Parallel Approximate Undirected Shortest Paths via Low Hop Emulators*

Alexandr Andoni†
andoni@cs.columbia.edu
Columbia University
New York, USA

Clifford Stein‡
cliff@cs.columbia.edu
Columbia University
New York, USA

Peilin Zhong§
peilin.zhong@columbia.edu
Columbia University
New York, USA

ABSTRACT

We present a $(1+\epsilon)$-approximate parallel algorithm for computing shortest paths in undirected graphs, achieving poly$(\log n)$ depth and $m$poly$(\log n)$ work for n-nodes m-edges graphs. Although sequential algorithms with (nearly) optimal running time have been known for several decades, near-optimal parallel algorithms have turned out to be a much tougher challenge. For $(1+\epsilon)$-approximation, all prior algorithms with poly$(\log n)$ depth perform at least $\Omega(mn^c)$ work for some constant $c > 0$. Improving this long-standing upper bound obtained by Cohen (STOC'94) has been open for 25 years.

We develop several new tools of independent interest. One of them is a new notion beyond hopsets — low hop emulator — a poly$(\log n)$-approximate emulator graph in which every shortest path has at most $O(\log \log n)$ hops (edges). Direct applications of the low hop emulators are parallel algorithms for poly$(\log n)$-approximate single source shortest path (SSSP), Bourgain's embedding, metric tree embedding, and low diameter decomposition, all with poly$(\log n)$ depth and $m$poly$(\log n)$ work.

To boost the approximation ratio to $(1+\epsilon)$, we introduce compressible preconditioners and apply it inside Sherman's framework (SODA'17) to solve the more general problem of uncapacitated minimum cost flow (a.k.a., transshipment problem). Our algorithm computes a $(1+\epsilon)$-approximate uncapacitated minimum cost flow in poly$(\log n)$ depth using $m$poly$(\log n)$ work. As a consequence, it also improves the state-of-the-art sequential running time from $m \cdot 2^{O(\sqrt{\log n})}$ to $m$poly$(\log n)$.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; **Shortest paths**; *Network flows*; *Massively parallel algorithms.*

*Full version of this paper appears as https://arxiv.org/pdf/1911.01956.pdf.

†Research supported in part by Simons Foundation (#491119 to Alexandr Andoni), NSF (CCF-1617955, CCF-1740833), and Google Research Award.
‡Research supported in part by NSF grants CCF-1714818 and CCF-1822809.
§Research supported in part by NSF grants CCF-1740833, CCF-1703925, CCF-1714818 and CCF-1822809, as well as the Google PhD Fellowship.

STOC '20, June 22–26, 2020, Chicago, IL, USA

ACM ISBN 978-1-4503-6979-4/20/06…$15.00
https://doi.org/10.1145/3357713.3384321

KEYWORDS

parallel algorithms, shortest paths, minimum cost flow, low hop emulators

ACM Reference Format:
Alexandr Andoni, Clifford Stein, and Peilin Zhong. 2020. Parallel Approximate Undirected Shortest Paths via Low Hop Emulators. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), June 22–26, 2020, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357713.3384321

1 INTRODUCTION

The problem of finding the shortest path between two vertices in an undirected weighted graph is one of the most fundamental problems in computer science. Standard sequential algorithms with (nearly) optimal running time have been known for several decades [27, 30, 57]. In contrast, parallelizing these algorithms has been a challenge, and existing parallel algorithms are far from attaining the efficiency we would like. Two standard measures of the efficiency of a parallel algorithm in the standard PRAM model of parallelism are work (total time over the processors)[1] and depth (parallel time). The exact shortest path can be computed by the standard path-doubling (Floyd-Warshall) algorithm in poly$(\log n)$ parallel time using $O(n^3)$ total work, for an n-node m-edge graph. This result has been improved in a long line of work [11, 13, 17, 26, 42, 55, 56]. Nevertheless, the state-of-the-art algorithms have either $\Omega(n^{2.1})$ work or $\Omega(n^{0.1})$ depth.

In order to achieve algorithms with better bounds on work and depth, researchers have turned to approximation algorithms. Building on the idea of hopsets [16], a series of papers, including [16, 18, 23, 41, 49] give $(1+\epsilon)$-approximation algorithms. Yet again, every prior algorithm with m poly$(\log n)$ work has at least $\Omega(n^\rho)$ depth, and the ones with poly$(\log n)$ depth do $\Omega(mn^\rho)$ work, where $\rho > 0$ is an arbitrary small constant. In particular, none of the prior algorithms achieve poly$(\log n)$ depth and m poly$(\log n)$ work simultaneously. In fact, there was no known parallel algorithm with poly$(\log n)$ parallel time and m poly$(\log n)$ work that approximates the shortest path even up to a poly$(\log n)$ factor. Hence, after [16], a major question which remained open for more than 25 years is:

Question 1.1. *Is there a parallel algorithm computing an approximate shortest path in* poly$(\log n)$ *depth and* m poly$(\log n)$ *work?*

In this paper, we answer this question positively by developing a parallel $(1+\epsilon)$-approximate shortest path algorithm with poly$(\log n)$ depth and m poly$(\log n)$ work.

[1] More precisely, the work is the running time required when only one processor can be used, i.e., the sequential running time when the algorithm is implemented in the vanilla RAM model.

1.1 Our Results and Comparison to Prior Approaches

To obtain our main result, we develop new tools, which we present next and which may be of independent interest. It is most natural to present these results in the context of two related approaches to parallel shortest path algorithms — hopsets and continuous optimization techniques.

We note that some of our results have new consequences beyond parallel algorithms, including faster sequential algorithms and constructions where none were previously known. Our input is a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ with weights $w : E \to \mathbb{Z}_{\geq 0}$ and $\max_{e \in E} w(e) \leq \text{poly}(n)$. The parallel algorithms from this paper are in the EREW PRAM model.

Hopsets. One iteration of Bellman-Ford can be implemented efficiently in parallel, and therefore, for graphs in which an approximate shortest path has a small number of hops (edges) we already have an efficient algorithm. Motivated by this insight, researchers have proposed adding edges to a graph in order to make an approximate shortest path with a small number of edges between every pair of vertices. Formally, for a given graph $G = (V, E, w)$ with weights $w : E \to \mathbb{R}_{\geq 0}$, a hopset is an edge-set H with weights $w_H : H \to \mathbb{R}_{\geq 0}$. Let $\widetilde{G}$ be the union graph $(V, E \cup H, w \cup w_H)$. We define $\text{dist}^{(h)}_{\widetilde{G}}(u, v)$, the h-hop distance in $\widetilde{G}$, to be the length of the shortest path between $u, v \in V$ which uses at most h hops (edges) in $\widetilde{G}$. Then H is an (h, ϵ)-*hopset* of G if $\forall u, v \in V$, $\text{dist}^{(h)}_{\widetilde{G}}(u, v)$ is always a $(1 + \epsilon)$-approximation to the shortest distance between u and v in the graph G. There is a three-way trade-off between h, ϵ, and $|H|$, which was studied in [16, 23, 34, 49, 59], leading to some of the aforementioned algorithms.

Surprisingly, a hard barrier arose: [1] showed that the size of H must be $\Omega(n^{1+\rho})$ for any $h \leq \text{poly}(\log n)$, $\epsilon < \frac{1}{\log n}$ and some constant $\rho > 0$. Thus, it is impossible to directly apply hopsets to compute a $(1 + \epsilon)$-approximate shortest path in $\text{poly}(\frac{\log n}{\epsilon})$ parallel time using $m\,\text{poly}(\frac{\log n}{\epsilon})$ work for sparse G, when $|E| = O(n)$.

Low Hop Emulator. To bypass this hardness, we introduce a new notion — *low hop emulator* — which has a weaker approximation guarantee than hopsets, but has stronger guarantees in other ways. A low hop emulator $G' = (V, E', w')$ of G is a sparse graph with $n\,\text{poly}(\log n)$ edges satisfying two properties. First, the distance between every pair of vertices in G' is a $\text{poly}(\log n)$ approximation to the distance in G. The second property is that G' has a low hop diameter, i.e., a shortest path between every pair of two vertices in G' only contains $O(\log \log n)$ number of hops (edges).

We give an efficient parallel sparse low hop emulator construction algorithm. To the best of our knowledge, it was not even clear whether sparse low hop emulators exist, and thus no previous algorithm was known even in the sequential setting.

Theorem 1.2 (Low hop emulator). *For any $k \geq 1$, any graph G admits a low hop emulator G', with expected size of $\widetilde{O}\left(n^{1+\frac{1}{k}}\right)$,[2] satisfying: $\forall u, v \in V, \text{dist}_G(u, v) \leq \text{dist}_{G'}(u, v) \leq \text{poly}(k) \cdot \text{dist}_G(u, v)$, and with the hop diameter at most $O(\log k)$. Furthermore, there is a PRAM algorithm computing the emulator G' in $\text{poly}\log(n)$ parallel time using $\widetilde{O}(m + n^{1+\frac{2}{k}})$ expected work.*

Notice that, setting $k = \log n$, we can compute a low hop emulator with expected size $\widetilde{O}(n)$ and hop diameter $O(\log \log n)$ in $\text{poly}\log(n)$ parallel time using $\widetilde{O}(m)$ expected work. The approximation ratio in this case is $\text{poly}\log(n)$.

We now highlight two main features that make a low hop emulator stronger than hopsets. Firstly, the low hop emulator can be computed in $\text{poly}(\log n)$ parallel time using $m\,\text{poly}(\log n)$ work while the same guarantees cannot be simultaneously achieved by hopsets. Secondly, the $O(\log \log n)$-hop distances in low hop emulator G' satisfy the *triangle inequality* while the h-hop distances in the union graph $\widetilde{G}$ of original graph G and the (h, ϵ)-hopset do not.

An immediate application of the first feature is a $\text{poly}(\log n)$-approximate single source shortest path (SSSP) algorithm using $\text{poly}(\log n)$ parallel time and $m\,\text{poly}(\log n)$ work. We remark that when using the hop-distance to approximate the exact distance in G, we only need to use edges from the low hop emulator while we also need to use original edges if we use hopsets.

The second feature is crucial for designing parallel algorithms for Bourgain's embedding [12], metric tree embedding [24, 28] and low diameter decomposition [50], using $\text{poly}(\log n)$ depth and $m\,\text{poly}(\log n)$ work. [28] introduced a notion similar to low hop emulators, and it also has the second feature mentioned above. In contrast, their emulator graph is a complete graph, and the construction is based on $\left(\text{poly}(\log n), \frac{1}{\text{poly}(\log n)}\right)$-hopsets.

Continuous optimization. To boost the approximation ratio of shortest path from $\text{poly}(\log n)$ to $(1+\epsilon)$, we employ continuous optimization techniques. Recently, continuous optimization techniques have been successfully applied to design new efficient algorithms for many classic combinatorial graph problems, e.g., [15, 19, 20, 38, 39, 44, 48, 52–54]. Most of them can be seen as "boosting" a coarse approximation algorithm to a more accurate approximation algorithm. Oftentimes, to fit into a general optimization framework, the "coarse" approximation must be for a more general problem — in our case, for the uncapacitated minimum cost flow, also known as the transshipment problem. Following this approach, the work of [8] develops near-optimal uncapacitated min-cost flow algorithms in the distributed and streaming settings based on the gradient descent algorithm. Their algorithm can be seen as boosting a $\text{poly}(\log n)$ approximate solver for the uncapacitated min-cost flow problem to an $(1 + \epsilon)$ approximate solver, but with one crucial difference: it requires a $\text{poly}(\log n)$ approximate solver for the *dual problem*. Hence it is not clear how to leverage their algorithm for our goal as the aforementioned techniques do not seem applicable to the dual of uncapacitated min-cost flow.

We develop an algorithm for the uncapacitated min-cost flow problem by opening up Sherman's framework [54] and combining it with new techniques. There is a fundamental challenge in adopting Sherman's framework, beyond implementing it in the parallel setting. Sherman's original algorithm solves the uncapacitated minimum cost flow problem in $m \cdot 2^{O(\sqrt{\log n})}$ sequential time. Hence, if we obtain a parallel uncapacitated min-cost flow algorithm with $m\,\text{poly}(\log n)$ total work, we cannot avoid improving this best-known running time of $m \cdot 2^{O(\sqrt{\log n})}$ to $m\,\text{poly}(\log n)$.

[2] $\widetilde{O}(f(n))$ denotes $f(n) \cdot \text{poly}\log(f(n))$.

Uncapacitated minimum cost flow and approximate $s-t$ shortest path. To handle the challenge mentioned above, we develop a novel *compressible* preconditioner. By using our compressible preconditioner inside Sherman's framework, we improve the running time of $(1+\epsilon)$-approximate uncapacitated min-cost flow from $m\cdot 2^{O(\sqrt{\log n})}$ to m poly$(\log n)$. Furthermore, we show that such compressible preconditioner can be computed in poly$(\log n)$ parallel time using m poly$(\log n)$ work. This preconditioner relies crucially on our low hop emulator ideas.

Formally, in the uncapacitated minimum cost flow problem, given a demand vector $b \in \mathbb{R}^n$ satisfying $\sum_{v\in V} b_v = 0$, the goal is to determine the flow on each edge such that the demand of each vertex is satisfied and the cost of the flow is minimized.

Theorem 1.3 (Parallel uncapacitated minimum cost flow). *Given a graph $G = (V, E, w)$, a demand vector $b \in \mathbb{R}^n$ and an error parameter $\epsilon \in (0, 0.5)$, there is a PRAM algorithm which outputs an $(1+\epsilon)$-approximate solution to the uncapacitated minimum cost flow problem with probability at least 0.99 in ϵ^{-2} poly$\log(n)$ parallel time using $\widetilde{O}(\epsilon^{-2}m)$ expected work.*

While the above techniques are sufficient for estimating the *value* of the shortest path, one additional challenge arises when we want to compute an $(1+\epsilon)$-approximate shortest *path*. In particular, the continuous optimization framework produces an approximate shortest path *flow*, which is not necessary integral and, more crucially, may contain cycles. We address this challenge by developing a novel recursive algorithm based on random walks, and which uses a coupling argument.

