Exponentially Faster Shortest Paths in the Congested Clique∗

Michal Dory  
Technion  
smichald@cs.technion.ac.il

Merav Parter  
Weizmann Institute  
merav.parter@weizmann.ac.il

ABSTRACT

We present improved deterministic algorithms for approximating shortest paths in the Congested Clique model of distributed computing. We obtain poly(log log n)-round algorithms for the following problems in unweighted undirected n-vertex graphs:

• (1 + ε)-approximation of multi-source shortest paths (MSSP) from O(√n) sources.
• (2 + ε, β)-approximation of all pairs shortest paths (APSP).
• (1 + ε, β)-approximation of APSP where β = O((log log n)^2 log log n).

These bounds improve exponentially over the state-of-the-art poly-logarithmic bounds due to [Censor-Hillel et al., PODC19]. It also provides the first nearly-additive bounds for the APSP problem in sub-polynomial time. Our approach is based on distinguishing between short and long distances based on some distance threshold t = O(β) where β = O((log log n)^2 log log n). Handling the long distances is done by devising a new algorithm for computing a sparse (1 + ε, β) emulator with O(n log log n) edges. For the short distances, we provide distance-sensitive variants for the distance tool-kit of [Censor-Hillel et al., PODC19]. By exploiting the fact that this tool-kit should be applied only on local balls of radius t, their round complexities get improved from poly(log n) to poly(log t).

Finally, our deterministic solutions for these problems are based on a derandomization scheme of a novel variant of the hitting set problem, which might be of independent interest.

1 INTRODUCTION

Shortest paths computation is one of the most fundamental graph problems, and as such it has been studied extensively in various computational models. In this work we consider the problem of approximating shortest path distances in the Congested Clique model of distributed computing [24]. This model attracts an increasing amount of attention over the last decade, due to its relation to practical parallel models such as MapReduce [18] and k-machines [21]. The fast evolution of shortest path algorithms in the Congested Clique model [1–4, 6, 12, 13, 19, 23, 26] is characterized by two main milestones.

First Era: Polynomial Complexity for Exact Distances. In their influential work [3], Censor-Hillel et al. presented the first efficient algorithm for fast matrix multiplication in the Congested Clique model, and demonstrated its applicability for exact distance computation. Specifically, they gave an O(n^{0.158})-round algorithm for computing all-pairs shortest paths (APSP) in unweighted undirected graphs. With weighted and directed graphs, they showed an O(n^{0.158})-round algorithm for (1 + o(1))-approximate APSP, and an O(n^{1/3})-round algorithm for exact APSP. In a follow-up work, Le Gall [23] extended this algebraic framework, and improved the complexity of exact APSP to O(n^{9/49}) rounds for weighted directed graphs with constant weights. Izumi and Le Gall showed that using quantum computing, the complexity of APSP can be improved to O(n^{1/4}) for weighted directed graphs [20]. Censor-Hillel, Leitersdorf and Turner [4] showed improved algorithms for sparse matrix multiplication. In particular, their algorithm multiplies two n × n matrices, each with O(n^{3/2}) non-zero entries, within a constant number of rounds. This led to an exact APSP computation in O(D · (m/n)^{1/3}) rounds in unweighted undirected graphs, where D is the diameter of the n-vertex graph and m is the number of edges.

Second Era: Poly-logarithmic Complexity for Approximate Distances. The above mentioned algorithms are all based on matrix multiplication. An alternative natural approach for approximating APSP is obtained by the notion of multiplicative spanners, namely, sparse subgraphs that preserve all distances up to a small multiplicative stretch. Specifically, by computing a sparse k-spanner in the graph, and collecting its edges at each vertex, one immediately gets a k-approximation for the APSP problem. Note, however, that due to the size vs. stretch tradeoff of multiplicative spanners, in poly-logarithmic time, the approximation value k is at least logarithmic. The problem of obtaining O(1) distance approximation in poly-logarithmic time was left widely open until the recent works of [1] and [4].

By using the ingenious method of continuous optimization [27], Becker et al. [1] presented the first poly-logarithmic algorithm for the (1+ε)-approximate single-source shortest paths (SSSP) problem
in weighted undirected graphs. The starting point of their algorithms is based on the $O(\log n)$-approximation provided by spanners. This approximation is then iteratively improved by applying a tailored gradient descent procedure. Their algorithms also have the additional benefit of being implemented in the more stringent Broadcast Congested Clique model\(^1\).

In a very recent work, Censor-Hillel et al. [2] presented poly-logarithmic time solutions for the more general problems of approximate multi-source shortest paths (MSSP) and APSP. In particular, among other results, they showed an $O(\log^2 n/e)$-round algorithm for the $(2+\epsilon)$-approximate APSP problem in unweighted undirected graphs, an $O(\log^2 n/e)$-round algorithm for $(3+\epsilon)$-approximate APSP in weighted undirected graphs; as well as $O(\log^2 n/e)$-round algorithm for the MSSP problem in weighted undirected graphs with respect to $O(\sqrt{n})$ sources. The approach of [2] is based on a clever reduction to sparse matrix multiplication algorithms from [2, 4]. This reduction is not immediate, for the following reason. The basic approach of using matrix multiplications to compute distances is based on starting with the input matrix of the graph $G$ and then iteratively square it $O(\log n)$ times. Even if the original graph is sparse (e.g., with $O(n^{1/2})$ edges), a single multiplication step might already result in a dense matrix. Since one needs to apply $O(\log n)$ multiplication steps, it is required to sparsify the intermediate matrices as well. To overcome this barrier, [2] developed a rich tool-kit for distance computation, that includes hopset constructions, fast computation of the $k$-nearest neighbours and more.

The poly-logarithmic time complexity appears to be a natural barrier in the matrix multiplications-based algorithms, for the following reason. Basically these algorithms work in iterations, where in iteration $i$, vertices learn about paths that have at most $2^i$ edges, hence $\Omega(\log n)$ iterations seem to be required in order to capture shortest paths with a polynomial number of hops. In this paper we ask if this barrier is indeed fundamental.

**Question 1.** Is it possible to provide $(1+\epsilon)$-approximation for SSSP and $(2+\epsilon)$-approximation for APSP in $O(\log n)$ rounds?\(^2\)

Another plausible explanation for viewing the logarithmic time complexity as a natural barrier comes from the following observation. The current state-of-the-art bound of many graph problems in the Congested Clique model is bounded from below by the logarithm of their respective Local time complexities\(^2\). In the more restrictive, yet quite related model of MPC, this limitation was conditionally proven by Ghaffari, Kuhn and Uitto [16]. Since the locality of the shortest path problem is linear, we reach again to the barrier of $O(\log n)$ rounds.

Another interesting question left by [2] is whether is it possible to design efficient algorithms for additive or near-additive approximation for APSP. Here, the goal is to get an estimate $\delta(u,v)$ for the distance between $u$ and $v$ such that $\delta(u,v) \leq (1+\epsilon)d(u,v) + \beta$ for some small $\beta$. Note that such guarantee is closer to $(1+\epsilon)$-approximation for long enough paths, which is much better than a multiplicative $(2+\epsilon)$-approximation, which raises the following question.

**Question 2.** Can we get near-additive approximation for APSP in sub-polynomial time?\(^3\)

We answer both questions in the affirmative by devising a unified tool for computing nearly-additive approximate distances in $poly(\log \log n)$ rounds. This immediately solves Question 2. To address Question 1, our approach is based on a mixture of the two current approaches for distance computation, namely, graph sparsiﬁcation and matrix multiplications. We start by describing our results, and then elaborate on our approach.

### 1.1 Our Contributions

In this work we break the natural logarithmic barrier for distance computation in the Congested Clique model. We present $poly(\log \log n)$ round approximation algorithms for the fundamental problems of MSSP and APSP.

$$((1+\epsilon) \text{ MSSP. Our first contribution is the first sub-logarithmic algorithm for computing distances from up to } O(\sqrt{n}) \text{ sources.})$$

**Theorem 3** $(1+\epsilon)$-MSSP in poly$(\log \log n)$ **Rounds**. For every $n$-vertex unweighted undirected graph $G=(V,E)$, a subset of $O(\sqrt{n})$ sources $S \subset V$ and a constant $\epsilon \in (0,1)$, there is a randomized algorithm in the Congested Clique model that computes $(1+\epsilon)$-approximation for the $S \times V$ distances with high probability within $O((\log \log n)^3)$ rounds.\(^4\)

We remark that even for computing $(1+\epsilon)$-approximate distances from a single source, the fastest previous algorithms require poly-logarithmic time [1, 2]. Similarly to [2], the barrier of handling at most $O(\sqrt{n})$ sources is rooted at the sparse matrix multiplication algorithm of [2, 4]. The latter takes $O(1)$ time as long as the density of the matrices is $O(\sqrt{n})$, for higher values the complexity is polynomial. We remark that the previous poly-logarithmic algorithms work also for weighted graphs. It is an interesting question whether sub-logarithmic algorithms can be obtained also for the weighted case.

$$(2\epsilon)\text{-APSP. Our next contribution is providing a } (2\epsilon)\text{-approximation for unweighted APSP in poly}(\log \log n) \text{ time, improving upon the previous poly}(\log \log n) \text{ algorithm [2].}$$

**Theorem 4** $(2+\epsilon)$-APSP in poly$(\log \log n)$ **Rounds**. For every $n$-vertex unweighted undirected graph $G = (V,E)$, and constant $\epsilon \in (0,1)$, there is a randomized algorithm in the Congested Clique model that computes $(2+\epsilon)$-approximation for APSP w.h.p within $O((\log \log n)^2)$ rounds.

