A Deterministic Algorithm for the MST Problem in Constant Rounds of Congested Clique

Krzysztof Nowicki*
University of Wroclaw
knowicki@cs.uni.wroc.pl

Abstract

In this paper, we show that the Minimum Spanning Tree problem can be solved deterministically, in $O(1)$ rounds of the Congested Clique model.

In the Congested Clique model, there are $n$ players that perform computation in synchronous rounds. Each round consist of a phase of local computation and a phase of communication, in which each pair of players is allowed to exchange $O(\log n)$ bit messages.

The studies of this model began with the MST problem: in the paper by Lotker et al. [SPAA’03, SICOMP’05] that defines the Congested Clique model the authors give a deterministic $O(\log \log n)$ round algorithm that improved over a trivial $O(\log n)$ round adaptation of Borůvka’s algorithm.

There was a sequence of gradual improvements to this result: an $O(\log \log \log n)$ round algorithm by Hegeman et al. [PODC’15], an $O(\log^* n)$ round algorithm by Ghaffari and Parter, [PODC’16] and an $O(1)$ round algorithm by Jurdziński and Nowicki, [SODA’18], but all those algorithms were randomized, which left the question about the existence of any deterministic $o(\log \log n)$ round algorithms for the Minimum Spanning Tree problem open.

Our result resolves this question and establishes that $O(1)$ rounds is enough to solve the MST problem in the Congested Clique model, even if we are not allowed to use any randomness. Furthermore, the amount of communication needed by the algorithm makes it applicable to some variants of the MPC model.

*This research is supported by the Polish National Science Centre, under projects number 2017/25/B/ST6/02010 and 2019/32/T/ST6/00566.
1 Introduction and related work

In this paper, we present a simple deterministic algorithm for the Minimum Weight Spanning Tree problem that needs only a constant number of rounds of Congested Clique.

The Congested Clique model is a model of distributed (or parallel) computation, in which we have $n$ players (processors) performing computation in synchronous rounds. Each player corresponds to a single vertex of the input graph, and initially knows all edges that are incident to this vertex.

A single round consists of a phase of local computation, in which all players simultaneously perform computation, and a phase of communication, in which each pair of players can simultaneously exchange a pair of messages of size $O(\log n)$ bits. In other words, it is a synchronous message passing model, in which the communication graph is a clique, the communicating players send and receive messages simultaneously, and the number of bits in a single message is $O(\log n)$.

As a result, we usually require that some player knows the answer computed by the algorithm. The exception are the problems for which the answer is too large to fit into memory of a single processor; then we allow that each player needs to know only a part of the result, but there are no general rules for this kind of problems and the specific requirements towards the output are usually tailored to the problem.

1.1 Congested Clique model

The Congested Clique model was proposed by Lotker et al. [LPPSP03, LPPP05]. Initially it was considered as a special case of the CONGEST model (in which the input graph is also the communication graph). The question that initiated the studies on Congested Clique was to establish the complexity of the MST algorithm in the CONGEST model, when the input graph is a clique. Apart from being just a special case of the CONGEST model, Congested Clique might be used to model the overlay networks and has ties to some models of parallel computing.

**Overlay networks:** Lotker et al. propose that Congested Clique may be a good theoretical model to study the overlay networks: an abstraction that separates the problems emerging from the topology of the communication network from the problems emerging from the structure of the problem we try to solve. In other words, it allows us to study a model in which each pair of nodes can communicate, and we do not consider any details of how this communication is executed by the underlying network.

**Parallel Computing:** Congested Clique can be also considered as a model of parallel computing, in particular, it is closely related to the Massively Parallel Computation (MPC) model. In MPC the computation is performed by a set of machines in synchronous rounds; each round consisting of a phase of local computation and a phase of communication. In the communication phase each pair of machines can exchange some number of messages, as long as each machine sends and receives a number of messages bounded by some parameter $S$, and total communication is bounded by $O(N)$, where $N$ is the size of the input.

In the Congested Clique model, the restrictions on the number of messages that can be exchanged between the processors are stronger, as each pair of processors can exchange only one message. The problem of exchanging larger amounts of messages is called the routing problem, and some of its variants can be solved in $O(1)$ rounds by the routing protocol proposed by Lenzen [Len13]. More precisely, we can route all messages to their destinations whenever this problem looks a little bit like communication in the MPC model, i.e. when each processor is a source and destination of $O(n)$ messages. This immediately gives that one can simulate some variants of the
MPC model (with $S \in O(n)$) in Congested Clique. In the remaining part of the paper, we use this result implicitly in several places, i.e. we show that each machine sends and receives only $O(n)$ messages, therefore exchanging all messages can be done in $O(1)$ rounds.

The connection between the models is bidirectional, as any algorithm that has small communication in the Congested Clique model can be applied to the MPC model [HP14, BDH18]. To clarify this statement: the routing protocol of Lenzen on its own requires $\Theta(n^2)$ messages, no matter what is the number of messages to be send. But in MPC model, we get communication that is handled by the routing protocol for free, therefore, by small communication we mean that in a single round the total number of messages that are send directly plus the total number messages that were routed by the routing protocol is $O(N)$. Therefore, we decided to measure the communication complexity of the algorithms without taking into account the cost of the routing protocol by Lenzen.

