Input-Dynamic Distributed Algorithms for Communication Networks

KLAUS-TYCHO FOERSTER, Faculty of Computer Science, University of Vienna, Austria
JANNE H. KORHONEN, IST Austria, Austria
AMI PAZ, Faculty of Computer Science, University of Vienna, Austria
JOEL RYBICKI, IST Austria, Austria
STEFAN SCHMID, Faculty of Computer Science, University of Vienna, Austria

Consider a distributed task where the *communication network* is fixed but the *local inputs* given to the nodes of the distributed system may change over time. In this work, we explore the following question: if some of the local inputs change, can an existing solution be updated efficiently, in a dynamic and distributed manner?

To address this question, we define the *batch dynamic* CONGEST model in which we are given a bandwidth-limited communication network and a *dynamic* edge labelling defines the problem input. The task is to maintain a solution to a graph problem on the labeled graph under *batch changes*. We investigate, when a batch of α edge label changes arrive,

– how much time as a function of α we need to update an existing solution, and
– how much information the nodes have to keep in local memory between batches in order to update the solution quickly.

Our work lays the foundations for the theory of input-dynamic distributed network algorithms. We give a general picture of the complexity landscape in this model, design both universal algorithms and algorithms for concrete problems, and present a general framework for lower bounds. In particular, we derive non-trivial upper bounds for two selected, contrasting problems: maintaining a minimum spanning tree and detecting cliques.

CCS Concepts: • **Networks → Network algorithms**; • **Theory of computation → Distributed computing models**; **Dynamic graph algorithms**.

Additional Key Words and Phrases: dynamic graph algorithms; congest model; distributed algorithms; communication networks; network management

ACM Reference Format:
Klaus-Tycho Foerster, Janne H. Korhonen, Ami Paz, Joel Rybicki, and Stefan Schmid. 2021. Input-Dynamic Distributed Algorithms for Communication Networks. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 1, Article 6 (March 2021), 33 pages. https://doi.org/10.1145/3447384

1 INTRODUCTION

There is an ongoing effort in the networking community to render networks more adaptive and "self-driving" by automating network management and configuration tasks [34]. To this end, it is

Authors' addresses: Klaus-Tycho Foerster, Faculty of Computer Science, University of Vienna, Vienna, Austria, klaus-tycho.foerster@univie.ac.at; Janne H. Korhonen, IST Austria, Klosterneuburg, Austria, janne.korhonen@ist.ac.at; Ami Paz, Faculty of Computer Science, University of Vienna, Vienna, Austria, ami.paz@univie.ac.at; Joel Rybicki, IST Austria, Klosterneuburg, Austria, joel.rybicki@ist.ac.at; Stefan Schmid, Faculty of Computer Science, University of Vienna, Vienna, Austria, stefan_schmid@univie.ac.at.

2476-1249/2021/3-ART6 $15.00
https://doi.org/10.1145/3447384

essential to design network protocols that solve various tasks, such as routing and traffic engineering, adaptively and *fast*.

Especially in large and long-lived networks, change is inevitable: new demand may increase congestion, network properties such as link weights may be updated, nodes may join and leave, and new links may appear or existing links fail. As distributed systems often need to maintain data structures and other information related to the operation of the network, it is important to update these structures efficiently and reliably upon changes. Naturally the naive approach of always recomputing everything from scratch after a change occurs might be far from optimal and inefficient. Rather, it is desirable that if there are only *few* changes, the existing solution could be efficiently utilised for computing a new solution.

However, developing robust, general techniques for *dynamic* distributed graph algorithms — that is, algorithms that reuse and exploit the existence of previous solutions — is challenging [11, 53]: even small changes in the communication topology may force communication and updates over long distances or interfere with ongoing updates. Nevertheless, a large body of prior work has focused on how to operate in dynamic environments where the underlying communication network changes: temporally dynamic graphs [19] model systems where the communication structure is changing over time; distributed dynamic graph algorithms consider solving individual graph problems when the graph representing the communication networks is changed by addition and removal of nodes and edges [11, 14, 16, 20, 22, 31, 45, 53]; and self-stabilisation considers recovery from arbitrary transient faults that may corrupt an existing solution [28, 29].