It is worth noting that a $(2+\epsilon)$-approximation is essentially the best we can hope for without improving the complexity of matrix multiplication in the Congested Clique model. As noted in [2], a reduction to matrix multiplication that appears in [8, 22] implies that one cannot compute a $(2-\epsilon)$ approximate APSP in sub-polynomial time, without obtaining sub-polynomial algorithms for matrix multiplication.

\(^1\)In this model, each vertex is limited to send the same message to all other vertices in a given round.

\(^2\)In the Local model, only vertices that are neighbors in the input graph can communicate in a given round. The messages are allowed to be of unbounded size.

\(^3\)The statements in this section are slightly simpler and assume that $\epsilon$ is constant just to simplify the presentation. The exact statements appear in Section 4.

\(^4\)The term $O(x)$ hides factors that are poly-logarithmic in $x$. 
(1 + $\epsilon$, $\beta$)-APSP. Finally, we provide poly(log log $n$)-round algorithm for the (1+$\epsilon$, $\beta$)-approximation for APSP for $\beta = O(\frac{\log \log n}{\epsilon^2})\log \log n$.

**Theorem 5 (Near-Additive APSP).** For any constant $\epsilon \in (0, 1)$, there exists a randomized algorithm in the Congested Clique model that w.h.p. computes a (1 + $\epsilon$, $\beta$)-approximation for the APSP problem in undirected unweighted graphs in $O((\log \log n)^2)$ rounds, for $\beta = O(\frac{\log \log n}{\epsilon^2})\log \log n$.

To the best of our knowledge, this is the first sub-polynomial algorithm to the problem. A polynomial algorithm can be obtained either by the exact algorithm for unweighted APSP that takes $O(n^{0.158})$ time [3], or by building a sparse near-additive spanner or emulator and letting the vertices learn it. This takes $O(n^3)$ rounds where $\rho$ is an arbitrary small constant, using the constructions in [10, 11]. We remark that these constructions work also in the more general CONGEST model, where vertices can only send $\Theta(\log n)$ bit messages to their neighbours.

Note that while [2] shows poly-logarithmic algorithms for multiplicative (2 + $\epsilon$)-approximation of APSP, prior to this work there was no nearly-additive approximation in sub-polynomial time. This nearly-additive approximation yields a near-exact (1 + $\epsilon$)-approximation for long distances (i.e., of distance $\Omega(\beta/\epsilon)$), improving upon the existing (2 + $\epsilon$)-approximation.

**Derandomization.** The randomized constructions presented above share a similar randomized core for which we devise a new derandomization algorithm. The current deterministic constructions of spanners and hopsets are mostly based on the derandomization of the hitting set problem [5, 15, 26]. In our context, the standard hitting set based arguments lead to a logarithmic overhead in the size of the emulator, and consequently also in the number of rounds to collect it. This is clearly too costly as we aim towards sub-logarithmic bounds. To handle that, we isolate the key probabilistic argument that leads to the desired sparsity of our emulators. By using the strong tool of pseudorandom generators for fooling DNF formulas [17], and extending the approach of [26], we show that this probabilistic argument can hold even when supplying all vertices a shared random seed of $O(\log n)$ bits. As we believe that this derandomization scheme might be useful in other contexts, we formalize the setting by defining the soft hitting set problem that captures our refined probabilistic arguments in a black-box manner, and show the following:

**Lemma 6.** The soft hitting set problem can be solved in $O((\log \log n)^3)$ rounds. This leads to $O((\log \log n)^4)$ deterministic solutions for Thm. 3, 4 and 5.

### 1.2 Our Techniques

One of the main ingredients in our algorithms is a new fast algorithm for constructing a near-additive emulator with $O(n \log \log n)$ edges. Roughly speaking, the near-additive emulator is a sparse graph\(^\dagger\) that preserves the distances up to a nearly-additive stretch. Collecting all the edges of this emulator at each vertex, already gives near-additive approximation for APSP. We start by explaining our emulator results and then explain how to use them for obtaining the approximate distances.

**Nearly Additive Emulators.** For an $n$-vertex unweighted graph $G = (V, E)$, a (1 + $\epsilon$, $\beta$) emulator $H = (V, E', \omega)$ is a sparse graph that preserves the distances of $G$ up to a (1 + $\epsilon$, $\beta$)-stretch. Nearly-additive emulators have been studied thoroughly mainly from the pure graph theoretic perspective [11, 28]. Our emulator algorithm is inspired by the two state-of-the-art constructions from Elkin and Neiman [11] and Thorup and Zwick [28]. The main benefit of our algorithm is its efficient implementation in the CONGEST model, which takes $O(\frac{\log \beta}{\epsilon})$ rounds w.h.p., for $\beta = O(\frac{\log \log n}{\epsilon^2})\log \log n$. For a constant $\epsilon$ this gives a complexity of $O((\log \log n)^2)$ rounds.

**Lemma 7 (Near-Additive Sparse Emulators).** For any $n$-vertex unweighted undirected graph $G = (V, E)$ and $0 < \epsilon < 1$, there is a randomized algorithm in the CONGEST model that computes (1 + $\epsilon$, $\beta$) emulator $H = (V, E', \omega)$ with $O(n \log \log n)$ edges within $O(\frac{\log \beta}{\epsilon})$ rounds w.h.p, where $\beta = O(\frac{\log \log n}{\epsilon^2})\log \log n$. We next explain how an emulator leads to fast algorithms for approximate shortest paths.

**Approximating shortest paths via near-additive emulators.** Assume we have a graph of size $O(n \log \log n)$ that preserves all the distances up to a (1 + $\epsilon$, $\beta$)-stretch for $\beta = O(\frac{\log \log n}{\epsilon^2})\log \log n$. Since this graph is sparse enough, all vertices can learn it in $O(\log \log n)$ rounds, which already gives a (1 + $\epsilon$, $\beta$)-approximation for APSP. For paths of length at least $t = O(\frac{\beta}{\epsilon})$, this actually gives a (1 + $\Theta(\epsilon)$)-approximation. It therefore remains to provide a near-exact approximation for all vertex pairs at distance at most $t = O(\frac{\beta}{\epsilon})$. The key observation is that since $t$ is small, one can get considerably faster algorithms for this task. Concretely, we reconstruct the tool-kit of [2], and turn them into distance-sensitive primitives, i.e., with a running time that depends on the given distance threshold $t$. For example, [2] show poly($\log n$) algorithms for computing the $k$-nearest vertices and for constructing hopsets.\(^5\) We show how to implement $t$-bounded variants of these tools in just poly($\log t$) time. Intuitively, if we only want to compute the $k$-nearest vertices at distance at most $t$, we need to employ only poly($\log t$) = poly($\log \log n$) steps of matrix multiplications, rather than poly($\log n$).

Using these tools together with the emulator leads to a poly($\log \log n$) round algorithm for (1 + $\epsilon$)-approximation for multi-source shortest paths. Using the recipe of [2], we can easily convert the (1 + $\epsilon$)-MSSP algorithm into a (3 + $\epsilon$)-APSP algorithm. Providing the ultimate (2 + $\epsilon$)-approximation turns out to be more technical, as following directly the approach in [2] adds logarithmic factors to the complexity, and to overcome it we add an additional sparsification phase to the algorithm.

**Constructing near-additive emulators.** We next give a high-level overview of our emulator construction. We build (1 + $\epsilon$, $O(\frac{\epsilon}{\log n})$)-emulator with $O(rn^{1+\frac{\epsilon}{\log n}})$ edges. For the choice $r = \log \log n$, we get a near-linear size emulator.

The construction is based on sampling sets $\emptyset = S_0 \subset S_1 \subset S_{r-1} \subset \ldots \subset S_1 \subset S_0 = V$ such that for $1 \leq i \leq r$, the set $S_i$ is

\(^\dagger\)Unlike spanners, the emulator is not necessarily a subgraph of the input graph. It might contain edges that are not in $G$, and it is allowed to be weighted, even when the graph $G$ is unweighted.