1.2 Minimum Spanning Tree problem in Congested Clique

In the Minimum Weight Spanning Tree problem, for a connected weighted input graph we have to compute the lightest acyclic set of edges that connects all vertices of the input graph. In the remaining part of the paper we will refer to this problem as the Minimum Spanning Tree problem or MST.

This problem was studies for over 90 years, and the first MST algorithm was proposed by Otakar Borůvka in 1926 [NMN01]. While the approach proposed by Borůvka seems to be a natural candidate for designing parallel algorithms, it turns out that for our purpose the more suitable choice is an algorithm proposed by Joseph Kruskal in 1956 [Kru56] – the properties of Kruskal’s algorithm that make it useful in the Congested Clique model were firstly observed by Hegeman et al. [HPP+15], and we discuss them in more detail in Section 5.

Congested Clique algorithms for Minimum Spanning Tree problem: In the initial paper on the Congested Clique model, Lotker et al. [LPPP05] proposed a deterministic $O(\log \log n)$ round algorithm for the MST problem.

While there was a sequence of papers improving on this complexity: an $O(\log \log \log n)$ round algorithm by Hegeman et al. [HPP+15], an $O(\log^* n)$ round algorithm by Ghaffari and Parter [GP16], and finally, an $O(1)$ round algorithm by Jurdziński and Nowicki [JN18], all those algorithms were randomized. The only deterministic alternative to the algorithm by Lotker et al. was proposed by Korhonen [Kor16], but the best upper bound on the round complexity for his algorithm is also $O(\log \log n)$. Therefore, the question about the existence of any deterministic $o(\log \log n)$ round algorithms for the MST problem remained open.

Our result resolves this question, and establishes that $O(1)$ rounds is enough to solve the MST problem in the Congested Clique model, even if we are not allowed to use any randomness. Besides the optimal bounds on the round complexity we achieve, we would like to emphasize that our algorithm is much simpler than all previous $o(\log \log n)$ algorithms.

2 Our results

An algorithm for the Spanning Forest problem: The main contribution of this paper is a deterministic algorithm for the Spanning Forest problem that requires only $O(1)$ rounds of communication in the Congested Clique model.

---

1To clarify: we want to compute an acyclic subgraph that has the same connected components as the input graph rather than a subgraph that is a forest and contains all vertices of the input graph; this variant is sometimes called a Maximal Spanning Forest problem.
Theorem 2.1. The Spanning Forest problem can be solved deterministically in $O(1)$ rounds of Congested Clique.

To obtain this result, we propose a rather straightforward extension of the sparsification technique by Korhonen [Kor16], which we combine with an approach based on some newly discovered properties of a deterministic part of the algorithm proposed by Jurdziński and Nowicki [JN18, JN17]. We give an algorithm that is a proof of Theorem 2.1 in Section 4.

An algorithm for the Minimum Spanning Tree problem: The most significant result presented in this paper is a deterministic algorithm for the Minimum Spanning Tree problem that requires only a constant number of rounds.

Theorem 2.2. The Minimum Spanning Forest problem can be solved deterministically in $O(1)$ rounds of Congested Clique.

This result follows from algorithm of Theorem 2.1, by a quite standard reduction by Hegeman et al. [HPP+15]. The reduction states that if we can solve in Congested Clique several instances of the Connected Components problem in parallel, then we can solve the MST problem. The original reduction was randomized, as it was based on the sampling approach by Karger, Klein, and Tarjan [KKT95], but this step can be replaced by Korhonen’s approach [Kor16]. We give an algorithm that is a proof of Theorem 2.2 in Section 5.

Application to the MPC model: Our algorithm for the MST, apart from the communication following from the Lenzen’s routing protocol, needs every processor to send and receive only $O(n)$ messages, and exchange $O(m)$ messages in each round. Therefore, it can be implemented in the MPC model with $O(n)$ memory per machine and $O(m)$ global memory.

Corollary 2.3. The Minimum Spanning Forest problem can be solved deterministically in $O(1)$ rounds of the MPC model that uses $O(m)$ global memory and $O(n)$ memory per machine.

2.1 Structure of the remaining part of this paper

In Section 3 we present a variant of Korhonen’s sparsification technique. Then, in Section 4 we present the algorithm for the Spanning Forest problem. Finally, in Section 5 we briefly explain how the algorithm for the Spanning Forest problem can be run in parallel to fit into the reduction by Hegeman et al.

3 Deterministic Sparsification in the Congested Clique

In this section we present a variant of the sparsification technique by Korhonen [Kor16] that can be applied to sparser graphs. Furthermore, we show that the communication complexity of this algorithm is $O(m)$.

Lemma 3.1. There is a deterministic, $O(1)$ round Congested Clique algorithm, using $O(m)$ messages that reduces an instance of the MST problem on a graph with $n$ vertices and $m$ edges to an instance of the MST problem that has $O(n)$ vertices and $O(\sqrt{mn})$ edges.