Theorem 1.4 (Parallel $(1+\epsilon)$-approximate $s-t$ shortest path). *Given a graph $G = (V, E, w)$, two vertices $s, t \in V$ and an error parameter $\epsilon \in (0, 0.5)$, there is a PRAM algorithm which can output a $(1+\epsilon)$-approximate $s-t$ shortest path with probability at least 0.99 in ϵ^{-2} poly$\log(n)$ parallel time using expected $\widetilde{O}(\epsilon^{-3}m)$ work.*

Massive Parallel Computing (MPC).. Although we present our parallel algorithms in the PRAM model, they can also be implemented in the Massive Parallel Computing (MPC) model [2, 7, 25, 29, 37] which is an abstract of massively parallel computing systems such as MapReduce [21], Hadoop [60], Dryad [36], Spark [61], and others. In particular, MPC model allows m^δ space per machine for some $\delta \in (0, 1)$. The computation in MPC proceeds in rounds, where each machine can perform unlimited local computation and exchange up to m^δ data in one round.

By applying the simulation methods [29, 37], our PRAM algorithm can be directly simulated in MPC. The obtained MPC algorithm has poly$(\log n)$ rounds and only needs $m \cdot$ poly$(\log n)$ total space. Furthermore, it is also fully scalable, i.e., the memory size per machine can be allowed to be m^δ for any constant $\delta \in (0, 1)$. To the best of our knowledge, this is the first MPC algorithm which computes $(1+\epsilon)$-approximate shortest path using poly$(\log n)$ rounds and m poly$(\log n)$ total space when the memory of each machine is upper bounded by $n^{1-\Omega(1)}$. Previous work on shortest paths in the MPC model include [22] when the memory size per machine is $o(n)$, and simulations of shortest path algorithms from the Congested Clique model [8, 14, 32, 33, 40, 47] when the memory size per machine is $\Omega(n)$ [9].

Independent work. Independently, [46] developed an alternative $\widetilde{O}(m)$ algorithm for the uncapacitated minimum cost flow problem (see Theorem 4.19). The first arXiv version of [46], released at the same time as the arXiv version of this paper [4], on November 5, 2019, also claimed a parallel algorithm with performance similar to one from Theorem 1.4. However, that version was missing a necessary component (how to compute a path from a flow in parallel, see Section 1.2.2), without which the algorithm could not even compute the cost in parallel. The author of [46] later developed an algorithm for this component, which appeared in a revision of his arXiv paper, on December 19, 2019.

1.2 Our Techniques

In this section, we give an overview of techniques that we use in our algorithms. Figure 1 sketches the dependencies between our techniques and the main results mentioned in this paper.

1.2.1 Low Hop Emulator. A concept closely related to low hop emulator are hopsets [16]. A hopset is a set of weighted shortcut edges such that for any two vertices s and t we can always find an approximate shortest path connecting them using small number of edges from the hopset and the original graph. Many hopset construction methods [10, 16, 23, 31, 32, 34, 42, 49, 58, 59] share some common features — they all choose a layer or multiple layers of leader vertices, and the hopset edges are some shortcut edges connecting to these leader vertices. However, when connecting shortcut edges to a layer of leader vertices, none of these algorithms can avoid processing information for all n vertices from the original graph, even though there may be a large fraction of vertices which are not connecting any of this layer's leader vertex in the final hopset. Furthermore, each of these algorithms needs either $n \cdot \log^{\omega(1)} n$ work (sequential time) or $\log^{\omega(1)} n$ depth to process n vertices for constructing shortcut edges for some layers. To improve the efficiency of these algorithms, a natural question is: can we reduce the number of vertices needed to be processed when constructing the shortcut edges?

Subemulator. Motivated by the above question, we introduce a new concept called subemulator. For $\alpha \geq 1$ and an integer $b \geq 1$, we say $H = (V', E', w')$ is an (α, b)-subemulator of $G = (V, E, w)$ if 1) V' is a subset of V; 2) for any vertex v in G, at least one of the b-closest neighbors[3] of v is in V'; 3) for any two vertices u, v in H, $\text{dist}_H(u, v)$ α-approximates $\text{dist}_G(u, v)$. In addition, if we can assign each vertex $v \in V$ a leader $q(v) \in V'$ such that $q(v)$ is one of the b-closest neighbors of v and for any two vertices $u, v \in V$ it always satisfies

$$\begin{aligned}\text{dist}_G(q(u), q(v)) &\leq \text{dist}_H(q(u), q(v)) \\ &\leq \text{dist}_G(q(u), u) + \beta \cdot \text{dist}_G(u, v) + \text{dist}_G(v, q(v))\end{aligned} \tag{1}$$

for some $\beta \geq 1$, we call H a strong (α, b, β)-subemulator of G. A subemulator H can be regarded as a sparsification of vertices of G. Two notions related to subemulators are vertex sparsifiers [45, 51] and distance-preserving minors [43]. The major difference between

[3] We assume that v is also a neighbor of v itself. Thus the closest (or 1-closest) neighbor of v is v itself.

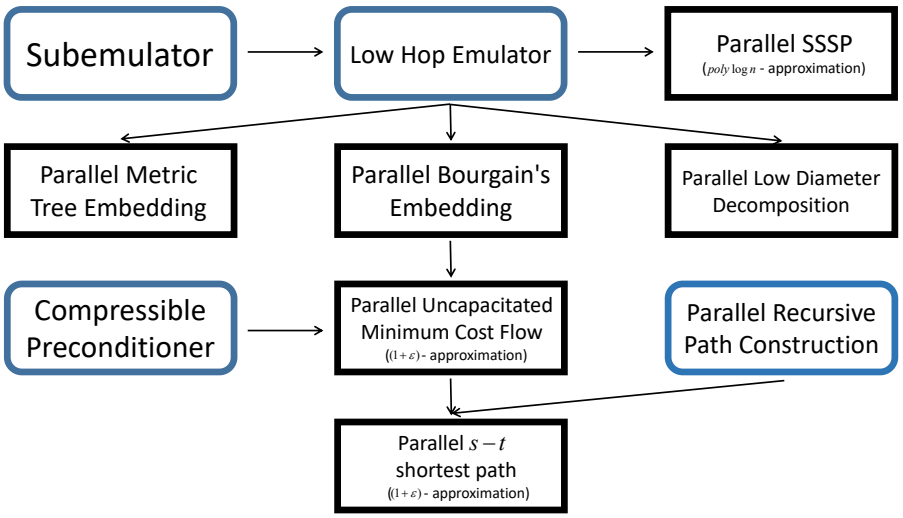

Figure 1: Techniques and results mentioned in this paper. Blue rounded rectangles indicate new techniques developed in this paper.

subemulators and vertex sparsifiers is that the vertex sparsifier approximately preserves flow/cut properties for the subset of vertices while the subemulator approximately preserves distances for the subset of vertices. Furthermore, both vertex sparsifiers and distance-preserving minors have given fixed vertex sets, whereas the vertex set of the subemulator is not given but should satisfy the condition 2) mentioned above, i.e., each vertex in G has a b-closest neighbor which is in the subemulator.

To construct a strong subemulator $H = (V', E', w')$, we need to construct both a vertex set V' and a edge set E'. For convenience, let us consider the case for $b \gg \log n$. Constructing V' is relatively easy. We can add each vertex of V to V' with probability $\Theta(\log(n)/b)$. By Chernoff bound, with high probability, each vertex has at least one of the b-closest neighbors in V' and the size of V' is roughly $\widetilde{O}(n/b)$. For each vertex $v \in V$, it is natural to set the leader vertex $q(v)$ to be the vertex in V' which is the closest vertex to v. The challenge remaining is to construct the edge set E' such that condition 3) and Equation (1) can be satisfied. In our construction, we add two categories of edges to E':

(1) For each edge $\{u, v\} \in E$, add an edge $\{q(u), q(v)\}$ with weight $\text{dist}_G(q(u), u) + w(u, v) + \text{dist}_G(v, q(v))$ to E'.
(2) For each $v \in V$ and for each u which is a b-closest neighbor of v, we add an edge $\{q(u), q(v)\}$ with weight $\text{dist}_G(q(u), u) + \text{dist}_G(u, v) + \text{dist}_G(q(v), v)$ to E'.

The first category of edges looks natural — for an edge $\{u, v\}$ of which two end points u, v are assigned to different leader vertices $q(u), q(v)$, we add a shortcut edge connecting those two leader vertices with a weight which is equal to the smallest length of the $q(u) - q(v)$ path crossing edge $\{u, v\}$. However, if we only have the edges from the first category, it is not good enough to preserve the distances between leader vertices. To fix this, we add the second category of edges. We now sketch the analysis. It follows from our construction that each edge in H corresponds to a path in G. Thus, $\forall u', v' \in V'$, $\text{dist}_G(u', v') \le \text{dist}_H(u', v')$. We only need to upper bound $\text{dist}_H(u', v')$. Let us fix a shortest path $u' = z_0 \to z_1 \to \cdots \to z_h = v'$ between u', v' in the original graph G. We want to construct a path in H with a short length. We use the following procedure to find some crucial vertices on the shortest path $z_0 \to \cdots \to z_h$:

(1) $y_0 \leftarrow u', k \leftarrow 0$. Repeat the following two steps:
(2) Let x_{k+1} be the last vertex on the path $z_0 \to \cdots \to z_h$ such that x_{k+1} is one of the b-closest neighbors of y_k. If x_{k+1} is z_h, finish the procedure.
(3) Set y_{k+1} to be the next vertex of x_{k+1} on the path $z_0 \to \cdots \to z_h$. $k \leftarrow k + 1$.

It is obvious that

$$\text{dist}_G(u', v') = \text{dist}_G(y_k, x_{k+1}) + \sum_{i=0}^{k-1} (\text{dist}_G(y_i, x_{i+1}) + w(x_{i+1}, y_{i+1})).$$

For $i = 0, 1, \cdots, k$, x_{i+1} is a b-closest neighbor of y_i. Thus, there is an edge $\{q(y_i), q(x_{i+1})\}$ in H from the second category of the edges. For $i = 1, 2, \cdots, k$, y_i is adjacent to x_i. Thus, there is an edge $\{q(x_i), q(y_i)\}$ in H from the first category of the edges. Thus $u' = q(y_0) \to q(x_1) \to q(y_1) \to q(x_2) \to q(y_2) \to \cdots \to q(x_{k+1}) = v'$ is a valid path (see Figure 2) in H and the length is

$$\text{dist}_G(u', v') + 2 \cdot \sum_{i=1}^{k} (\text{dist}_G(x_i, q(x_i)) + \text{dist}_G(y_i, q(y_i))).$$

By our choice of $q(\cdot)$, we have $\forall v \in V, \text{dist}_G(v, q(v)) = \text{dist}_G(v, V')$. So, $\forall i = 1, 2, \cdots, k$,

$$\text{dist}_G(x_i, q(x_i)) \le \text{dist}_G(y_{i-1}, q(y_{i-1})) + \text{dist}_G(y_{i-1}, x_i).$$

Since y_i is not a b-closest neighbor of y_{i-1} but $q(y_{i-1})$ is a b-closest neighbor of y_{i-1}, $\forall i = 1, 2, \cdots, k$,

$$\text{dist}_G(y_{i-1}, q(y_{i-1})) \le \text{dist}_G(y_{i-1}, x_i) + w(x_i, y_i).$$

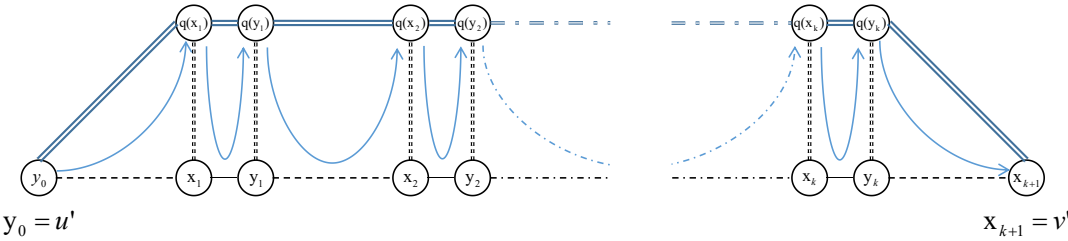

Figure 2: For $u', v' \in V'$ and a shortest path between u', v' in G, we can find a corresponding path between u', v' in the subemulator H. A single dashed line denotes a shortest path in G between y_{i-1} and x_i. A single solid line denotes an edge $\{x_i, y_i\}$ in G. A double dashed line denotes a shortest path in G between a vertex and its leader vertex. A double solid blue line denotes an edge in the subemulator H with a weight which is equal to the length of the path in G represented by the corresponding blue arc.

Since $x_{k+1} \in V'$, we have $\text{dist}_G(y_k, q(y_k)) \leq \text{dist}_G(y_k, x_{k+1})$. Then $\sum_{i=1}^{k} \text{dist}_G(x_i, q(x_i)) \leq 2 \cdot \text{dist}_G(u', v')$ and $\sum_{i=1}^{k} \text{dist}_G(y_i, q(y_i)) \leq \text{dist}_G(u', v')$. Thus, we can conclude $\text{dist}_H(u', v') \leq 8 \cdot \text{dist}_G(u', v')$. We now argue that our construction of E' also satisfies Equation (1) with $\beta = 22$. There are two cases. The first case is that either u is a b-closest neighbor of v or v is a b-closest neighbor of u. In this case, E' contains an edge from the second category with weight $\text{dist}_G(q(u), u) + \text{dist}_G(u, v) + \text{dist}_G(v, q(v))$ which implies Equation (1). The second case is that neither u is a b-closest neighbor of v nor v is a b-closest neighbor of u. In this case, we have

$$\begin{aligned}\text{dist}_H(q(u), q(v)) &\leq 8\,\text{dist}_G(q(u), q(v)) \\ &\leq 8(\text{dist}_G(q(u), u) + \text{dist}_G(u, v) + \text{dist}_G(v, q(v))) \\ &\leq \text{dist}_G(q(u), u) + \text{dist}_G(v, q(v)) + 22\,\text{dist}_G(u, v),\end{aligned}$$

where the last step follows from $\text{dist}_G(u, q(u)), \text{dist}_G(v, q(v)) \leq \text{dist}_G(u, v)$.

