching.

More formally our algorithm is as follows. Suppose we are given an instance of edge-capacitated \( b \)-matching on bipartite graph \( G = (V, E) \). Let \( L \) and \( R \) be the corresponding bipartition of vertices of \( G \). We construct the following instance of length-constrained flow on digraph \( D = (V', A) \) with \( h = 3 \) as follows. Each \( v \in V \) has two copies \( v^{(l)} \) and \( v^{(o)} \) in \( V' \). We add arc \((v^{(l)}, v^{(o)})\) to \( A \) with capacity \( b(v) \). If \( \{u, v\} \in E \) when \( u \in L \) and \( v \in R \) then we add arc \((u^{(o)}, v^{(l)})\) with capacity \( U_e \) to \( A \). Lastly, we let \( S = \{u^{(l)} : u \in L\} \), \( T = \{v^{(o)} : v \in R\} \) and the length of each arc in \( D \) be 1. Next, we apply Theorem 3.1 to compute a \((1 - \varepsilon_1)\)-approximate maximum \( 3 \)-length \( S \)-\( T \) flow \( f \) on \( D \) for some small \( \varepsilon_1 \) to be chosen later. Since \( D \) is a \( 3 \)-layer \( S \)-\( T \) \( \Delta \)-DAG we may interpret this as a (non-length-constrained) flow where the flow value on arc \( a \) is \( f(a) \).

We then apply Lemma 9.2 to this non-length-constrained flow to get integral \( S \)-\( T \) flow \( f' \) satisfying \( \text{val}(f') \geq (1 - \varepsilon_2) \cdot \text{val}(f) \) for some small \( \varepsilon_2 \) to be chosen later. We return as our solution the \( b \)-matching which naturally corresponds to \( f' \). Namely, if \( e = \{u, v\} \) then \( f' \) is integral it assigns arc \((u^{(o)}, v^{(l)})\) a value in \( \{0, 1, \ldots, U_e\} \). We let \( x_e \) be this value for \( e = \{u, v\} \) and we return as our \( b \)-matching solution \( (x_e) \).

\( f' \) is a \((1 - \varepsilon_1)(1 - \varepsilon_2)\)-approximate maximum \( S \)-\( T \) flow. Letting \( \text{OPT} \) be the value of the optimal \( b \)-matching solution, it is easy to see that the maximum \( S \)-\( T \) flow has value \( \text{OPT} \) and so the solution we return has value at least \((1 - \varepsilon_1)(1 - \varepsilon_2) \cdot \text{OPT} \). Letting \( \varepsilon_1 \) and \( \varepsilon_2 \) be \( \Theta(\varepsilon) \) for an appropriately small hidden constant we get that \((1 - \varepsilon_1)(1 - \varepsilon_2) \cdot \text{OPT} \geq (1 - \varepsilon) \cdot \text{OPT} \).

Lastly, we argue our running time. Our running time is dominated by one call to Theorem 3.1 with \( \varepsilon_1 = \Theta(\varepsilon) \) which takes \( \tilde{O} \left( \frac{1}{\varepsilon} + \frac{1}{\varepsilon^2} \cdot (p_{CC})^{10} \right) \) and one call to Lemma 9.2 with \( \varepsilon_2 = \Theta(\varepsilon) \) which takes \( \tilde{O} \left( \frac{1}{\varepsilon} \cdot (p_{CC})^{10} \right) \). Combining these running times gives the overall running time of our algorithm. \( \square \)

### 13 CONCLUSION AND FUTURE WORK

In this work we gave the first efficient randomized and deterministic algorithms for computing \((1 - \varepsilon)\)-approximate length-constrained flows both in parallel and in the CONGEST model of distributed computation. We used these algorithms to give new results in maximal and maximum disjoint path problems, expander decompositions, bipartite \( b \)-matching and length-constrained cutmatchings. We conclude with open questions and directions for future work.

1. Our length-constrained flow algorithms have a dependence of \( \text{poly}(h) \) which when plugged into the techniques of [26] give CONGEST algorithms for many distributed problems, e.g., MST, whose running time is \( \text{poly}(\text{OPT}) \) (up to sub-linear factors) where OPT is the optimal CONGEST running time for the input problem. It would be exciting to improve the dependence on \( h \) of our algorithms to, say, \( O(h) \) as this result when combined with those of [26] would give CONGEST algorithms running in time \( O(\text{OPT}) \) (up to sub-linear factors).