Our variant, if executed as in the paper by Korhonen, needs only $O(\log \log \Delta_A)$ rounds to complete computation, where $\Delta_A$ is an average degree of the input graph. More importantly, our implementation allows us to reduce an instance of the MST problem with $n$ vertices and $m$ edges to an instance with $O(n)$ vertices and $O(\sqrt{mn})$ edges in $O(1)$ rounds.
The remaining part of this section is a proof of Lemma 3.1. Firstly, we recall the sparsification algorithm by Korhonen, then we show some initial reduction that allows us to use it to obtain an algorithm that is a proof of Lemma 3.1.

### 3.1 Deterministic Sparsifying via $\Delta$–partitions

A $\Delta$–partition of the set vertices $V$ is a partition into disjoint sets $V_1, V_2, \ldots, V_\Delta$, such that

- for each $i$, $|V_i| \in \mathcal{O}(n/\Delta)$,
- for each $i, j \geq 2$, $|\{\{u, v\}, u \in V_i, v \in V_j\}| \in \mathcal{O}(n/\Delta)$.

This notion of $\Delta$–partition corresponds to the notion of $\varepsilon$–partition from the paper by Korhonen, with $\Delta = n^{\varepsilon}$. One of the useful properties is that a graph with $\Delta$–partition has only $\mathcal{O}(n\Delta)$ edges, as if we sum the number of edges between the sets $V_i, V_j$, over all $i$ and $j$, we get $\mathcal{O}(n/\Delta) \cdot \Delta^2 = \mathcal{O}(n\Delta)$.

The main contribution of the paper by the Korhonen is an $\mathcal{O}(1)$ round Congested Clique algorithm that given a graph with $\Delta$–partition computes a graph with a $\sqrt{\Delta}$–partition, while preserving all edges of the minimum spanning forest of the input graph. We state this algorithm as Algorithm 1.

**Algorithm 1: Sparsify(G, $\Delta$–partition)[Kor16]**

1. let $V'_i = \bigcup_{j=(i-1)\sqrt{\Delta}+1}^{i\sqrt{\Delta}} V_j$
2. partition the edges in such a way that for all $i \leq j$ the edges $E_{i,j} = \{\{u, v\}, u \in V'_i, v \in V'_j\}$ are in the memory of a single processor
3. for all $i \leq j$ compute minimum spanning forest $F_{i,j}$ of a graph consisting of edges $E_{i,j}$
4. return graph $(V, \bigcup_{i,j} \text{edges of } F_{i,j})$ with partition $V'_1, \ldots, V'_{\sqrt{\Delta}}$

Firstly, we recall some properties of the sparsification technique by Korhonen (Lemma 3.2), then we discuss its implementation in Congested Clique.

**Lemma 3.2.** [Kor16] Algorithm 1 returns a graph $G'$ with a $\sqrt{\Delta}$–partition, such that the minimum spanning forest of $G'$ is also the minimum spanning forest of $G$.

**Proof.** To claim that we preserve minimum spanning tree we use the cycle property[Tar83]. In a brief summary, this property says that any edge that is the heaviest edge on some cycle in a graph $G$ cannot be in the minimum spanning tree of $G$. Here, we firstly observe that if an edge $\{u, v\}$ is not a part of the minimum spanning forest of $E_{i,j}$, then $u$ and $v$ have to be connected over the edges of $E_{i,j}$, and $\{u, v\}$ is heavier than all the edges on the path connecting $u$ and $v$. Therefore, in $G$ exists a cycle in which the edge $\{u, v\}$ is the heaviest edge. Hence, $\{u, v\}$ cannot be a part of the minimum spanning tree of $G$.

$\sqrt{\Delta}$–partition: between the sets $V'_i, V'_j$ we only keep the edges from $F_{i,j}$. Since $|V'_i \cup V'_j| \in \mathcal{O}(n/\sqrt{\Delta})$, the size of the spanning forest $F_{i,j}$ on vertices from $V'_i \cup V'_j$ is also $\mathcal{O}(n/\sqrt{\Delta})$. This concludes the proof of Lemma 3.2.

**Congested Clique implementation:** The partition of vertices into sets depends only on the identifiers of vertices, therefore it can be carried out locally. To compute the spanning forests of $E_{i,j}$, Korhonen proposed that each pair $i, j$ gets a dedicated vertex of the clique (we call such vertex

\[2\text{note that this also means } i = j\]
a coordinator), which gathers all edges of $E_{i,j}$ and computes its minimum spanning forest in the local memory. To show that this implementation can be carried out, it is enough to show that $|E_{i,j}| \in \mathcal{O}(n)$.

The set of vertices incident to edges of $E_{i,j}$ consists of $\mathcal{O}(\sqrt{\Delta})$ sets $V_{\alpha}, V_{\alpha+1}, \ldots, V_{\alpha+\Theta(\sqrt{\Delta})}$, that are part of a $\Delta$–partition of the graph. By definition of $\Delta$–partition there are at most $\mathcal{O}(n/\Delta)$ edges between vertices from sets $V_{\alpha_1}, V_{\alpha_2}$, for any $\alpha_1, \alpha_2 \in \{\alpha, \alpha + 1, \ldots, \alpha + \Theta(\sqrt{\Delta})\}$. Therefore, $|E_{i,j}| \in \mathcal{O}(\sqrt{\Delta}^2 \cdot n/\Delta) = \mathcal{O}(n)$.