The bottleneck of computing a subemulator is to obtain b-closest neighbors for each vertex. We can use the truncated broadcasting technique [3, 55] to handle this in poly$(\log n)$ parallel time using $\widetilde{O}(m + nb^2)$ total work. The output subemulator has $\widetilde{O}(n/b)$ vertices and $O(m + nb)$ edges. As we can see, there is a trade-off between total work used and the number of vertices in the subemulator: if we can afford more work for the construction of the subemulator, fewer vertices appear in the subemulator.

Low hop emulator via subemulator. Now, we describe how to use strong subemulators to construct a low hop emulator. Consider a weighted undirected graph $G = (V, E, w)$. Suppose we obtain a sequence of subemulators $H_0 = (V_0, E_0, w_0), H_1 = (V_1, E_1, w_1), \cdots, H_t = (V_t, E_t, w_t)$ where $H_0 = G$ and $\forall i = 0, \cdots, t-1$, H_{i+1} is a strong $(8, b_i, 22)$-subemulator of H_i for some integer $b_i \geq 1$. We have $V = V_0 \supseteq V_1 \supseteq V_2 \supseteq \cdots \supseteq V_t$. For $v \in V_i$, let us denote $q_i(v) \in V_{i+1}$ as the corresponding assigned leader vertex of v in the subemulator H_{i+1} satisfying Equation (1). We add following three types of edges to the graph $G' = (V, E', w')$ and we will see that G' is a low hop emulator of G:

(1) $\forall i = 0, \cdots, t-1$, $\forall v \in V_i$, add an edge $\{v, q_i(v)\}$ with weight $27^{t-i-1} \cdot \text{dist}_{H_i}(v, q_i(v))$ to G'.
(2) $\forall i = 0, \cdots, t$, for each edge $\{u, v\} \in E_i$, add an edge $\{u, v\}$ with weight $27^{t-i} \cdot w_i(u, v)$ to G'.
(3) $\forall i = 0, \cdots, t$, $\forall v \in V_i$, add an edge $\{v, u\}$ with weight $27^{t-i} \cdot \text{dist}_{H_i}(v, u)$ to G' for each u which is one of the b_i-closest neighbors of v in H_i (define $b_t = |V_t|$).

Roughly speaking, we can imagine that G' is obtained from flattening a graph with $t+1$ layers. Each layer corresponds to a subemulator. The lowest layer corresponds to the original graph G, and the highest layer corresponds to the last subemulator H_t. The first type of edges connect the vertices in the lower layer to the leader vertices in the higher layer. The second type of edges correspond to the subemulators on all layers. The third type of edges shortcut the close vertices from the same layer. Furthermore, the weights of the edges on the lower layers have larger penalty factor, i.e., the penalty factor of the edges on the layer i is 27^{t-i}.

By Equation (1) of strong subemulator, we can show that $\forall u, v \in V$, $\text{dist}_G(u, v) \leq \text{dist}_{G'}(u, v)$. Consider the first layer. By the second type edges, we know that $\forall u, v \in V$, $\text{dist}_{G'}(u, v) \leq 27^t\,\text{dist}_G(u, v)$. In particular, for $t = O(\log \log n)$, G' preserves the distances in G up to a poly$(\log n)$ factor. Now we want to show that $\forall u, v \in V$, there is always a shortest path connecting u, v in G' such that the number of hops (edges) of the path is at most $4t$. For convenience, we conceptually split each vertex of G' into vertices on different layers based on the construction of G'. Consider a shortest path $u = z_0 \to z_1 \to z_2 \to \cdots \to z_h = v$ using the smallest number of hops in G' with splitting vertices. By the constructions of three types of edges we know that $\forall j = 0, 1, \cdots, h-1$, z_j, z_{j+1} are either on the same layer or on the adjacent layers, and z_0, z_h are on the lowest layer which is corresponding to $H_0 = G$. We will claim two properties of the shortest path $z_0 \to \cdots \to z_h$. Suppose z_j, z_{j+1} are on the same layer corresponding to H_i. We claim that z_{j+2} cannot be on the same layer as z_j and z_{j+1}. Intuitively, this is because if z_{j+2} is on the same layer of z_j then there are two cases which both lead to contradictions: in the first case, z_{j+2} is close to z_j such that there is a third type edge connecting z_j, z_{j+2} which implies that z_{j+1} is redundant; in the second case, z_{j+2} is far away from z_j such that $\text{dist}_{H_i}(z_j, q_i(z_j)) + \text{dist}_{H_{i+1}}(q_i(z_j), q_i(z_{j+2})) + \text{dist}_{H_i}(q_i(z_{j+2}), z_{j+2})$ is a good approximation to $\text{dist}_{H_i}(z_j, z_{j+2})$, and due to a smaller penalty factor, the length of the path $z_j \to q_i(z_j) \to$ (shortest path) $\to q_i(z_{j+2}) \to z_{j+2}$ is smaller than the length of $z_j \to z_{j+1} \to z_{j+2}$. We claim another property of $z_0 \to \cdots \to z_h$ as the following. If the layer of z_{j+1} is lower than the layer of z_j, the layer of any of $z_{j+2}, z_{j+3}, \cdots, z_h$ must be lower than the layer of z_j. At a high level, this is because of Equation (1) and the smaller penalty factor for higher layers: if we move from higher layer to lower layer then come back to the higher layer, it is always worse than we only move in the higher layers. Due to these two claims, the shortest path in G' should have the following shape: the path starts from the lowest layer, then keeps moving to the

non-lower layers until reach some vertex, and finally keeps moving to the non-higher layers until reach the target. Furthermore, there are no three consecutive vertices on the path which are on the same layer. Hence we can conclude that the shortest path has number of hops at most $4t$. Based on above analysis, the shortest path in G' will never use the second type edges. Thus, in our final construction of G', we only need the first type and the third type of edges.

The size of G' is at most $\sum_{i=0}^{t} |V_i| \cdot b_i$. The bottleneck of the construction of G' is to compute the third type edges. This can be done by truncated broadcasting technique [3, 55] in $t \cdot \text{poly}(\log n)$ parallel time using $\sum_{i=0}^{t} \left(|E_i| + |V_i| \cdot b_i^2\right) \cdot \text{poly}(\log n)$ total work. The problem remaining is to determine the sequence of b_i. As we discussed previously, we are able to use $\text{poly}(\log |V_i|)$ parallel time and $\widetilde{O}(|E_i| + |V_i|b_i)$ total work to construct a subemulator H_{i+1} with $\widetilde{O}(|V_i|/b_i)$ vertices and $O(|E_i| + |V_i|b_i)$ edges. By double exponential problem size reduction technique [3], we can make b_i grow double exponentially fast in this situation. More precisely, if we set $b_0 \leftarrow \text{poly}(\log n)$, $b_{i+1} \leftarrow b_i^{1.25}$, and $t \leftarrow O(\log \log n)$, then in this case, the result low hop emulator can be computed in $\text{poly}(\log n)$ parallel time and $\widetilde{O}(m+n)$ total work. Furthermore, the size of the result low hop emulator is $\widetilde{O}(n)$, the approximation ratio is $\text{poly}(\log n)$, and the hop diameter is $O(\log \log n)$.

Applications of low hop emulator. We can build a useful oracle based on a low hop emulator: given a query subset S of vertices, the oracle can output a $\text{poly}(\log n)$ approximations to $\text{dist}_G(v, S)$ for all $v \in V$. Furthermore, the output approximate distances always satisfy triangle inequality. To implement such oracle, we preprocess an $\widetilde{O}(n)$ size low hop emulator G' with $\text{poly}(\log n)$ approximation ratio and $O(\log \log n)$ hop diameter in $\text{poly}(\log n)$ parallel time using $\widetilde{O}(m+n)$ work. For each oracle query, we can just run Bellman-Ford on G' with source S. The work needed for each Bellman-Ford iteration is at most $\widetilde{O}(n)$. Since the hop diameter is $O(\log \log n)$, the number of iterations needed is $O(\log \log n)$. Therefore, each query can be handled in $\text{poly}(\log n)$ parallel time and $\widetilde{O}(n)$ total work. The triangle inequality is always satisfied since the output approximate distances are exact distances in the graph G'. Several parallel applications such as Bourgain's embedding, metric tree embedding and low diameter decomposition directly follow the oracle.

1.2.2 Minimum Cost Flow and Shortest Path.

Uncapacitated minimum cost flow. At a high level, our uncapacitated minimum cost flow algorithm is based on Sherman's framework [54]. Sherman's algorithm has several recursive iterations. It first uses the multiplicative weights update method [5] to find a flow which almost satisfies the demands and has nearly optimal cost. If the unsatisfied parts of demands are sufficiently small, it routes them naively to make the flow truly feasible without increasing the cost by too much. Otherwise, it updates the demands to be the unsatisfied parts of the original demands and recursively routes the new demands. [54] shows that if the problem is well conditioned, then the final solution can be computed by the above process efficiently. However, most of the time the natural form of the uncapacitated minimum cost flow problem is not well-conditioned. Thus, a preconditioner, i.e., a linear operator $P \in \mathbb{R}^{r \times n}$ applied to the flow constraints, is needed to make the problem well-condtitioned. Consider a given graph $G = (V, E, w)$. Sherman shows that if for any valid demands $b \in \mathbb{R}^n$ we always have $\text{OPT}(b) \le \|Pb\|_1 \le \gamma \cdot \text{OPT}(b)$, then P can make the condition number of the flow problem on G be upper bounded by γ, where $\text{OPT}(b)$ denotes the optimal cost of the flow on G satisfying the demands b. Sherman gives a method to construct such P. However, to have a smaller approximation ratio γ, the time of computing matrix-vector multiplication with P must increase such that the running time of the multiplicative weights update step increases. To balance the trade-off, Sherman constructs P with $\gamma = 2^{O(\sqrt{\log n})}$ approximation ratio and $\text{nnz}(x) \cdot 2^{O(\sqrt{\log n})}$ time for matrix-vector multiplication $P \cdot x$, where $\text{nnz}(x)$ denotes the number of non-zero entries of x. Thus, its final running time is $m \cdot 2^{O(\sqrt{\log n})}$. To design a parallel minimum cost flow algorithm using $\text{poly}(\log n)$ parallel time and $m\,\text{poly}(\log n)$ work, we cannot avoid improving the sequential running time of minimum cost flow to $m\,\text{poly}(\log n)$ time in sequential setting. By the above discussion, a natural way is to find a linear transformation P which can embed the minimum cost flow into ℓ_1 with $\text{poly}(\log n)$ approximation ratio and the running time for matrix-vector multiplication $P \cdot x$ needs to be $\text{nnz}(x) \cdot \text{poly}(\log n)$. Next, we will introduce how to construct such embedding P.

First, we compute a mapping φ which embeds the vertices into ℓ_1^d for $d = O(\log^2 n)$ such that $\forall u, v \in V$, $\|\varphi(u) - \varphi(v)\|_1$ is a $\text{poly}(\log n)$ approximation to $\text{dist}_G(u, v)$. This step can be done by Bourgain's embedding. The parallel version of Bourgain's embedding is one of the applications of low hop emulator as we mentioned previously. Then we can reduce the minimum cost flow problem to the geometric transportation problem. The geometric transportation problem is also called Earth Mover's Distance (EMD) problem. Specifically, it is the following minimization problem:

$$\min_{\pi: V \times V \to \mathbb{R}_{\ge 0}} \sum_{(u,v) \in V \times V} \pi(u, v) \cdot \|\varphi(u) - \varphi(v)\|_1$$
$$s.t.\ \forall u \in V, \sum_{v \in V} \pi(u, v) - \sum_{v \in V} \pi(v, u) = b_u.$$

We denote $\text{OPT}_{\text{EMD}}(b)$ as the optimal cost of the above EMD problem. It is easy to see that $\text{OPT}_{\text{EMD}}(b)$ is a $\text{poly}(\log n)$ approximation to $\text{OPT}(b)$. Therefore, it suffices to construct P such that for any valid demand vector $b \in \mathbb{R}^n$,

$$\text{OPT}_{\text{EMD}}(b) \le \|Pb\|_1 \le \text{poly}(\log n) \cdot \text{OPT}_{\text{EMD}}(b).$$

One known embedding of EMD into ℓ_1 is based on randomly shifted grids [35]. We can without loss of generality assume that the coordinates of $\varphi(v)$ are integers in $\{1, \cdots, \Delta\}$ for some Δ which is a power of 2 and upper bounded by $\text{poly}(n)$. We create $1 + \log \Delta$ levels of cells. We number each level from 0 to $\log \Delta$. Each cell in level $\log \Delta$ has side length Δ. Each cell in level $i+1$ is partitioned into 2^d equal size cells in level i and thus each cell in level i has side length 2^i. Therefore each cell in level 0 can contain at most one point $\varphi(v)$ for $v \in V$. According to [35], for any valid demand

vector $b \in \mathbb{R}^n$,

$$\mathop{\mathbf{E}}_{\tau\sim\{0,1,\cdots,\Delta-1\}}\left[\sum_{i=0}^{\log\Delta}\sum_{C\text{: a cell in level } i} 2^i\cdot\left|\sum_{v\in V:\varphi(v)+\tau\cdot\mathbf{1}_d \text{ is in } C} b_v\right|\right] \tag{2}$$

is always a poly($\log n$) approximation to $\mathrm{OPT}_{\mathrm{EMD}}(b)$, where τ is drawn uniformly at random from $\{0,1,\cdots,\Delta-1\}$, and $\varphi(v)+\tau\cdot\mathbf{1}_d$ is the point obtained after shifting each coordinate of $\varphi(v)$ by τ. Since each cell in level i has side length 2^i, Equation (2) is equal to

$$\begin{aligned}&\sum_{i=0}^{\log\Delta}\frac{1}{2^i}\sum_{\tau=0}^{2^i-1}\sum_{C\text{: a cell in level } i} 2^i\cdot\left|\sum_{v\in V:\varphi(v)+\tau\cdot\mathbf{1}_d \text{ is in the cell } C} b_v\right|\\ =&\sum_{i=0}^{\log\Delta}\sum_{C\text{: a cell in level } i}\sum_{\tau=0}^{2^i-1}\left|\sum_{v\in V:\varphi(v)+\tau\cdot\mathbf{1}_d \text{ is in the cell } C} b_v\right|.\end{aligned} \tag{3}$$