\(^5\)For a formal definition of a hopset, see Section 2.
constructed by adding each vertex of $S_{i-1}$ to $S_i$ with probability $p_i$. The probabilities $p_i$ are chosen such that the final emulator would have $O(r^i \cdot n^{1+\frac{1}{3r}})$ edges.

We add edges to the emulator as follows. For $0 \leq i \leq r$, each vertex $v \in S_i \setminus S_{i+1}$ looks at a ball $B(u, \delta_i, G)$ of radius $\delta_i = \Theta(\frac{1}{\sqrt{r}})$ around it. If this ball contains a vertex from $S_{i+1}$, it adds an edge to it, and otherwise it adds edges to all vertices in $B(u, \delta_i, G) \cap S_i$. Intuitively if there are not too many vertices in $S_i$ at distance $\delta_i$ from $v$ it adds edges to them, otherwise there would be a vertex from $S_{i+1}$ there and it adds an edge to it.

The intuition for the stretch analysis is as follows. We define a notion of $i$-clustered vertices, that in some sense captures the density of neighbourhoods around vertices. A vertex is $i$-clustered if there is a vertex from $S_i$ close-by. For example, all vertices are $0$-clustered, all high-degree vertices are $1$-clustered, all high-degree vertices that also have a vertex from $S_2$ close-by are $2$-clustered and so on. For a formal definition see Section 3. Now, if we take $u, v \in V$ such that all vertices in the shortest path between them, $\pi(u, v)$, are at most $i$-clustered we work as follows. We break $\pi(u, v)$ into segments of length $\frac{1}{\sqrt{r}}$ in each such segment we show that there is an additive stretch of $\Theta(\frac{1}{\sqrt{r}})$ between the first and last $i$-clustered vertices in the segment $w_1, w_2$. Summing up over all $d(u, v)e^t$ such segments leads to an additive $\Theta(d(u, v)e)$ term. To handle the parts of the segments not between $w_1$ and $w_2$ we use an inductive argument. Each one of the iterations of the algorithm adds $\Theta(d(u, v)e)$ term, which eventually leads to a stretch of $(1 + \Theta(re), O(\frac{1}{\sqrt{r}}))$. By rescaling, we get a stretch of $(1 + e, O(\frac{1}{e})^i)$. To get a sparse emulator of size $O(n \log \log n)$, we choose $r = \log \log n$, which gives a stretch of $(1 + e, O(\log \log \log n \cdot \log \log n))$.

### Implementation in the CONGESTED CLIQUE model

The intuition for the implementation is as follows. During the algorithm vertices inspect their local balls of radius at most $\delta_r = O(\log \log n \log \log n)$, and add edges to some of the vertices in these balls. Ideally, we would like to exploit the small radius of the balls to get a complexity of $\log(\log \delta_r) = \log(\log n)$. The key challenge is that it is not clear how vertices can learn their $t$-hop neighborhood in $\log t$ rounds. This sets a major barrier in the case where the $t$-hop ball of a vertex is dense (i.e., containing $o(\sqrt{n})$ vertices). Our key idea to overcome this barrier is based on separating vertices to heavy and light based on the sparsity level of their $t$-hop ball. For sparse vertices, whose $t$-hop ball has $O(n^{2/3})$ vertices, the algorithm collects the balls in $\log(\log t)$ rounds using our distance sensitive tool-kit. For the remaining dense vertices, we will be using the fact that a random collection of $O(\sqrt{n})$ vertices $S$ hit their balls. These ideas allow to implement all the iterations of the algorithm, except the last one. In the last iteration vertices from the last set of the emulator $S_r$ should add edges to all vertices of $S_r$ at distance at most $t = \delta_r$, here we exploit the fact that the set $S_r$ is small and use it together with our bounded hopset to get an efficient implementation.

### Deterministic algorithms

To derandomize our algorithms, we introduce the notion of soft hitting sets, which is roughly defined as follows. The input to the problem is given by two sets of vertices $R, L$, and an integer $\Delta \leq |L|$. Each vertex $u_i$ in $L$ holds a set $S_i \subseteq R$ of size at least $\Delta$. A set $S_i$ is hit by a subset of vertices $Z \subseteq R$ if $S_i \cap Z \neq \emptyset$. A subset of vertices $Z \subseteq R$ is a soft hitting set for the $S_i$'s if (i) $|Z| = O(n/\Delta)$ and (ii) the total size of all the $S_i$ sets that are not hit by $Z$ is $O(\Delta \cdot |L|)$. The main benefit of this definition over the standard hitting set definition is in property (i). In the hitting set definition, the size of the hitting set is $O(n \log \log n)$ (i.e., larger by an $O(\log n)$ factor).

The derandomization of the soft hitting set problem is based on the PRGs of Gopalan et al. [17]. These PRGs can fool a family of DNF formula of $n$ variables with a seed of length $O(\log \log n) \log n)$. Parter and Yogev [26] observed that the covering conditions of the hitting set problem can be stated as a read-once DNF formula, and used these PRGs to compute hitting sets in $O(\log \log n)^3$ CONGESTED CLIQUE rounds. We extend this framework and show that the soft hitting problem can be stated as a function that depends on the probability that a certain DNF formula is satisfiable, which allows us to compute the soft hitting set in $O(\log \log n)^3$ rounds. Overall, the derandomization adds $O((\log \log n)^4)$ term to the complexity of our algorithms.

We note that under the unbounded local computation assumption, our deterministic bounds can in fact match the randomized ones. Although this assumption is considered to be legit in the CONGESTED CLIQUE model (as well as in other standard distributed models), it is clearly less desirable. Thus in our main constructions we prefer to avoid it by paying an extra $\log \log \log n$ term. Nevertheless for completeness, we briefly explain the idea of such an optimal derandomization assuming that the vertices can perform unbounded local computation. The well-known PRGs by Nisan and Wigderson [25] provide a logarithmic seed for our probabilistic arguments. Since in our setting, we can compute a chunk of $\log n$ random bits in $O(1)$ rounds, the entire seed can be computed in $O(1)$ rounds. The only caveat of this PRG is its construction time. However, since in our algorithms the PRG is computed locally, the vertices can use this PRG as a black-box in our derandomization algorithms.

### Comparison to existing nearly-additive emulator constructions

The most related constructions to ours are the centralized emulator constructions of Elkin and Neiman [11] and Thorup and Zwick [28]. The first construction also has distributed implementations in the CONGEST model in $O(n^\rho)$ time where $\rho$ is an arbitrary small constant [10, 11]. For the sake of the discussion, we say that an algorithm is local if its exploration radius is sub-polynomial (i.e., every vertex explores a sub-polynomial ball around it in order to define its output), and otherwise it is global. Also, we say that an algorithm is cluster-centric if in the clustering procedure of the algorithm, all vertices in the cluster make a collective decision, on behalf of the cluster. An algorithm is vertex-centric, if each vertex makes its own individual decisions.

With this rough characterization in hand, the EN algorithm is local and cluster-centric. The TZ algorithm, on the other hand, is global and vertex-centric. Our algorithm appears to be an hybrid of the two: it is local and vertex-centric. Both of these properties are important for its efficient implementation. The vertex-centric approach allows to give a very short and simple algorithm. This led to a very fast implementation in the CONGESTED CLIQUE model. The local approach is crucial to obtaining a sub-logarithmic complexity. Implementing the global approach of TZ in the CONGESTED CLIQUE model seems to require at least polynomial time. We remark
that while the description of our algorithm is different from the description of EN, they seem to be two different ways of viewing a similar process. This connection is useful for the analysis, and indeed some elements in our stretch analysis are inspired by [10, 11].

For a more detailed comparison between the algorithms, see [9]. For a recent survey on near-additive spanners and emulators see [14].

Roadmap. The rest of the paper is organized as follows. In Section 2, we discuss basic definitions and tools. Our emulator construction appears in Section 3. Our shortest paths algorithms appear in Section 4, and the deterministic variants are discussed in Section 5. Full details and proofs appear in the full version [9].

2 PRELIMINARIES

To implement the algorithm, we need the following definitions and tools.

Notation. Given a graph $G = (V, E)$, we denote by $d_G(u, v)$ the distance between the vertices $u$ and $v$ in $G$. If $G$ is clear from the context, we use the notation $d(u, v)$ for $d_G(u, v)$. The $t$-hop distance between $u$ and $v$, denoted by $d^t_G(u, v)$, is the distance of the shortest path between $u$ and $v$ that uses at most $t$ edges. We denote by $B(\delta, G)$ the ball of radius $\delta$ around $v$ in $G$. We use the term w.h.p for an event that happens with probability at least $1 - \frac{1}{n^c}$ for some constant $c$. We use the notation $S' \leftarrow \text{Sample}(S, p)$ for a subset $S'$ that is sampled from $S$ by adding each vertex to $S'$ independently with probability $p$.