1.1 Input-dynamic distributed algorithms

In contrast to most prior work which focuses on efficiently maintaining solutions in distributed systems where the underlying communication network itself may abruptly change, we instead investigate how to deal with *dynamic inputs* without changes in the topology: we assume that the local inputs (e.g. edge weights) of the nodes may change, but the underlying communication network remains static and reliable. We initiate the study of input-dynamic distributed graph algorithms, with the goal of laying the groundwork for a comprehensive theory of this setting. Indeed, we will see that this move from dynamic topology towards a setting more closely resembling centralised dynamic graph algorithms [46], where input changes and the computational model are similarly decoupled from each other, enables a development of a general theory of input-dynamic distributed algorithms.

1.2 Motivation: towards dynamic network management

While the input-dynamic distributed setting is of theoretical interest, it is largely motivated by practical questions arising in network management and optimisation. In wired communication networks the communication topology is typically relatively static (e.g. the layout and connections of the physical network equipment), but the input is highly dynamic. For example, network operators perform link weight updates for dynamic traffic engineering [41] or to adjust link layer forwarding in local area networks [68, 77], content distribution providers dynamically optimise cache assignments [42], and traffic patterns naturally evolve over time [9, 42]. In all these cases, the underlying topology of the communication network remains fixed, but only some input parameters change.

Formally, the above network tasks can often be modelled as algorithmic graph problems, where the input, in the form of edge weights or communication demands, changes over time, while the network topology remains fixed or changes infrequently. In the light of the current efforts to render networks more autonomous and adaptive [35, 61, 76], it is interesting to understand the power and limitations of such dynamic distributed optimisations, also compared to a model where the communication topology frequently changes.

To elucidate the connection between fundamental graph problems and basic network management tasks, we discuss some motivating examples.

Example 1: Link layer spanning trees. In the context of link layer networking, a typical task is to maintain a spanning tree on the network with e.g. the (rapid) Spanning Tree Protocol (STP) [68]. A standard approach to compute spanning trees is that the nodes first elect a root (leader) and then pick their parent in the tree according to the shortest distance to the root. However, when link weights change, the leader is tasked with broadcasting such changes, and can hence take a long time to converge to a new (minimum) spanning tree. In the case of centralised solutions, such as software-defined networking (SDN) [77], we run into the same conceptual issues, namely that the changes need to be gathered at a (logically) centralised location, and pushed out to the network.

From the theoretical perspective, there are well-established lower bounds for computing a minimum spanning tree *from scratch* in communication-bounded networks: in the classic CONGEST model, the problem requires $\Omega(\sqrt{n} + D)$ rounds, where n is the number of nodes and D is the diameter of the network [26]. However, it is not a priori clear how efficiently *maintaining* an MST under input changes can be done in a distributed manner. As one of our results, we show that (1) it is possible to replace the dependency on $\sqrt{n}$ to be linear in the number edge weight changes while maintaining a small local memory footprint *and* (2) there is a matching lower bound showing that this is essentially the best input-dynamic algorithm one can hope for.

Example 2: Traffic engineering and shortest paths routing. The reliable and efficient data delivery in ISP networks typically relies on a clever traffic engineering algorithm [10]. Adaptive traffic engineering is usually performed using dynamic link weight changes, steering the traffic along certain (approximately) shortest paths in the network. The re-computation of routes can be quite time consuming and exhibit longer convergence times, including the classic all-pairs shortest paths (APSP) algorithms based on as distance-vector and link-state protocols [69]. There are theoretical limits on how fast exact and approximate all-pairs shortest paths can be computed from scratch in a distributed manner, even when the networks are sparse [1].

In this paper we will show that while similar limitations also extend to the case of input-dynamic algorithms, there is a simple universal, near-optimal dynamic algorithm for these types of tasks.

Example 3: Detecting network substructures. In many networking applications, it is desirable to detect whether the network contains certain substructures and locate them efficiently. For example, the task of cycle detection is a special case of the task of *loop detection*, i.e., detecting whether a routing scheme contains a directed cycle. On the other hand, clique detection can be used for maintaining and assigning e.g. failover nodes and links.

Once again, we can formally characterise the complexity of such tasks, establishing both fast algorithms and proving non-trivial lower bounds in the case of input-dynamic network algorithms, for several subgraph detection problems.

