Mind the $\tilde{\mathcal{O}}$: asymptotically better, but still impractical, quantum distributed algorithms

Phillip A. Kerger[1,2,3], David E. Bernal Neira[2,3,4], Zoe Gonzalez Izquierdo[2,3], and Eleanor G. Rieffel[3]

[1]Department of Applied Mathematics and Statistics, Johns Hopkins University
[2]Research Institute of Advanced Computer Science, USRA
[3]Quantum Artificial Intelligence Laboratory, NASA Ames Research Center
[4]Davidson School of Chemical Engineering, Purdue University

June 15, 2023

Abstract

The CONGEST and CONGEST-CLIQUE models have been carefully studied to represent situations where the communication bandwidth between processors in a network is severely limited. Messages of only $\mathcal{O}(\log(n))$ bits of information each may be sent between processors in each round. The quantum versions of these models allow the processors instead to communicate and compute with quantum bits under the same bandwidth limitations. This leads to the following natural research question: What problems can be solved more efficiently in these quantum models than in the classical ones? Building on existing work, we contribute to this question in two ways. Firstly, we present two algorithms in the Quantum CONGEST-CLIQUE model of distributed computation that succeed with high probability; one for producing an approximately optimal Steiner Tree, and one for producing an exact directed minimum spanning tree, each of which uses $\tilde{\mathcal{O}}(n^{1/4})$ rounds of communication and $\tilde{\mathcal{O}}(n^{9/4})$ messages, where n is the number of nodes in the network. The algorithms thus achieve a lower asymptotic round and message complexity than any known algorithms in the classical CONGEST-CLIQUE model. At a high level, we achieve these results by combining classical algorithmic frameworks with quantum subroutines. An existing framework for using a distributed version of Grover's search algorithm to accelerate triangle finding lies at the core of the asymptotic speedup. Secondly, we carefully characterize the constants and logarithmic factors involved in our algorithms as well as related algorithms, otherwise commonly obscured by $\tilde{O}$ notation. The analysis shows that some improvements are needed to render both our and existing related quantum and classical algorithms practical, as their asymptotic speedups only help for very large values of n.

Keywords— Quantum Computing, Distributed Computing, Steiner Tree, Directed Minimum Spanning Tree

1 Introduction

The classical CONGEST-CLIQUE Model (cCCM henceforth) in distributed computing has been carefully studied as a model central to the field, e.g., (Korhonen & Suomela, 2017; Saikia & Karmakar, 2019; Fischer & Oshman, 2021; Lenzen, 2012; Dolev, Lenzen, & Peled, 2012; Nowicki, 2019). In this model, processors in a network solve a problem whose input is distributed across the nodes under significant communication limitations, described in detail in §2. For example, a network of aircraft or spacecraft, satellites, and control stations, all with large distances between them, may have severely limited communication bandwidth to be modeled in such a way. The quantum version of this model, in which quantum bits can be sent between processors, the quantum CONGEST-CLIQUE Model (qCCM), as well as the quantum CONGEST model, have been the subject of recent research (Izumi & Gall, 2019; Censor-Hillel, Fischer, Le Gall, Leitersdorf, & Oshman, 2022; van Apeldoorn & de Vos, 2022; Elkin, Klauck, Nanongkai, &

Pandurangan, 2012) in an effort to understand how quantum communication may help in these distributed computing frameworks. For the quantum CONGEST Model, however, (Elkin et al., 2012) showed that many problems cannot be solved more quickly than in the classical model. These include shortest paths, minimum spanning trees, Steiner trees, min-cut, and more; the computational advantages of quantum communication are thus severely limited in the CONGEST setting, though a notable positive result is sub-linear diameter computation in (Le Gall & Magniez, 2018). No comparable negative results exist for the qCCM, and in fact, (Izumi & Gall, 2019) provides an asymptotic quantum speedup for computing all-pairs shortest path (APSP henceforth) distances. Hence, it is apparent that the negative results of (Elkin et al., 2012) cannot transfer over to the qCCM, so investigating these problems in the qCCM presents an opportunity for contribution to the understanding of how quantum communication may help in these distributed computing frameworks. In this paper, we contribute to this understanding by formulating algorithms in the qCCM for finding approximately optimal Steiner trees and exact directed minimum spanning trees using $\tilde{\mathcal{O}}(n^{1/4})$ rounds – asymptotically fewer rounds than any known classical algorithms. This is done by augmenting the APSP algorithm of (Izumi & Gall, 2019) with an efficient routing table scheme, which is necessary to make use of the shortest *paths* information instead of only the APSP *distances*, and using the resulting subroutine with existing classical algorithmic frameworks. Beyond asymptotics, we also characterize the complexity of our algorithms as well as those of (Izumi & Gall, 2019; Censor-Hillel et al., 2016; Saikia & Karmakar, 2019; Fischer & Oshman, 2021) to include the logarithmic and constant factors involved to estimate the scales at which they would be practical, which was not included in the previous work. It should be noted that, like APSP, these problems cannot see quantum speedups in the CONGEST (non-clique) setting as shown in (Elkin et al., 2012). Our Steiner tree algorithm is approximate and based on a classical polynomial-time centralized algorithm of (Kou, Markowsky, & Berman, 1981). Our directed minimum spanning tree problem algorithm follows an approach similar to (Fischer & Oshman, 2021), which effectively has its centralized roots in (Lovasz, 1985).

2 Background and Setting

This section provides the necessary background for our algorithms' settings and the problems they solve.

2.1 The CONGEST and CONGEST-CLIQUE Models of Distributed Computing

In the standard CONGEST model, we consider a graph of n processor nodes whose edges represent communication channels. Initially, each node knows only its neighbors in the graph and associated edge weights. In rounds, each processor node executes computation locally and then communicates with its neighbors before executing further local computation. The congestion limitation restricts this communication, with each node able to send only one message of $\mathcal{O}(\log(n))$ classical bits in each round to its neighbors, though the messages to each neighbor may differ. In the cCCM, we separate the communication graph from the problem input graph by allowing all nodes to communicate with each other, though the same $\mathcal{O}(\log(n))$ bits-per-message congestion limitation remains. Hence, a processor node could send $n-1$ different messages to the other $n-1$ nodes in the graph, with a single node distributing up to $\mathcal{O}(n \cdot \log(n))$ bits of information in a single round. Taking advantage of this way of dispersing information to the network is paramount in many efficient CONGEST-CLIQUE algorithms. The efficiency of algorithms in these distributed models is commonly measured in terms of the *round complexity*, the number of rounds of communication used in an algorithm to solve the problem in question. A good overview of these distributed models can be found in (Ghaffari, 2020).

2.2 Quantum Versions of CONGEST and CONGEST-CLIQUE

The quantum models we work in are obtained via the following modification: Instead of restricting to messages of $\mathcal{O}(\log(n))$ classical bits, we allow messages to consist of $\mathcal{O}(\log(n))$ quantum bits, qubits. For background on qubits and the fundamentals of quantum computing, we refer the reader to (Rieffel & Polak, 2011). We formally define the qCCM, the setting for our algorithms, as follows:

Definition 2.1 (Quantum CONGEST-CLIQUE)**.** The Quantum CONGEST-CLIQUE Model (qCCM) is a distributed computation model in which an input graph $G = (V, E, W)$ is distributed over a network of n processors, where each processor is represented by a node in V. Each node is assigned a unique ID number in $[n]$. Time passes in *rounds*, each of which consists of the following:

1. Each node may execute unlimited local computation.
2. Each node may send a message consisting of either a register of $\mathcal{O}(\log n)$ qubits or a string of $\mathcal{O}(\log n)$ classical bits to each other node in the network. Each of those messages may be distinct.
3. Each node receives and saves the messages the other nodes send it.

The input graph G is distributed across the nodes as follows: Each node knows its own ID number, the ID numbers of its neighbors in G, the number of nodes n in G, and the weights corresponding to the edges it is incident upon. The output solution to a problem must be given by having each node $v \in V$ return the restriction of the global output to $\mathcal{N}_G(u) := \{v : uv \in E\}$, its neighborhood in G. No entanglement is shared across nodes initially.

This is an analog of the cCCM, except that quantum bits may be sent in place of classical bits. To clarify the output requirement, in the Steiner tree problem, we require node u to output the edges of the solution tree that are incident upon u. Since many messages in our algorithms need not be sent as qubits, we define the qCCM slightly unconventionally, allowing either quantum or classical bits to be sent. We specify those that may be sent classically. However, even without this modification, the quantum versions of CONGEST and cCCM are at least as powerful as their classical counterparts. This is because any n-bit classical message can be instead sent as an n-qubit message of unentangled qubits; for a classical bit reading 0 or 1, we can send a qubit in the state $|0\rangle$ or $|1\rangle$ respectively, and then take measurements with respect to the $\{|0\rangle, |1\rangle\}$ basis to read the same message the classical bits would have communicated. Hence, one can also freely make use of existing classical algorithms in the qCCM. Further, the assumption that IDs are in $[n]$, with n known, is not necessary but is convenient; without this assumption, we could have all nodes broadcast their IDs to the entire network and then assign a new label in $[n]$ to each node according to an ordering of the original IDs, resulting in our assumed situation.

Remark 2.2. Definition 2.1 does not account for how the information needs to be stored. In this paper, it suffices for all information regarding the input graph to be stored classically as long as there is quantum access to that data. We provide some details on this in §8.4 of the appendix.

Remark 2.3. No entanglement being shared across nodes initially in definition 2.1 results in quantum teleportation not being a trivial way to solve problems in the qCCM.

Example 2.4. To provide some intuition on how allowing communication through qubits in this distributed setting can be helpful, we now describe and give an example of distributed Grover search, first described in (Le Gall & Magniez, 2018). The high-level intuition for why quantum computing gives an advantage for search is that quantum operations use quantum interference effects to have canceling effects among non-solutions. Grover search has a generalization called "amplitude amplification" we will use; see (Rieffel & Polak, 2011) for details on these algorithms. Now, for a processor node u in the network and a Boolean function $g : X \to \{0,1\}$, suppose there exists a classical procedure $\mathcal{C}$ in the cCCM that allows u to compute $g(x)$, for any $x \in X$ in r rounds. The quantum speedup will come from computing $\mathcal{C}$ in a quantum superposition, which enables g to be evaluated with inputs in superposition so that amplitude amplification can be used for inputs to g. Let $A_i : \{x \in X : g(x) = i\}$, for $i = 0, 1$, and suppose that $0 < |A_1| \leq |X|/2$. Then classically, node u can find an $x \in A_1$ in $\Theta(r|X|)$ rounds by checking each element of X. Using the quantum distributed Grover search of (Le Gall & Magniez, 2018) enables u to find such an x with high probability in only $\tilde{\mathcal{O}}(r\sqrt{|X|})$ rounds by evaluating the result of computing g on a superposition of inputs.

We illustrate this procedure in an example case where a node u wants to inquire whether one of its edges uv is part of a triangle in G. We first describe a classical procedure for this, followed by the corresponding quantum-distributed search version. For $v \in \mathcal{N}_G(u)$, denote by $\mathcal{I}_v : V \to \{0,1\}$ the indicator function of $\mathcal{N}_G(v)$, and by $g_{uv} : \mathcal{N}_G(u) \to \{0,1\}$ its restriction to inputs in $\mathcal{N}_G(u)$. Classically, node u can evaluate $g_{uv}(w)$ in two rounds for any $w \in \mathcal{N}_G(u)$ by sending the ID of w (of length $\log n$) to v, and having v send back the answer $\mathcal{I}_v(w)$. Then u can check $g_{uv}(w)$ for each $w \in \mathcal{N}_G(u)$ one at a time to determine whether uv is part of a triangle in G or not in $2 \cdot |\mathcal{N}_G(u)|$ rounds.