Our observation is that in order to carry out the implementation of this step, we need only $\mathcal{O}(m/n)$ coordinator vertices, rather than $\Theta(\Delta^2)$. The reason is that each edge of the graph ends up being a member of $E_{i,j}$ for exactly one pair $i,j$. Therefore, the total size of the sets of edges we have to gather is $\mathcal{O}(m)$, and the maximal size is still $\mathcal{O}(n)$. Hence, $\mathcal{O}(m/n)$ coordinator vertices are enough to store the edges in all sets $E_{i,j}$. The assignment of pairs $i,j$ to processors can be done by a parallel prefix computation, i.e. for each pair $i,j$ we compute $\sum_{(i',j') \leq (i,j)} |E_{i',j'}|$ which is enough to compute the ID of processor that needs to handle $E_{i,j}$.

### 3.2 Obtaining a graph with an $\left(\frac{m}{n}\right)$–partition

In this subsection, we provide a simple reduction that transforms an $n$–vertex $m$–edge graph $G$ to an $\mathcal{O}(n)$–vertex, $\mathcal{O}(m)$–edge graph $G'$ with $(m/n)$–partition, such that knowing the edges of the MST of $G'$ allows us identify the edges of the MST of $G$. Then, we apply Algorithm 1 on $G'$, which gives us a graph with $\sqrt{(m/n)}$–partition, hence having only $\mathcal{O}(\sqrt{mn})$ edges, which concludes the proof of Lemma 3.1.

**Initial reduction:** Firstly, we transform a graph $G$ with $n$ vertices, $m$ edges, and average degree $\Delta_A = m/n$ into a graph $G'$ with $\mathcal{O}(n)$ vertices, and maximal degree $\Delta_A$, in a way that computing the edges of the MST of $G'$ allows us to identify the edges of the MST of $G$. To obtain $G'$, we split each vertex with more than $\Delta_A$ vertices into several vertices of degree $\Delta_A$, connected by a path; we assign to each edge on this path a weight that is smaller than all weights in the input graph.

**Initial reduction – implementation:** To obtain a partition of vertices into vertices of degree at most $\Delta_A$, it is enough to gather all degrees of vertices in the memory of a single processor. This processor then decides for each vertex what is the number of vertices it has to be splitted to, and what are the ID’s of the newly created vertices. We assign the new ID’s in a way that each vertex gets splitted into several vertices that get new ID’s that form a sequence of consecutive numbers. This allows to communicate the number of vertices and their identifiers as two messages: one that is the number of vertices to be created, and the other that is the smallest ID of a created vertex. Therefore, the total number of messages needed to be send by the processor that computes the splitting is $\mathcal{O}(n)$, and communicating those messages can be done in $\mathcal{O}(1)$ rounds.

**Initial reduction – preserving MST:** The Kruskal’s algorithm considers the edges form the lightest to the heaviest, hence it considers all edges created by the initial reduction before the edges that correspond to the edges of the input graph. After processing all created edges it computes a set of connected components that corresponds to the vertices of $G$. The remaining edges correspond to the edges of $G$, and all edges included in the MST of $G'$ from this point correspond to the edges of the MST of $G$.

**Initial reduction – number of vertices:** Since we only split the vertices that have more incident edges than $\Delta_A$, creating at least one vertex with degree $\Delta_A$, we can create at most $n$ new vertices. That is because creating more would imply that in $G$ there are more than $n\Delta_A$ edges. Therefore, in the obtained graph we have $\mathcal{O}(n)$ vertices with maximal degree $\Delta_A$. 


Computing \( \left( \frac{m}{n} \right) \)-partition: Let us consider an arbitrary partition of vertices into sets of size \( \Theta(n/\Delta_A) \). If we treat those sets as sets \( V'_i \) from Algorithm 1, and execute remaining steps of the algorithm, the result is a graph with a \( \Delta \)-partition.

The guarantees for the resulting graph follow from exactly the same analysis. To show that it can be implemented, it is enough to show that \( |E_{i,j}| \in \mathcal{O}(n) \). For any pair of sets the number of vertices is \( \mathcal{O}(n/\Delta_A) \) and maximal degree is \( \mathcal{O}(\Delta_A) \). Hence, the total number of edges incident to those vertices has to be \( \mathcal{O}(\Delta_A) \cdot \mathcal{O}(n/\Delta_A) = \mathcal{O}(n) \).

The other issue is that a single vertex of the clique can simulate several vertices. The number of messages that have to be sent to the coordinators and received from the coordinators is bounded by the degree of vertices. Since the sum of degree of all vertices simulated by a single vertex of degree \( \delta \) is at most \( \delta + \delta/(\frac{mn}{n}) \in \mathcal{O}(\delta) \), the overall the number of messages to be send by a single vertex increases only by a constant factor, and communication still can be executed in \( \mathcal{O}(1) \) rounds.