Outlook: Towards scalable and efficient network management. Above we gave three examples how our work can benefit the study of standard network management tasks. In general, we believe that taking the viewpoint of input-dynamic algorithms has the potential to lead to a paradigm shift in the design of efficient and scalable network management protocols. Current state-of-the-art approaches to dealing with network management essentially come in two flavours: distributed control with recomputations from scratch and centralised control (e.g, SDNs). While the former has the drawbacks discussed above, limiting the granularity at which optimisations can be performed, the latter can entail scalability issues. For example, the indirection via a controller, even if it is

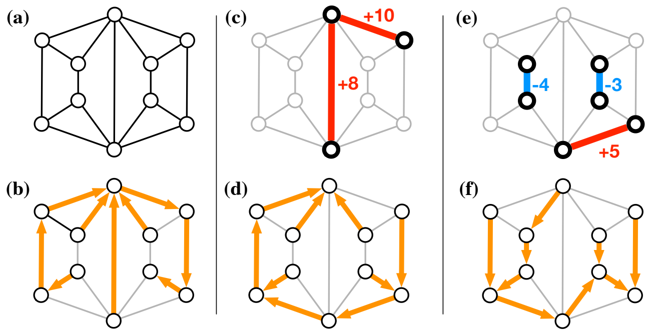

Fig. 1. Examples of input-dynamic minimum-weight spanning tree. (a) The underlying communication graph, with all edges starting with weight 1. (b) A feasible minimum-weight spanning tree. (c) A batch of two edge weight increments. (d) Solution to the new input labelling. (e) A new batch of three changes: two decrements and one increment. (f) An updated solution.

only logically centralised but physically distributed, can result in delays: in terms of reaction and computation time at the controller and in terms of the required synchronisation in case of multiple controllers (e.g., to keep states consistent) [59].

In contrast to the two prior approaches, the paradigm of input-dynamic distributed algorithms aims to realise the best of both worlds: design algorithms that benefit from previously collected state in distributed protocols, rapidly generate outputs based on existing solutions, and update the auxiliary data structures for the next set of changes.

1.3 Batch dynamic CONGEST model

Our aim is hence to develop a rigorous theoretical framework for reasoning about distributed input-dynamic algorithms. While there are standard models for *non-dynamic* distributed computations [67], there is no established model for *input-dynamic* graph algorithms so far (in Section 3 we overview prior approaches in modelling other aspects of dynamic networks).

To remedy this, we introduce the *batch dynamic* CONGEST model, which allows us to formally develop a theory of input-dynamic graph algorithms. In brief, the model is a dynamic variant of the standard CONGEST model of distributed computation with the following characteristics:

(1) The communication network is represented by a static graph $G = (V, E)$ on $|V| = n$ nodes. The nodes can communicate with each other over the edges, with $O(\log n)$ bandwidth per round. (This is the standard CONGEST model [67].)
(2) The input is given by a *dynamic* edge labelling of G. The input labelling may change and once this happens nodes need to compute a new feasible solution for the new input labelling. The labelling can denote, e.g., edge weights or mark a subgraph of G. We assume that the labels can be encoded using $O(\log n)$ bits so that communicating a label takes a single round.
(3) The goal is to design a distributed algorithm which maintains a solution to a given graph problem on the labelled graph under *batch changes*: up to α labels can change simultaneously, and the nodes should react to these changes. The nodes may maintain a *local auxiliary state* to store, e.g., the current output and auxiliary data structures, in order to facilitate efficient updates upon subsequent changes.

Figure 1 gives an example of an input-dynamic problem: maintaining a minimum-weight spanning tree. We define the model in more detail in Section 4.

Table 1. Upper and lower bounds for select problems in batch dynamic CONGEST. Upper bounds marked with † follow from the universal algorithms. The lower bounds apply in a regime where α is sufficiently small compared to n, with the threshold usually corresponding to the point where the lower bound matches the complexity of computing a solution from scratch in CONGEST; see Section 8 for details. All upper bounds are deterministic. The lower bounds hold for both deterministic and randomised algorithms.

	Upper bound		Lower bound	
Problem	Time	Space	Time	Ref.
any problem	$O(\alpha + D)$	$O(m \log n)$	—	§5
any LOCAL(1) problem	$O(\alpha)$	$O(m \log n)$	—	§5
minimum spanning tree	$O(\alpha + D)$	$O(\log n)$	$\Omega(\alpha/\log^2 \alpha + D)$	§7, §8.4
k-clique	$O(\alpha^{1/2})$	$O(m \log n)$	$\Omega(\alpha^{1/4}/\log \alpha)$	§6, §8.3
4-cycle	$O(\alpha)^{\dagger}$	$O(m \log n)^{\dagger}$	$\Omega(\alpha^{2/3}/\log \alpha)$	§8.3
k-cycle, $k \geq 5$	$O(\alpha)^{\dagger}$	$O(m \log n)^{\dagger}$	$\Omega(\alpha^{1/2}/\log \alpha)$	§8.3
diameter, $(3/2 - \varepsilon)$-apx.	$O(\alpha + D)^{\dagger}$	$O(m \log n)^{\dagger}$	$\Omega(\alpha/\log^2 \alpha + D)$	§8.3
APSP, $(3/2 - \varepsilon)$-apx.	$O(\alpha + D)^{\dagger}$	$O(m \log n)^{\dagger}$	$\Omega(\alpha/\log^2 \alpha + D)$	§8.3