For the distributed quantum implementation, u can instead initialize a register of $\log n$ qubits as $|\psi\rangle_0 := \frac{1}{\sqrt{|\mathcal{N}_G(u)|}} \sum_{x \in \mathcal{N}_G(u)} |x\rangle$, all the inputs for g_{uv} in equal superposition. To do a Grover search, u needs to be able to evaluate g_{uv} with inputs $|\psi\rangle$ in superposition. For the quantum implementation of $\mathcal{C}$, u sends a quantum register in state $|\psi\rangle|0\rangle$ to node v, and has node v evaluate a quantum implementation of $\mathcal{I}_v$, which we will consider as a call to an oracle mapping $|x\rangle|0\rangle$ to $|x\rangle|\mathcal{I}_v(x)\rangle$ for all $x \in V$. Node v sends back the resulting qubit register, and node u has evaluated $g_{uv}(|\psi\rangle)$ in 2 rounds. Now, since u can evaluate g_{uv} in superposition, node u may proceed using standard amplitude amplification, using 2 rounds of communication for each evaluation of g_{uv}, so that u can find an element $w \in \mathcal{N}_G(u)$ satisfying $g_{uv}(w) = 1$ with high probability in $\tilde{\mathcal{O}}(r\sqrt{|\mathcal{N}_G(u)|})$ rounds if one exists. We note that in this example, v cannot execute this procedure by itself since it does not know $\mathcal{N}_G(u)$ (and sending this information to v would take $|\mathcal{N}_G(u)|$ rounds), though it is able to evaluate $\mathcal{I}_v$ in superposition for any $w \in \mathcal{N}_G(u)$. For any classical procedure $\mathcal{C}$ evaluating a different function from this specific g (that can be implemented efficiently classically and, therefore, translated to an efficient quantum implementation), the same idea results in the square-root advantage to find a desired element such that g evaluates to 1.

2.3 Notation and Problem Definitions

For an integer-weighted graph $G = (V, E, W)$, we will denote $n := |V|, m := |E|$, and W_e the weight of an edge $e \in E$ throughout the paper. Let $\delta(v) \subset V$ be the set of edges incident on node v, and $\mathcal{N}_G(u) := \{v : uv \in E\}$ the neighborhood of $u \in G$. Denote by $d_G(u, v)$ the shortest-path distance in G from u to v. For a graph $G = (V, E, W)$ two sets of nodes U and U', let $\mathcal{P}_G(U, U') := \{uv \in E : u \in U, w \in U'\}$ be the set of edges connecting U to U'. Let $\mathcal{P}(U) := \mathcal{P}(U, U)$ as shorthand. All logarithms will be taken with respect to base 2, unless otherwise stated.

Definition 2.5 (Steiner Tree Problem)**.** Given a weighted, undirected graph $G = (V, E, W)$, and a set of nodes $\mathcal{Z} \subset V$, referred to as *Steiner Terminals*, output the minimum weight tree in G that contains $\mathcal{Z}$.

Definition 2.6 (Approximate Steiner Tree)**.** For a Steiner Tree Problem with terminals $\mathcal{Z}$ and solution $\mathcal{S}_{OPT}$ with edge set $E_{\mathcal{S}_{OPT}}$, a tree T in G containing $\mathcal{Z}$ with edge set E_T such that

$$\sum_{uv \in E_T} W_{uv} \leq r \cdot \sum_{uv \in E_{\mathcal{S}_{OPT}}} W_{uv}$$

is called an approximate Steiner Tree with approximation factor r.

Definition 2.7 (Directed Minimum Spanning Tree Problem (DMST))**.** Given a directed, weighted graph $G = (V, E, W)$ and a root node $r \in V$, output the minimum weight directed spanning tree for G rooted at r. This is also known as the *minimum weight arborescence* problem.

3 Contributions

We provide an algorithm for the qCCM that produces an approximate Steiner Tree with high probability (w.h.p.) in $\tilde{\mathcal{O}}(n^{1/4})$ rounds and an algorithm that produces an exact Directed Minimum Spanning Tree w.h.p. in $\tilde{\mathcal{O}}(n^{1/4})$ rounds. To do this, we enhance the quantum APSP algorithm of (Izumi & Gall, 2019) in an efficient way to compute not only APSP distances but also the corresponding routing tables (described in §4) that our algorithms rely on. Further, in addition to these $\tilde{\mathcal{O}}$ results, in sections 4.7, 5.4, and 6.3, we characterize the constants and logarithmic factors involved in our algorithms as well as related classical algorithms to contribute to the community's understanding of their implementability. This reveals that the factors commonly obscured by $\tilde{\mathcal{O}}$ notation in related literature, especially the logarithms, have a severe impact on practicality.

We summarize the algorithmic results in the following two theorems:

Theorem 3.1. *There exists an algorithm in the Quantum CONGEST-CLIQUE model that, given an integer-weighted input graph $G = (V, E, W)$, outputs a $2(1 - 1/l)$ approximate Steiner Tree with probability of at least $1 - \frac{1}{poly(n)}$, and uses $\tilde{\mathcal{O}}(n^{1/4})$ rounds of computation, where l denotes the number of terminal leaf nodes in the optimal Steiner Tree.*

Theorem 3.2. *There exists an algorithm in the Quantum CONGEST-CLIQUE model that, given a directed and integer-weighted input graph $G = (V, E, W)$, produces an exact Directed Minimum Spanning Tree with high probability, of at least $1 - \frac{1}{poly(n)}$, and uses $\tilde{\mathcal{O}}(n^{1/4})$ rounds of computation.*

4 APSP and Routing Tables

We first describe an algorithm for the APSP problem with routing tables in the qCCM, for which we combine an algorithm of (Izumi & Gall, 2019) with a routing table computation from (Zwick, 2000). For this, we reduce APSP with routing tables to triangle finding via *distance products* as in (Censor-Hillel et al., 2016).

4.1 Distance Products and Routing Tables

Definition 4.1. A *routing table* for a node v is a function $R_v : V \to V$ mapping a vertex u to the first node visited in the shortest path going from v to u other than v itself.

Definition 4.2. The *distance product* between two $n \times n$ matrices A and B is defined as the $n \times n$ matrix $A \star B$ with entries:

$$(A \star B)_{ij} = \min_k \{A_{ik} + B_{kj}\}. \tag{4.1}$$

The distance product is also sometimes called the min-plus or tropical product. For shortest paths, we will repeatedly square the graph adjacency matrix with respect to the distance product. For a $n \times n$ matrix W and an integer k, let us denote $W^{k,\star} := W \star (W \star (\dots (W \star W)) \dots)$ as the k^{th} power of the distance product. For a graph $G = (V, E, W)$ with weighted adjacency matrix W (assigning $W_{uv} = \infty$ if $uv \notin E$), $W_{uv}^{k,\star}$ is the length of the shortest path from v to u in G using at most k hops. Hence, for any $N \geq n$, $W^{N,\star}$ contains all the shortest path distances between nodes in G. As these distance products obey standard exponent rules, we may take $N = 2^{\lceil \log n \rceil}$ to recursively compute the APSP distances via taking $\lceil \log n \rceil$ distance product squares:

$$W^{2,\star} = W \star W, \;\; W^{4,\star} = \left(W^{2,\star}\right)^{2,\star}, \dots, \;\; W^{2^{\lceil \log n \rceil},\star} = \left(W^{2^{\lceil \log n \rceil - 1},\star}\right)^{2,\star}. \tag{4.2}$$

This procedure reduces computing APSP distances to computing $\lceil \log n \rceil$ distance products. In the context of the CONGEST-CLIQUE model, each node needs to learn the row of W^n that represents it. As we also require nodes to learn their routing tables, we provide a scheme in §4.3 that is well-suited for our setting to extend (Izumi & Gall, 2019) to also compute routing tables.

4.2 Distance Products via Triangle Finding

Having established reductions to distance products, we turn to their efficient computation. The main idea is that we can reduce distance products to a binary search in which each step in the search finds negative triangles. This procedure corresponds to (Izumi, Le Gall, & Magniez, 2020, Proposition 2), which we describe here, restricting to finding the distance product square needed for Eq. (4.2).

A negative triangle in a weighted graph is a set of edges $\Delta^- = (uv, vw, wu) \subset E^3$ such that $\sum_{e \in \Delta^-} W_e < 0$. Let us denote the set of all negative triangles in a graph G as Δ_G^-. Specifically, we will be interested in each node v being able to output edges $vu \in \delta(v)$ such that vu is involved in at least one negative triangle in G. Let us call this problem `FindEdges`, and define it formally as:

`FindEdges`

Input: An integer-weighted (directed or undirected) graph $G = (V, E, W)$ distributed among the nodes, with each node v knowing $\mathcal{N}_G(v)$, as well as the weights W_{vu} for each $u \in \mathcal{N}_G(v)$.

Output: For each node v, its output is all the edges $vu \in E$ that are involved in at least one negative triangle in G.

Proposition 4.3. If `FindEdges` on a n-node integer-weighted graph $G = (V, E, W)$ can be solved in $T(n)$ rounds, then the distance product $A \star B$ of two $n \times n$ matrices A and B with entries in $[M]$ can be computed in $T(3n) \cdot \lceil \log_2(2M) \rceil$ rounds.

Proof. Let A and B be arbitrary $n \times n$ integer-valued matrices, and D be an $n \times n$ matrix initialized to $\mathbf{0}$. Let each $u \in V$ simulate three copies of itself,u_1, u_2, u_3, writing V_1, V_2, V_3 as the sets of copies of nodes in V. Consider the graph $G' = (V_1 \cup V_2 \cup V_3, E', W')$, by letting $u_i v_j \in E'$ for $u_i \in V_i, v_j \in V_j, i \neq j$, taking $W'_{u_1 v_2} = A_{uv}$ for $u_1 \in V_1, v_2 \in V_2$, $W'_{u_2 v_3} = B_{uv}$ for $u_2 \in V_2, v_3 \in V_3$, and $W'_{u_3 v_1} = D_{uv}$ for $u_3 \in V_3, v_1 \in V_1$. An edge zv is part of a negative triangle in G' exactly whenever

$$\min_{u \in V} \{A_{vu} + B_{uz}\} < -D_{zv}.$$

Assuming we can compute `FindEdges` for a k-node graph in $T(n)$ rounds, with a non-positive matrix $D = \mathbf{0}$ initialized we can apply simultaneous binary searches on D_{zv}, with values between $\{-2M, 0\}$, updating it for each node v after each run of `FindEdges` to find $\min_{u \in V}\{A_{vu} + B_{uz}\}$ for every other node z in $T(3n) \cdot \lceil \log(\max_{v,z \in V}\{\min_{u \in V}\{A_{vu} + B_{uv}\}\}) \rceil$ rounds, since G' is a tripartite graph with $3n$ nodes. □

Remark 4.4. This procedure can be realized in a single n-node distributed graph by letting each node represent the three copies of itself since G' is tripartite. The $T(3n)$ stems from each processor node possibly needing to send one message for each node it is simulating in each round of `FindEdges`. If bandwidth per message is large enough (3 times the bandwidth needed for solving `FindEdges` in $T(n)$ rounds), then this can be done in $T(n)$ rounds.

So for this binary search, each node v initializes and locally stores $D_{vz} = 0$ for each other $z \in V$, after which we solve `FindEdges` on G'. The node then updates each D_{vz} according to whether or not the edge copies of vz were part of a negative triangle in G', after which `FindEdges` is computed with the updated values for D. This is repeated until all the $\min_{u\in V}\{A_{vu} + B_{uz}\}$ have been determined.

4.3 Routing Tables via Efficient Computation of Witness Matrices

For the routing table entries, we also need each node v to know the intermediate node u that is being used to attain $\min_{u\in V}\{W_{vu} + W_{uz}\}$.

Definition 4.5. For a distance product $A \star B$ of two $n \times n$ matrices A, B, a *witness matrix* C is an $n \times n$ matrix such that

$$C_{ij} \in argmin_{k\in[n]}\{A_{ik} + B_{kj}\}$$

Put simply, a witness matrix contains the intermediate entries used to attain the values in the resulting distance product. We present here a simple way of computing witness matrices along with the distance product by modifying the matrix entries appropriately, first considered by (Zwick, 2000). The approach is well-suited for our algorithm, as we only incur $\mathcal{O}(\log n)$ additional calls to `FindEdges` for a distance product computation with a witness matrix.

For an $n \times n$ integer matrix W, obtain matrices W' and W'' by taking $W'_{ij} = nW_{ij} + j - 1$ and $W''_{ji} = nW_{ji}$. Set $K = W' \star W''$.