Equation (3) can be written in the from of $\|Pb\|_1$ where each row of P corresponds to a cell C and a shift value τ, and each column of P corresponds to a vertex v. Figure 3 shows how does P look like: for an entry $P_{i,j}$ corresponding to a cell C, a shift value τ and a vertex v, we have $P_{i,j}=1$ if the point $\varphi(v)+\tau\cdot\mathbf{1}_d$ is in the cell C and $P_{i,j}=0$ otherwise. Therefore, P can be used to precondition the minimum cost flow problem on G with condition number at most poly($\log n$). However, such matrix P is dense and have poly(n) number of rows. It is impossible to naively write down the whole matrix. Fortunately, we will show that P has a good structure and we can write down a compressed representation of P. Consider a cell C in level i and a vertex v. If there exists $\tau\in\{0,1,\cdots,2^i-1\}$ such that $\varphi(v)+\tau\cdot\mathbf{1}_d$ is in the cell C, then there must exist τ_1,τ_2 such that $\varphi(v)+\tau\cdot\mathbf{1}_d$ is in the cell C if and only if $\tau\in\{\tau_1,\tau_1+1,\cdots,\tau_2\}$. In other words, the shift values τ that can make $\varphi(v)+\tau\cdot\mathbf{1}_d$ be in C are consecutive. Another important property that we can show is that the number of cells in level i that can contain at least one of the shifted points $\varphi(v),\varphi(v)+\mathbf{1}_d,\varphi(v)+2\cdot\mathbf{1}_d,\cdots,\varphi(v)+(2^i-1)\cdot\mathbf{1}_d$ is at most $d+1$. Now consider a column of P corresponding to some vertex v. The entries with value 1 in this column should be in several consecutive segments. The number of such segments is at most $(d+1)\cdot(1+\log\Delta)\le\mathrm{poly}(\log n)$. Thus, for each column of P, we can just store the beginning and the ending positions of these segments. The whole matrix P can be represented by $n\,\mathrm{poly}(\log n)$ segments. The only problem remaining is to use this compressed representation to do matrix-vector multiplication. Suppose we want to compute $y=P\cdot x$ for some $x\in\mathbb{R}^n$. It is equivalent to the following procedure:

(1) Initialize y to be an all-zero vector.
(2) For each column i and for each segment $[l,r]$ in column i, increase all $y_l,y_{l+1},\cdots,y_r$ by x_i.

We can reduce the above procedure to the following one:

(1) Initialize z to be an all-zero vector.
(2) For each column i and for each segment $[l,r]$ in column i, increase z_l by x_i and increase z_{r+1} by $-x_i$.
(3) Compute $y_j\leftarrow\sum_{k=1}^{j}z_k$.

In the above procedure, we only need to compute a prefix sum for z. Since each column of P has at most poly($\log n$) segments, the total number of segments involved is at most $\mathrm{nnz}(x)\cdot\mathrm{poly}(\log n)$. The total running time is $\widetilde{O}(\mathrm{nnz}(x)\cdot\mathrm{poly}(\log n))$. Notice that even though y has a large dimension, it can be decomposed into $\widetilde{O}(\mathrm{nnz}(x)\cdot\mathrm{poly}(\log n))$ segments where the entries of each segment have the same value. Thus, we just store the beginning and the ending positions of each segment of y.

Each step of computing the compressed representation can be implemented in poly($\log n$) parallel time and each step of computing the matrix-vector multiplication can also be implemented in poly($\log n$) parallel time. We obtained a desired preconditioner. By plugging this preconditioner into Sherman's framework, we can obtain a parallel $(1+\epsilon)$-approximate uncapacitated minimum cost flow algorithm with poly($\log n$) depth and $\epsilon^{-2}m\cdot\mathrm{poly}(\log n)$ work.

Parallel $(1+\epsilon)$-approximate $s-t$ shortest path. $s-t$ Shortest path is closely related to uncapacitated minimum cost flow. If we set demand $b_s=1, b_t=-1$ and $b_v=0$ for $v\neq s,t\in V$, then the optimal cost of the flow is exactly $\mathrm{dist}_G(s,t)$. Thus, computing a $(1+\epsilon)$-approximation to $\mathrm{dist}_G(s,t)$ can be achieved by our flow algorithm. However, the flow algorithm can only output a flow but not a path. We need more effort to find a path from s to t with length at most $(1+\epsilon)\cdot\mathrm{dist}_G(s,t)$. As mentioned by [8], if the $(1+\epsilon)$-approximate flow does not contain any cycles, then for each vertex $v\neq t$ we can choose an out edge with probability proportional to the magnitude of its out flow, and the expected length of the path found from s to t is exactly the cost of the flow which is $(1+\epsilon)\cdot\mathrm{dist}_G(s,t)$. Unfortunately, the flow outputted by our flow algorithm may create cycles. If we randomly choose an out edge for each vertex $v\neq t$ with probability proportional to the magnitude of the out flow, we may stuck in some cycle and may not find a path from s to t. To handle cycles, we propose the following procedure to find a path from s to t.

(1) If the graph only has constant number of vertices, find the shortest path from s to t directly.
(2) Otherwise, compute the $(1+\epsilon')$-approximate minimum cost flow from s to t for $\epsilon'=\Theta(\epsilon/\log n)$.
(3) For each vertex except t, choose an out edge with probability proportional to its out flow.
(4) Consider the graph with $n-1$ chosen edges. Each connected component in the graph is either a tree or a tree plus an edge. A component is a tree if and only if t is in the component. For each component, we compute a spanning tree. If the component contains t, we set t as the root of the spanning tree. Otherwise, we set an arbitrary end point of the non-tree edge as the root of the spanning tree.
(5) Construct a new graph of which vertices are roots of spanning trees. For each edge $\{u,v\}$ in the original graph, we add an edge connecting the root of u and the root of v with weight

(distance from u to the root of u on the spanning tree) + $w(u,v)$
+(distance from v to the root of u on the spanning tree).

(6) Recursively find a $(1+\epsilon')$-approximate shortest path from the root of s to t in the new graph. Recover a path in the original graph from the path in the new graph.

In the above procedure, only $1/2$ vertices can be root vertices. Thus, the procedure can recurse at most $\log n$ times which implies that

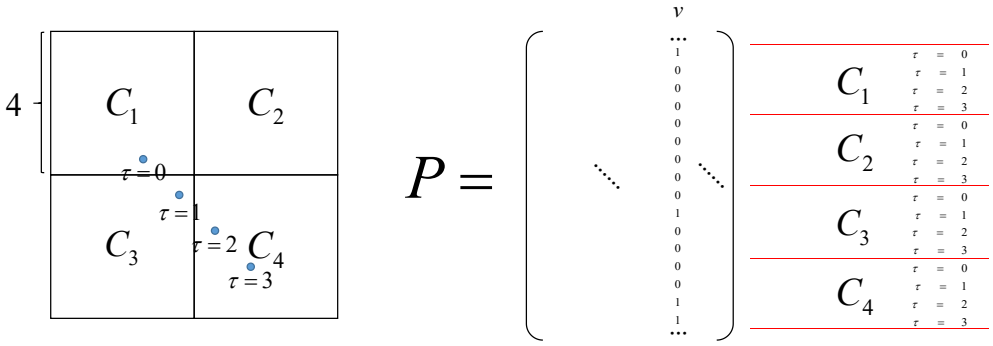

Figure 3: Consider cells C_1, C_2, C_3, C_4 shown above with side length 4. Blue dots denote the positions of $\varphi(v) + \tau \cdot \mathbf{1}_d$ for some vertex v and $\tau = 0, 1, 2, 3$. The entries of P in the column corresponding to v and in the rows corresponding to (C, τ) for $C = C_1, C_2, C_3, C_4$ and $\tau = 0, 1, 2, 3$ are shown on the right.

the parallel time of the algorithm is at most poly$(\log n)$ and the total work is still $\sim m$ poly(n). Now analyze the correctness. It is easy to see that each edge in the new graph corresponds to a path between two root vertices in the original graph. Thus a path from the root of s to t in the new graph corresponds to an $s - t$ path in the original graph. We only need to show that the distance between the root of s and t in the new graph can not be much larger than the distance between s and t in the original graph. To prove this, we show that if we do a random walk starting from s and for each step we choose the next vertex with probability proportional to the out flow, the expected length of the random walk to reach t is exactly the cost of the flow. By coupling argument, we can prove that the expected length of the distance between the root of s and t in the new graph is at most $(1 + O(\epsilon')) \cdot$ (the cost of the flow). Thus, the expected length of the final $s - t$ path is at most $(1 + O(\epsilon'))^{\log n} \cdot \text{dist}_G(s, t) \leq (1 + \epsilon) \cdot \text{dist}_G(s, t)$.

1.3 A Roadmap

We introduce notation and preliminaries in Section 2. We describe the construction of low hop emulators in Section 3. We describe our uncapacitated minimum cost flow algorithm in sequential setting in Section 4. For parallel implementations, applications of low hop emulators, parallel recursive path construction algorithm and all missing proofs, we refer reader to the full version[4].

2 PRELIMINARIES

Let $[n]$ denote the set $\{1, 2, \cdots, n\}$. For a set V, 2^V denotes the family of all the subsets of V, i.e., $2^V = \{S \mid S \subseteq V\}$. In this paper, we will only consider graphs with non-negative weights. Let $G = (V, E, w)$ be a connected undirected weighted graph with vertex set V, edge set E, and weights of the edges $w : E \to \mathbb{Z}_{\geq 0}$. Let both $\{u, v\}, \{v, u\}$ denote an undirected edge between u and v. For each edge $e = \{u, v\} \in E$, let both $w(u, v), w(v, u)$ denote $w(e)$. For $v \in V$, let $w(v, v)$ be 0. Consider a tuple $p = (u_0, u_1, u_2, \cdots, u_h) \in V^{h+1}$. If $\forall i \in [h]$, either $u_i = u_{i-1}$ or $\{u_{i-1}, u_i\} \in E$, then p is a path between u_0 and u_h. The number of hops of p is h, and the length of p is defined as $w(p) = \sum_{i=1}^{h} w(u_{i-1}, u_i)$. For $u, v \in V$, let $\text{dist}_G(u, v)$ denote the length of the shortest path between u, v, i.e., $\text{dist}_G(u, v) = w(p^*)$, where the path p^* between u, v satisfies that $\forall$path p between u, v, $w(p^*) \leq w(p)$. Similarly, $\text{dist}_G^{(h)}(u, v)$ denotes the h-hop distance between u, v, i.e., $\text{dist}_G^{(h)}(u, v) = w(p')$, where the h-hop path p' between u, v satisfies that $\forall h$-hop path p between u, v, $w(p') \leq w(p)$. The diameter $\text{diam}(G)$ of G is defined as $\max_{u,v \in V} \text{dist}_G(u, v)$. The hop diameter of G is defined as the minimum value of $h \in \mathbb{Z}_{\geq 0}$ such that $\forall u, v \in V, \text{dist}_G(u, v) = \text{dist}_G^{(h)}(u, v)$. For $S \subseteq V, v \in V$, we define $\text{dist}_G(v, S) = \text{dist}_G(S, v) = \min_{u \in S} \text{dist}(u, v)$. Similarly, we define $\text{dist}_G^{(h)}(v, S) = \text{dist}_G^{(h)}(S, v) = \min_{u \in S} \text{dist}_G^{(h)}(u, v)$. If G is clear in the context, we use $\text{dist}(\cdot, \cdot)$ and $\text{dist}^{(h)}(\cdot, \cdot)$ for short.

Consider two weighted graphs $G = (V, E, w)$ and $G' = (V, E', w')$. If $\forall u, v \in V, \text{dist}_G(u, v) \leq \text{dist}_{G'}(u, v) \leq \alpha \cdot \text{dist}_G(u, v)$ for some $\alpha \geq 1$, then G' is called an α-emulator of G.

Given $r \in \mathbb{Z}_{\geq 0}$, for $v \in V$, we define $\text{Ball}_G(v, r) = \{u \in V \mid \text{dist}_G(u, v) \leq r\}$, and $\text{Ball}_G^\circ(v, r) = \{u \in V \mid \text{dist}_G(u, v) < r\}$. Given $b \in [|V|]$, for $v \in V$, let $r_{G,b}(v)$ satisfy that $|\text{Ball}_G(v, r_{G,b}(v))| \geq b$ and $|\text{Ball}_G^\circ(v, r_{G,b}(v))| < b$. We define $\text{Ball}_{G,b}(v) = \text{Ball}_G(v, r_{G,b}(v))$, and $\text{Ball}_{G,b}^\circ(v) = \text{Ball}_G^\circ(v, r_{G,b}(v))$. If there is no ambiguity, we just use $\text{Ball}(v, r)$, $\text{Ball}^\circ(v, r)$, $r_b(v)$, $\text{Ball}_b(v)$, $\text{Ball}_b^\circ(v)$ to denote $\text{Ball}_G(v, r)$, $\text{Ball}_G^\circ(v, r)$, $r_{G,b}(v)$, $\text{Ball}_{G,b}(v)$, $\text{Ball}_{G,b}^\circ(v)$ respectively for short.