Model. In the Congested Clique model, we have a communication network of $n$ vertices, communication happens in synchronous rounds, and per round, each two vertices exchange $O(\log n)$ bits. The input and output are local, in the sense that initially each of the $n$ vertices knows only its incident edges in the graph $G = (V, E)$, and at the end each vertex should know the part of the output adjacent to it. For example, its distances from other vertices.

Near-additive emulator. Given an unweighted graph $G = (V, E)$, a weighted graph $H = (V, E')$ on the same set of vertices is a $(1 + \epsilon, \beta)$-emulator for $G$ if for any pair of vertices $u, v \in V$, it holds that $d_H(u, v) \leq d_G(u, v) \leq d_H(u, v) + \beta$.

Hitting sets. Let $S_v \subseteq V$ be a set of size at least $k$. We say that $A$ is a hitting set of the sets $\{S_v\}_{v \in V}$, if $A$ has a vertex from each one of the sets $S_v$. We can construct hitting sets easily by adding each vertex to $A$ with probability $p = O(\frac{\log n}{k})$, which gives the following. The proof is deferred to [9].

Lemma 8. Let $V' \subseteq V$, and let $\{S_v \subseteq V\}_{v \in V'}$ be a set of subsets of size at least $k$. There exists a randomized algorithm in the Congested Clique model that constructs a hitting set of size $O(n \log n/k)$ w.h.p, without communication.

A recent deterministic construction of hitting sets in the Congested Clique model is given in [26], which show the following (see Corollary 17 in [26]).

Lemma 9. Let $V' \subseteq V$, and let $\{S_v \subseteq V\}_{v \in V'}$ be a set of subsets of size at least $k$, such that $S_v$ is known to $v$. There exists a deterministic algorithm in the Congested Clique model that constructs a hitting set of size $O(n \log n/k)$ in $O((\log \log n)^3)$ rounds.

The $(k, d)$-nearest problem. In the $(k, d)$-nearest problem, we are given integers $k, d$, and the goal of each vertex is to learn the distances to the closest $k$ vertices of distance at most $d$. If there are less than $k$ vertices at distance at most $d$ from $v$, it learns the distances to all vertices of distance at most $d$. We call the $k$ closest vertices of distance at most $d$ from $v$, the $(k, d)$-nearest vertices to $v$, and denote them by $N_{k,d}(v)$. We focus on solving this problem in unweighted graphs.

This problem is an extension of the $k$-nearest problem discussed in [2], where the goal is to find the $k$-nearest vertices in the whole graph, without restriction on the distance. Extending ideas from [2], we show that the $(k, d)$-nearest problem can be solved in time $poly(\log d)$ if we only consider distances bounded by $d$ and $k = O(n^{2/3})$. The proof appears in [9].

Theorem 10. The $(k, d)$-nearest problem can be solved in unweighted graphs in $O\left(\left(\frac{k}{n^{2/3}} + \log d\right) \log d\right)$ rounds in Congested Clique.

The $(S, d)$-source detection problem. In the $(S, d)$-source detection problem, we are given a set of sources $S \subseteq V$ and an integer $d$, and the task is to compute for each vertex $v$ the set of sources within hop distance at most $d$, as well as the distances to those sources using paths of at most $d$ hops. The input graph to the problem can be weighted. The following is shown in [2].

Theorem 11. The $(S, d)$-source detection problem can be solved in weighted graphs in $O\left(\frac{m^{1/3}|S|^{2/3}}{n} + 1\right)d$ rounds in Congested Clique, where $m$ is the number of edges in the input graph.

Bounded hopsets. Note that in Theorem 11 there is a linear dependence on $d$. Hopsets would allow us to solve the same task in time $poly(\log d)$ at the price of obtaining $(1 + \epsilon)$-approximations for the distances.

Given a graph $G$, a $(\beta, \epsilon)$-hopset $H$ is a weighted graph on the same vertices, such that the hop-distances in $G \cup H$ give $(1 + \epsilon)$-approximation for the distances in $G$. The importance of hopsets comes from the fact that they allow to focus only on short paths in $G \cup H$ which is crucial for obtaining a fast algorithm. In [2] it is shown that hopsets can be constructed in $O(\frac{\log^2 n}{\epsilon})$ rounds in the Congested Clique, which is too expensive for our purposes. Here, we focus on constructing bounded hopsets, that give good approximation only to paths of at most $t$ hops, and show that they can be constructed in time which is just $poly(\log t)$.

Bounded hopsets are defined as follows. Given a graph $G$, a $(\beta, \epsilon, t)$-hopset $H$ is a weighted graph on the same vertices, such that the hop-distances in $G \cup H$ give $(1 + \epsilon)$-approximation for all pairs of vertices in $G$ where $d_G(u, v) = d^t_H(u, v)$. In unweighted
graphs, these are all pairs of vertices at distance at most $t$ from each other. For any such pair, we have $d_{G}(u,v) \leq d_{G,H}(u,v)$ and $d_{G}(u,v) \leq d_{G,H}(u,v) + (1 + \epsilon)d_{G}(u,v)$. Extending ideas from [2], we show the following. The proof appears in [9].

**Theorem 12.** Let $G$ be an unweighted undirected graph and let $0 < \epsilon < 1$.

1. There is a randomized construction of a $(\beta, \epsilon, t)$-hopset with $O(n^{1/2} \log n)$ edges and $\beta = O(\log(t)/\epsilon)$ that takes $O((\log t) / \epsilon)$ rounds w.h.p. in the CONGESTED CLIQUE model.

2. There is a deterministic construction of a $(\beta, \epsilon, t)$-hopset with $O(n^{1/2} \log n)$ edges and $\beta = O(\log(t)/\epsilon)$ that takes $O((\log t) / \epsilon + (\log \log n)^{3})$ rounds in the CONGESTED CLIQUE model.

3. $(1 + \epsilon, \beta)$-EMULATORS

### 3.1 Warm-up: $(1 + \epsilon, \Theta(\frac{1}{\epsilon}))$-emulator with $\tilde{O}(n^{1+\epsilon})$ edges.

To illustrate the intuition behind our algorithm, we start by describing a simplified variant with $\tilde{O}(n^{1+\epsilon})$ edges. The construction samples two random sets $S_1, S_2$ as follows: $S_1$ is a random set of size $O(n^{1/4})$, defined by adding each vertex to $S_1$ with probability $\frac{1}{n^{1/4}}$, and $S_2$ is a random set of size $O(n^{1/4})$ defined by adding each vertex of $S_1$ to $S_2$ with probability $\frac{1}{n^{1/4}}$. We add to the emulator the following (possibly weighted) edges:

1. All edges adjacent to low-degree vertices of degree at most $n^{1/4} \log n$. In addition, each vertex of degree at least $n^{1/4} \log n$ adds an edge to a neighbour in $S_1$, which exists w.h.p. as $S_1$ is a random set of size $n^{1/4}$.

2. Vertices $v \in S_1$ consider their ball $B(v, \delta, G)$ of radius $\delta = \frac{1}{2} + 2 \log n$. If this ball has at most $\sqrt{n} \log n$ vertices from $S_1$, $v$ adds an edge to each of them. Otherwise, w.h.p., this ball has a representative from $S_2$, and $v$ adds a weighted edge to it.

3. The vertices in $S_2$ add edges to all the vertices.

Each edge $(u, v)$ added to the emulator has a weight of $d_{G}(u, v)$. Note that in Line 1 each vertex adds at most $n^{1/4} \log n$ edges. In Line 2, each vertex of $S_1$ adds at most $\sqrt{n} \log n$ edges, as $|S_1| = O(n^{1/4})$ is sums up to $O(n^{1+\epsilon})$ edges. Finally, in Line 3, we add at most $O(n^{1+\epsilon})$ edges as $|S_2| = n^{1/4}$. Hence, in total the emulator has $\tilde{O}(n^{1+\epsilon})$ edges.