Claim 1. With W, W', W'', and K as defined immediately above,

(i) $\left\lfloor \dfrac{K}{n} \right\rfloor = W^{2,\star}$

(ii) $(K \mod n) + 1$ is a witness matrix for $W^{2,\star}$.

The claim follows from routine calculations of the quantities involved and can be found in the Appendix, §8.1.

Hence, we can obtain witness matrices by simply changing the entries of our matrices by no more than a multiplicative factor of n and an addition of n. Since the complexity of our method depends on the magnitude of the entries of W logarithmically, we only need logarithmically many more calls to `FindEdges` to obtain witness matrices along with the distance products, making this simple method well-suited for our approach. More precisely, we can compute W^2 with a witness matrix using $\lceil \log\big(2n \cdot \max_{i,j}\{W^2_{ij} < \infty\}\big)\rceil$. calls to `FindEdges`. We obtain the following corollary to proposition 4.3 to characterize the exact number of rounds needed:

Corollary 4.6. If `FindEdges` on an n-node integer-weighted graph $G = (V, E, W)$ can be solved in $T(n)$ rounds, then the distance product square $W^{2,\star}$, along with a witness matrix H, can be computed in $T(3n) \cdot \lceil \log_2(n \cdot \max_{v,z\in G}\{\min_{u\in V}\{W_{vu} + W_{uv}\}\} + n)\rceil$ rounds.

Proof. This follows from claim 1 and proposition 4.3 upon observing that
$\max_{v\in V}\{\min_{u\in V}\{W'_{vu} + W''_{uv})\}\} \le n \cdot \max_{v,z\in G}\{\min_{u\in V}\{W_{vu} + W_{uv}\}\} + n$. □

Once we obtain witness matrices along with the distance product computations, constructing the routing tables for each node along the way of computing APSP is straightforward. In each squaring of W in Eq. (4.2), each node updates its routing table entries according to the corresponding witness matrix entry observed. It is worth noting that these routing table entries need only be stored and accessed classically so that we avoid using unnecessary quantum data storage.

4.4 Triangle Finding

Given the results from sections 4.3 and 4.2, we have reduced finding both the routing tables and distance product to having each edge learn the edges involved in a negative triangle in the graph. This section will thus describe the procedure to solve the `FindEdges` subroutine. We state here a central result from (Izumi & Gall, 2019):

Proposition 4.7. There exists an algorithm in the quantum CONGEST-CLIQUE model that solves the `FindEdges` subroutine in $\tilde{\mathcal{O}}(n^{1/4})$ rounds.

We will proceed to describe each step of the algorithm to describe the precise round complexity beyond the $\tilde{O}(n^{1/4})$ to characterize the constants involved in the interest of assessing the future implementability of our algorithms.

As a preliminary, we give a message routing lemma of (Dolev et al., 2012) for the congested clique, which will be used repeatedly:

Lemma 4.8. Suppose each node in G is the source and destination for at most n messages of size $\mathcal{O}(\log n)$ and that the sources and destinations of each message are known in advance to all nodes. Then all messages can be routed to their destinations in 2 rounds.

We introduce the subproblem `FindEdgesWithPromise` (`FEWP` henceforth). Let $\Gamma(u, v)$ denote the number of nodes $w \in V$ such that (u, v, w) forms a negative triangle in G.

`FEWP`

Input: An integer-weighted graph $G = (V, E, W)$ distributed among the nodes and a set $S \subset \mathcal{P}(V)$, with each node v knowing $\mathcal{N}_G(v)$ and S.

Promise: For each $uv \in S, \Gamma(u, v) \leq 90 \log n$.

Output: For each node v, its output is the edges $vu \in S$ that satisfy $\Gamma(u, v) > 0$.

We give here a description of the procedure of (Izumi & Gall, 2019) to solve `FindEdges` given an algorithm $\mathcal{A}$ to solve `FEWP`. Let $\varepsilon_{\mathcal{A}}$ be the failure probability of the algorithm $\mathcal{A}$ for an instance of `FEWP`.

```
FindEdgesViaFEWP
  1: $S := \mathcal{P}; M := \emptyset; i := 0.$
  2: WHILE $60 \cdot 2^i \log n \leq n$:
       a): Each node samples each of its edges with probability $\sqrt{\frac{60 \cdot 2^i \log n}{n}}$, so that we obtain a distributed
           subgraph $G'$ of $G$ consisting of the sampled edges
       b): Run $\mathcal{A}$ on $(G', S)$. Denote the output by $S'$.
       c): $S \leftarrow S \setminus S'; M \leftarrow M \cup S; i \leftarrow i + 1.$
  3: Run $\mathcal{A}$ on $(G, S)$, and call $S''$ the output.
  4: Output $M \cup S$.
```

From step 2 of this above algorithm, it is straightforward to check that this requires a maximum of $c_n := \lceil \log\left(\frac{n}{60 \log n}\right) \rceil + 1$ calls to the $\mathcal{A}$ subroutine to solve `FEWP`. Further, it succeeds with probability at least $1 - c_n/n^3 - c_n/n^2 8 - (c_n + 1)\varepsilon_{\mathcal{A}}$. We refer the reader to (Izumi & Gall, 2019, §3) for the proof of correctness. We now turn toward constructing an efficient algorithm for FEWP.

To solve this subroutine, we must first introduce an additional labeling scheme over the nodes that will determine how the search for negative triangles will be split up to avoid communication congestion in the network. Assume for simplicity that $n^{1/4}, \sqrt{n}, n^{3/4}$ are integers. Let $\mathcal{M} = [n^{1/4}] \times [n^{1/4}] \times [\sqrt{n}]$. Clearly, $|\mathcal{M}| = n$, and $\mathcal{M}$ admits a total ordering lexicographically. Since we assume each node $v_i \in V$ is labeled with unique integer ID $i \in [n]$, v_i can select the element in $\mathcal{M}$ that has place i in the lexicographic ordering of $\mathcal{M}$ without communication occurring. Hence, each node $v \in V$ is associated with a unique triple $(i, j, k) \in \mathcal{M}$. We will refer to the unique node associated with $(i, j, k) \in \mathcal{M}$ as node $v_{(i,j,k)}$.

The next ingredient is a partitioning scheme of the space of possible triangles. Let $\mathcal{U}$ be a partition of V into $n^{1/4}$

subsets containing $n^{3/4}$ nodes each, by taking $U_i := \{v_j : j \in \{(i-1) \cdot n^{3/4}, \dots, i \cdot n^{3/4}\}\}$ for $i = 1, \dots, n^{1/4}$, and $\mathcal{U} := \{U_1, \dots, U_{n^14}\}$. Apply the same idea to create a partition $\mathcal{U}'$ of $\sqrt{n}$ sets of size $\sqrt{n}$, by taking $U'_i := \{v_j : j \in \{(i-1) \cdot \sqrt{n}, \dots, i \cdot \sqrt{n}\}\}$ for $i = 1, \dots, \sqrt{n}$, and $\mathcal{U} := \{U_1, \dots, U_{\sqrt{n}}\}$. Let $\mathbb{V} = \mathcal{U} \times \mathcal{U} \times \mathcal{U}'$. Each node $v_{(i,j,k)}$ can then locally determine its association with the element $(U_i, U_j, U'_k) \in \mathbb{V}$ since $|\mathbb{V}| = n$. Further, if we use one round to have all nodes broadcast their IDs to all other nodes, each node $v_{(i,j,k)}$ can locally compute the (U_i, U_j, U'_k) it is assigned to, so this assignment can be done in one round.

We present here the algorithm ComputePairs used to solve the FEWP subroutine.

```
ComputePairs
Input: An integer-weighted graph $G = (V, E, W)$ distributed among the nodes, a partition of $V \times V \times V$ of $(U_i, U_j, U'_k)$ associated with each node as above, and a set $S \subset \mathcal{P}(V)$ such that for $uv \in S, \Gamma(u, v) \leq 90 \log n$.
Output: For each node $v$, its output is the edges $vu \in S$ that satisfy $\Gamma(u, v) > 0$.
  1: Every node $v_{(i,j,k)}$ receives the weights $W_{uv}$, $W_{vw}$ for all $uv \in \mathcal{P}(U_i, U_j)$ and $vw \in \mathcal{P}(U_j, U'_k)$.
  2: Every node $v_{(i,j,k)}$ constructs the set $\Lambda_k(U_i, U_j) \subset \mathcal{P}(U_i, U_j)$ by selecting every $uv \in \mathcal{P}(U_i, U_j)$ with probability $10 \cdot \frac{\log n}{\sqrt{n}}$. If $|\{v \in U_1 : uv \in \Lambda_k(U_i, U_j)\}| > 100n^{1/4} \log n$ for some $u \in U_j$, abort the algorithm and report failure. Otherwise, $v_{(i,j,k)}$ keeps all pairs $uv \in \Lambda_k(U_i, U_j) \cap S$ and receives the weights $Wuv$ for all of those pairs. Denote those elements of $\Lambda_k(U_i, U_j) \cap S$ as $u_1^k v_1^k, \ldots, u_m^k v_m^k$.
  3: Every node $v_{(i,j,k)}$ checks for each $l \in [m]$ whether there is some $U \in \mathcal{U}'$ that contains a node $w$ such that $(u_l^k, v_l^k, w)$ forms a negative triangle, and outputs all pairs $u_l^k v_l^k$ for which a negative triangle was found.
```

With probability at least $1 - 2/n$, the algorithm ComputePairs does not terminate at step 2 and every pair $(u, v) \in S$ appears in at least one $\Lambda_k(U_i, U_j)$. The details for this result can be found in (Izumi & Gall, 2019, Lemma 2).

Step 1 requires $2n^{1/4} \lceil \frac{\log W}{\log n} \rceil$ rounds and can be implemented fully classically without any qubit communication. Step 2 requires at most $200 \log n \lceil \frac{\log W}{\log n} \rceil$ rounds and can also be implemented classically. Step 3 can be implemented in $\tilde{\mathcal{O}}(n^{1/4})$ rounds quantumly taking advantage of distributed Grover search but would take $\mathcal{O}(\sqrt{n})$ steps to implement classically. The remainder of this section is devoted to illustrating how this step can be done in $\tilde{\mathcal{O}}(n^{1/4})$ rounds.

Define the following quantity:

Definition 4.9. For node $v_{(i,j,k)}$, let

$$\Delta(i, j, k) := \{(u, v) \in \mathcal{P}(U_i, U_j) \cap S : \exists w \in U'_k \textit{ with } (u, v, w) \textit{ forming a negative triangle in } G\}$$

For simultaneous quantum searches, we divide the nodes into different classes based on the number of negative triangles they are a part of with the following routine:

```
IdentifyClass
Input: An integer-weighted graph $G = (V, E, W)$ distributed among the nodes, and a set $S \subset E$ as in FEWP.
Output: For each node $v$, a class $\alpha$ the node belongs to.
  1: Every node $u_{(i,j,k)} \in V$ samples each node in $\{v \in V : (u_{(i,j,k)}, v) \in S\}$ with probability $\frac{10 \log n}{n}$, creating a set $\Lambda(u)$ of sampled vertices. If $\max_u |\Lambda(u)| > 20 \log n$, abort the algorithm and report a failure. Otherwise, have each node broadcast $\Lambda(u)$ to all other nodes, and take $R := \cup_{u \in V} \{uv | v \in \Lambda(u)\}$.
  2: Each $v_{(i,j,k)} \in V$ computes $d_{i,j,k} := |\{uv \in \mathcal{P}(i, j) \cap R : \ \exists w \in U'_k$ such that $\{u, v, w\}$ forms a negative triangle in $G\}|$, then determines its class $\alpha$ to be $min\{c \in \mathbb{N} : d_{i,j,k} < 10 \cdot 2^c \log n\}$.
```

This uses at most $20 \log n$ rounds (each node sends at most that many IDs to every other node) and can be implemented by having all exchanged messages consist only of classical bits. Using Chernoff's bound, one can show that the procedure succeeds with probability of at least $1 - 1/n$ as seen in (Izumi et al., 2020, Proposition 5).