For a vector $x \in \mathbb{R}^m$ we use $\|x\|_1$ to denote the ℓ_1 norm of x, i.e., $\|x\|_1 = \sum_{i=1}^m |x_i|$. We use $\|x\|_\infty$ to denote the ℓ_∞ norm of x, i.e., $\|x\|_\infty = \max_{i \in [m]} |x_i|$. Given a matrix $A \in \mathbb{R}^{n \times m}$, we use A_i, A^j and $A_{j,i}$ to denote the i-th column, the j-th row and the entry in the i-th column and the j-th row of A respectively. We use $\|A\|_{1 \to 1}$ to denote the operator ℓ_1 norm of A, i.e., $\|A\|_{1 \to 1} = \sup_{x: x \neq 0} \frac{\|Ax\|_1}{\|x\|_1}$. A well-known fact is that $\|A\|_{1 \to 1} = \max_{i \in [m]} \|A_i\|_1$. We use $\mathbf{1}_n$ to denote an n dimensional all-one vector. We use $\text{sgn}(a)$ to denote the sign of a, i.e., $\text{sgn}(a) = 1$ if $a \geq 0$, and $\text{sgn}(a) = -1$ otherwise. We use $\text{nnz}(\cdot)$ to denote the number of non-zero entries of a matrix or a vector.

3 LOW HOP EMULATOR

Given a weighted undirected graph G, we give a new construction of the graph emulator of G. For any two vertices in our constructed emulator, there is always a shortest path with small number of hops. Furthermore, our construction can be implemented in parallel efficiently.

[4] https://arxiv.org/pdf/1911.01956.pdf

3.1 Subemulator

In this section, we introduce a new concept which we called *subemulator*.
an emulator with low hop diameter.

Definition 3.1 (Subemulator). *Consider two connected undirected weighted graphs $G = (V, E, w)$ and $H = (V', E', w')$. For $b \in [|V|]$ and $\alpha \geq 1$, if H satisfies*

(1) $V' \subseteq V$,
(2) $\forall v \in V, \text{Ball}_{G,b}(v) \cap V' \neq \emptyset$,
(3) $\forall u, v \in V', \text{dist}_G(u, v) \leq \text{dist}_H(u, v) \leq \alpha \cdot \text{dist}_G(u, v)$,

then H is an (α, b)-subemulator of G. Furthermore, if there is a mapping $q : V \to V'$ which satisfies $\forall v \in V, q(v) \in \text{Ball}_{G,b}(v)$ and

$$\forall u, v \in V, \text{dist}_H(q(u), q(v)) \leq \text{dist}_G(u, q(u)) + \text{dist}_G(v, q(v)) + \beta \cdot \text{dist}_G(u, v)$$

for some $\beta \geq 1$, then H is a strong (α, b, β)-subemulator of G, $q(\cdot)$ is called a leader mapping, and $q(v)$ is the leader of v.

In Algorithm 1, we show how to construct a strong subemulator.

Algorithm 1 Construction of the Subemulator

procedure Subemulator($G = (V, E, w)$, $b \in [|V|]$)
  Output: $H = (V', E', w')$, $q : V \to V'$
  $V' \leftarrow$Samples(G, b). ▷ Constructing vertices.
  $H, q \leftarrow$Connects(G, V', b). ▷ Constructing edges and leaders.
  Return H, q.
end procedure
procedure Samples($G = (V, E, w)$, $b \in [|V|]$)
  Initialize $S, V' \leftarrow \emptyset$, $n \leftarrow |V|$
  For $v \in V$, add v into S with probability $\min(50 \log(n)/b, 1/2)$.
  For $v \in V$, if $v \in S$ or $\text{Ball}_{G,b}(v) \cap S = \emptyset$, $V' \leftarrow V' \cup \{v\}$.
  Return V'.
end procedure
procedure Connects($G = (V, E, w)$, $V' \subseteq V$, $b \in [|V|]$)
  Output: $H = (V', E', w')$, $q : V \to V'$
  For $v \in V$, $q(v) \leftarrow \arg\min_{u \in \text{Ball}_{G,b}(v) \cap V'} \text{dist}_G(u, v)$.
  Initialize $E' = \emptyset$.
  For $\{u, v\} \in E$, $E' \leftarrow E' \cup \{q(u), q(v)\}$.
  For $v \in V$, $u \in \text{Ball}^{\circ}_{G,b}(v)$, $E' \leftarrow E' \cup \{q(u), q(v)\}$.
  For $e' \in E'$, initialize $w'(e') \leftarrow \infty$.
  For $\{u, v\} \in E$, consider $e' = \{q(u), q(v)\} \in E'$,
  $w'(e') \leftarrow \min(w'(e'), \text{dist}_G(q(u), u) + w(u, v) + \text{dist}_G(v, q(v)))$.
  For $v \in V$, $u \in \text{Ball}^{\circ}_{G,b}(v)$, consider $e' = \{q(u), q(v)\}$,
  $w'(e') \leftarrow \min(w'(e'), \text{dist}_G(q(u), u) + \text{dist}_G(u, v) + \text{dist}_G(v, q(v)))$.
  Return $H = (V', E', w')$ and $q : V \to V'$.
end procedure

Theorem 3.2 (Construction of the subemulator). *Consider a connected n-vertex m-edge undirected weighted graph $G = (V, E, w)$ and a parameter $b \in [n]$. Subemulator(G, b) (Algorithm 1) will output an undirected weighted graph $H = (V', E', w')$ and $q : V \to V'$ such that H is a strong $(8, b, 22)$-subemulator of G, and q is a corresponding leader mapping (Definition 3.1). Furthermore, $\mathbf{E}[|V'|] \leq \min(75 \log(n)/b, 3/4)n$, $|E'| \leq m + nb$.*

3.2 A Warm-up Algorithm: Distance Oracle

Given a weighted undirected graph, a distance oracle is a static data structure which uses small space and can be used to efficiently return an approximate distance between any pair of query vertices. In this section, we give a warm-up algorithm which is a direct application of subemulator. In section 3.3, we will show how to apply the preprocessing procedure PreProc (Algorithm 2) in our construction of low hop emulator.

Algorithm 2 Distance Oracle

procedure PreProc($G = (V, E, w)$, k)
  $n \leftarrow |V|$, $m \leftarrow |E|$.
  $t \leftarrow 0$, $H_0 = (V_0, E_0, w_0) \leftarrow G$, $b_0 \leftarrow \max\left(\lceil (75 \log n)^2 \rceil, n^{1/(2k)}\right)$.
  $n_0 \leftarrow |V_0|$, $m_0 \leftarrow |E_0|$
  while $n_t \geq b_t$ **do**
    $H_{t+1} = (V_{t+1}, E_{t+1}, w_{t+1})$, $q_t \leftarrow$Subemulator(H_t, b_t). ▷ See Algorithm 1.
    $\forall v \in V_t$, let $B_t(v) \leftarrow \text{Ball}^{\circ}_{H_t, b_t}(v) \cup \{q_t(v)\}$ and compute and store $\text{dist}_{H_t}(v, u)$ for every $u \in B_t(v)$.
    $n_{t+1} \leftarrow |V_{t+1}|$, $m_{t+1} \leftarrow |E_{t+1}|$.
    $b_{t+1} \leftarrow b_t^{1.25}$.
    $t \leftarrow t + 1$.
  end while
  For $v \in V_t$, $B_t(v) \leftarrow V_t$, compute $\text{dist}_{H_t}(v, u)$ for $u \in V_t$, and $q_t(v) \leftarrow x$ where $x \in V_t$ is smallest.
end procedure
procedure Query(u, v)
  Output: $d \in \mathbb{Z}_{\geq 0}$
  $l \leftarrow 0$, $d_0 \leftarrow 0$, $u_0 \leftarrow u$, $v_0 \leftarrow v$.
  while $v_l \notin B_l(u_l)$ and $u_l \notin B_l(v_l)$ **do**
    $d_l \leftarrow \text{dist}_{H_l}(u_l, q_l(u_l)) + \text{dist}_{H_l}(v_l, q_l(v_l))$.
    $u_{l+1} = q_l(u_l)$, $v_{l+1} = q_l(v_l)$.
    $l \leftarrow l + 1$
  end while
  $d_l \leftarrow \text{dist}_{H_l}(u_l, v_l)$.
  Return $d = \sum_{i=0}^{l} d_i$.
end procedure

Lemma 3.3 (Properties of the preprocessing algorithm). *Given a connected weighted graph $G = (V, E, w)$ with $|V| = n$, $|E| = m$, and a parameter $k \in [0.5, 0.5 \log n]$, let t be the value at the end of PreProc(G, k) (Algorithm 2). For $i > t$, define $n_i = m_i = 0, b_i = b_{i-1}^{1.25}, V_i = \emptyset$. We have following properties:*

(1) $t \leq 4\lceil \log(k) + 1 \rceil$.
(2) *For $i \in \mathbb{Z}_{\geq 0}$,*
  - $\mathbf{E}[n_i] \leq \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_i^2$,
  - $\mathbf{E}[m_i] \leq m + 2 \cdot \max(n^{1+1/(2k)}, n \cdot (75 \log n)^2)$,
  - $\mathbf{E}\left[\sum_{v \in V_i} |B_i(v)|\right] \leq \max(n^{1+1/k}, n \cdot (75 \log n)^4)/b_i$.

Lemma 3.4 (Correctness of the query algorithm). *Given a connected weighted graph $G = (V, E, w)$ with $|V| = n$, $|E| = m$, and a parameter $k \in [0.5, 0.5 \log n]$, run preprocessing PreProc(G, k) (Algorithm 2). Then $\forall u, v \in V$, the output d of Query(u, v) (Algorithm 2) satisfies $\text{dist}_G(u, v) \leq d \leq 26^{4\lceil \log(k)+1 \rceil} \text{dist}_G(u, v)$. The running time of Query(u, v) is $O(\log(4k))$.*

3.3 Low Hop Emulator

In this section, we construct an emulator graph such that the distance is approximately preserved and there always exists a low hop shortest path between any pair of vertices in the emulator.

Algorithm 3 Low Hop Emulator

procedure LowHopDimEmulator($G = (V, E, w)$, k)
  Output: $G' = (V', E', w')$
  Run the processing procedure PreProc(G, k), and let t be the value at the end of the procedure. $\forall i \in \{0, 1, \cdots, t\}$, let $H_i = (V_i, E_i, w_i)$, $q_i : V_i \to V_{i+1}$, $B_i : V_i \to 2^{V_i}$, b_i be computed by the such procedure. ▷ See Algorithm 2.
  Initialize $E' \leftarrow \emptyset$.
  For $i \in \{0, 1, \cdots, t-1\}$, for each $v \in V_i$, $E' \leftarrow E' \cup \{v, u\}$, where $u = q_i(v)$.
  For $i \in \{0, 1, \cdots, t\}$, for each $v \in V_i$, for each $u \in B_i(v)$, $E' \leftarrow E' \cup \{u, v\}$.
  For each $e' \in E'$, initialize $w'(e') \leftarrow \infty$.
  For $i \in \{0, 1, \cdots, t-1\}$, for each $v \in V_i$, consider $e' = \{v, u\}$ where $u = q_i(v)$. Let $w'(e') \leftarrow \min(w'(e'), 27^{t-i-1} \cdot \text{dist}_{H_i}(u, v))$.
  For $i \in \{0, 1, \cdots, t\}$, for each $v \in V_i$, for each $u \in B_i(v)$, consider $e' = \{u, v\}$. Let $w'(e') \leftarrow \min(w'(e'), 27^{t-i} \cdot \text{dist}_{H_i}(u, v))$.
  Output $G' = (V, E', w')$.
end procedure

Theorem 3.5. *Consider an n-vertex m-edge connected undirected weighted graph $G = (V, E, w)$ and $k \in [0.5, 0.5 \log n]$. Let $G' = (V, E', w')$ be the output of LowHopDimEmulator(G, k) (Algorithm 3). Then, $\mathbf{E}[|E'|] \le O(n^{1+1/(2k)} + n \log^2 n)$ and $\forall u, v \in V$,*

$$\text{dist}_G(u, v) \le \text{dist}_{G'}(u, v) \le 27^{4\lceil \log(k)+1 \rceil} \cdot \text{dist}_G(u, v).$$

Furthermore, $\forall u, v \in V$, $\text{dist}_{G'}(u, v) = \text{dist}_{G'}^{(16\lceil \log(k)+1 \rceil)}(u, v)$.

4 UNCAPACITATED MINIMUM COST FLOW

Given an undirected graph $G = (V, E, w)$ with $|V| = n$ vertices and $|E| = m$ edges, the vertex-edge incidence matrix $A \in \mathbb{R}^{n \times m}$ is defined as the following: $\forall i \in [n], j \in [m]$,

$$A_{i,j} = \begin{cases} 1 & \{i, v\} \in E \text{ is the } j\text{-th edge of } G \text{ and } i < v, \\ -1 & \{i, v\} \in E \text{ is the } j\text{-th edge of } G \text{ and } i > v, \\ 0 & \text{Otherwise.} \end{cases}$$

The weight matrix $W \in \mathbb{R}^{m \times m}$ is a diagonal matrix. The i-th diagonal entry of W is $w(e)$, where $e \in E$ is the i-th edge. Given a demand vector $b \in \mathbb{R}^n$ with $\mathbf{1}_n^\top b = 0$, i.e., $\sum_{i=1}^n b_i = 0$, the uncapacitated minimum cost flow (transshipment) problem is to solve the following problem: $\min_{f \in \mathbb{R}^m} \|Wf\|_1, s.t.\ Af = b$.

If b only has two non-zero entries $b_i = 1$ and $b_j = -1$, then the optimal cost is the length of the shortest path between vertex i and vertex j. Without loss of generality, we can suppose that each edge has positive weight. Otherwise, we can contract the edges with weight 0, and the contraction will not affect the value of the solution. Let $x = Wf$, then the problem becomes

$$\min_{x \in \mathbb{R}^m} \|x\|_1 \quad s.t.\ AW^{-1}x = b. \tag{4}$$

In this section, we will focus on finding a $(1+\epsilon)$-approximation to problem (4).