4 Deterministic algorithm for the Spanning Forest problem

In this section we propose a deterministic algorithm for the Spanning Forest problem in \( \mathcal{O}(1) \) rounds of Congested Clique.

**Theorem 2.1.** The Spanning Forest problem can be solved deterministically in \( \mathcal{O}(1) \) rounds of Congested Clique.

The remaining part of this section is a proof of Theorem 2.1. Our algorithm is based on a simple deterministic algorithm proposed by Jurdziński and Nowicki [JN17, JN18]. In the Congested Clique model it was used for reducing a single instance of a Spanning Forest problem to two instances, one with degree bounded by \( s \), the other with known partition into \( n/s \) components, for any parameter \( s \). For graphs that are almost regular, i.e. in which all vertices have degree \( \Theta(\delta) \), for some parameter \( \delta \), this technique together with sparsification algorithm can solve the Spanning Forest problem. Using the technique we can compute a partition into components of size \( \Omega(\delta) \), which gives us a component graph that has therefore obtaining a graph with \( \mathcal{O}(n\delta) \) edges and \( \mathcal{O}(\frac{n}{\delta}) \) vertices, which after application of sparsification algorithm would become a graph with \( \mathcal{O}(\sqrt{\frac{n\delta}{\delta}}) = \mathcal{O}(n) \) edges.

In this section, we show how to use this approach to handle graphs that are not necessarily almost regular. To do so, we partition the edges of the input graph into \( \Theta(\log n) \) disjoint sets: in the \( i \)th set we include the edges that connect components of size at least \( 2^{i-1} \) and less than \( 2^i \). Then, we apply on such graphs the sparsification technique from Lemma 3.1, which reduces the number of remaining edges to \( \mathcal{O}(n) \).

4.1 A technique reducing the number of components

Firstly, let us recall the relevant part of the algorithm by Jurdziński and Nowicki. This algorithm consists of two stages, in each stage for each vertex we choose some edge, and using selected edges we compute connected components. In the first stage, for each vertex \( v \) we select an edge connecting \( v \) to a neighbour with the highest degree. Then, in the second stage, if there are some edges incident to \( v \) that were not used in the first stage, we select for each \( v \) one of those edges. We state a more precise formulation of this algorithm as Algorithm 2. The key property of this algorithm is that it computes a partition into connected components with a guarantee that a vertex is part of a component that is at least as large as its degree (see Lemma 4.1).

**Lemma 4.1.** [JN18] After execution of Algorithm 2, a vertex that has degree \( \delta \) becomes a member of a component of size at least \( \delta + 1 \).
Algorithm 2: Reduce_components(G)

1: each vertex v marks an edge connecting it to a neighbour with the highest degree (break ties towards higher ID)
2: each vertex v notifies all neighbours, whether the edge between them was marked
3: each vertex v marks an edge connecting it to a vertex u that didn’t mark the edge \{u, v\} (if such u exists)
4: each vertex v sends the marked edges to the coordinator vertex
5: the coordinator vertex computes connected components using gathered edges

Proof. Here, for the sake of completeness, we give a proof of Lemma 4.1. Consider a vertex v and let u be the vertex with the lexicographically largest \((\text{degree}(u), \text{ID}(u))\) in the connected component of node v. We claim that all neighbors of u in the original graph are in the same connected component. Otherwise, u has neighbors that did not choose u in the first step; let w be the neighbor among these that u chose in the second step. Node w chose to connect to some other vertex \(u'\) such that \((\text{degree}(u'), \text{ID}(u')) > (\text{degree}(u), \text{ID}(u))\). But now u is connected to \(u'\) and the existence of such a node \(u'\) in this component is in contradiction with the choice of u. Thus, all neighbors of u are in the same component, which means this component has at least \text{degree}(u) + 1 \geq \text{degree}(v) + 1\) vertices. □

4.2 Our application

The intuition is that applying Lemma 4.1 should cause a significant reduction of the number of connected components in some sufficiently dense subgraphs. Let consider a component graph \(G'\) that is a result of Algorithm 2, i.e. a graph in which vertices correspond to the connected components computed by Algorithm 2, where we removed all self loops. Let assign to each vertex v of \(G'\) a weight that is equal to the number of vertices of \(G\) that are in the component corresponding to v. Let \(V_i\) be a set of vertices of \(G'\) of weight at least \(2^{i-1}\), and less than \(2^i\), and let \(x_i\) be a sum of weights of vertices in \(V_i\), and \(y \in O(\log n)\) be the maximal index of non empty \(V_i\).

Let us consider graphs \(G_1, \ldots, G_y\), where \(G_i\) consists of vertices of \(V_i\), the vertices of \(\bigcup_{j=i}^{y} V_j\) that are neighbours of vertices of \(V_i\) and all edges that have one endpoint in \(V_i\) and other in \(\bigcup_{j=i}^{y} V_j\). We observe two properties of graphs \(G_i\), states as Facts 4.2 and 4.3.

Fact 4.2. The number of edges of \(G_i\) is smaller than \(x_i 2^i\).