Let us make the convenient assumption that $\alpha = 0$ for all $v_{i,j,k}$, which avoids some technicalities around congestion in the forthcoming triangle search. Note that $\alpha \leq \frac{1}{2} \log n$, so we can run successive searches for each α for nodes in with class α in the general case. The general case is discussed in §8.2 of the appendix and can also be found in (Izumi & Gall, 2019), but this case is sufficient to convey the central ideas.

We have all the necessary ingredients to describe the implementation of step 3 of the ComputePairs procedure.

```
  3.1: Each node executes the IdentifyClass procedure.
  3.2: For each $\alpha$, for every $l \in [m]$, every node $v_{(i,j,k)}$ in class $\alpha$ executes a quantum search to find whether there is a $U'_k \in \mathcal{U}'$ with some $w \in U'_k$ forming a negative triangle $(u_l^k, v_l^k, w)$ in $G$, and then reports all the pairs $u_l^k v_l^k$ for which such a $U'_k$ was found.
```

This provides the basis of the triangle-searching strategy. To summarize the intuition of the asymptotic speedup in this paper: Since the U'_k have size $\sqrt{n}$ (recall that $|\mathcal{U}'| = \sqrt{n}$), if each node using a quantum search can search through its assigned U'_k in $\tilde{\mathcal{O}}(n^{1/4})$ rounds, simultaneously, we will obtain our desired complexity. We will complete this argument in §4.6 and first describe the quantum searches used therein in the following subsection.

4.5 Distributed Quantum Searches

With this intuition in mind, we now state two useful theorems of (Izumi & Gall, 2019) for the distributed quantum searches. Let X denote a finite set throughout this subsection.

Theorem 4.10. Let $g : X \to \{0,1\}$, if a node u can compute $g(x)$ in r rounds in the CONGEST-CLIQUE model for any $x \in X$, then there exists an algorithm in the Quantum CONGEST-CLIQUE that has u output some $x \in X$ with $g(x) = 1$ with high probability using $\tilde{\mathcal{O}}(r\sqrt{|X|})$ rounds.

This basic theorem concerns only single searches, but we need a framework that can perform multiple simultaneous searches. Let $g_1, \ldots, g_m : X \to \{0,1\}$ and

$$A_i^0 := \{x \in X : g_i(x) = 0\}, A_i^1 := \{x \in X : g_i(x) = 1\}, \forall i \in [m].$$

Assume there exists an r-round classical distributed algorithm C_m that allows a node u upon an input $\chi = (x_1, \ldots, x_m) \in X^m$ to determine and output $(g_1(x_1), \ldots, g_m(x_m))$. In our use of distributed searches, X will consist of nodes in the network, and searches will need to communicate with those nodes for which the functions g_i are evaluated. To avoid congestion, we will have to consider those $\chi \in X^m$ that have many repeated entries carefully. We introduce some notation for this first. Define the quantity

$$\alpha(\chi) := \max_{I \subset [m]} |\{\chi_i = \chi_j \quad \forall i, j \in I\}|,$$

the maximum number of entries in χ that are all identical.

Next, given some $\beta \in \mathbb{N}$, assume that in place of C_m we now have a classical algorithm $\tilde{C}_{m,\beta}$ such that upon input $\chi = (x_1, \ldots, x_m) \in X^m$, a node u outputs $g_1(x_1), \ldots, g_m(x_m)$ if $\alpha(\chi) \le \beta$ and an arbitrary output otherwise. The following theorem summarizes that such a $\tilde{C}_{m,\beta}$ with sufficiently large β is enough to maintain a quantum speedup as seen in the previous theorem:

Theorem 4.11. For a set X with $|X| < m/(36 \log m)$, suppose there exists such an evaluation algorithm $C_{m,\beta}$ for some $\beta > 8m/|X|$ and that $\alpha(\chi) \le \beta$ for all $\chi \in A_1^1 \times \cdots \times A_m^1$. Then there is a $\tilde{\mathcal{O}}(r\sqrt{|X|})$-round quantum algorithm that outputs an element of $A_1^1 \times \cdots \times A_m^1$ with probability at least $1 - 2/m^2$.

The proof can be found in (Izumi & Gall, 2019, Theorem 3).

4.6 Final Steps of the Triangle Finding

We continue here to complete the step 3.2 of the ComputePairs procedure, armed with Theorem 4.11. We need simultaneous searches to be executed by each node $v_{(i,j,k)}$ to determine the triangles in $U_i \times U_j \times U'_k$. We provide a short lemma first that ensures the conditions for the quantum searches:

Lemma 4.12. The following statements hold with probability at least $1 - 2/n^2$:

(i): $|\Delta(i,j,k)| \le 2n$

(ii): $|\Lambda_k(U_i, U_j) \cap \Delta(i,j,k)| \le 100 \cdot \sqrt{n} \log n$ for $i, j \in [n^{1/4}]$.

The proofs of these statements are technical but straightforward, making use of Chernoff's bound and union bounds; hence we skip them here. To invoke Theorem 4.11, we describe a classical procedure first, beginning with an evaluation step, EvaluationA implementable in $\tilde{\mathcal{O}}(1)$ rounds.

```
EvaluationA
Input: Every node $v_{(i,j,k)}$ receives $m$ elements $(u_1^{i,j,k}, \ldots, u_m^{i,j,k})$ of $\mathcal{U}'$
Promise: For every node $v_{i,j,k}$ and every $\mathbf{w} \in \mathcal{U}'$, $|L_{\mathbf{w}}^{i,j,k}| \le 800\sqrt{n}\log n$.
Output: Each node outputs a list of exactly those $u_l^{i,j,k}$ such that there is a negative triangle in $U_i \times U_j \times u_l^{i,j,k}$.

    1. Every node $v_{(i,j,k)}$, for each $r \in \sqrt{n}$ routes the list $L_{\mathbf{w}}^{i,j,t}$ to node $v_{(i,j,t)}$.
    2. Every node $v_{(i,j,k)}$, for each $vu$ it received in step 1, sends the truth value of the inequality

           $$\min_{w \in U'_k} \{W_{uw} + W_{wv}\} \le W_{vu} \qquad (4.3)$$

       to the node that sent $vu$.
```

Each node is the source and destination of up to $800n \log n$ messages in step 1, meaning that this step can be implemented in $1600 \log n$ rounds. The same goes for step 2, noting that the number of messages is the same, but they need only be single-bit messages (the truth values of the inequalities). Hence, the evaluations for Theorem 4.11 can be implemented in $3200 \log n$ rounds. Now, applying the theorem with $X = \mathcal{U}', \beta = 800\sqrt{n} \log n$, noting that then the assumptions of the theorem hold with probability at least $1 - 2/n^2$ due to Lemma 4.12, implies that step 3.2 is implementable in $\tilde{\mathcal{O}}(n^{1/4})$ rounds, with a success probability of at least $1 - 2/m^2$.

For the general case in which we do not assume $\alpha = 0$ for all i, j, k in `IdentifyClass`, covered in the appendix, one needs to modify the `EvaluationA` procedure in order to implement load balancing and information duplication to avoid congestion in the simultaneous searches. These details can be found in the appendix, where a new labeling scheme and different evaluation procedure `EvaluationB`, are described for this, or in (Izumi & Gall, 2019).

4.7 Complexity

As noted previously and in (Izumi & Gall, 2019), this APSP scheme uses $\tilde{\mathcal{O}}(n^{1/4})$ rounds. Let us characterize the constants and logarithmic factors involved to assess this algorithm's practical utility. Suppose that in each round, $2 \cdot \log n$ qubits can be sent in each message (so that we can send two IDs or one edge with each message), where n is the number of nodes. For simplicity, let's assume $W \ll n$ and drop W.

1. APSP with routing tables needs $\log(n)$ distance products with witness matrices.
2. Computing the i^{th} distance product square for Eq. (4.2) with a witness matrix needs up to $\log(2^i) = i$ calls to `FindEdges`, since the entries of the matrix being squared may double each iteration. Then APSP and distance products together make $\sum_{i=1}^{\lceil \log n \rceil} i = \frac{\lceil \log(n) \rceil (\lceil \log(n) \rceil + 1)}{2}$ calls to `FindEdges`.
3. Solving `FindEdges` needs $\log\left(\frac{n}{60 \log n}\right)$ calls to `FEWP`, using `FindEdgesViaFEWP`.
4. Step 1 of `ComputePairs` needs up to $2 \cdot n^{1/4}$ rounds and step 2 takes up to $200 \log n$ rounds.
5. Step 1 of `IdentifyClass` needs up to $20 \log n$ rounds.
6. In step 2 of `IdentifyClass`, the c_{uvw} are up to $\frac{1}{2} \log n$ large, and hence α may range up to $\frac{1}{2} \log n$.
7. Step 0 of the `EvaluationB` procedure needs $n^{1/4}$ rounds. Steps 1 and 2 of the `EvaluationB` (or `EvaluationA`, in the $\alpha = 0$ case) procedure use a total of $3200 \log n$ rounds.
8. `EvaluationB` (or `EvaluationA`) procedure is called up to $\log(n) n^{1/4}$ times for each value of α in step 3.2 of `ComputePairs`.

Without any improvements, we get the following complexity, using $3n$ in place of n for the terms of steps 3-8 due to corollary 4.6:

$$\frac{\lceil \log(n) \rceil (\lceil \log(n) \rceil + 1)}{2} \log\left(\frac{3n}{60 \log 3n}\right) \Big(2(3n)^{1/4} + 220 \log 3n + 2(3n)^{1/4} + \frac{1}{2} \log 3n \cdot \log 3n \cdot (3n)^{1/4} 3200 (\log 3n)\Big), \tag{4.4}$$

which we will call $f(n)$, so that $f(n) = \mathcal{O}(n^{1/4} \log^6(n))$, with the largest term being about $800 \log^6(n) n^{1/4}$, and we have dropped W to just consider the case $W \ll n$. We can solve the problem trivially in the (quantum or classical) CONGEST-CLIQUE within $n \log(W)$ rounds by having each node broadcast its neighbors and the weight on the edge. Let us again drop W for the case $W \ll n$ so that in order for the quantum algorithm to give a real speedup, we will need

$$f(n) < n,$$

which requires $n > 10^{18}$ (even with the simpler under-approximation $800 \log^6(n) n^{1/4}$ in place of f). Hence, even with some potential improvements, the algorithm is impractical for a large regime of values of n even when compared to the trivial CONGEST-CLIQUE n-round strategy.

For the algorithm of (Izumi & Gall, 2019) computing only APSP *distances*, the first term in 4.4 becomes simply $\lceil \log n \rceil$, so that when computing only APSP distances the advantage over the trivial strategy begins at roughly $n \approx 10^{16}$.

Remark 4.13. In light of logarithmic factors commonly being obscured by $\tilde{\mathcal{O}}$ notation, we point out that even an improved algorithm needing only $\log^4(n)n^{1/4}$ would not be practical unless $n > 10^7$, for the same reasons. Recall that n is the number of *processors* in the distributed network – tens of millions would be needed to make this algorithm worth implementing instead of the trivial strategy. Practitioners should mind the $\tilde{\mathcal{O}}$ if applications are of interest, since even relatively few logarithmic factors can severely limit practicality of algorithms, and researchers should be encouraged to fully write out the exact complexities of their algorithms for the same reason.

4.7.1 Memory Requirements

Although in definition 2.1 we make no assumption on the memory capacities of each node, the trivial n-round strategy uses at least $2\log(n)|E|^2 \cdot \log(W)$ memory at the leader node that solves the problem. For the APSP problem in question, using the Floyd-Warshall algorithm results in memory requirements of $2n^2\log(n) \cdot \log(nW)$ at the leader node. Hence, we may ask whether the quantum APSP algorithm leads to lower memory requirements. The memory requirement is largely characterized by up to $720n^{7/4}\log(n)\log(nW)$ needed in step 0 of the `EvaluationB` procedure, which can be found in the appendix. This results in a memory advantage for quantum APSP over the trivial strategy beginning in the regime of $n > 1.6 \cdot 10^{10}$.