4.1 Sherman's Framework

Before we present our algorithm, let us review Sherman's algorithm [54], and completely open his black box.

Definition 4.1 (ℓ_1 Non-linear condition number). *Given a matrix $B \in \mathbb{R}^{r \times m}$, the ℓ_1 non-linear condition number of B is defined as $\kappa(B) = \inf_S \|B\|_{1\to 1} \cdot \sup_{x \in \mathbb{R}^m : Bx \neq 0} \frac{\|S(Bx)\|_1}{\|Bx\|_1}$, where the range of $S : \mathbb{R}^r \to \mathbb{R}^m$ is over all maps such that $\forall x \in \mathbb{R}^m$, $B \cdot S(Bx) = Bx$.*

It is easy to show that an equivalent definition of $\kappa(B)$ is $\|B\|_{1\to 1} \cdot \max_{g \in \{y \in \mathbb{R}^r \mid y = Bx, x \in \mathbb{R}^m\} \setminus \{0\}} \min_{x : Bx = g} \frac{\|x\|_1}{\|g\|_1}$.

Definition 4.2 ((α, β)-Solution). *Given a matrix $B \in \mathbb{R}^{r \times m}$ and a vector $g \in \{y \in \mathbb{R}^r \mid y = Bx, x \in \mathbb{R}^m\}$, let $x^* = \arg\min_{x : Bx = g} \|x\|_1$. If $\|x\|_1 \le \alpha \|x^*\|_1$ and $\|Bx - g\|_1 \le \beta \|B\|_{1\to 1} \|x^*\|_1$, then x is called an (α, β)-solution with respect to (B, g). Given a matrix $B \in \mathbb{R}^{r \times m}$, if an algorithm can output an (α, β)-solution with respect to (B, g) for any vector $g \in \{y \in \mathbb{R}^r \mid y = Bx, x \in \mathbb{R}^m\}$, then the algorithm is called an (α, β)-solver for B.*

Definition 4.3 (Composition of the solvers). *Suppose F_1 is an (α_1, β_1)-solver for $B \in \mathbb{R}^{r \times m}$ and F_2 is an (α_2, β_2)-solver for B. For any input vector $g \in \{y \in \mathbb{R}^r \mid y = Bx, x \in \mathbb{R}^m\}$, the composition $F_2 \circ F_1$ firstly runs F_1 to obtain an (α_1, β_1)-solution $x \in \mathbb{R}^m$ with respect to (B, g), then runs F_2 to obtain an (α_2, β_2)-solution $x' \in \mathbb{R}^m$ with respect to $(B, g - Bx)$, and finally outputs $x + x'$.*

Lemma 4.4 ([54]). *Suppose F_1 is an $(\alpha_1, \beta_1/\kappa)$-solver for $B \in \mathbb{R}^{r \times m}$ and F_2 is an $(\alpha_2, \beta_2/\kappa)$-solver for B, where κ is the ℓ_1 non-linear condition number of B, i.e., $\kappa = \kappa(B)$. Then $F_2 \circ F_1$ is an $(\alpha_1 + \alpha_2\beta_1, \beta_1\beta_2/\kappa)$-solver for B.*

Corollary 4.5 ([54]). *Let $\epsilon \in (0, 0.5)$. Suppose F is an $(1+\epsilon, \epsilon/\kappa)$ solver for $B \in \mathbb{R}^{r \times m}$, where κ is the ℓ_1 non-linear condition number of B, i.e., $\kappa = \kappa(B)$. Define $F^1 = F$, and $F^t = F^{t-1} \circ F$. Then F^t is an $(1 + 4\epsilon, \epsilon^t/\kappa)$ solver.*

Corollary 4.6 ([54]). *Let $\epsilon \in (0, 0.5)$, $t, M \in \mathbb{R}_{\ge 0}$. Suppose F_1 is an $(1 + 4\epsilon, \epsilon^t/\kappa)$-solver for $B \in \mathbb{R}^{r \times m}$, and F_2 is an $(M, 0)$-solver for B, where $\kappa = \kappa(B)$. Then $F_2 \circ F_1$ is an $(1 + 4\epsilon + M\epsilon^t, 0)$-solver for B.*

Let us come back to the minimum cost flow problem, problem (4). One observation is that if a matrix $P \in \mathbb{R}^{r \times m}$ has full column rank, then $PAW^{-1}x = Pb \Leftrightarrow AW^{-1}x = b$. So, instead of solving Equation (4) directly, we can design a matrix $P \in \mathbb{R}^{r \times m}$ with full column rank, and try to solve

$$\min_{x \in \mathbb{R}^m} \|x\|_1 \quad s.t.\ PAW^{-1}x = Pb. \tag{5}$$

Notice that since P has full column rank, problem (5) is exactly the same as problem (4). Although an $(\alpha, 0)$-solver for PAW^{-1} is also an $(\alpha, 0)$-solver for AW^{-1}, an (α, β)-solver for PAW^{-1} may not be an (α, β)-solver for AW^{-1} for $\beta > 0$. As shown in [54], if $\kappa(PAW^{-1})$ is smaller, then it is much easier to design a $(1+\epsilon, \epsilon/\kappa(PAW^{-1}))$-solver for PAW^{-1}. If $\kappa(PAW^{-1})$ is small, then we say P is a good preconditioner for AW^{-1}. Before we discuss how to construct P, let us assume $\kappa(PAW^{-1}) \le \kappa$, and review how to solve problem (5).

According to [54], there is a simple $(n, 0)$-solver to problem (5).

Lemma 4.7 ([54]). *Given a connected undirected weighted graph $G = (V, E, w)$, let $A \in \mathbb{R}^{n\times m}$ be the corresponding vertex-edge incidence matrix, and let $W \in \mathbb{R}^{m\times m}$ be the corresponding diagonal weight matrix, where $n = |V|, m = |E|$. There is an algorithm MSTRouting, such that for any demand vector $b \in \mathbb{R}^n$ with $\mathbf{1}_n^\top b = 0$, the output $f \in \mathbb{R}^m$ of MSTRouting(G, b) satisfies $Af = b$ and $\|Wf\|_1 \le n \cdot \min_{f':Af'=b} \|Wf'\|_1$. MSTRouting$(G, b)$ takes running time $\widetilde{O}(m)$.*

By above lemma, if we set $x = Wf$, we have $PAW^{-1}x = Pb$, and $\|x\|_1 \le n \cdot \min_{x':PAW^{-1}x'=Pb} \|x'\|_1$. Thus, x is an $(n, 0)$-solution to problem (5). Suppose $\epsilon < 0.5$. By Corollary 4.6, if we have a $(1+4\epsilon, \epsilon^{1+\log n}/\kappa)$ solver for PAW^{-1}, then together with Lemma 4.7, we can obtain a $(1+5\epsilon, 0)$-solver for PAW^{-1}, and thus we can finally find a $(1+5\epsilon)$ approximation to problem (4). If we have a $(1+\epsilon, \epsilon/\kappa)$-solver for PAW^{-1}, then according to Corollary 4.5, we can apply $(1+\epsilon, \epsilon/\kappa)$-solver $1+\log n$ times to obtain a $(1+4\epsilon, \epsilon^{1+\log n}/\kappa)$-solver. It suffices to design a $(1+\epsilon, \epsilon/\kappa)$ solver for PAW^{-1}.

4.1.1 A $(1+\epsilon, \epsilon/\kappa)$-Solver. In this section, we will have a detailed discussion of how [39, 54] used multiplicative weights update algorithm [5] to find a $(1+\epsilon, \epsilon/\kappa)$-solution with respect to (PAW^{-1}, Pb), where $\kappa \ge \kappa(PAW^{-1})$ is an upper bound of the condition number (see Definition 4.1) of PAW^{-1}, and $\epsilon \in (0, 0.5)$ is an arbitrary real number.

Let $x^* = \arg\min_{x:PAW^{-1}x=Pb} \|x\|_1$. It is not hard to show

$$\frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to1}} \le \|x^*\|_1 \le \kappa \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to1}}.$$

Then, we can reduce the optimization problem to a feasibility problem. We want to binary search $s \in \{1, 1+\epsilon, (1+\epsilon)^2, \cdots, (1+\epsilon)^{\lceil \log_{1+\epsilon} \kappa \rceil}\}$, and want to find s such that $s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to1}} \le (1+\epsilon)\|x^*\|_1$ and find $x \in \mathbb{R}^m$ which satisfies $\|x\|_1 \le s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to1}}$ and $\|PAW^{-1}x - Pb\|_1 \le \frac{\epsilon}{2\kappa} \cdot \|PAW^{-1}\|_{1\to1} \cdot s \cdot \frac{\|Pb\|_1}{\|PAW^{-1}\|_{1\to1}}$. The binary search will takes $O(\log(\log_{1+\epsilon} \kappa))$ rounds.

Let $x' = x \cdot \frac{\|PAW^{-1}\|_{1\to1}}{\|Pb\|_1} \cdot \frac{1}{s}$. Then the problem becomes the following feasibility problem: given $s \ge 1$, either find $x' \in \mathbb{R}^m$ such that

$$\|x'\|_1 \le 1 \quad \text{and} \quad \left\| \frac{PAW^{-1}}{\|PAW^{-1}\|_{1\to1}} x' - \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \right\|_1 \le \frac{\epsilon}{2\kappa} \tag{6}$$

or find a certificate such that

$$\|x'\|_1 \le 1 \quad \text{and} \quad \frac{PAW^{-1}}{\|PAW^{-1}\|_{1\to1}} x' = \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} \tag{7}$$

is not feasible. Next, we show how to use multiplicative weights update algorithm [5, 39, 54] to solve problem (6)-(7).

Lemma 4.8 ([39, 54]). *Consider $P \in \mathbb{R}^{r\times n}, A \in \mathbb{R}^{n\times m}, W \in \mathbb{R}^{m\times m}, b \in \mathbb{R}^n, s \ge 1, \epsilon \in (0, 0.5), \kappa \ge 1$. MWU$(P, A, W, b, s, \epsilon, \kappa)$ (Algorithm 4) takes $T = O\left(\frac{\kappa^2}{\epsilon^2} \log m\right)$ iterations. The output $x' \in \mathbb{R}^m$ satisfies Equation (6) if MWU$(P, A, W, b, s, \epsilon, \kappa)$ does not return FAIL. Otherwise, $\bar{y} = \frac{1}{T}\sum_{t=1}^T y_t$ is a certificate that Equation (7) is not feasible. In particular,*

$$\forall j \in [m], \quad \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} < \frac{\bar{y}^\top (PAW^{-1})_j}{\|PAW^{-1}\|_{1\to1}}, \quad \frac{1}{s} \cdot \frac{Pb}{\|Pb\|_1} < -\frac{\bar{y}^\top (PAW^{-1})_j}{\|PAW^{-1}\|_{1\to1}}.$$

Algorithm 4 Solving the Feasibility Problem

procedure MWU$(P \in \mathbb{R}^{r\times n}, A \in \mathbb{R}^{n\times m}, W \in \mathbb{R}^{m\times m}, b \in \mathbb{R}^n, s \ge 1, \epsilon \in (0, 0.5), \kappa \ge 1)$
  Output: $x' \in \mathbb{R}^m$
  Initialize weights: $\forall i \in [m], \psi_1^+(i) \leftarrow 1, \psi_1^-(i) \leftarrow 1$.
  Initialize $T \leftarrow \frac{64\kappa^2 \ln(2m)}{\epsilon^2}$, $\eta \leftarrow \frac{\epsilon}{8\kappa}$, $B \in \mathbb{R}^{n\times 2m}$:
  $B \leftarrow \left(\frac{AW^{-1}}{\|PAW^{-1}\|_{1\to1}} - \frac{1}{s} \cdot \frac{b\cdot \mathbf{1}_m^\top}{\|Pb\|_1} \quad -\frac{AW^{-1}}{\|PAW^{-1}\|_{1\to1}} - \frac{1}{s} \cdot \frac{b\cdot \mathbf{1}_m^\top}{\|Pb\|_1} \right)$.
  for $t = 1 \to T$ **do**
    $\Psi_t \leftarrow \sum_{i=1}^m \psi_t^+(i) + \sum_{i=1}^m \psi_t^-(i)$.
    For $i \in [m]$, $p_t^+(i) \leftarrow \psi_t^+(i)/\Psi_t$, $p_t^-(i) \leftarrow \psi_t^-(i)/\Psi_t$.
    Set $p_t \in \mathbb{R}^{2m}$ s.t. $\forall i \in [m]$, the i-th entry of p_t is $p_t^+(i)$, and the $(i+m)$-th entry of p_t is $p_t^-(i)$.
    If $\|PBp_t\|_1 \le \frac{\epsilon}{2\kappa}$, return $x' \in \mathbb{R}^m$ such that $\forall i \in [m], x'_i = p_t^+(i) - p_t^-(i)$.
    Otherwise, set $y_t \in \{+1, -1\}^r$ such that $\forall i \in [r], (y_t)_i = \operatorname{sgn}\left((PBp_t)_i\right)$.
    For $i \in [m]$, $\phi_t^+(i) \leftarrow y_t^\top PB_i/2$, $\phi_t^-(i) \leftarrow y_t^\top PB_{i+m}/2$.
    For $i \in [m]$, $\psi_{t+1}^+(i) \leftarrow \psi_t^+(i) \cdot \left(1 - \eta\phi_t^+(i)\right)$, $\psi_{t+1}^-(i) \leftarrow \psi_t^-(i) \cdot \left(1 - \eta\phi_t^-(i)\right)$.
  end for
  Return FAIL.
end procedure

4.2 Preconditioner Construction

As discussed in the previous section, if we find a good preconditioner such that $\kappa(PAW^{-1})$ is small, we can use a small number of iterations to compute a good solution. Before we describe how to choose a good preconditioner, let us introduce the following lemma.