Model discussion. As discussed earlier, the underlying motivation for our work is to study how changes to the input can be efficiently handled, while suppressing interferences arising from changing communication topology. A natural starting point for studying communication-efficient solutions for graph problems in networks is the standard CONGEST model [67], which we extend to model the input-dynamic setting.

Assuming that the communication topology remains static allows us to adopt the basic viewpoint of centralised dynamic algorithms, where an algorithm can fully process changes to input before the arrival of new changes. While this may initially seem restrictive, our algorithms can in fact also tolerate changes arriving during an update: we can simply delay the processing of such changes, and fix them the next time the algorithm starts. Indeed, this parallels centralised dynamic algorithms, where the processing of changes is not disrupted by a newer change arriving.

While this model has not explicitly been considered in the prior work, we note that input-dynamic distributed algorithms of similar flavour have been studied before in limited manner. In particular, Peleg [66] gives an elegant minimum spanning tree update algorithm that, in our language, is a batch dynamic algorithm for minimum spanning tree. Very recently, a series of papers has investigated batch dynamic versions of MPC and k-machine models, mainly focusing on the minimum spanning tree problem [27, 43, 63]. However, the MPC and k-machine models assume a fully-connected communication topology, i.e., every pair of nodes share a direct communication link, making them less suitable for modelling large-scale networks.

Finally, we note that in practice, the communication topology rarely remains static throughout the entire lifetime of a network. However, if the changes in the communication topology are infrequent enough compared to the changes in the inputs, then recomputing a new auxiliary state from scratch, whenever the underlying communication network changes, will have a small cost in the amortised sense. Moreover, any *lower bounds* for the batch dynamic model also hold in the case of networks with changing communication topology.

2 CONTRIBUTIONS

In this work, we focus on the following fundamental questions. When a batch of α edge label changes arrive, and the communication graph has diameter D,

(a) how much time does it take to update an existing solution, as a function of α and D, and
(b) how much information does a node need to keep in its local memory between batches, in order to achieve optimal running time?

With these questions, we lay the foundations for the theory of input-dynamic distributed graph algorithms. We draw a general picture of the complexity landscape in the batch dynamic CONGEST model as summarised in Table 1. Our main results are as follows.

2.1 Universal upper bounds

As an almost trivial baseline, we observe that *any* graph problem can be solved in $O(\alpha + D)$ rounds. Moreover, any graph problem where the output of a node depends only on the constant-radius neighbourhood of the node – that is, a problem solvable in $O(1)$ rounds in the LOCAL model[1] – can be solved in $O(\alpha)$ rounds. However, these universal algorithms come at a large cost in space complexity: storing the auxiliary state between batches may require up to $O(m \log n)$ bits, where m is the number of edges — in the input graph if the input marks a subgraph, and in the communication graph if the input represents edge weights. (Section 5.)

2.2 Intermediate complexity: clique enumeration

We give an algorithm for enumerating k-cliques in $O(\alpha^{1/2})$ rounds, beating the universal upper bound for local problems, and showing that there exist non-trivial problems that can be solved in $o(\alpha)$ rounds. To complement this result, we show that dynamic clique detection requires $\Omega(\alpha^{1/4})$ rounds. This is an example of a natural problem with time complexity that is neither constant nor $\Theta(\alpha)$. (Section 6.)

2.3 Saving space: minimum-weight spanning trees

We show that a minimum-weight spanning tree can be maintained in $O(\alpha + D)$ rounds using only $O(\log n)$ bits per node for storing the auxiliary state; this exponentially improves the storage requirements of a previous distributed dynamic algorithm of Peleg [66], which uses $O(n \log n)$ bits of memory per node. In addition, we show that our result is tight, in terms of update time, up to poly $\log \alpha$: for any $\alpha \leq n^{1/2}$, maintaining a minimum-weight spanning tree requires $\Omega(\alpha/\log^2 \alpha + D)$ rounds. (Section 7.)