4.7.2 Complexity of the Classical Analogue

For completeness, we provide here a characterization of the complexity of a closely related classical algorithm for APSP with routing tables in the CONGEST-CLIQUE as proposed in (Censor-Hillel et al., 2016) that has complexity $\tilde{\mathcal{O}}(n^{1/3})$. In their framework, the approach to finding witness matrices requires $\mathcal{O}(\log(n)^3)$ calls to the distance product (Censor-Hillel et al., 2016, §3.4), and similarly to our approach $\log(n)$ distance products are required. Their classical algorithm computes distance products in $\mathcal{O}(n^{1/3})$ rounds, or under $2\log n$ message bandwidth in up to

$$20n^{1/3}\log(n)^4 =: g(n) \tag{4.5}$$

rounds, the details of which can be found in the appendix, §8.2.1. Then $g(n) > n$ up until about $n \approx 2.6 \cdot 10^{11}$. As with the quantum APSP, though this algorithm gives the best known asymptotic complexity of $\tilde{\mathcal{O}}(n^{1/3})$ in the classical CONGEST-CLIQUE, it also fails to give any real improvement over the trivial strategy across a very large regime of values of n. Consequently, algorithms making use of this APSP algorithm, such as (Saikia & Karmakar, 2019) or (Fischer & Oshman, 2021), suffer from the same problem of impracticality. However, the algorithm only requires within $4n^{4/3}\log(n)\log(nW) + n\log(n)\log(nW)$ memory per node, which is less than required for the trivial strategy even for $n \geq 4$.

5 Approximately Optimal Steiner Tree Algorithm

5.1 Algorithm Overview

We present a high-level overview of the proposed algorithm to produce approximately optimal Steiner Trees, divided into four steps.

Step 1 - APSP and Routing Tables: Solve the APSP problem as in (Izumi & Gall, 2019) and add an efficient routing table scheme via triangle finding in $\tilde{\mathcal{O}}(n^{1/4})$ rounds, with success probability $(1 - 1/poly(n))$ (this step determines the algorithm's overall success probability).

Step 2 - Shortest-path Forest: Construct a shortest-path forest (SPF), where each tree consists of exactly one source terminal and the shortest paths to the vertices whose closest terminal is that source terminal. This step can be completed in one round and n messages, per (Saikia & Karmakar, 2019, §3.1). The messages can be in classical bits.

Step 3 - Weight Modifications: Modify the edge weights depending on whether they belong to a tree (set to 0), connect nodes in the same tree (set to ∞), or connect nodes from different trees (set to the shortest path distance between root terminals of the trees that use the edge). This uses one round and n messages.

Step 4 - Minimum Spanning Tree: Construct a minimum spanning tree (MST) on the modified graph in $\mathcal{O}(1)$ rounds as in (Nowicki, 2019), and prune leaves of the MST that do not connect terminal nodes since these are not needed for the Steiner Tree.

The correctness of the algorithm follows from the correctness of each step together with the analysis of the classical results of (Kou et al., 1981), which uses the same algorithmic steps of constructing a shortest path forest and building it into an approximately optimal Steiner Tree.

5.2 Shortest Path Forest

After the APSP distances and routing tables have been found, we construct a *Shortest Path Forest* (SPF) based on the terminals of the Steiner Tree.

Definition 5.1. (Shortest Path Forest): For a weighted, undirected graph $G = (V, E, W)$ together with a given set of terminal nodes $Z = \{z_1, \dots, z_k\}$, a subgraph $F = (V, E_F, W)$ of G is called a *shortest path forest* if it consists of $|Z|$ disjoint trees $T_z = (V_z, E_z, W)$ satisfying

i) $z_i \in T_{z_j}$ if and only if $i = j$, for $i, j \in [k]$.

ii) For each $v \in Z_i, d_G(v, z_i) = \min_{z \in Z} d_G(v, z)$, and a shortest path connecting v to z_i in G is contained in T_{z_i}

iii) The V_{z_i} form a partition of V, and $E_{z_1} \cup E_{z_2} \cdots \cup E_{z_k} = E_F \subset E$

In other words, an SPF is a forest obtained by gathering, for each node, a shortest path in G connecting it to the closest Steiner terminal node.

For a node v in a tree, we will let $par(v)$ denote the parent node of v in that tree, $s(v)$ the Steiner Terminal in the tree that v will be in, and $ID(v) \in [n]$ the ID of node $v \in V$. Let $\mathcal{Q}(v) := \{z : d_G(v, z) = \min_{z \in Z} d_G(v, z)\}$ be the set of Steiner Terminals closest to node v. We make use of the following procedure for the SPF:

```
DistributedSPF
Input: For each node $v \in G$, APSP distances and the corresponding routing table $R_v$.
Output: An SPF distributed among the nodes.
  1: Each node $v$ sets $s(v) := \text{argmin}_{z \in \mathcal{Q}(v)} ID(z)$ using the APSP information.
  2: Each node $v$ sets $par(v) := R_v(s(v))$, $R_v$ being the routing table of $v$, and sends a message to $par(v)$ to
     indicate this choice. If $v$ receives such a message from another node $u$, it registers $u$ as its child in the
     SPF.
```

Step 1 in DistributedSPF requires no communication since each node already knows the shortest path distances to all other nodes, including the Steiner Terminals, meaning it can be executed locally. Each node v choosing $par(v)$ in step 2 can also be done locally using routing table information, and thus step 2 requires 1 round of communication of $n - |Z|$ classical messages, since all non-Steiner nodes send one message.

Claim 2. After executing the DistributedSPF procedure, the trees $T_{z_k} = (V_{z_k}, E_{z_k}, W)$ with $V_{z_k} := \{v \in V : s(v) = z_k\}$ and $E_{z_k} := \{v, par(v)\} : v \in V_{z_k}\}$ form an SPF.

Proof. i) holds since each Steiner Terminal is closest to itself. iii) is immediate. To see that ii) holds, note that for $v \in V_{z_k}$, $par(v) \in V_{z_k}$ and $\{v, par(v)\} \in E_{z_k}$ as well. Then $par(par(\dots par(v) \dots)) = z_k$ and the entire path to z_k lies in T_{z_k}. □

Hence, after this procedure, we have a distributed SPF across our graph, where each node knows its label, parent, and children of the tree it is in.

5.3 Weight Modified MST and Pruning

Finally, we introduce a modification of the edge weights before constructing an MST on that new graph that will be pruned into an approximate Steiner Tree. These remaining steps stem from a centralized algorithm first proposed by (Kou et al., 1981) whose steps can be implemented efficiently in the distributed setting, as in (Saikia & Karmakar, 2019). We first modify the edge weights as follows:

Partition the edges E into three sets – *tree edges* E_F as in 5.1 that are part of the edge set of the SPF, *intra-tree edges* E_{IT} that are incident on two nodes in the same tree T_i of the SPF, and *inter-tree edges* E_{XT} that are incident on two nodes in different trees of the SPF. Having each node know which of these its edges belong to can be done in one round by having each node send its neighbors the ID of the terminal it chose as the root of the tree in the SPF that is a part of. Then the edge weights are modified as follows, denoting the modified weights as W':

(i): For $e = (u, v) \in E_T, W'(u, v) := 0$

(ii): For $e = (u, v) \in E_{IT}, W'(u, v) := \infty$

(iii): For $e = (u, v) \in E_{XT}, W'(u, v) := d(u, Z_u) + W(u, v) + d(v, Z_v)$,

noting that $d_G(u, s(u))$ is the shortest-path distance in G from u to its closest Steiner Terminal.

Next, we find a minimum spanning tree on the graph $G' = (V, E, W')$, for which we may implement the classical $\mathcal{O}(1)$ round algorithm proposed by (Nowicki, 2019). On a high level, this constant-round complexity is achieved by sparsification techniques, reducing MST instances to sparse ones, and then solving those efficiently. We skip the details here and refer the interested reader to (Nowicki, 2019). After this step, each node knows which of its edges are part of this weight-modified MST, as well as the parent-child relationships in the tree for those edges.

Finally, we prune this MST by removing non-terminal leaf nodes and the corresponding edges. This is done by each node v sending the ID of its parent in the MST to every other node in the graph. As a result, each node can locally compute the entire MST and then decide whether or not it connects two Steiner Terminals. If it does, it decides it is part of the Steiner Tree; otherwise, it broadcasts that it is to be pruned. Each node that has not been pruned then registers the edges connecting it to non-pruned neighbors as part of the Steiner Tree. This pruning step takes 2 rounds and up to $n^2 + n$ classical messages.

5.4 Overall Complexity and Correctness

In algorithm 5.1, after step 1, steps 2, and 3 can each be done within 2 rounds. Walking through (Nowicki, 2019) reveals that the MST for step 4 can be found in 54 rounds, with an additional 2 rounds sufficing for the pruning. Hence, the overall complexity remains dominated by Eq. (4.4). Hence, the round complexity is $\tilde{\mathcal{O}}(n^{1/4})$, which is faster than any known classical CONGEST-CLIQUE algorithm to produce an approximate Steiner tree of the same approximation ratio. However, as a consequence of the full complexity obtained in §4.7, the regime of n in which this algorithm beats the trivial strategy of sending all information to a single node is also $n > 10^{18}$. For the same reason, the classical algorithm provided in (Saikia & Karmakar, 2019) making use of the APSP subroutine from (Censor-Hillel et al., 2016) discussed in §4.7.2 has its complexity mostly characterized by Eq. (4.5), so that the regime in which it provides an advantage over the trivial strategy lies in $n > 10^{11}$. Our algorithm's correctness follows from the correctness of each step together with the correctness of the algorithm by (Kou et al., 1981) that implements these steps in a classical, centralized manner.

6 Directed Minimum Spanning Tree Algorithm

This section will be concerned with establishing Theorem 3.2 for the Directed Minimum Spanning Tree (DMST) problem, in definition 2.7. Like (Fischer & Oshman, 2021), we follow the algorithmic ideas first proposed by (Lovasz, 1985), implementing them in the quantum CONGEST-CLIQUE. Specifically, we will use $\log n$ calls to the APSP and routing tables scheme described in §4, so that in our case, we retrieve complexity $\tilde{\mathcal{O}}(n^{1/4})$ and success probability $(1 - \frac{1}{poly(n)})^{\log n} = 1 - \frac{1}{poly(n)}$.

Before describing the algorithm, we need to establish some preliminaries and terminology for the procedures executed during the algorithm, especially the ideas of shrinking vertices into *super-vertices* and tracking a set H of specific edges as first described in (Edmonds et al., 1967). We use the following language to discuss super-vertices and related objects.

Definition 6.1. A *super-vertex set* $\mathbb{V}^* := \{V_1^*, \ldots, V_t^*\}$ for a graph $G = (V, E, W)$ is a partition of V, and each V_i^* is called a *super-vertex.* We will call a super-vertex *simple* if V^* is a singleton. The corresponding *minor* $G^* := (\mathbb{V}^*, E^*, W^*)$ is the graph obtained by creating edges (V_i^*, V_j^*) with weight $W^*(V_i^*, V_j^*) := \min\{W(v_i, v_j) : v_i \in V_i^*, v_j \in V_j^*\}$.

Notably, we continue to follow the convention of an edge of weight ∞ being equivalent to not having an edge. We will refer to creating a super-vertex V^* as *contracting* the vertices in V^* into a super-vertex.

6.1 Edmonds' Centralized DMST Algorithm

We provide a brief overview of the algorithm proposed in (Edmonds et al., 1967), which presents the core ideas of the super-vertex-based approach. The following algorithm produces a DMST for G:

```
Edmonds DMST Algorithm
Input: An integer-weighted digraph and a root node $r$.
Output: A DMST for $G$ rooted at $r$.
  1. Initialize a subgraph $H$ with the same vertex set as G by subtracting for each node the minimum incoming edge weight from all its incoming edges, and selecting exactly one incoming zero-weight edge for each non-root node of $G$. Set $G_0 = G, H_0 = H, t = 0$.
  2. WHILE $H_t$ is not a tree:
      (a) For each cycle of $H$, contract the nodes on that cycle into a super-vertex. Consider all non-contracted nodes as simple super-vertices, and obtain a new graph $G_{t+1}$ as the resulting minor.
      (b) If there is a non-root node of $G_{t+1}$ with no incoming edges, report a failure. Otherwise, obtain a subgraph $H_{t+1}$ by, for each non-root node of $G_{t+1}$, subtracting the minimum incoming edge weight from all its incoming edges, and selecting exactly one incoming zero-weight edge for each non-root, updating $t \leftarrow t + 1$.
  3. Let $B_t = H_t$. FOR $k \in (t, t-1, \ldots, 1)$:
      (a) Obtain $B'_{k-1}$ by expanding the non-simple super-vertices of $B_k$ and selecting all but one of the edges for each of the previously contracted cycles of $H_k$ to add to $B_{k-1}$.
  4. Return $B_0$.
```

Note that the edge weight modifications modify the weight of all directed spanning trees equally, so optimality is unaffected. In step 2., if H_t is a tree, it is an optimal DMST for the current graph G_t. Otherwise, it contains at least one directed cycle, so that indeed step 2. is valid. Hence, at the beginning of step 3., B_t is a DMST for G_t. Then the first iteration produces B_{t-1} a DMST for G_{t-1} since only edges of zero weight were added, and B_{t-1} will have no cycles. The same holds for $B_{t-2}, B_{t-3}, \ldots, B_0$, for which B_0 corresponds to the DMST for the original graph G. If the algorithm reports a failure at some point, no spanning tree rooted at r exists for the graph, since a failure is reported only when there is an isolated non-root connected component in G_{t+1}.