Proof. By definition, each edge of \(G_i\) has to have an endpoint in set \(V_i\), which consists of vertices that have weight less than \(2^i\). In other words, each vertex in \(V_i\) corresponds to a component computed by Algorithm 2 that has size less than \(2^i\). By Lemma 4.1, all vertices that are in such components has to have degree smaller than \(2^i\). Since \(x_i\) is exactly the number of the vertices of the original graph that form the components corresponding to the vertices in \(V_i\), the total number of edges incident to those vertices is smaller than \(x_i 2^i\). □

Fact 4.3. The number of vertices of \(G_i\) is at most \(\frac{1}{2^y} \sum_{j=i}^{y} x_j / 2^{j-1-i}\).

Proof. By definition, the vertices of \(V_i\) have to have weight at least \(2^{i-1}\), which means that each correspond to a component computed by Algorithm 2 of size at least \(2^{i-1}\). Since \(x_i\) is exactly the number of the vertices of original graph that form components corresponding to the vertices in \(V_i\), the total number of vertices of \(G'\) in \(V_i\) is at most \(x_i / 2^{i-1}\). By definition, the set of vertices of \(G_i\) consists only of vertices from \(\bigcup_{j=i}^{y} V_j\), hence it cannot be larger than \(\sum_{j=i}^{y} x_j / 2^{j-1-i} = \frac{1}{2^y} \sum_{j=i}^{y} x_j / 2^{j-1-i}\). □
Reducing the number of edges: The next step of the algorithm is to apply Lemma 3.1 to all graphs \( G_1, G_2, \ldots, G_y \), in parallel. Let \( G_1^R, G_2^R, \ldots, G_y^R \) be a sequence of obtained graphs.

**Lemma 4.4.** The total number of edges in \( G_1^R, G_2^R, \ldots, G_y^R \) is \( O(n) \).

**Proof.** Using Facts 4.2 and 4.3 and Lemma 3.1 we have that applying Algorithm 1 to \( G_i \), gives a reduced graph \( G_i^R \) with the number of edges that can be bounded by \( O\left(\sqrt{x_i 2^i \cdot \sum_{j=1}^y x_j / 2^j - 1 - i}\right) \), and the expression under the \( O \) notation can be bounded as follows.

\[
\sqrt{x_i 2^i \cdot \frac{1}{2^i} \cdot \sum_{j=1}^y x_j / 2^j - 1 - i} \leq \sqrt{\left(\sum_{j=1}^y x_j / 2^j - 1 - i\right) \cdot \left(\sum_{j=1}^y x_j / 2^j - 1 - i\right)} = \sum_{j=1}^y x_j / 2^j - 1 - i
\]

Therefore, the total number of edges in all reduced graphs is \( O(\sum_{i=1}^y \sum_{j=1}^y x_j / 2^j - 1 - i) \), but if we look on the contribution to the sum from the point of view of the vertices from \( V_i \), we get the following:

\[
O\left(\sum_{i=1}^y \sum_{j=1}^y x_j / 2^j - 1 - i\right) = O\left(\sum_{j=1}^y \sum_{i=1}^y x_j / 2^j - 1 - i\right) = O\left(\sum_{j=1}^y 4x_j\right) = O(n)
\]

**Congested Clique implementation:** Here we briefly discuss the execution of the algorithm presented in this section in the Congested Clique model. Firstly, Algorithm 2 can be clearly implemented in Congested Clique, as it requires only communication over the edges of the input graph, and one coordinator vertex that computes the partition into connected components. After that, each vertex has knowledge on which edges belong to each \( G_i \).

Then, we need to run several instances of sparsification algorithm from Lemma 3.1 in parallel. There are two parts that we need to address: reduction of the graph to one with maximal degree that is at most as large as the average degree of the input graph, and running algorithm Algorithm 1 in parallel.

The main difference for the degree reducing part is that this time we work on component graphs, so it could be that a single vertex does not see all incident edges. Let consider a single \( G_i \). If a single vertex gathers degrees of all vertices in \( G_i \) (omitting the edges that are inside components), together with ID of a component that holds the vertex, the processor that sees all those degrees sees what is the degree of the component, and can compute the number of parts it has to be splitted to. Then, each vertex gets the ID of a part it belongs to (or number of parts it has to be split to, if it is a high degree vertex). Such a reduction, even though does not provide that a simulated vertex sees all incident edges, is sufficient to carry out Algorithm 1. Thus, this reduction can be executed for a single \( G_i \). Still, we want to show that executing it for all \( i \) simultaneously can be done with small communication. This follows from that a message is sent by a vertex \( v \) to the coordinator of the \( i \)th instance only if there is an edge incident to \( v \) in \( G_i \). This gives that the amount of communication cannot be larger than the number of edges in all graphs \( G_i \). Since \( G_i \) are edge disjoint, it is \( O(m) \).

To see that Algorithm 1 can be executed in parallel, we only need to observe that for a single vertex, the communication is proportional to its degree, and each coordinator receives \( O(n) \) edges in total. Therefore, even though a single vertex may participate in many instances of the algorithm, its communication cannot be larger than sum of degrees of all vertices that it simulates. Furthermore,
the number of coordinators is still $O(m/n)$, as our reduction increase the number of edges at most by an additive $O(n)$, all instances that we process are edge disjoint, and a single instance needs space proportional to the number of edges.