**Stretch analysis.** We next sketch the stretch analysis. Let $u, v \in V$ such that $\pi(u, v)$ is the shortest $u \rightarrow v$ path. If $\pi(u, v)$ only contains low degree vertices it is contained in the emulator. Otherwise, $\pi(u, v)$ has at least one vertex $w$ with degree at least $n^{1/4} \log n$. As $S_1$ is a random set of size $O(n^{3/4})$, then $w$ has a neighbour $s \in S_1$. There are 2 options for $s$: either it adds edges to all vertices from $S_1$ of radius $\delta = \frac{1}{2} + 2 \log n$, or it adds one edge to a vertex in $S_2$ at distance at most $\delta$. In the first case, we say that $w$ is 1-clustered and in the second case we say that $w$ is 2-clustered. We break into cases according to this. The case that $w$ is 2-clustered is actually easy, as in this case there is a vertex from $S_2$ at distance $\frac{1}{\epsilon} \theta$ from $\pi(u, v)$, since we added edges between all vertices to vertices in $S_2$, it follows that the emulator has a $u \rightarrow v$ path with additive stretch $\Theta(\frac{1}{\epsilon})$. The more interesting case is that all high-degree vertices are 1-clustered, in this case we can break the path to sub-paths of length $\frac{1}{\epsilon}$ and show that in each of them we get an additive $4 \epsilon$ stretch. The intuition is that if we take the first and last high degree vertices in each such sub-path, the emulator has a short path between them since their neighbours in $S_1$ have an edge between them. As $\pi(u, v)$ has $ed(u, v)$ segments of length $\frac{1}{\epsilon}$, this results in a $(1 + 4 \epsilon, 4 \epsilon)$ stretch overall, where the last additive term is for the case that $d(u, v) < \frac{1}{\epsilon}$. If we sum up over all cases, we get a stretch of $O(1 + 4 \epsilon, \Theta(\frac{1}{\epsilon}))$. By rescaling of $\epsilon$ we can get a $(1 + \epsilon, \Theta(\frac{1}{\epsilon}))$ stretch.

### 3.2 Algorithm for $(1 + \epsilon, \beta)$-Emulators

We next describe an algorithm that builds an emulator of size $O(n \log \log n)$ edges and stretch of $(1 + \epsilon, O(\log \log n \log \log n \log \log n))$. Later we show a variant of the algorithm that can be efficiently implemented in the CONGESTED CLIQUE.

Let $\psi = S_{r+1} \subset S_r \subset S_{r-1} \ldots \subset S_1 \subset S_0 = V$ be random subsamples of vertices, such that $S_i \leftarrow \text{Sample}(S_{i-1}, p_i)$ for every $i \in \{1, \ldots, r\}$, where $p_i = n^{-1/2i}$ for $1 \leq i \leq r - 1$, and $p_r = n^{-\frac{1}{2r}}$. For every $i \in \{0, \ldots, r\}$, we define the values $R_i, \delta_i$. Intuitively, $\delta_i$ is a radius such that vertices in $S_i$ consider the ball of radius $\delta_i$ around them, and add edges to certain vertices there, and $R_i$ would bound the distance between certain vertices to the closest vertex in $S_i$. The values $R_i, \delta_i$ are defined as follows. $R_0 = 0, R_1 = \sum_{i=0}^{r-1} \delta_i$, where $\delta_i = \frac{1}{\epsilon} + 2R_i$. For every $i \in \{0, \ldots, r\}$, define $s \in S_i$ to be $i$-dense if $B(s, \delta_i, G) \cap S_{i+1} \neq \emptyset$, and otherwise it is $i$-sparse. The emulator includes edges from each $i$-sparse vertex $s \in S_i$ to all vertices in $S_i$ of distance at most $\delta_i$, and one edge from each $i$-dense vertex $s \in S_i$ to a vertex in $S_{i+1}$ of distance at most $\delta_i$.

For each vertex $u$ and $i \in \{0, \ldots, r\}$, we define $c_i(u)$ to be a vertex in $S_i$ of distance at most $R_i$ from $u$ if such exists, $c_i(u)$ is defined inductively as follows. Set $c_0(u) = u$. The vertex $c_{r+1}(u)$ exists only if $c_i(u)$ exists and is $i$-dense. In this case, we define $c_{r+1}(u)$ to be the closest vertex to $c_i(u)$ in $B(c_i(u), \delta_i, G) \cap S_{i+1}$, such vertex exists since $c_i(u)$ is $i$-dense. We say that a vertex is $i$-clustered if $c_i(u)$ exists. In particular, all vertices are $0$-clustered, and each $i$-clustered vertex is also $i'$-clustered for any $0 \leq i' \leq i$.

**The edges of the emulator.** For every $v \in V \setminus S_{i+1}$, we add to the emulator $H$ the following edges:

- If $v$ is $i$-dense, we add one edge to $c_{i+1}(v)$.
- If $v$ is $i$-sparse, we add edges to all vertices $u \in B(v, \delta_i, G) \cap S_i$.

For each edge $(u, v)$ added to the emulator, its weight is the weight of the shortest $u \rightarrow v$ path in $G$. Note that each vertex is exactly in one of the sets $S_i \setminus S_{i+1}$, hence it adds edges only once in the process. An inductive argument shows that the distance between $v$ and $c_i(v)$ in the emulator is indeed at most $R_i$, the proof is deferred to [9].

---

\footnote{The definition of $p_i$ would be useful for the CONGESTED CLIQUE implementation, for other purposes it is possible to define it in the same way as the other $p_i$ values.}
Claim 13. Every $i$-clustered vertex $v$ is connected to $c_i(v)$ in the emulator by a path of at most $i$ edges and of total weight at most $R_i$.

Size analysis. For lack of space, the size analysis appears in [9]. It is based on showing the following. First, for $0 \leq i < r$, a simple calculation shows that $S_i$ is of size $n^{1 - \frac{2i}{c^i+1}}$ in expectation. In addition, by the choice of the probabilities $p_i$, each vertex $s \in S_i \setminus S_{i+1}$ adds at most $O(n^{\frac{i}{c^i}})$ edges to the emulator in expectation. This sums up to $O(n^{1+\frac{i}{c^i}})$ edges added by vertices in $S_i \setminus S_{i+1}$. In addition, we show that $S_r = O(\sqrt{n})$ w.h.p., hence the total number of edges added between vertices in $S_r$ is bounded by $O(n)$. Overall, the expected number of edges in the emulator is bounded by $O(r \cdot n^{1+\frac{r}{c^r}})$.

3.3 Stretch analysis

The stretch analysis appears in Lemma 18. To prove it, we need the following technical claims, for proofs see [9]. We start by bounding the value of $R_i$.

Claim 14. $R_i = \sum_{j=0}^{i} \frac{1}{c^j} \cdot 3^{j-i-1}$.

Claim 15. Let $0 < \epsilon < \frac{1}{c}$, then $R_i \leq \frac{2}{\epsilon^2}$.

We next define a value $\beta_i$ that would give a bound on the additive stretch of paths of $i$-clustered vertices in Lemma 18. It is defined as follows. Define $\beta_0 = 0$, $\beta_i = 4 \sum_{j=1}^{i} 2^{-j} R_j$.

Claim 16. $4R_i + 2\beta_{i-1} = \beta_i$.

Claim 17. Let $0 < \epsilon < \frac{1}{15}$, then $\beta_i \leq \frac{10}{\epsilon^2}$.

Let $\pi(u, v)$ be any shortest $u - v$ path. We denote by $d_H(u, v)$ the distance between $u$ and $v$ in the emulator, and by $d(u, v)$ the distance between them in $G$.

Lemma 18. Assume that all the vertices in $\pi(u, v)$ are at most $i$-clustered, except maybe one that is $j$-clustered for $j > i$, then $d_H(u, v) \leq (1 + 20e) d(u, v) + \beta_i$.

Proof. The proof is by induction. If $i = 0$, all the vertices in $\pi(u, v)$ are at most 0-clustered, except maybe one. As all vertices that are at most 0-clustered are 0-spars and add all their adjacent edges to the emulator, it holds that all the edges of $\pi(u, v)$ are taken to the emulator. Hence, $d_H(u, v) = d(u, v)$. Assume that the claim holds for $i - 1$, and we show that it holds for $i$.

Case 1: $\pi(u, v)$ has length at most $\frac{1}{c^i}$. If there is only one vertex in $\pi(u, v)$ that is $j$-clustered for $j \geq i$, then the claim follows from the induction assumption. Hence, we focus on the case that there are at least two $i$-clustered vertices in $\pi(u, v)$. Denote the first and last such vertices by $u'$ and $v'$ respectively. From Claim 13, for every $i$-clustered vertex $w$, the distance in the emulator between $w$ and $c_i(w)$ is at most $R_i$. Hence, $d_H(u', c_i(u')) \leq R_i$ and $d_H(v', c_i(v')) \leq R_i$. Since $d(u', v') \leq \frac{1}{c^i}$, it follows that

$$d(c_i(u'), c_i(v')) \leq d(c_i(u'), u') + d(u', v') + d(v', c_i(v')) \leq d(u', v') + 2R_i \leq \frac{1}{c^i} + 2R_i.$$

Also, from the statement of the claim at least one of $u'$, $v'$ is at most $i$-clustered. Assume w.l.o.g. that this is $u'$. This means that $c_i(u')$ adds edges to all vertices of $S_j$ at distance at most $\delta_i = \frac{1}{c^i} + 2R_i$, and in particular there is an edge between $c_i(u')$ and $c_i(v')$ in the emulator of weight at most $d(u', v') + 2R_i$. It follows that

$$d_H(u', v') \leq d_H(u', c_i(u')) + d_H(c_i(u'), c_i(v')) + d_H(c_i(v'), v') \leq R_i + d(u', v') + 2R_i = d(u', v') + 4R_i.$$

Now, the subpaths of $\pi(u, v)$ between $u$ and $u'$ and between $v'$ and $v$ are also shortest paths $\pi(u, u')$, $\pi(v', v)$, which have only one vertex that is $j$-clustered for $j \geq i$. Hence from the induction hypothesis,

$$d_H(u, u') \leq (1 + 20e(i - 1))d(u, u') + \beta_{i-1},$$

$$d_H(v', v) \leq (1 + 20e(i - 1))d(v', v) + \beta_{i-1}.$$

Combining it all, we get that

$$d_H(u, v) \leq d_H(u, u') + d_H(u', v') + d_H(v', v) \leq (1 + 20e(i - 1))d(u, v) + 4R_i + 2\beta_{i-1}.$$

From Claim 16, we get

$$d_H(u, v) \leq (1 + 20e(i - 1))d(u, v) + \beta_i,$$

which concludes the proof for this case.