Note that in iteration t of step 2., H has one cycle for each of its connected components that does not contain the root node. Hence, the drawback of this algorithm is that we may apply up to $\mathcal{O}(n)$ steps of shrinking cycles. This shortcoming is remedied by a more efficient method of selecting how to shrink nodes into super-vertices in (Lovasz, 1985), such that only $\log n$ shrinking cycle steps take place.

6.2 Lovasz' Shrinking Iterations

We devote this subsection to discuss the shrinking step of (Lovasz, 1985) that will be repeated $\log n$ times in place of step 2. of Edmonds' algorithm to obtain Lovasz' DMST algorithm.

`Lovasz' Shrinking Iteration LSI`
Input: A directed, weighted graph $G = (V, E, W)$ and a root node $r \in V$.
Output: Either a new graph G^*, or a success flag and a DMST H of G.

1. If there is a non-root node of G with no incoming edges, report a failure. Otherwise, for each non-root node of G, subtract the minimum incoming edge weight from all its incoming edges. Select exactly one incoming zero-weight edge for each non-root node to create a subgraph H of G with those edges.

2 Find all cycles of H, and denote them $H_1, \dots, H_C$. If H has no cycles, abort the iteration and return (SUCCESS, H). For $j = 1, \dots, C$, find the set V_j of nodes that dipaths in H from H_j can reach.

3. Compute the All-Pairs-Shortest-Path distances in G.

4. For each node $v \in V$, denote $d_j(v) := \min\{d(v, u) : u \in H_j\}$. For each $j = 1, \dots, C$, set $\beta_j := \min\{d_j(v) : v \in V(G) \setminus \mathbb{V}_j\}$ and $U_j := \{u \in V_j : d_j(u) \leq \beta_j\}$.

5. Create a minor G^* by contracting each U_j into a super-vertex U_j^*, considering all other vertices of G as simple super-vertices $V_1^*, \dots, V_k^*$. For each vertex N^* of G^*, let the edge weights in G^* be:

$$\begin{aligned} W^*_{N^* U_j^*} &= \min\{W_{vu} : v \in N^*, u \in U_j^*\} - \beta_j + \min\{d_j(u) : u \in U_j^*\} \\ &\quad \text{for all } j = 1, \dots, C, \text{ and} \\ W^*_{N^* V^*} &= \min\{W_{vV^*} : v \in N^*\} \\ &\quad \text{for all the simple super-vertices } V^* \text{ of } G^*. \end{aligned}$$

6. Return G^*.

To summarize these iterations: The minimum-weight incoming edge of each node is selected. That weight is subtracted from the weights of every incoming edge to that node, and one of those edges with new weight 0 is selected for each node to create a subgraph H. If H is a tree, we are done. Otherwise, we find all cycles of the resulting directed subgraph, then compute APSP and determine the V_j, U_j, and β_j, which we use to define a new graph with some nodes of the original G contracted into super-vertices.

The main result for the DMST problem in (Lovasz, 1985) is that replacing (a) and (b) of step 2. in the `Edmonds DMST Algorithm`, taking the new H obtained at each iteration to be H_{t+1} and the G^* to be G_{t+1}, leads to no more than $\lceil \log n \rceil$ such shrinking iterations needed before a success is reported.

6.2.1 Quantum Distributed Implementation

Our goal is to implement the Lovasz iterations in the quantum distributed setting in $\tilde{\mathcal{O}}(n^{1/4})$ rounds by making use of quantum APSP of §4. In the distributed setting, processor nodes cannot directly be shrunk into super-vertices. As in (Fischer & Oshman, 2021), we reconcile this issue by representing the super-vertex contractions within the nodes through *soft contractions.*

First, note that a convenient way to track what nodes we want to consider merging into a super-vertex is to keep a mapping $sID : V \to S$, where S is a set of super-vertex IDs, which we can just take to be the IDs of the original nodes. We will refer to a pair of (G, sID) as an *annotated graph.* An annotated graph naturally corresponds to some *minor* of G, namely, the minor obtained by contracting all vertices sharing a super-vertex ID into a super-vertex.

Definition 6.2 (Soft Contractions)**.** For an annotated graph (G, sID), a set of active edges H, and active component H_i with corresponding weight modifiers β_i, and a subset $A \subset S$ of super-vertices, the *soft contraction* of H_i in G is the annotated graph (G^{H_i}, sID') obtained by taking $G^{H_i} = (V, E, W')$ with

- $W'_{uv} = 0$ if $sID(u) = sID(v)$
- $W'_{uv} = W_{uv} + dist_{G(A)}(v, C(H_i)) - \beta_i$ if $u \in V \setminus A$ and $v \in A$
- $W'_{uv} = W_{uv}$ otherwise

and updating the mapping sID to sID' defined by $sID'(v) = sID(v), \forall v \notin A$, $sID'(v) = \min\{sID(u) : u \in A\}$.

6.2.2 Quantum Distributed Lovasz' Iteration

We provide here a quantum distributed implementation of Lovasz' iteration that we will form the core of our DMST algorithm.

```
Quantum Distributed Lovasz' Iteration QDLSI
Input: A directed, weighted, graph $G = (V, E, W)$ with annotations $sID$ and a subgraph $H$.
Output: A new graph $G^*$ with annotations $sID'$, or a success flag and a DMST $H$ of $G$.
  1: Have all nodes learn all edges of $H$, as well as the current super-vertices.
  2: For each connected component $H_i \subset H$, denote by $C(H_i)$ the cycle of $H_i$. Let $c(H_i)$ be the node with
     maximal ID in $C(H_i)$, which each node can locally compute.
  3: Run the quantum algorithm for APSP and routing tables described in §4 on this graph, or report a failure
     if it fails.
  4: For each $i$, determine an edge $v_i u_i$, $v_i \notin H_i, u_i \in H_i$ minimizing $\beta_i := W_{v_i u_i} + d_G(u_i, c(H_i))$, and broadcast
     both to all nodes in $H_i$.
  5: Each node $v_i$ in each $H_i$ applies the following updates locally:
       – Soft-contract $H_i$ at level $\beta_i$ to soft-contract all super-vertices with distance $\beta_i$ to $C(H_i)$ into one
         super-vertex, with each contracted node updating its super-vertex ID to $c(H_i)$
       – add edge $v_i u_i$ to $H$, effectively merging $H_i$ with another active component of $H$
```

We can follow exactly the steps of Lovasz's DMST algorithm, distributedly by replacing steps 2-5 of the LSI with this quantum-distributed version. The following ensues:

Lemma 6.3. If none of the APSP and routing table subroutines fail, within $\lceil \log n \rceil$ iterations of the QDLSI, H is a single connected component.

Lemma 6.4. With probability $(1 - \frac{1}{poly(n)})^{\log n}$, all the APSP and routing table subroutines in step 3 succeed.

Lemmas 6.3 and 6.4 then together imply Theorem 3.2. Within $\lceil \log n \rceil$ iterations, only one active component remains: the root component. This active component can then be expanded to a full DMST on G within $\lceil \log n \rceil$ rounds, as detailed in (Fischer & Oshman, 2021, §7) or the Unpacking procedure in §8.3 of the appendix. All messages in the algorithm other than those for computing the APSP in QDLSI may be classical. We provide here the full algorithm for completeness:

```
Quantum DMST Algorithm
Input: An integer-weighted digraph and a root node $r$.
Output: A DMST for $G$ rooted at $r$.
  1. Initialize a subgraph $H$ with the same vertex set as G by subtracting for each node the minimum incoming
     edge weight from all its incoming edges, and selecting exactly one incoming zero-weight edge for each non-
     root node of $G$. Set $t = 0, H_0 = H$, and $G_0 = G$ with annotations $sID_0$ to be the identity mapping.
  2. WHILE: $H_t$ is not a single component
     (a) Run QDLSI with inputs $H_t$, $(G_t, sID_t)$ to obtain $H_{t+1}$, $(G_{t+1}, sIDt + 1)$ as outputs. Increment
         $t \leftarrow t + 1$.
  3. Let $T_t := H_t$. For $k = t, \dots, 1$: For each super-vertex of the $k^{th}$ iteration of QDLSI applied, simultaneously
     run the Unpacking procedure with input tree $T_k$ to obtain $T_{k-1}$.
  4. Return $T_0$ as the distributed minimum spanning tree.
```

6.3 Complexity

In the QDLSI, all steps other than the APSP step 3 of the quantum Lovasz iteration can be implemented within 2 rounds. In particular, to have all nodes know some tree on G for which each node knows its parent, every node can simply broadcast its parent edge and weight. Since this iteration is used up to $\lceil \log(n) \rceil$ times and expanding the DMST at the end of the algorithm also takes logarithmically many rounds, we obtain a complexity dominated by the APSP computation of $\tilde{\mathcal{O}}(n^{1/4})$, a better asymptotic rate than any known classical CONGEST-CLIQUE algorithm. However, beyond the $\tilde{\mathcal{O}}$, the complexity is largely characterized by $\log(n) \cdot f(n)$, with $f(n)$ as in Eq. (4.4). In order to have $\log(n) f(n) < n$ to improve upon the trivial strategy of having a single node solve the problem, we then need $n > 10^{21}$. Using the classical APSP from (Censor-Hillel et al., 2016) in place of the quantum APSP of §4 as done in (Fischer & Oshman, 2021) to attain the $\tilde{\mathcal{O}}(n^{1/3})$ complexity in the cCCM, one would need $\log(n) \cdot g(n) < n$ to beat the trivial strategy, with g as in Eq. (4.5), or more than $n > 10^{14}$.

7 Discussion and Future Work

We have provided algorithms in the Quantum CONGEST-CLIQUE model for computing approximately optimal Steiner Trees and exact Directed Minimum Spanning trees that use asymptotically fewer rounds than their classical known counterparts. As Steiner Tree and Minimum Spanning Trees cannot benefit from quantum communication in the CONGEST (non-clique) model, the algorithms reveal how quantum communication can be exploited thanks to the CONGEST-CLIQUE setting. A few open questions remain as well. In particular, there exist many generalizations of the Steiner Tree problem, so these may be a natural starting point to attempt to generalize the results. A helpful overview of Steiner-type problems can be found in (Hauptmann & Karpinski, 2015). Regarding the DMST, it may be difficult to generalize a similar approach to closely related problems. Since the standard MST can be solved in a (relatively small) constant number of rounds in the classical CONGEST-CLIQUE, no significant quantum speedup is possible. Other interesting MST-type problems are the bounded-degree and minimum-degree spanning tree problems. However, even the bounded-degree decision problem on an unweighted graph, "does G have a spanning tree of degree at most k?" is NP-complete, unlike the DMST, so we suspect that other techniques would need to be employed. (Dinitz, Halldorsson, Izumi, & Newport, 2019) provides a classical distributed approximation algorithm for the problem. Additionally, we have traced many constants and log factors throughout our description of the above algorithms, which, as shown, would need to be significantly improved for these and related algorithms to be practical. Hence, a natural avenue for future work is to work towards such practical improvements. Beyond the scope of the particular algorithms involved, we hope to help the community recognize the severity with which the practicality of algorithms is affected by logarithmic factors that may be obscured by $\tilde{\mathcal{O}}$ notation, and thus encourage fellow researchers to present the full complexity of their algorithms beyond asymptotics. Particularly in a model like CONGEST-CLIQUE, where problems can always be solved trivially in n rounds, these logarithmic factors should clearly not be taken lightly. Further, a question of potential practical interest would be to ask the following: What algorithms solving the discussed problems are the most efficient with respect to rounds needed in the CONGEST-CLIQUE in the regimes of n in which the discussed algorithms are impractical?