Lemma 4.9 ([39, 54]). *Given $P \in \mathbb{R}^{r\times n}$ with full column rank, $A \in \mathbb{R}^{n\times m}, W \in \mathbb{R}^{m\times m}$, if $\forall b \in \{y \in \mathbb{R}^n \mid y = AW^{-1}x, x \in \mathbb{R}^m\}$,*

$$\|x^*\|_1 \le \|Pb\|_1 \le \gamma\|x^*\|_1,$$

where $x^* = \arg\min_{x\in\mathbb{R}^m:AW^{-1}x=b} \|x\|_1$, then $\kappa(PAW^{-1}) \le \gamma$.

By above lemma, our goal is to find a linear operator P such that for any demand vector b, $\|Pb\|_1$ can approximate the minimum cost flow with demand vector b very well. Instead of using Sherman's original lattice algorithm, we propose to use randomly shifted grids based algorithm [35].

4.2.1 Embedding Minimum Cost Flow into ℓ_1 via Randomly Shifted Grids. In this section, we review the embedding method of [35] and describe how to construct the preconditioner. Suppose we have a mapping $\varphi : V \to [\Delta]^d$ such that $\forall u, v \in V$,

$$\operatorname{dist}_G(u, v) \le \|\varphi(u) - \varphi(v)\|_1 \le \alpha \cdot \operatorname{dist}_G(u, v).$$

We can reduce estimating the minimum cost flow on G to approximating the cost of the geometric transportation problem. The geometric transportation problem is also called Earth Mover's Distance (EMD) problem. In particular, it is the following minimization problem:

$$\min_{\pi:V\times V\to\mathbb{R}_{\ge0}} \sum_{(u,v)\in V\times V} \pi(u, v) \cdot \|\varphi(u) - \varphi(v)\|_1 \tag{8}$$
$$s.t.\ \forall u \in V, \sum_{v\in V} \pi(u, v) - \sum_{v\in V} \pi(v, u) = b_u.$$

It is obvious that if we can obtain a β-approximation to the optimal cost of (8), we can obtain an $\alpha\beta$-approximation to the cost of original minimum cost flow problem on G.

For a sequential algorithm, the such embedding φ can be obtained by Bourgain's Embedding.

Lemma 4.10 (Bourgain's Embedding [12]). *Given an undirected graph $G = (V, E, w)$ with $|V| = n$ vertices and $|E| = m$ edges, there is a randomized algorithm which can output a mapping $\varphi : V \to [\Delta]^d$ for $d = O(\log^2 n)$ with probability 0.99 in $O(m \log^2 n)$ time, such that*

$$\forall u, v \in V, \mathrm{dist}_G(u, v) \le \|\varphi(u) - \varphi(v)\|_1 \le O(\log n) \cdot \mathrm{dist}_G(u, v),$$

where $\Delta \le \sum_{e \in E} w(e)$.

In the remaining of this section, we focus on approximating (8). Without loss of generality, we suppose Δ is a power of 2. Let $L = 1 + \log \Delta$. We create L levels grids $G_0, G_1, \cdots, G_{L-1}$, where G_i partitions $[2\Delta]^d$ into disjoint cells with side length 2^i. In particular, $\forall i \in \{0, 1 \cdots, L-1\}$, the i-th level grid G_i is:

$$\Big\{ C \;\Big|\; C = \{a_1, \cdots, a_1 + 2^i - 1\} \times \cdots \times \{a_d, \cdots, a_d + 2^i - 1\}, \\ \forall j \in [d], a_j \bmod 2^i = 1, a_j \in [2\Delta] \Big\}.$$

Instead of shifting the gird, we shift the points. For each dimension, we can use the same shift value τ [6]. Let τ be a random variable with uniform distribution over $[\Delta]$. We can construct a vector $h \in \mathbb{R}^{\sum_{i=0}^{L-1} |G_i|}$ with one entry per cell in $G_0 \cup G_1 \cup \cdots \cup G_{L-1}$. Let $h_{(i,C)}$ correspond to the cell $C \in G_i$. For each $i \in \{0, 1, \cdots, L-1\}$ and each cell $C \in G_i$, we set $h_{(i,C)}$ as: $h_{(i,C)} = d \cdot 2^i \cdot \sum_{v \in V : \varphi(v) + \tau \cdot \mathbf{1}_d \in C} b_v$. Let $\mathrm{OPT}_{\mathrm{EMD}}(b)$ denote the optimal solution of the EMD problem (8). As shown by [35], $\|h\|_1$ is a good approximation to $\mathrm{OPT}_{\mathrm{EMD}}(b)$.

Lemma 4.11. *Let $h \in \mathbb{R}^{\sum_{i=0}^{L-1} |G_i|}$ be constructed as above. Then,*

(1) $\mathbf{E}_\tau[\|h\|_1] \le 2Ld \cdot \mathrm{OPT}_{EMD}$,
(2) $\|h\|_1 \ge \mathrm{OPT}_{\mathrm{EMD}}(b)$.

An observation is that since each cell in G_i has side length 2^i, shifting each point by $\tau \cdot \mathbf{1}_d$ is equivalent to shifting each point by $(\tau \bmod 2^i) \cdot \mathbf{1}_d$ for the cells in G_i. Thus, if we modify the construction of h as the following: $\forall i \in \{0, 1, \cdots, L-1\}, C \in G_i$,

$$h_{(i,C)} = d \cdot 2^i \cdot \sum_{v \in V : \varphi(v) + (\tau \bmod 2^i) \cdot \mathbf{1}_d \in C} b_v,$$

Lemma 4.11 still holds. Next, we describe how to construct $h' \in \mathbb{R}^{\sum_{i=0}^{L-1} 2^i |G_i|}$. The entry $h'_{(i,C,\tau)}$ corresponds to the cell $C \in G_i$ and the shift value τ. For each $i \in \{0, 1, \cdots, L-1\}$, each cell $C \in G_i$ and each shift value $\tau \in [2^i]$, we set $h'_{(i,C,\tau)}$ as:

$$h'_{(i,C,\tau)} = \frac{1}{2^i} \cdot d \cdot 2^i \cdot \sum_{v \in V : \varphi(v) + \tau \cdot \mathbf{1}_d \in C} b_v = d \cdot \sum_{v \in V : \varphi(v) + \tau \cdot \mathbf{1}_d \in C} b_v.$$

It is clear that $\|h'\|_1 = \mathbf{E}[\|h\|_1]$. By Lemma 4.11, we have

$$\mathrm{OPT}_{\mathrm{EMD}}(b) \le \|h'\|_1 \le 2Ld \cdot \mathrm{OPT}_{\mathrm{EMD}}(b).$$

Observe that h' can be written as a linear map of b, i.e., $h' = P'b$, where $P' \in \mathbb{R}^{(\sum_{i=0}^{L-1} 2^i |G_i|) \times n}$. Each row of P' is indexed by a tuple (i, C, τ) for $i \in \{0, 1, \cdots, L-1\}, C \in G_i$ and $\tau \in [2^i]$, and each column of P' is indexed by a vertex $v \in V$. For $i \in \{0, 1, \cdots, L-1\}, C \in G_i, \tau \in [2^i], v \in V$,

$$P'_{(i,C,\tau),v} = \begin{cases} d & \varphi(v) + \tau \cdot \mathbf{1}_d \in C, \\ 0 & \text{Otherwise.} \end{cases}$$

Consider $i = 0, \tau = 1, \forall v \in V$, there is a unique cell $C \in G_0$ which contains $\varphi(v) + \mathbf{1}_d$. Thus, P' has full column rank. According to Lemma 4.9, since $\forall b \in \{y \in \mathbb{R}^n \mid y = AW^{-1}x, x \in \mathbb{R}^m\}$,

$$\min_{x \in \mathbb{R}^m : AW^{-1}x = b} \|x\|_1 \le \mathrm{OPT}_{\mathrm{EMD}}(b) \le \|P'b\|_1 \\ \le 2Ld \cdot \mathrm{OPT}_{\mathrm{EMD}}(b) \le 2Ld\alpha \cdot \min_{x \in \mathbb{R}^m : AW^{-1}x = b} \|x\|_1,$$

we have $\kappa(P'AW^{-1}) \le 2Ld\alpha$. However, since the size of P' is too large, we cannot apply P' directly in Algorithm 4, and thus it is unclear how to construct a $(1+\epsilon, \epsilon/\kappa(P'AW^{-1}))$-solver for $P'AW^{-1}$.

4.3 Fast Operations for the Preconditioner

One of our main contributions is to develop several fast operations for P' such that we can implement Algorithm 4 efficiently.

4.3.1 Preconditioner Compression.

Removing useless cells. The first observation is that though P' has a large number of rows, most rows of P' are zero. Thus, we can remove them. Precisely, for each $i \in \{0, 1, \cdots, L-1\}$, let $C_i = \{C \in G_i \mid \exists v \in V, \tau \in [2^i],\ s.t.\ \varphi(v) + \tau \cdot \mathbf{1}_d \in C\}$. Then we can set $P \in \mathbb{R}^{(\sum_{i=0}^{L-1} 2^i |C_i|) \times n}$ such that $\forall i \in \{0, 1, \cdots, L-1\}, C \in C_i, \tau \in [2^i], v \in V$,

$$P_{(i,C,\tau),v} = \begin{cases} d & \varphi(v) + \tau \cdot \mathbf{1}_d \in C, \\ 0 & \text{Otherwise.} \end{cases}$$

Lemma 4.12. *$\forall i \in \{0, 1, \cdots, L-1\}, |C_i| \le n \cdot (d+1)$.*

By Lemma 4.12, we know that P has at most $2\Delta \cdot n(d+1)$ rows.

Compressed representation. Another observation is that, P may have many identical rows. Thus, we want to handle these rows simultaneously. To achieve this goal, we introduce a concept called compressed representation.

Definition 4.13 (Compressed representation of a vector). *Let $I = \{([a_1, b_1], c_1), ([a_2, b_2], c_2), \cdots, ([a_s, b_s], c_s)\}$, where $c_i \in \mathbb{R}$, $[a_i, b_i] \subseteq [1, r]$ for some $r \in \mathbb{Z}_{\ge 1}$, and $\forall i \ne j \in [s], [a_i, b_i] \cap [a_j, c_j] = \emptyset$. Let $x \in \mathbb{R}^r$. If $\forall i \in [s], j \in [a_i, b_i], x_j = c_i$ and $\forall j \in [1, r] \setminus \bigcup_{i \in [s]} [a_i, b_i], x_j = 0$, then I is a compressed representation of x. The size of the compressed representation I is $|I| = s$.*

The compressed representation of x may not be unique.

Definition 4.14 (Compressed representation of a matrix). *Let $I = (I_1, I_2, \cdots, I_n)$. Given a matrix $P \in \mathbb{R}^{r \times n}$, if $\forall i \in [n]$, I_i is a compressed representation of P_i, then I is called a compressed representation of P. Furthermore, the size of the compressed representation I is defined as $\sum_{i=1}^n |I_i|$.*

Lemma 4.15 (Computing a compressed representation of P). *Given an undirected graph $G = (V, E, w)$ with $|V| = n, |E| = m$ and a mapping $\varphi : V \to [\Delta]^d$ for some Δ, d, such that*

$$\forall u, v \in V, \mathrm{dist}_G(u, v) \le \|\varphi(u) - \varphi(v)\|_1 \le \alpha \cdot \mathrm{dist}_G(u, v),$$

the output $I = (I_1, I_2, \cdots, I_n)$ of ImplicitP(φ) (Algorithm 5) is a compressed representation of a matrix P with full column rank and

Algorithm 5 Computing a compressed representation of P

procedure ImplicitP($\varphi : V \to [\Delta]^d$)
  Output: I
  $n \leftarrow |V|$, $L \leftarrow 1 + \log \Delta$, $\forall i \in \{0, 1, \cdots, L-1\}$, $C_i \leftarrow \emptyset$, and create grids $G_0, G_1, \cdots, G_{L-1}$.
  $\forall i \in \{0, 1, \cdots, L-1\}$, $v \in V$, $C_i \leftarrow C_i \cup \{C \in G_i \mid \exists \tau \in [2^i], \varphi(v) + \tau \cdot \mathbf{1}_d \in C\}$.
  for the i-th vertex $v \in V$ **do**
    $I_i \leftarrow \emptyset$.
    for $l \in \{0, 1, \cdots, L-1\}$ **do**
      For each $C \in C_l$ with $\exists \tau \in [2^l]$, $\varphi(v) + \tau \cdot \mathbf{1}_d \in C$, find $\tau_1, \tau_2 \in [2^l]$ such that $\tau_1 = \min_{\tau \in [2^l]: \varphi(v)+\tau \cdot \mathbf{1}_d \in C} \tau$, $\tau_2 = \max_{\tau \in [2^l]: \varphi(v)+\tau \cdot \mathbf{1}_d \in C} \tau$.
      Suppose C is the k-th cell in C_l. $a \leftarrow (k-1)2^l + \sum_{j=0}^{l-1} 2^j |C_j|$. $I_i \leftarrow I_i \cup \{([a + \tau_1, a + \tau_2], d)\}$.
    end for
  end for
  Return $I = (I_1, I_2, \cdots, I_n)$.
end procedure

$\kappa(PAW^{-1}) \le O(\alpha L d)$, *where* $L = 1 + \log \Delta$, $A \in \mathbb{R}^{n \times m}$ *is the vertex-incidence matrix, and* $W \in \mathbb{R}^{m \times m}$ *is the diagonal weight matrix. Furthermore, for* $i \in [n]$, *the size of* I_i *is at most* $(d+1)L$. *The running time of* ImplicitP(φ) *is* $n \cdot \mathrm{poly}(dL \log n)$.

4.3.2 Operations under Compressed Representations. In this section, we introduce how to implement some important operations under compressed representations.