Case 2: $\pi(u, v)$ has length greater than $\frac{1}{c^i}$. Here we divide $\pi(u, v)$ to subpaths of length exactly $\frac{1}{c^i}$, except maybe one of length at most $\frac{1}{c^i}$. In each of these subpaths we can follow the analysis from Case 1 and get a stretch of $(1 + 20e(i - 1), \beta_i)$ for the subpath. From Claim 17, $\beta_i \leq \frac{10}{\epsilon^2}$, hence, for each subpath of length $\frac{1}{c^i}$ we add at most additive stretch of $\frac{10}{\epsilon^2}$. As $\epsilon \leq \frac{1}{15}$, we have that $\frac{1}{c^i} \geq \frac{1}{c} - 1 \geq \frac{10}{\epsilon^2}$. Hence, there are at most $d(u, v) \cdot 2e^i$ subpaths of length exactly $\frac{1}{c^i}$. As each one of them adds an additive stretch of at most $\frac{10}{\epsilon^2}$, this adds at most $\frac{10}{\epsilon^2} \cdot d(u, v) \cdot 2e^i = 20ed(u, v)$ to the total stretch. The last subpath of length at most $\frac{1}{c^i}$ adds $\beta_i$ to the additive stretch. In total we get

$$d_H(u, v) \leq (1 + 20e(i - 1))d(u, v) + 20ed(u, v) + \beta_i \leq (1 + 20e) d(u, v) + \beta_i,$$

which completes the proof.

Conclusion. The analysis in previous sections gives the following (see [9] for a proof).

Theorem 19. Let $G$ be an unweighted undirected graph, and let $0 < \epsilon < 1$, there is a randomized algorithm that builds an emulator $H$ with $O(r \cdot n^{1+\frac{r}{c^r}})$ edges in expectation, and stretch of $(1 + e, O(\frac{\log \log n}{\epsilon})^{-1})$. For the choice $r = \log \log n$, we have $O(n \log \log n)$ edges in expectation and stretch of $(1 + e, O(\frac{\log \log n}{\epsilon} \log \log \log n))$.

3.4 Constructing emulators in the CONGESTED CLIQUE

We next explain how to implement the algorithm from Section 3.2 efficiently in the CONGESTED CLIQUE model. This algorithm is composed of two parts, first we sample the sets $S_i$, and then each vertex $v \in S_j$ looks at a certain $\delta_i$-neighbourhood around it and adds edges to certain vertices there. The sampling is a completely local task that can be simulated locally by each vertex. For the second part, we focus first on vertices that are in $S_j \setminus S_{i+1}$ for $i < r$. Such vertices are either $i$-sparse, in which case they add edges to all vertices in $S_i$ in radius $\delta_i$, or $i$-dense in which case they add one edge to the closest vertex from $S_{i+1}$. To implement it we use the $(k, d)$-NEAREST algorithm. The intuition is simple. If the
δ₁-neighbourhood around a vertex v ∈ Sᵢ \ Sᵢ₊₁ has at most \( n^{2/3} \) vertices, running the (k, d)-nearest algorithm with k = n²/₃, d = δ₁ actually allows v to learn its entire δ₁-neighbourhood and simulate the algorithm. If, on the other hand, the δ₁-neighbourhood around v has more than \( n^{2/3} \) vertices, then since \( Sᵢ \) is a random set of O(√n) vertices, the δ₁-neighbourhood of v contains a vertex from \( Sᵢ \) w.h.p. As \( Sᵢ \subseteq Sᵢ₊₁ \), it follows that v is i-dense, and only needs to add one edge to the closest vertex from \( Sᵢ₊₁ \). Running the (k, d)-nearest algorithm allows to learn the distance to this vertex. To summarize, all vertices not in \( Sᵢ \) can actually add all their adjacent edges to the emulator using the (k, d)-nearest algorithm.

To deal with vertices in \( Sᵢ \) we use a different approach. Note that since \( Sᵢ₊₁ = \emptyset \), all the vertices in \( Sᵢ \) are r-sparse and should add edges to all vertices in \( Sᵢ \) in their δᵢ-neighbourhood. However, this neighbourhood may be large, and there is no clear way to learn the whole neighbourhood. Here we exploit the fact that \( Sᵢ \) is of size O(√n), so we only need to compute distances to at most O(√n) vertices. We show that using the source detection algorithm and the bounded hopset we can compute approximations to all these distances efficiently. Using the hopset is crucial to get a poly(log δᵢ) = poly(log log n) complexity, instead of a much higher \( O(δᵢ) \) complexity.

We next describe the algorithm in detail, and explain the changes needed in the analysis since we compute only approximations to distances in the final stage.

**Sampling the sets \( Sᵢ \).** We sample the sets \( Sᵢ \) exactly as described in Section 3.2, this is a local process computed by each vertex. For a vertex v, let \( iᵥ \) be the maximum index i such that v ∈ \( Sᵢ \). At the end of the sampling, each vertex sends one message to all other vertices with the index \( iᵥ \).

**Adding edges to the emulator.** To describe the algorithm, we need the following definitions. For a vertex v, let \( Bᵥ = B(v, δᵢᵥ, G) \). We say that v is heavy if \( |Bᵥ| > n^{2/3} \) and otherwise it is light. Let \( N_k,δᵢᵥ(v) \) be a set of k = n²/₃ closest vertices to v of distance at most δᵢᵥ. Note that if v is heavy, there are at least \( n^{2/3} \) vertices in the δᵢᵥ-neighbourhood of v. Since \( Sᵢ \) contains each vertex with probability \( \frac{1}{\sqrt{n}} \) we have the following (the proof is deferred to [9]).

**Claim 20.** For all heavy vertices v, there is a vertex in \( N_k,δᵢᵥ(v) \cap Sᵢ \) w.h.p. for k = n²/₃.

We will show that for each vertex with \( iᵥ < r \), we can add all the adjacent edges to the emulator by just learning the (k, d)-nearest vertices, and for vertices in \( Sᵢ \), we will use the bounded hopsets to get approximations to the distances.

**Claim 21.** All the edges of the emulator with at least one endpoint in \( V \setminus Sᵢ \) can be added to the emulator with correct distances in O(log² δᵢ) rounds w.h.p.

**Proof.** To implement the algorithm, we compute the (k, d)-nearest for k = n²/₃, d = δ₁, which takes O(log² δ₁) time by Theorem 10. Let \( N_k,d(v) \) be the set of k closest vertices of distance at most d to v computed by the algorithm. Let v ∈ \( Sᵢ \), and let \( iᵥ \) be the maximum index such that v ∈ \( Sᵢ \). If v is light \( |Bᵢ| = |B(v, δᵢᵥ, G)| ≤ n^{2/3} \). Since δ₁ᵥ ≤ δ₁ = d, it follows that \( Bᵢ \subseteq N_k,d(v) \). Hence, v already knows the distances to all vertices in \( Bᵢ \), and since it also knows which of them belong to each set \( Sᵢ \), it can add all the relevant edges to vertices in \( Bᵢ \cap Sᵢ \), as follows. Since v computed the set \( Bᵢ \), it knows whether \( Bᵢ \cap Sᵢ₊₁ \neq \emptyset \), in this case it is i-dense, and adds an edge to the closest vertex in \( Bᵢ \cap Sᵢ₊₁ \). Otherwise, it is i-sparse, and adds edges to all vertices in \( Bᵢ \cap Sᵢ \).

We next consider the case that v is heavy. Then, by Claim 20, there is a vertex u ∈ \( N_k,δᵢᵥ(v) \cap Sᵢ \) w.h.p. Now v knows the distances to all vertices in \( N_k,u \) and in particular to u. Also, \( u \in Sᵢ \subseteq Sᵢ₊₁ \), which means that v is i-dense. Hence, v only needs to add one edge to \( cᵢᵥ(u) \) which is the closest vertex from \( Sᵢ₊₁ \), since v knows the distance to u and also to all other vertices strictly closer than u, it can add the relevant edge as needed. □

**Claim 22.** All the edges in the emulator with two endpoints in \( Sᵢ \) can be added to the emulator with (1 + ε')-approximate distances in O(\( \log^2 \deltaᵢ / \epsilon' \)) rounds w.h.p.