5 The Algorithm for the Minimum Spanning Tree problem

In this section we show that our algorithm for the Spanning Forest problem can be applied to the Minimum Spanning Forest, proving Theorem 2.2.

Theorem 2.2. The Minimum Spanning Forest problem can be solved deterministically in $O(1)$ rounds of Congested Clique.

We prove Theorem 2.2, using already proven Lemma 3.1 and Theorem 2.1. Firstly, let us recall the reduction from a single instance of the MST problem to several instances of the Connected Components problem [HPP+15].

Lemma 5.1. There is a deterministic, $O(1)$ round Congested Clique algorithm that for given $n$ vertex $m$ edge graph $G$ reduces the problem of computing the MST of $G$ to $O(\sqrt{m/n})$ independent instances of the Connected Components problem, defined by sets of edges of total size $O(m)$.

Proof. On the top level, the reduction relies on some properties of Kruskal’s algorithm for the MST problem. In particular it uses that while the algorithm considers some edge $e$, the only information needed to decide whether $e$ is an edge of the minimum spanning tree is partition into connected components spanned by the edges that are lighter than $e$.

We apply this idea in a following way. Firstly, we sort the edges by weight, using $O(1)$ sorting algorithm by Lenzen [Len13]. Then, we split the sorted sequence of edges into sets $E_1, E_2, \ldots, E_{m/n}$, each of size $O(n)$. For each $i$ we compute a spanning forest $F_i$ of a graph with edges $\bigcup_{j=1}^{i-1} E_j$, applying the Spanning Forest algorithm from Theorem 2.1 in parallel. Then a single processor can fit $F_i$ and $E_i$ in the local memory, and simulate the steps of Kruskal’s algorithm on $E_i$.

The only issue with applying this approach in a straightforward way is that starting from $m$ edge graph, this gives $\frac{m^2}{n}$ instances of total size that could be $\Theta((m/n)^2) \cdot \Theta(n)$. To bypass this issue, Hegeman et al. used a random sampling approach proposed by Karger Klein and Tarjan [KKT95], which can be used to reduce a single instance of the MST problem to two instances of this problem that have to be executed one after the other, each of size $O(\sqrt{mn})$. Here, we instead use a variant of Korhonen’s algorithm described in Lemma 3.1, which gives us a reduction to a graph with $O(\sqrt{mn})$ edges in $O(1)$ rounds, deterministically.

Applying this technique on an $O(\sqrt{mn})$-edge graph gives that the resulting instances of the Connected Components problem have only $O((\sqrt{m/n})^2) \cdot \Theta(n) = O(m)$ edges in total. The only remaining part is to show that we can duplicate some of the edges in a way that gives us independent instances, so we can solve them easily in parallel. Our goal is to have $k$ copies of an edge that appears in $k$ instances; this way we can treat the copies (of vertices and edges) appearing in several instances as completely non related objects. This allows us to provide a rather clean way of running the algorithm from Theorem 2.1 in parallel.

To perform the duplication efficiently, to each processor that holds a set of edges that should participate in $k$ instances we assign $k-1$ helper processors. Let us denote the set consisting of this processor and the helper processors by $P_k$. Duplication can be executed in two stages. In the

\footnote{This step uses $\Theta(n^2)$ communicates, but it is only because it uses a routing protocol; if we have routing for free, the communication complexity is proportional to the number of sorted elements}
first stage, each processor of $P_k$ receives from the original processor part of the set of edges of size $O(n/k)$. In the second stage, each processor sends received part to all other processors in $P_k$.

The first step of the protocol can be executed as the original processor sends $O(n)$ edges in total, and each processor receives only $O(n/k)$ messages. Then, in the second step each processor sends $|P_k| \cdot O(n/k) = O(n)$ messages and receives the set of all edges that was stored in the memory of the original processor that has size $O(n)$. Therefore, this step also can be executed without violating the memory and communication constraints.

5.1 Running the algorithm for Spanning Forest problem in parallel

In this subsection, we show that we can solve the instances of Connected Components problem obtained by the reduction from Lemma 5.1 using the Spanning Forest algorithm from Theorem 2.1 in parallel, which completes the algorithm for the Minimum Spanning Forest problem, and a proof of Theorem 2.2.

The short argument \(^4\) is that the Spanning Forest algorithm from Theorem 2.1 is in fact an MPC algorithm. Since after the edge duplication step we get a set of Connected Components problems of total size $O(m)$, each defined by a set of edges, Congested Clique can solve all of them by running the MPC algorithm for the Spanning Forest problem in parallel.

Simulation in Congested Clique: In the remaining part of this section, for the sake of completeness, we give some details of the parallel execution of the algorithm from Theorem 2.1 in the Congested Clique model, without referencing simulation of MPC algorithms.