Acknowledgements

We are grateful for support from the NASA Ames Research Center, from the NASA SCaN program, and from DARPA under IAA 8839, Annex 130. PK and DB acknowledge support from the NASA Academic Mission Services (contract NNA16BD14C – under SAA2-403506). The authors thank Ojas Parekh for helpful input and discussions regarding the arborescence problem, Shon Grabbe for ongoing useful discussions, and Filip Maciejewskifor for helpful feedback on the work.

References

Booth, K. E. C., O'Gorman, B., Marshall, J., Hadfield, S., & Rieffel, E. (2021, sep). Quantum-accelerated constraint programming. *Quantum*, *5*, 550. Retrieved from https://arxiv.org/abs/2103.04502 doi: doi:10.22331/q-2021-09-28-550

Censor-Hillel, K., Fischer, O., Le Gall, F., Leitersdorf, D., & Oshman, R. (2022). *Quantum Distributed Algorithms for Detection of Cliques.* arXiv. Retrieved from https://arxiv.org/abs/2201.03000 doi: doi:10.48550/ARXIV.2201.03000

Censor-Hillel, K., Kaski, P., Korhonen, J. H., Lenzen, C., Paz, A., & Suomela, J. (2016, mar). Algebraic methods in the congested clique. *Distributed Computing*, *32*(6), 461–478. doi: doi:10.1007/s00446-016-0270-2

Dinitz, M., Halldorsson, M. M., Izumi, T., & Newport, C. (2019). Distributed Minimum Degree Spanning Trees. In *Proceedings of the 2019 acm symposium on podc* (p. 511–520). New York, NY, USA: ACM. doi: doi:10.1145/3293611.3331604

Dolev, D., Lenzen, C., & Peled, S. (2012). "Tri, Tri Again": Finding Triangles and Small Subgraphs in a Distributed Setting - (Extended Abstract). *ArXiv*, *abs/1201.6652*.

Edmonds, J., et al. (1967). Optimum branchings. *Journal of Research of the national Bureau of Standards B*, *71*(4), 233–240.

Elkin, M., Klauck, H., Nanongkai, D., & Pandurangan, G. (2012). *Can Quantum Communication Speed Up Distributed Computation?* arXiv. Retrieved from https://arxiv.org/abs/1207.5211 doi: doi:10.48550/ARXIV.1207.5211

Fischer, O., & Oshman, R. (2021). A distributed algorithm for directed minimum-weight spanning tree. *Distributed Computing*, 1–31.

Ghaffari, M. (2020). *Distributed Graph Algorithms (Lecture Notes).* Retrieved from https://disco.ethz.ch/courses/podc/lecturenotes/LOCAL.pdf

Giovannetti, V., Lloyd, S., & Maccone, L. (2008, apr). Quantum random access memory. *Physical Review Letters*, *100*(16). Retrieved from https://doi.org/10.1103%2Fphysrevlett.100.160501 doi: doi:10.1103/physrevlett.100.160501

Hauptmann, M., & Karpinski, M. (2015). *A Compendium on Steiner Tree Problems.* Retrieved from http://theory.cs.uni-bonn.de/info5/steinerkompendium

Izumi, T., & Gall, F. L. (2019, jul). Quantum Distributed Algorithm for the All-Pairs Shortest Path Problem in the CONGEST-CLIQUE Model. In *Proceedings of the 2019 ACM symposium on podc.* ACM. Retrieved from https://arxiv.org/abs/1906.02456 doi: doi:10.1145/3293611.3331628

Izumi, T., Le Gall, F., & Magniez, F. (2020). *Quantum Distributed Algorithm for Triangle Finding in the CONGEST Model.* Schloss Dagstuhl - Leibniz-Zentrum für Informatik. Retrieved from https://drops.dagstuhl.de/opus/volltexte/2020/11884/ doi: doi:10.4230/LIPICS.STACS.2020.23

Korhonen, J. H., & Suomela, J. (2017). *Towards a complexity theory for the congested clique.* arXiv. Retrieved from https://arxiv.org/abs/1705.03284 doi: doi:10.48550/ARXIV.1705.03284

Kou, L. T., Markowsky, G., & Berman, L. (1981). A fast algorithm for Steiner trees. *Acta Informatica*, *15*, 141–145. Retrieved from http://aturing.umcs.maine.edu/~markov/SteinerTrees.pdf

Le Gall, F., & Magniez, F. (2018, jul). Sublinear-Time Quantum Computation of the Diameter in CONGEST Networks. In *Proceedings of the 2018 ACM symposium on podc.* ACM. Retrieved from https://arxiv.org/pdf/1804.02917.pdf doi: doi:10.1145/3212734.3212744

Lenzen, C. (2012). *Optimal Deterministic Routing and Sorting on the Congested Clique.* arXiv. doi: doi:10.48550/ARXIV.1207.1852

Lovasz, L. (1985). Computing ears and branchings in parallel. In *26th annual symposium on foundations of computer science (sfcs 1985)* (pp. 464–467). doi: doi:10.1109/SFCS.1985.16

Nowicki, K. (2019). *A Deterministic Algorithm for the MST Problem in Constant Rounds of Congested Clique.* arXiv. doi: doi:10.48550/ARXIV.1912.04239

Rieffel, E., & Polak, W. (2011). *Quantum Computing: A Gentle Introduction* (1st ed.). The MIT Press.

Saikia, P., & Karmakar, S. (2019). *Distributed Approximation Algorithms for Steiner Tree in the CONGESTED CLIQUE.* arXiv. Retrieved from https://arxiv.org/abs/1907.12011 doi: doi:10.48550/ARXIV.1907.12011

van Apeldoorn, J., & de Vos, T. (2022). *A Framework for Distributed Quantum Queries in the CONGEST Model.* arXiv. Retrieved from https://arxiv.org/abs/2202.10969 doi: doi:10.48550/ARXIV.2202.10969

Zwick, U. (2000). *All Pairs Shortest Paths using Bridging Sets and Rectangular Matrix Multiplication.* arXiv. doi: doi:10.48550/ARXIV.CS/0008011

8 Appendix

8.1 Proof of claim 1

For an $n \times n$ integer matrix W, obtain matrices W' and W'' by taking $W'_{ij} = nW_{ij} + j - 1$ and $W''_{ji} = nW_{ji}$. Set $D = W' \star W''$. We aim to show that $\left\lfloor \frac{D}{n} \right\rfloor = W^{2,\star}$ and $(D \mod n) + 1$ is a witness matrix for $W^{2,\star}$.

Proof.

(i) We have

$$\left\lfloor \frac{D}{n} \right\rfloor_{ij} = \left\lfloor \min_{k \in [n]} \{nW_{ik} + k - 1 + nW_{kj}\} / n \right\rfloor = \left\lfloor \min_{k \in [n]} \left\{ W_{ik} + W_{kj} + \frac{k-1}{n} \right\} \right\rfloor$$
$$= W^2_{ij} + \left\lfloor \min_{k \in [n]} \left\{ \frac{k-1}{n} : W_{ik} + W_{kj} = W^2_{ij} \right\} \right\rfloor = W^2_{ij}.$$

(ii) Next,

$$D_{ij} = nW^2_{ij} + \left\lfloor \min_{k \in [n]} \left\{ k - 1 : W_{ik} + W_{kj} = W^2_{ij} \right\} \right\rfloor$$

gives us

$$(D \mod n) + 1 = \left\lfloor \min_{k \in [n]} \left\{ k - 1 : W_{ik} + W_{kj} = W^2_{ij} \right\} \right\rfloor + 1 = \min_{k \in [n]} \left\{ k : W_{ik} + W_{kj} = W^2_{ij} \right\},$$

which proves the claim.

□

8.2 The α ¿ 0 case

The strategy will be to assign each $v_{(i,j,k)} \in \mathbb{V}$ into classes in accordance with approximately how many negative triangles are in $U_i \times U_j \times U'_k$ before starting the search.

To assign each node to a class, we use the routine `IdentifyClass` of (Izumi & Gall, 2019), also described in the main text.

The main body of this paper discussed the special case assuming $\alpha = 0$. Hence we now consider the $\alpha > 0$ case. For each $\alpha \in \mathbb{N}$, let us denote $c_{i,j,k}$ the smallest nonnegative integer satisfying $d_{i,j,k} < 10 \cdot 2^c \log n$, and

$$V_\alpha := \{v_{(i,j,k)} : c_{i,j,k} = \alpha\} \tag{8.1}$$
$$V_\alpha[i,j] := \{U'_k \in \mathcal{U}' : v_{(i,j,k)} \in V_\alpha\} \tag{8.2}$$

for any $i, j \in [n^{1/4}]$. Notably, $\mathcal{P}(i,j)$ contains at most $\sqrt{n}$ edges, so that $d_{i,j,k} \leq \sqrt{n}$ as well. Hence, $c = \frac{1}{2} \log n$ provides an upper bound for the minimum in step 2. The important immediate consequence is that we only need to consider V_α up to at most $\alpha = \frac{1}{2} \log n$.

Lemma 8.1. The `IdentifyClass` algorithm and the resulting V_α satisfy the following statements with probability at least $1 - 2/n$:

(i): The algorithm does not abort

(ii): $|\Delta(i,j,k)| \leq 2n$

(iii): For $\alpha > 0$, $v_{(i,j,k)} \in V_\alpha$, we have $2^{\alpha-3} n \leq |\Delta(i,j,k)| \leq 2^{\alpha+1} n$.

(iv): $|\Lambda_x(i,j) \cap \Delta(i,j,k)| \leq 100 \cdot 2^\alpha \sqrt{n} \log n$ for $i, j \in [n^{1/4}]$ and $\alpha \in \mathbb{N}$.

This provides an adapted version of lemma 4.12 for the $\alpha > 0$ case.

The following lemma provides a tool that will allow for ”duplication” of information to avoid message congestion in the network in the `EvaluationB` procedure.

Lemma 8.2. For all $\alpha \geq 0$ and $i, j \in [n^{1/4}]$,

$$|V_\alpha[i,j]| \leq \frac{720\sqrt{n}\log n}{2^\alpha} \tag{8.3}$$

Proof. The $\alpha = 0$ case is immediate since $|\mathcal{U}'| = \sqrt{n}$, so consider $\alpha \geq 1$. The "promise" in the FEWP subroutine we are in guarantees that for all $(u,v) \in S, \Gamma(u,v) \leq 90\log n$, so that for any $i, j \in [n^{1/4}]$, each edge in $\mathcal{P}(U_i, U_j) \cap S$ has at most $90 \log n$ other nodes forming a negative triangle with it, leading to the inequality

$$\sum_{k: v_{(i,j,k)} \in V_\alpha} |\Delta(i,j,k)| \leq 90 n^{3/2} \log n.$$

Using $|\Delta(i,j,k)| \geq 2^{\alpha-3} n$ from part (i) of lemma 8.1, the conclusion follows. □

We now describe the implementation of step 3 of the `ComputePairs` procedure for the $\alpha > 0$ case.

3.1: Each node executes the `IdentifyClass` procedure.

3.2: For each α:
For every $l \in [m]$, every node $v_{(i,j,k)}$ executes a quantum search to find whether there is a $U'_k \in V_\alpha[U_i, U_j]$ with some $w \in U'_k$ forming a negative triangle (u_l^k, v_l^k, w) in G, and then reports all the pairs $u_l^k v_l^k$ for which such a U'_k was found.