2. The running time of many of our algorithms depends on \( p_{CC} \), the best quality of a CONGEST algorithm for cycle cover (as defined in Definition 5.2). It is known that \( p_{CC} \leq 2^\Omega(\sqrt{\log n}) \) but it would be extremely interesting to show that \( p_{CC} \leq O(1) \). Such an improvement would immediately improve the dependency on \( n \) from \( n^{O(1)} \) to \( O(1) \) for our CONGEST algorithms for deterministic length-constrained flows, deterministic maximal and maximum disjoint paths, \((1 - \varepsilon)\)-approximate \( b \)-matching and length-constrained cutmatchings. Such a result does not seem to be known even for the randomized case.

3. Lastly, many classic problems can be efficiently solved by reducing to flows but, in particular, by reducing to length-constrained flows with a length-constraint \( h = O(1) \). Indeed this is how we were able to give new algorithms for \( b \)-matching in this work. It would be interesting to understand which additional classic problems our length-constrained flow algorithms give new algorithms for in CONGEST.

### ACKNOWLEDGMENTS

Hershkowitz and Haeupler supported in part by NSF grants CCF-1527110, CCF-1618280, CCF-1814603, CCF-1910588, NSF CAREER award CCF1750808, a Sloan Research Fellowship, funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (ERC grant agreement 949272) and the Swiss National Foundation (project grant 200021-184735). Hershkowitz also supported by the Air Force Office of Scientific Research under award number FA9550-20-1-0080. Saranurak supported by NSF CAREER grant 2238138.

### REFERENCES

[1] Mohamad Ahmadi, Fabian Kuhn, and Rotem Oshman. 2018. Distributed approximate maximum matching in the CONGEST model. In *International Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[2] Kateřina Altmanová, Petr Kolman, and Jan Voborník. 2019. On polynomial-time combinatorial algorithms for maximum \( b \)-bounded flow. In *Algorithms and Data Structures Symposium (WADS)* Springer, 14–27.

[3] Alexandr Andoni, Clifford Stein, and Peilin Zhong. 2020. Parallel approximate combinatorial algorithms for maximum \( l \)-bounded flow. In *Algorithm Engineering and Experimental Algorithms (ALENEX)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[4] Sanjeev Arora, Boaz Barak, and David Steurer. 2015. Subexponential algorithms for planar graphs and multicommodity flows: a primal-dual approach. In *Annual ACM Symposium on Theory of Computing (STOC)*. 322–335.

[5] Baruch Awerbuch and Robh Khandekar. 2007. Distributed network monitoring and multicommodity flows: a primal-dual approach. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 284–291.
Alok Baveja and Aravind Srinivasan. 2000. Approximation algorithms for disjoint paths and related routing and packing problems. *SIAM J. Comput.* 29, 5 (1999), 1560–1580.

Yi-Jun Chang and Thatchaphol Saranurak. 2020. Deterministic distributed edge partition decomposition and routing with applications in distributed derandomization. In *Conference on Computer Science and Cybersecurity* (CCS). ACM, 1–12.

Erik D. Demaine, MohammadTaghi Hajiaghayi, and Hsien-Chih Liao. 2009. Higher lower bounds for bipartite graph partitioning and multicommodity flow problems. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA). SIAM, 916–924.

Bernhard Haeupler, Harald Raecke, and Mohsen Ghaffari. 2022. Hop-Constrained Expander Decompositions: Oblivious Routing, and Distributed Universal Optimality. In *Annual ACM Symposium on Theory of Computing* (STOC).

Bernhard Haeupler, David Wajc, and Goran Zuzic. 2020. Network coding gaps for completion times of multiple unicasts. In *Symposium on Foundations of Computer Science* (FOCS). IEEE, 494–505.

Bernhard Haeupler, David Wajc, and Goran Zuzic. 2021. Universally-optimal distributed algorithms for known topologies. In *Annual ACM Symposium on Theory of Computing* (STOC).

Venkatesan Guruswami, Sanjeev Khanna, Rajmohan Rajaraman, Bruce Shepherd, and Mihalis Yannakakis. 2003. Near-optimal hardness results and approximation algorithms for disjoint paths problems. In *Proceedings of the Annual ACM Symposium on Theory of Computing* (STOC). SIAM, 67, 3 (2003), 473–496.