**Proof.** Vertices in \( Sᵢ \) should add edges to all vertices in \( Sᵢ \), of distance at most δᵢ. For this, we start by running the bounded hopset algorithm with \( t = δᵢ \), which takes O(\( \log^2 \deltaᵢ / \epsilon' \)) time and constructs a (β, ε', t)-hopset \( H' \) with \( β = O(\log \deltaᵢ / \epsilon') \) by the randomized construction in Theorem 12. By definition, for all pairs of vertices \( u, v \) of distance at most \( t \) there is a β-hop path in \( G \cup H' \) of length at most \( (1 + \epsilon')d_G(u, v) \). To learn about those paths we run the (S, d)-source detection algorithm on the graph \( G \cup H' \) with \( S = \bigcup Sᵢ \), \( d = β = O(\log \deltaᵢ / \epsilon') \) which takes O(\( n^{2/3}n^{1/3} / n + 1 \log \deltaᵢ / \epsilon' \)) time w.h.p by Theorem 11, as the size of \( Sᵢ \) is O(\( \sqrt{n} \)) w.h.p. This gives (1 + ε')-approximations for the distances to all sources in \( Sᵢ \) of distance at most δᵢ as needed, and hence all vertices in \( Sᵢ \) can add all the relevant edges to the emulator with approximate distances. The overall time complexity is O(\( \log^2 \deltaᵢ / \epsilon' \)) rounds from constructing the hopsets. □

To conclude, we have the following.

**Lemma 23.** The time complexity of the algorithm is O(\( \log^2 \deltaᵢ / \epsilon' \)) rounds w.h.p.

**Proof.** Sampling the sets \( Sᵢ \) and informing all vertices about it takes one round, adding edges to the emulator takes O(\( \log^2 \deltaᵢ / \epsilon' \)) rounds w.h.p by Claims 21 and 22.

**Analysis and conclusion.** Since we compute only approximations to distances of edges with both endpoints in \( Sᵢ \), this changes slightly the stretch analysis in the final stage, see [9] for full details. To conclude, we get a (1 + ε, β) emulator with O(r · n¹⁺ β / ε) edges in O(\( \log^2 \beta / \epsilon \)) rounds, where \( β = O(\log n) \). Later we focus mostly on the case that \( r = \log n \). In this case, \( \log β = O(\log \log n - \log \log n / \epsilon) \). If ε is constant we get that a complexity of O(\( \log^2 \beta / \epsilon \)) is roughly O(\( \log^2 n \)).

**Theorem 24.** Let \( G \) be an unweighted undirected graph, 0 < ε < 1 and let \( r ≥ 2 \) be an integer; there is a randomized algorithm that builds an emulator \( H \) with O(r · n¹⁺ β / ε) edges in expectation, and stretch of (1 + ε, β), in O(\( \log^2 \beta / \epsilon \)) rounds w.h.p, where \( β = O(\log n) \). For
the choice \( r = \log \log n \), we have \( O(n \log \log n) \) edges in expectation, and \( \beta = O\left(\frac{\log \log n}{\epsilon}\right)\log \log n \).

A variant that works w.h.p. In the algorithm described above, the number of edges is \( O(n \log \log n) \) in expectation. For our applications, it would be useful to have this number of edges w.h.p. we next describe a variant that obtains this.

From the size analysis, all vertices not in \( S_r \) add \( O(rn^{1+\frac{2}{\beta}}) \) edges to the emulator in expectation. In addition, the size of \( S_r \) is \( O(\sqrt{n}) \) w.h.p. We showed that \( |S_r|^2 = O(n) \) w.h.p. hence the number of edges added between vertices in \( S_r \) is at most \( O(n) \) w.h.p. Also, from Claim 20, for all heavy vertices \( v \), there is a vertex in \( N_{\frac{\beta}{2}}(v) \cap S_r \) w.h.p. We would like to find a run where all the above events hold. From Markov’s inequality, we have that with constant probability the number of edges added by all vertices not in \( S_r \) is \( O(n^{1+\frac{2}{\beta}}) \), since the other events hold w.h.p. we have that with constant probability all the above events hold. Hence, if we run the algorithm for \( O(\log n) \) times in parallel, w.h.p we have a run where all events hold. In such a run the total number of edges in the emulator is \( O(rn^{1+\frac{2}{\beta}}) \). In [9], we show how to simulate \( O(\log n) \) runs in parallel. The intuition is that each vertex only needs to let all vertices know the maximum index \( i \) such that \( v \in S_i \). As \( i \leq \log \log n \), this index can be represented by \( O(\log \log n) \) bits, and we can send \( O(\log n) \) such values efficiently. Based on these ideas, we show the following.

**Theorem 25.** Let \( G \) be an unweighted undirected graph, let \( 0 < \epsilon < 1 \) and let \( r \geq 2 \) be an integer, there is a randomized algorithm that builds an emulator \( H \) with \( O(r \cdot n^{1+\frac{2}{\beta}}) \) edges w.h.p. and stretch of \( (1 + \epsilon, \beta) \), in \( O(\frac{\log \beta}{\epsilon}) \) rounds w.h.p. where \( \beta = O\left(\frac{\beta}{\epsilon}\right)^{-1} \). For the choice \( r = \log \log n \), we have \( O(n \log \log n) \) edges w.h.p. and \( \beta = O\left(\frac{\log \log n}{\epsilon}\right)\log \log n \).

### 4 APPLICATIONS

**\((1 + \epsilon, \beta)\)-approximation of APSP.** Using the emulator, we directly get a near-additive approximation for APSP, by building a sparse emulator and letting all vertices learn it. This gives the following (see [9] for a full proof).

**Theorem 26.** Let \( 0 < \epsilon < 1 \), there is a randomized \((1 + \epsilon, \beta)\)-approximation algorithm for unweighted undirected APSP in the Congested Clique model that takes \( O(\frac{\log \beta}{\epsilon}) \) rounds w.h.p. where \( \beta = O\left(\frac{\log \log n}{\epsilon}\right)\log \log n \).

**\((1 + \epsilon)\)-approximation of multi-source shortest paths.** In [9], we show how to get \((1 + \epsilon)\)-approximation for SSSP or multi-source shortest paths as long as the number of sources is \( O(\sqrt{n}) \) in just \( \text{poly}(\log \log n) \) rounds in unweighted undirected graphs. The idea is simple, for vertices that are far away, a \((1 + \epsilon, \beta)\)-approximation already gives a \((1 + \epsilon)\)-approximation. So we just need to take care of close by vertices of distance around \( O(\frac{\beta}{\epsilon}) \) from each other, for this we can use the bounded hopset. We show the following.

**Theorem 27.** Let \( 0 < \epsilon < 1 \) and let \( G \) be an unweighted undirected graph, there is a randomized \((1 + \epsilon)\)-approximation algorithm for multi-source shortest paths in the Congested Clique model from a set of sources \( S \) of size \( O(\sqrt{n}) \) that takes \( O(\frac{\log \beta}{\epsilon}) \) rounds w.h.p. where \( \beta = O\left(\frac{\log \log n}{\epsilon}\right)\log \log n \).

**\(2+\epsilon)\)-approximation of APSP.** We showed how to get a \((1 + \epsilon, \beta)\)-approximation for APSP, next we discuss a \((2 + \epsilon)\)-approximation, which gives a better approximation for short paths. As we showed before, for long paths of length around \( t = O(\frac{\beta}{\epsilon}) \), we already have a \((1 + \epsilon)\)-approximation from the emulator, so we need to take care only of short paths of length at most \( t \). To explain the intuition, we start by describing a simple \((3 + \epsilon)\)-approximation, and then explain how to improve the approximation. Assume we sample a random set \( A \) of \( \sqrt{n} \) vertices, then each vertex has a vertex from \( A \) among its \( k = \sqrt{n} \log n \) closest vertices w.h.p. Now each vertex learns its \((k, t)\)-nearest vertices, which are the \( k \) closest vertices of distance at most \( t \). For any vertex \( u \), there are 2 cases, either the \((k, t)\)-nearest vertices to \( u \) contain its entire \( t \)-neighbourhood, in which case, \( u \) already knows all the distances to vertices at distance at most \( t \), or there are at least \( k \) vertices in the \( t \)-neighbourhood of \( u \), in which case there is also a vertex from \( A \) there, \( p_A(u) \). Now, for a pair of vertices \( u, v \) of distance at most \( t \), if \( v \) is in the \((k, t)\)-closest vertices to \( u \), we are done. Otherwise, \( d(u, p_A(u)) \leq d(u, v) \), which gives

\[ d(u, p_A(u)) + d(p_A(u), v) \leq d(u, v) + d(p_A(u), u) + d(u, v) \leq 3d(u, v) \]

Hence, if we compute the distance from \( u \) to \( v \) through \( p_A(u) \) we get a \((3 + \epsilon)\)-approximation for the distance. To do so, we let all vertices learn approximate distances to all vertices in \( A \) at distance at most \( 2t \), this can be implemented efficiently using the bounded hopset and source detection algorithms. Since we only approximate the distances to \( A \), this results in a \((3 + \epsilon)\)-approximation.