The $\alpha = 0$ case was described in the main text. We proceed to describe the classical procedure for invoking theorem 4.11 to obtain the speedup for the general α case, as in (Izumi & Gall, 2019, §5.3.2). Some technical precautions must be taken to avoid congestion of messages between nodes. This crucially relies on information duplication to effectively increase bandwidth between nodes. Lemma 8.2 provides a strong bound for the size of each V_α. For this duplication of the information stored by the relevant nodes, a new labeling scheme is convenient. Suppose for simplicity that $C_\alpha := 2^\alpha/(720 \log n)$ is an integer, and assign each node a label $(\mathbf{u}, \mathbf{v}, \mathbf{w}, y) \in V_\alpha \times [C_\alpha]$, which is possible due to the bound of lemma 8.2. The following `EvaluationB` implementable in $\mathcal{O}(\log n)$ rounds (using a slightly sharper complexity analysis than (Izumi & Gall, 2019)) can then be used for invoking theorem 4.11:

`EvaluationB`
Input: A list $(\mathbf{w}_1^k, \ldots, \mathbf{w}_m^k)$ of elements of $V_\alpha[\mathbf{u}, \mathbf{v}]$ assigned to each node $k = (\mathbf{u}, \mathbf{v}, x)$.
Promise: $|L^k_{\mathbf{w}}| \leq 800 \cdot 2^\alpha \sqrt{n} \log n$ for each node k and all $|w \in V_\alpha[\mathbf{u}, \mathbf{v}]$.
Output: Every node $k = (\mathbf{u}, \mathbf{v}, x)$ outputs for each $\ell \in [m]$ whether some $w \in \mathbf{w}_l^k$ forms a negative triangle $\{u_\ell^k, v_\ell^k, w\}$.

0. Every node $(\mathbf{u}, \mathbf{v}, \mathbf{w}) \in V_\alpha$ broadcasts the edge information loaded in step 1 of `ComputePairs` to $(\mathbf{u}, \mathbf{v}, \mathbf{w}, y)$ for each $y \in [C_\alpha]$.
1. Every node $(\mathbf{u}, \mathbf{v}, x)$ splits each $L^k_{\mathbf{w}}$ into smaller sublists $L^k_{\mathbf{w},1}, \ldots, L^k_{\mathbf{w},C_\alpha}$ for each $\mathbf{w}$, with each sublist containing up to $\lceil |L^k_{\mathbf{w}}|/C_\alpha \rceil = \lceil 800 \cdot 720 \sqrt{n} \log^2 n \rceil$ elements,and sends each $L^k_{\mathbf{w},y}$ to node $(\mathbf{u}, \mathbf{v}, \mathbf{w}, y)$ along with the relevant edge weights.
2. Every $(\mathbf{u}, \mathbf{v}, \mathbf{w}, y)$ node returns the truth value

$$\min_{w \in \mathbf{w}} \{W_{uw} + W_{wv}\} \leq W_{vu}$$

to node k for each $uv \in L^k_{\mathbf{w},y}$ received in step 1.

For each value of α, we separately solve step 3.2 of the `ComputePairs` procedure. Since lemma 8.2 tells us that there are C_α times more nodes not in V_α than there are in V_α, every node in V_α can use C_α of those nodes not in V_α to relay messages and effectively increase its message bandwidth, which is exactly what `EvaluationB` takes advantage of. Steps 1 and 2 of the procedure take up to $2 \cdot \lceil |L^k_{\mathbf{w}}| \rceil / n \leq 1600 \cdot \log n$ rounds, since lists of size $\lceil |L^k_{\mathbf{w}}|/C_\alpha \rceil$ are sent to C_α nodes, and the bound on α gives $\lceil |L^k_{\mathbf{w}}| \rceil \leq 800 n \log(n)$.

8.2.1 Complexity of the Classical Analogue

This subsection of the appendix serves to provide some supplemental information to §4.7.2 discussing the complexity of an algorithm for APSP with routing tables in the CONGEST-CLIQUE as proposed in (Censor-Hillel et al., 2016) that has complexity $\tilde{\mathcal{O}}(n^{1/3})$. Note that (Censor-Hillel et al., 2016, corollary 6) applied to APSP *distance* computations only, whereas the routing table computations are discussed in (Censor-Hillel et al., 2016, §3.4). As shown there, $\mathcal{O}(\log^3)$ distance products (without witnesses) are needed to compute one distance product with a witness matrix. More precisely:

1. Obtaining a witness matrix when witnesses are unique requires $\log(n)$ distance products.
2. The procedure for finding witnesses in the general case calls the procedure to find witnesses in the unique witness case $\mathcal{O}(\log^2 n)$ times, or $2 \cdot \log^2 n$ times if $c = 2$ is deemed as sufficient for the success probability.
3. $\log n$ such distance products with witnesses are needed for the APSP algorithm with routing tables.

Then $2\log^4 n$ distance products are computed in total for one distance product with witnesses. The distance product via the semi-ring matrix multiplication algorithm of (Censor-Hillel et al., 2016, §2.1) uses $10n^{1/3}$ rounds ($4n^{1/3}$ for its steps 1 and 2, and $2n^{1/3}$ for step 3) using lemma 4.8, and hence one obtains the full round complexity of

$$10n^{1/3} \cdot 2\log(n)^4 = g(n). \tag{8.4}$$

8.3 Expanding the DMST in the Distributed Setting

We handle the expansion of the DMST in the same way as in (Fischer & Oshman, 2021, §7), borrowing much of their discussion for our description here. However, as we have computed APSP distances along the way in place of SSSP, 'unpacking' the DMST becomes a bit simpler in our case.

Consider a component H_i in one of the iterations of QDLSI, with input graph for the iteration being G_i. For each contraction in QDLSI, we determined edges $v_i u_i$, $v_i \notin H_i, u_i \in H_i$ minimizing $\beta_i := W_{v_i u_i} + d_G(u_i, c(H_i))$ to contract nodes. Recall that what happens in the iteration is that the cycle $c(H_i)$ and all nodes that have distance β_i to $c(H_i)$ are contracted into one super-vertex. Denote that super-vertex by $V^*_{H_i,\beta_i}$. Let G_{i+1} denote the graph obtained after this contraction. Then our goal, given a DMST T_{i+1} for G_{i+1}, is to recover G_i along with a DMST T_i for G_i. We make use of the following Unpacking operation of (Fischer & Oshman, 2021, §7):

```
Unpacking

Input: A digraph $G_{i+1}$ with a DMST $T_{i+1}$ with root $r$, a set of edges $H_i$ as in Quantum DMST Algorithm, a node $V^*_{H_i,\beta_i}$ of $G_{I+1}$ marked as a super-vertex, a set $c(H_i)$ of the nodes contracted into it, and $G_i$ the graph before contracting $c(H_i)$.

Output: A DMST $T_i$ for $G_i$ rooted at $r$.

  1: For any $v_1, v_2 \notin V^*_{H_i,\beta_i}$, let edge $v_1 v_2 \in T_i$ iff $v_1 v_2 \in T_{i+1}$.
  2: For $uV^*_{H_i,\beta_i} \in T_{i+1}$, which exists since $T_{i+1}$ is a DMST for $G_{i+1}$, denote the edge
     $uv^* := argmin_{uv: v \in V^*_{H_i,\beta_i}, u: \exists uv \in G_{i+1}} W_{vu} + d_G(u, c(H_i))$. Add $uv^*$ and the shortest path $\zeta$ connecting $v^*$
     to $c(H_i)$ to $T_i$.
  3: For any edge $V^*_{H_i,\beta_i} u \in T_{i+1}$ outgoing from the contracted super-vertex, add the edge
     $argmin_{vu: v \in V^*_{H_i,\beta_i}} W^{G_i}_{vu}$ to $T_i$.
  4: Add all edges $H_i \setminus \delta^{in}(\zeta)$ to $T_i$, where $\delta^{in}(\zeta)$ denotes all edges incoming on $\zeta$.
```

At the end of this procedure, T_i is a DMST for G_i (Fischer & Oshman, 2021, lemma 8). We now describe how it can be implemented distributedly, needing only classical messages and information. For every contracted super-vertex, the following steps can be implemented at the same time, as will become clear in how the steps are executed for the nodes of each contracted super-vertex. Let us focus on unpacking one super-vertex $V^*_{H_i,\beta_i}$. Each node knows its neighbors in G_i, and every node's super-vertex ID in G_i and G_{i+1}, since each node stores this information before the initial contraction to G_{i+1} in QDLSI happens. Hence, step 1 can be done locally at each node without any communication. Step 2 can be done by first having each node $v \in V^*_{H_i,\beta_i}$ send $\beta(u, v)$ to the other nodes in $V^*_{H_i,\beta_i}$, in one round, and then having each node of $V^*_{H_i,\beta_i}$ send to v^* the routing table entry corresponding to its shortest path to $c(H_i)$ in G_i, also in one round (the nodes have already computed this information in QDLSI. Then v^* notifies the nodes that are part of ζ, which can then add the appropriate edge to T_i, needing yet another round, so that step 2 can be done in three rounds of classical communication only. Step 3 is handled similarly. For the outgoing edge, each node in $V^*_{H_i,\beta_i}$ sends $W^{G_i}_{vu}$ to the other nodes in $V^*_{H_i,\beta_i}$ so that the appropriate edge to add to T_i can be determined (in case of a tie, the node with smaller ID can be the one to add the edge), so this can be done in one round. For step 4, every node in ζ notifies its neighbors that it is in ζ, after which every node can determine which edges to add to T_i. For the unpacking of $V^*_{H_i,\beta_i}$, the information and communication for implementing its unpacking is contained in the nodes of $V^*_{H_i,\beta_i}$, so we can indeed unpack all vertices synchronously to obtain G_i even when multiple super-vertices were contracted to get G_{i+1}. Hence, one layer of unpacking using this procedure can be implemented in 5 rounds (making use of the APSP and routing table information computed earlier before the contractions in `QDLSI`). Since there are at most $\lceil \log n \rceil$ contraction steps, the unpacking procedure can be implemented in $5 \cdot \lceil \log n \rceil$ rounds.

8.4 Information access

In remark 2.2, we mention that it suffices for all information regarding the input graph to be stored classically, with quantum access to it. Here, we expand on what we mean by that and refer the interested reader to (Booth, O'Gorman, Marshall, Hadfield, & Rieffel, 2021) for further details.

While our algorithms use quantum subroutines, the problem instances and their solutions are encoded as classical information. The required quantum access refers to the ability to access the classical data so that computation in superposition of this data is possible. For instance, in the standard (non-distributed) Grover search algorithm, with a problem instance described by a function $g : X \to 0, 1$, we need the ability to apply the unitary $U_w|x\rangle = (-1)^{g(x)}|x\rangle$ to an N-qubit superposition state $|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$. This unitary is also referred to as the “oracle”, and a call to it as a “query”. If we wish to use the distributed Grover search in example 2.4, in which the node u leading the search tries to determine whether each edge uv incident on it is part of a triangle in graph G, the unitary that node v must be able to evaluate is the indicator function of its neighborhood, and u must be able to apply the Grover diffusion unitary restricted to its neighborhood. Then after initializing the N-qubit equal superposition, nodes u and v can send a register of qubits back and forth between each other, with v evaluating the unitary corresponding to the indicator of its neighborhood and u applying the Grover diffusion operator restricted to its neighborhood.The same ideas transfer over to a distributed quantum implementation of the `EvaluationA` (or `EvaluationB`) procedure. There, instead of evaluating unitaries corresponding to indicators, in step 2, each node $v_{(i,j,k)}$ evaluates the unitary corresponding to the truth values of inequality 4.3 for the evaluation steps. That information is then returned to the node that sent it, which can then apply the appropriate Grover diffusion operator.

In general, quantum random access memory (QRAM) is the data structure that allows queries to the oracle. We can use circuit QRAM in our protocols or could make use of special-purpose hardware QRAM if it were to be realized. This choice does not affect the number of rounds of communication but would affect the efficiency of computation at each node. A main component of the distributed algorithms discussed in this work is quantum query access for each node to its list of edges and their weights in some graph G. This information is stored in memory, and the QRAM implementing the query to retrieve it can be called in time $\mathcal{O}(\log n)$, resulting in a limited overhead for our algorithms. This retrieval of information takes place *locally* at each node; hence, this overhead does not add to the round complexity of our algorithms in the CONGEST-CLIQUE setting. We refer to (Giovannetti, Lloyd, & Maccone, 2008) for more details on QRAM.