We are given sets of edges of $O(\sqrt{m/n})$ graphs, with $O(m)$ edges in total. Our goal is to compute a representation of each of those graphs that is a vertex partition rather than an edge partition: i.e. we want that each processor gets a set of vertices and all edges incident, and for each edge the processor knows the ID of a processor holding the other endpoint. Furthermore, we want to partition the vertices in such a way that executing the Spanning Forest algorithm for all graphs in parallel can be efficiently simulated by the processors of the Congested Clique. Basically, the goal is to partition the vertices of all instances in such a way that:

- each processor simulates vertices that have $O(n)$ incident edges in total,
- vertex knows what is the ID of a processor that handles simulation of their neighbour.

Clearly, those two properties guarantee that we can execute communication between the neighbours in the simulated graphs. Furthermore, it also implies that communication with coordinators can be executed efficiently.

In Algorithm 1 a vertex communicates with a coordinator only if it has an edge that has to be send to this coordinator, which limits the amount of communication to $O(n)$ (as that is the total number of edges incident to vertices simulated by one processor).

In Algorithm 2 a vertex communicates with some coordinator, only if it has an edge participating in an instance that uses this coordinator, since the instances are edge disjoint, the total number of messages send by one vertex is always no larger than the sum of degrees of simulated vertices, which is $O(n)$.

Remark 5.2. Bounding the required communication by the number of edges automatically gives us an $O(m)$ bound on the global communication, which means that the MST algorithm also can be efficiently executed in the MPC model.

Partition of simulated vertices: To obtain a partition of simulated vertices allowing this parallel execution, we start by copying each edge $\{u, v\}$ twice, one copy for an edge outgoing from $u$, and

\(^4\)short but not that clear for the readers not familiar with the relation between the MPC and Congested Clique
one copy for an edge outgoing from $v$. Then, we sort this set of edges, to obtain that all edges outgoing from a single vertex $v$ are in the memory of a single processor (and we have that for all $v$ simultaneously).

We can sort the edges using the $O(1)$ round sorting protocol [Len13], but as a result of sorting, we do not have guarantee that for each $v$ we see all the edges in the memory of a single processor. Still, if that is not the case, then the edges outgoing form $v$ are in the memory of processors with consecutive IDs. Therefore, each processor can communicate with processors of neighbouring ID and if some two processors have the edges that are outgoing from a single vertex $v$ the one with smaller ID can send them to the one with larger ID, which can be also done in $O(1)$ rounds.

From now on, a processor that holds the edges outgoing from $v$ will simulate $v$, and the remaining work we have to do is to find which processors simulate the other endpoints of the edges incident to $v$. To do so, we again use a sorting algorithm. To each of two copies of an edge we attach additional information stating which processor simulates one of the endpoints, and we sort the edges by the original endpoints. This way some processor $p$ will see two copies of the edge, each having ID of a processor simulating one endpoint. Then, $p$ can notify the processors simulating the endpoints of an edge, what is the ID of a processor simulating the other endpoint.

Since as a result of sorting, each processor gets $O(n)$ edges, it has to send at most $O(n)$ notifications. Furthermore, each processor simulates vertices of degrees that sum to $O(n)$, therefore, it need to receive $O(n)$ notifications. Therefore, this step can be executed in $O(1)$ rounds.

References

[BDH18] Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief Announcement: Semi-MapReduce Meets Congested Clique. CoRR, abs/1802.10297, 2018.

[GP16] Mohsen Ghaffari and Merav Parter. MST in Log-Star Rounds of Congested Clique. In Proceedings of PODC, PODC ’16, pages 19–28, New York, NY, USA, 2016. ACM.

[HP14] James W. Hegeman and Sriram V. Pemmaraju. Lessons from the Congested Clique Applied to MapReduce. In Proceedings of SIROCCO, pages 149–164, 2014.

[HPP+15] James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Sequizzato. Toward Optimal Bounds in the Congested Clique: Graph Connectivity and MST. In Proceedings of PODC, PODC ’15, pages 91–100, New York, NY, USA, 2015. ACM.

[JN17] Tomasz Jurdziński and Krzysztof Nowicki. Brief Announcement: On Connectivity in the Broadcast Congested Clique. In Proceedings of DISC, pages 54:1–54:4, 2017.

[JN18] Tomasz Jurdziński and Krzysztof Nowicki. MST in $O(1)$ Rounds of Congested Clique. In Proceedings of SODA, pages 2620–2632, 2018.

[KKT95] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. J. ACM, 42(2):321–328, March 1995.

[Kor16] Janne H. Korhonen. Brief Announcement: Deterministic MST Sparsification in the Congested Clique. In Proceedings of DISC, 2016.

[Kru56] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical society, 7(1):48–50, 1956.
[Len13] Christoph Lenzen. Optimal Deterministic Routing and Sorting on the Congested Clique. In Proceedings of PODC, PODC ’13, pages 42–50, New York, NY, USA, 2013. ACM.

[LPPP05] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-Weight Spanning Tree Construction in $O(\log \log n)$ Communication Rounds. SIAM J. Comput., 35(1):120–131, 2005.

[LPPSP03] Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. MST construction in $O(\log \log n)$ communication rounds. In Proceedings of SPAA, pages 94–100. ACM, 2003.

[NMN01] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. Discrete Mathematics, 233(1):3–36, 2001.

[Tar83] Robert Endre Tarjan. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.