Fact 4.16. *Let* $I = \{([a_1, b_1], c_1), \cdots, ([a_s, b_s], c_s)\}$ *be a compressed representation of a vector* $x \in \mathbb{R}^r$. *Then,* $\|x\|_1 = \sum_{i=1}^{s} (b_i - a_i + 1) \cdot |c_i|$. *Let* $y \in \mathbb{R}^r$ *be a vector satisfying* $\forall i \in [r], y_i = \mathrm{sgn}(x_i)$. *Then* $I' = \{([a_1, b_1], \mathrm{sgn}(c_1)), \cdots, ([a_s, b_s], \mathrm{sgn}(c_s))\}$ *is a compressed representation of* y. *Let* $z = t \cdot x$ *for some* $t \in \mathbb{R}$. $I'' = \{([a_1, b_1], tc_1), \cdots, ([a_s, b_s], tc_s)\}$ *is a compressed representation of* z. *Furthermore,* $\|x\|_1$, I' *and* I'' *can be computed in* $O(s)$ *time.*

Algorithm 6 Compressed Matrix-Vector Multiplication

procedure MatrixVec($I = (I_1, I_2, \cdots, I_n)$, $g \in \mathbb{R}^n$)
  Output: $\widehat{I}$
  $S \leftarrow \emptyset$, $\widehat{I} \leftarrow \emptyset$.
  for $i \in [n] : g_i \neq 0$ **do**
    For each $([a, b], c) \in I_i$, $S \leftarrow S \cup \{(a, cg_i), (b+1, -cg_i)\}$.
  end for
  Sort $S = \{(q_1, z_1), \cdots, (q_k, z_k)\}$ such that $q_1 \le q_2 \le \cdots \le q_k$.
  For each $j \in \{2, 3, \cdots, k\} : q_j > q_{j-1}$, $\widehat{I} \leftarrow \widehat{I} \cup \{([q_{j-1}, q_j - 1], \sum_{t: q_t < q_j} z_t)\}$.
  Return $\widehat{I}$.
end procedure

Lemma 4.17 (Compressed matrix-vector multiplication). *Given a compressed representation* $I = (I_1, I_2, \cdots, I_n)$ *of a matrix* $P \in \mathbb{R}^{r \times n}$ *with* $\forall i \in [n], |I_i| \le s$, *and a vector* $g \in \mathbb{R}^n$, *the output* $\widehat{I}$ *of* MatrixVec(I, g) *(Algorithm 6) is a compressed representation of* Pg. *Furthermore,* $|\widehat{I}| \le 2s \cdot \mathrm{nnz}(g)$, *and the running time is at most* $O(s \,\mathrm{nnz}(g) \cdot \log(s \,\mathrm{nnz}(g)))$.

Lemma 4.18 (Compressed vector-matrix multiplication). *Given a compressed representation* I *of a vector* $y \in \mathbb{R}^r$ *with* $|I| \le s$ *and a compressed representation* $I' = (I_1, I_2, \cdots, I_n)$ *of a matrix* $P \in \mathbb{R}^{r \times n}$ *with* $\forall i \in [n], |I_i| \le s'$, *the output* $g^\top \in \mathbb{R}^n$ *of* VectorMat(I, I') *(Algorithm 7) is* $P^\top y$. *Furthermore, the running time is* $O((s + ns') \log s)$.

Algorithm 7 Compressed Vector-Matrix Multiplication

procedure VectorMat($I, I' = (I_1, I_2, \cdots, I_n)$)
  Output: $g^\top \in \mathbb{R}^n$
  $g \leftarrow (0, 0, \cdots, 0)$.
  Fill I such that $\forall j \in [r]$, $\exists([a, b], c) \in I, j \in [a, b]$.
  Sort $I = \{([a_1, b_1], c_1), ([a_2, b_2], c_2), \cdots, ([a_s, b_s], c_s)\}$ such that $a_1 < a_2 < \cdots < a_s$.
  $\forall j \in [s]$, compute the prefix sum $p_j = \sum_{t=1}^{j} (b_t - a_t + 1) \cdot c_t$.
  for $i \in [n]$, $([a, b], c) \in I_i$ **do**
    Run binary search to find $j_1 \le j_2$ such that $a \in [a_{j_1}, b_{j_1}]$, $b \in [a_{j_2}, b_{j_2}]$.
    If $j_1 = j_2$, $g_i \leftarrow g_i + c \cdot c_{j_1} \cdot (b - a + 1)$.
    If $j_1 < j_2$, $g_i \leftarrow g_i + c \cdot (c_{j_1} \cdot (b_{j_1} - a + 1) + c_{j_2} \cdot (b - a_{j_2} + 1) + (p_{j_2 - 1} - p_{j_1}))$.
  end for
  Return $g^\top$.
end procedure

4.4 Uncapacitated Minimum Cost Flow

By plugging our preconditioner in Algorithm 4, we obtain the uncapacitated minimum cost flow algorithm.

Theorem 4.19. *Given an* $\epsilon \in (0, 0.5)$, *a connected* n*-vertex* m*-edge undirected graph* $G = (V, E, w)$ *with* $w : E \to \mathbb{Z}_{\ge 0}$, *and a demand vector* $b \in \mathbb{R}^n$ *with* $\mathbf{1}_n^\top b = 0$, *there is a randomized algorithm which can output an* $(1 + \epsilon)$*-approximate solution to the uncapacitated minimum cost flow problem in* $\epsilon^{-2} m \cdot (\log n \log \Lambda)^{O(1)}$ *time with probability at least* 0.99, *where* $\Lambda = \sum_{e \in E} w(e)$.

ACKNOWLEDGMENTS

We thank Aaron Bernstein, Yan Gu, Hossein Esfandiari, Jakub Łącki, Vahab Mirrokni and Ruosong Wang for helpful discussions. Part of this work was done while Peilin Zhong was an intern at Google New York.

REFERENCES

[1] Amir Abboud, Greg Bodwin, and Seth Pettie. 2018. A hierarchy of lower bounds for sublinear additive spanners. *SIAM J. Comput.* 47, 6 (2018), 2203–2236.
[2] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. 2014. Parallel algorithms for geometric graph problems. https://doi.org/10.1145/2591796.2591805 Full version at http://arxiv.org/abs/1401.0042.
[3] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. 2018. Parallel graph connectivity in log diameter rounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 674–685.
[4] Alexandr Andoni, Clifford Stein, and Peilin Zhong. 2019. Parallel Approximate Undirected Shortest Paths Via Low Hop Emulators. *arXiv preprint arXiv:1911.01956* (2019).
[5] Sanjeev Arora, Elad Hazan, and Satyen Kale. 2012. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing* 8, 1 (2012), 121–164.
[6] Artūrs Bačkurs and Piotr Indyk. 2014. Better embeddings for planar earth-mover distance over sparse sets. In *Proceedings of the thirtieth annual symposium on Computational geometry*. ACM, 280.
[7] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*. ACM, 273–284.
[8] Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. 2017. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[9] Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. 2018. Brief Announcement: Semi-MapReduce Meets Congested Clique. CoRR, abs/1802.10297, 2018. *arXiv preprint arXiv:1802.10297* (2018).

[10] Aaron Bernstein. 2009. Fully dynamic (2+ ϵ) approximate all-pairs shortest paths with fast query and close to linear update time. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 693–702.

[11] Guy E Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. 2016. Parallel shortest paths using radius stepping. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 443–454.

[12] Jean Bourgain. 1985. On Lipschitz embedding of finite metric spaces in Hilbert space. *Israel Journal of Mathematics* 52, 1-2 (1985), 46–52.

[13] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D Zaroliagis. 1998. A parallel priority queue with constant time operations. *J. Parallel and Distrib. Comput.* 49, 1 (1998), 4–21.

[14] Keren Censor-Hillel, Michal Dory, Janne H. Korhonen, and Dean Leitersdorf. 2019. Fast Approximate Shortest Paths in the Congested Clique. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*. ACM, New York, NY, USA, 74–83. https://doi.org/10.1145/3293611.3331633

[15] Paul Christiano, Jonathan A Kelner, Aleksander Madry, Daniel A Spielman, and Shang-Hua Teng. 2011. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*. ACM, 273–282.

[16] Edith Cohen. 1994. Polylog-time and near-linear work approximation scheme for undirected shortest paths. In *Proceedings of the 26th Annual ACM SIGACT Symposium on Theory of Computing*, Vol. 26. 16–26.

[17] Edith Cohen. 1997. Using selective path-doubling for parallel shortest-path computations. *Journal of Algorithms* 22, 1 (1997), 30–56.

[18] Edith Cohen. 2000. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)* 47, 1 (2000), 132–166.

[19] Michael B Cohen, Aleksander Mądry, Piotr Sankowski, and Adrian Vladu. 2017. Negative-Weight Shortest Paths and Unit Capacity Minimum Cost Flow in Õ (m 10/7 log W) Time*. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 752–771.

[20] Samuel I Daitch and Daniel A Spielman. 2008. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*. ACM, 451–460.

[21] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[22] Michael Dinitz and Yasamin Nazari. 2019. Brief Announcement: Massively Parallel Approximate Distance Sketches. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[23] Michael Elkin and Ofer Neiman. 2016. Hopsets with Constant Hopbound, and Applications to Approximate Shortest Paths. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 128–137.

[24] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. 2004. A tight bound on approximating arbitrary metrics by tree metrics. *J. Comput. System Sci.* 69, 3 (2004), 485–497.

[25] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. 2010. On distributing symmetric streaming computations. *ACM Transactions on Algorithms* 6, 4 (2010). Previously in SODA'08.

[26] Sebastian Forster and Danupon Nanongkai. 2018. A faster distributed single-source shortest paths algorithm. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 686–697.

[27] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)* 34, 3 (1987), 596–615.

[28] Stephan Friedrichs and Christoph Lenzen. 2018. Parallel metric tree embedding based on an algebraic view on moore-bellman-ford. *Journal of the ACM (JACM)* 65, 6 (2018), 43.

[29] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, Searching, and Simulation in the MapReduce Framework.. In *ISAAC*, Vol. 7074. Springer, 374–383.

[30] Thomas Dueholm Hansen, Haim Kaplan, Robert E Tarjan, and Uri Zwick. 2015. Hollow Heaps. In *International Colloquium on Automata, Languages, and Programming*. Springer, 689–700.

[31] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2014. Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*.

[32] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2016. An almost-tight distributed algorithm for computing single-source shortest paths. 2016. STOC.

[33] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2019. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. *SIAM J. Comput.* 0 (2019), STOC16–98.

[34] Shang-En Huang and Seth Pettie. 2019. Thorup–Zwick emulators are universally optimal hopsets. *Inform. Process. Lett.* 142 (2019), 9–13.

[35] Piotr Indyk and Nitin Thaper. 2003. Fast image retrieval via embeddings. In *Workshop on Statistical and Computational Theories of Vision (at ICCV)*.

[36] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 59–72.

[37] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A model of computation for MapReduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 938–948.

[38] Jonathan A Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. 2014. 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*. SIAM, 217–226.

[39] Andrey Boris Khesin, Aleksandar Nikolov, and Dmitry Paramonov. 2019. Preconditioning for the Geometric Transportation Problem. *arXiv preprint arXiv:1902.08384* (2019).

[40] Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. 2015. Distributed computation of large-scale graph problems. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 391–410.

[41] Philip N Klein and Sairam Sairam. 1992. A parallel randomized approximation scheme for shortest paths. In *Proceedings of the 24th Annual ACM SIGACT Symposium on Theory of Computing*, Vol. 92. 750–758.

[42] Philip N Klein and Sairam Subramanian. 1997. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms* 25, 2 (1997), 205–220.

[43] R. Krauthgamer, H. Nguyen, and T. Zondiner. 2014. Preserving Terminal Distances Using Minors. *SIAM Journal on Discrete Mathematics* 28, 1 (2014), 127–141. https://doi.org/10.1137/120888843

[44] Yin Tat Lee and Aaron Sidford. 2014. Path finding methods for linear programming: Solving linear programs in $\widetilde{O}(\sqrt{\text{rank}})$ iterations and faster algorithms for maximum flow. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. IEEE, 424–433.

[45] F Thomson Leighton and Ankur Moitra. 2010. Extensions and limits to vertex sparsification. In *Proceedings of the forty-second ACM symposium on Theory of computing*. ACM, 47–56.

[46] Jason Li. 2020. Faster Parallel Algorithm for Approximate Shortest Path. In *Proceedings of the ACM SIGACT Symposium on Theory of Computing*. First appeared as arXiv:1911.01626.

[47] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. 2005. Minimum-weight spanning tree construction in O (log log n) communication rounds. *SIAM J. Comput.* 35, 1 (2005), 120–131.

[48] Aleksander Madry. 2013. Navigating central path with electrical flows: From flows to matchings, and back. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE, 253–262.

[49] Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. 2015. Improved Parallel Algorithms for Spanners and Hopsets. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM, 192–201.

[50] Gary L Miller, Richard Peng, and Shen Chen Xu. 2013. Parallel graph decompositions using random shifts. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 196–203.

[51] Ankur Moitra. 2009. Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 3–12.

[52] Jonah Sherman. 2013. Nearly maximum flows in nearly linear time. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE, 263–269.

[53] Jonah Sherman. 2017. Area-convexity, l_∞ regularization, and undirected multicommodity flow. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 452–460.

[54] Jonah Sherman. 2017. Generalized preconditioning and undirected minimum-cost flow. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 772–780.

[55] Hanmao Shi and Thomas H Spencer. 1999. Time–work tradeoffs of the single-source shortest paths problem. *Journal of algorithms* 30, 1 (1999), 19–32.

[56] Thomas H Spencer. 1997. Time-work tradeoffs for parallel algorithms. *Journal of the ACM (JACM)* 44, 5 (1997), 742–778.

[57] Mikkel Thorup. 1999. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)* 46, 3 (1999), 362–394.

[58] Mikkel Thorup and Uri Zwick. 2005. Approximate distance oracles. *Journal of the ACM (JACM)* 52, 1 (2005), 1–24.

[59] Mikkel Thorup and Uri Zwick. 2006. Spanners and emulators with sublinear distance errors. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. Society for Industrial and Applied Mathematics, 802–809.

[60] Virginia Vassilevska Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing (STOC)*. ACM, 887–898.

[61] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.