Obtaining a better approximation of \((2 + \epsilon)\) requires several changes to the algorithm and analysis. At a high-level the algorithm starts by dealing with paths that have at least one high-degree vertex of degree at least \( \sqrt{n} \log n \). For such paths it is relatively easy to find a good approximation using hitting set arguments. Then, we are left with a sparser graph of size \( O(n^{3/2}) \). In this graph, we want to implement an algorithm similar to the \((3 + \epsilon)\)-approximation described above, compute a random set of vertices \( A \), compute the \((k, t)\)-nearest vertices and compute distances to close by sources in \( A \). However, since now we work only in a sparse graph, we can afford computing distances to a larger set \( A \) of size around \( n^{3/4} \), which allows focusing on \( k \) around \( n^{1/4} \). We show that we can exploit this sparsity and get a better approximation in this case using matrix multiplication of 3 sparse matrices. This approach generally follows [2], however following the algorithm described there directly adds some logarithmic factors when we multiply the 3 matrices, and to avoid them we need to sparsify the graph even further. In [9], we describe the algorithm in detail, and show the following.

**Theorem 28.** Let \( 0 < \epsilon < 1 \), there is a randomized \((2 + \epsilon)\)-approximation algorithm for unweighted undirected APSP in the Congested Clique model that takes \( O(\frac{\log \beta}{\epsilon}) \) rounds w.h.p. where \( \beta = O\left(\frac{\log \log n}{\epsilon}\right)\log \log n \).
5 DETERMINISTIC ALGORITHMS

We next explain how to derandomize our algorithms. Most of the randomized parts in our algorithms are based on hitting set arguments, which can be derandomized easily by using Lemma 9 instead of Lemma 8. This adds $O((\log \log n)^3)$ term to the complexity. However, derandomizing the emulator algorithm requires a more careful process. Intuitively, using hitting set arguments directly to derandomize the process would add a logarithmic factor to the size of the emulator, which leads to additional logarithmic term in the complexity of our shortest paths algorithms, which is too expensive. To avoid it, we introduce the soft hitting set problem, a new variant of the hitting set problem that captures more accurately the randomized process required in the emulator construction. This problem is defined as follows.

For a given two subsets of vertices $V_1, V_2$, define the soft-hitting set function $SH(V_1, V_2)$ by

$$SH(V_1, V_2) = \begin{cases} 0, & \text{if } V_1 \cap V_2 = \emptyset \\ |V_1|, & \text{otherwise.} \end{cases}$$

**Definition 29 (Soft Hitting Set).** Consider a graph $G = (V, E)$ with two special sets of vertices $L \subseteq V$ and $R \subseteq V$ with the following properties: each vertex $v \in L$ has a subset $S_v \subseteq R$ where $|S_v| \geq \Delta$. A set of vertices $R' \subseteq R$ is soft hitting set for $L$ if: (i) $|R'| = O(|R|/\Delta)$ and (ii) $\sum_{u \in L} SH(S_u, R') = O(|L| \cdot \Delta)$.

The above definition can be viewed as an adaption of the hitting-set definition by Ghaffari and Kuhn in [15]. In [9], we show:

**Lemma 30 (Det. Construction of Soft Hitting Sets).** Let $L, R \subseteq V$ be subsets of vertices where each vertex $u \in L$ knows a set $S_u \subseteq R$ of at least $\Delta$ vertices. There exists an $O((\log \log n)^3)$-round deterministic algorithm in the CONGESTED CLIQUE model that computes a soft-hitting set $R' \subseteq R$ for $L$, where $|R'| \leq c|L|/\Delta$ for a constant $c$.

Using soft hitting sets leads to deterministic construction of emulators with essentially the same parameters as the randomized construction, in the cost of additional $O((\log \log n)^3)$ term to the complexity. This comes from the fact that in each one of the $O(\log \log n)$ iterations of the algorithm, we construct $S_i$ a soft hitting set which takes $O((\log \log n)^3)$ time. Using the deterministic emulator we get deterministic variants for all our applications, with a cost of additional $O((\log \log n)^3)$ term in the complexity. For full details see [9].

Acknowledgments. We would like to thank Keren Censor-Hillel for many fruitful discussions. The research is supported in part by the Israel Science Foundation (grant no. 1696/14 and 2084/18), the European Union’s Horizon 2020 Research and Innovation Program under grant agreement no. 755839, and the Minerva grant no. 713238.

REFERENCES

[1] Ruben Becker, Andreas K venebauer, Sebastian Kinninger, and Christoph Lenzen. 2017. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. In 31st International Symposium on Distributed Computing.

[2] 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 2019.

[3] Keren Censor-Hillel, Petteri Kaski, Janne H Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. 2019. Algebraic methods in the congested clique. Distributed Computing 32, 6 (2019), 661–678.

[4] Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner. 2019. Sparse matrix multiplication and triangle listing in the congested clique model. Theoretical Computer Science (2019).

[5] Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. 2017. Derandomizing Local Distributed Algorithms under Bandwidth Restrictions. In 31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria. 11:1–11:16.

[6] Shiri Chechik and Doron Malkhiar. 2019. Reachability and Shortest Paths in the Broadcast CONGEST Model. In 33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary. 11:1–11:13.

[7] 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.

[8] Dorit Dor, Shay Halperin, and Uri Zwick. 2000. All-pairs almost shortest paths. SIAM J. Comput. 29, 5 (2000), 1740–1759.

[9] Michal Dory and Merav Parter. 2020. Exponentially Faster Shortest Paths in the Congested Clique. CoRR abs/2003.03058 (2020).

[10] Michael Elkin and Shaked Matar. 2019. Near-additive spanners in low polynomial deterministic CONGEST time. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC). 531–540.

[11] Michael Elkin and Ofer Neiman. 2018. Efficient algorithms for constructing very sparse spanners and emulators. ACM Transactions on Algorithms (TALG) 15, 1 (2018), 1–29.

[12] Michael Elkin and Ofer Neiman. 2019. Hopsets with constant hopbound, and applications to approximate shortest paths. SIAM J. Comput. 48, 4 (2019), 1436–1480.

[13] Michael Elkin and Ofer Neiman. 2019. Linear-size hopsets with small hopbound, and constant-hopbound hopsets in RNC. In The 31st ACM Symposium on Parallelism in Algorithms and Architectures. 333–341.

[14] Michael Elkin and Ofer Neiman. 2020. Near-Additive Spanners and Near-Exact Hopsets, A Unified View. arXiv:2001.07477 [cs.DS].

[15] Mohsen Ghaffari and Fabian Kuhn. 2018. Derandomizing distributed algorithms with small messages: Spanners and dominating set. In 32nd International Symposium on Distributed Computing (DISC 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[16] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. 2019. Conditional hardness results for massively parallel computation from distributed lower bounds. In 2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS). IEEE, 1650–1663.

[17] Parikshit Gopalan, Raghu Meka, Omer Reingold, Luca Trevisan, and Salil Vadhan. 2012. Better pseudorandom generators from milder pseudorandom restrictions. In 2012 IEEE 53rd Annual Symposium on Foundations of Computer Science. IEEE, 120–129.

[18] James W Hegeman and Sriram V Pemmaraju. 2015. Lessons from the congested clique model and applications to graph-theoretic problems. In 39th International Conference on Parallel Processing Workshops (ICPPW 2010). IEEE Computer Society.

[19] Stephan Holzer and Nathan Pinsker. 2016. Approximation of Distances and Sparse Spanners and Emulators. In 2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS). IEEE, 682–689.

[20] James Hegeman and Sriram Pemmaraju. 2015. Lessons from the congested clique model and applications to graph-theoretic problems. In 39th International Conference on Parallel Processing Workshops (ICPPW 2010). IEEE Computer Society.

[21] Noam Nisan and Avi Wigderson. 1994. Hardness vs Randomness. J. Comput. System Sci. 49, 2 (1994), 149–167.

[22] Merav Parter and Eylon Yogev. 2018. Congested Clique Algorithms for Graph Spanners. In 32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018. 401–408.

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

[24] Mikkel Thorup and Uri Zwick. 2006. Spanners and emulators with sublinear distance errors. In Proc. SODA 2006. 802–809.