2.4 A general framework for lower bounds

We develop a framework for lifting CONGEST lower bounds into the batch dynamic CONGEST model, providing a vast array of non-trivial lower bounds for input-dynamic problems. These include lower bounds for classic graph problems, such as cycle detection, clique detection, computing the diameter, approximating all-pairs shortest paths, and computing minimum spanning trees. The lower bounds hold for both deterministic and randomised algorithms. (Section 8.)

2.5 Dynamic congested clique

We explore the dynamic variant of the *congested clique* model, which arises as a natural special case of the batch dynamic CONGEST. We show that triangle counting can be solved in $O((\alpha/n)^{1/3} + 1)$ rounds in this model using $O(n \log n)$ bits of auxiliary state by applying a *dynamic matrix*

[1]The LOCAL model is similar to the CONGEST model, but without the $O(\log n)$ limitation on the message sizes [67].

multiplication algorithm. To contrast this, we show that any problem can be solved in $O(\lceil \alpha/n \rceil)$ rounds using $O(m \log n)$ bits of auxiliary state. (Section 9.)

2.6 Summary and open questions

As a key takeaway, we have established that the possible time complexities in batch dynamic CONGEST range from constant to linear-in-α, and that there are truly intermediate problems in between. However, plenty of questions remain unanswered; we highlight the following objectives as particularly promising future directions:

– *Upper bounds*: Develop new algorithmic techniques for batch dynamic CONGEST.
– *Understanding space*: Develop lower bound techniques for space complexity. In particular, are there problems that exhibit *time-space tradeoffs*, i.e. problems where optimal time and space bounds cannot be achieved at the same time?
– *Symmetry-breaking problems*: Understand how problems with subpolynomial complexity in CONGEST– in particular, symmetry-breaking problems such as maximal independent set – behave in the batch dynamic CONGEST model.

2.7 Technical overview and methodological advancements

The main *conceptual* contribution of our work is the introduction of the batch dynamic model, which allows the development of a robust complexity theory of input-dynamic distributed algorithms. A particularly attractive feature of our model is that we can easily leverage standard machinery developed for non-dynamic CONGEST model in the input-dynamic setting. This for example immediately yields the baseline results given in Section 5 and the fast triangle counting algorithms for batch dynamic congested clique in Section 9.

However, to obtain *efficient* input-dynamic algorithms, it is necessary to develop new algorithmic and analysis techniques. As our main technical contributions, we analyse two different algorithmic problems, clique enumeration (a local problem) and maintaining minimum-spanning trees (a global problem), and devise a general framework for proving lower bounds for input-dynamic distributed algorithms.

Clique enumeration. The clique enumeration problem is a local problem: the nodes need to decide whether their local neighbourhood contains a clique of a certain size. In the dynamic setting, the main challenge is to deal with the fact that nodes do not know the number α of changes in advance, but the running time should be bounded in terms of α. When dealing with a global problem that requires $\Omega(D)$ rounds in networks with diameter D, we can simply broadcast the number of changes in the network using standard broadcasting techniques. However, as clique enumeration is a local problem, we wish to obtain running times independent of the diameter of the communication network.

To this end, we observe that the subgraph defined by the changed input edges has an useful graph theoretical property, namely, it has bounded degeneracy. This allows us to distributively compute the Nash–Williams decomposition between after a batch of updates, which can be efficiently used to route information about the local changes to input while avoiding congestion. This resembles to approach taken by e.g. Korhonen and Rybicki [54], who use a distributed version of the Nash–Williams decomposition by Barenboim and Elkin [15], to detect cycles in bounded degeneracy graphs. The main difference to this work is that here we show how to use this approach in the batch dynamic model and we show how to interleave the computation of this decomposition and clique enumeration in a way where nodes only locally have to determine when to halt, without knowing the total number of changes in advance.

Minimum-weight spanning trees. Recent work almost exclusively has focused on maintaining minimum-weight spanning trees in fully-connected communication topologies. The main challenge in our work is that we consider general communication topologies, where nodes may need to communicate via large distances. In order to achieve small space complexity, we use a distributed implementation of the standard Eulerian tour tree data structure, which can be used to recover the minimum-weight spanning tree as long as we can maintain the said data structure.

Recently, Eulerian tour trees have also been used in *fully-connected* dynamic models [27, 43, 52], where direct communication is possible between any pair of nodes. In our model, the analysis is complicated by the fact, that communication has to be done over the network e.g. via broadcast trees – to avoid congestion, the changes to the input need to be broadcast in a pipelined fashion. The key observation is that the steps required for the Eulerian tour tree update can be formulated as maximum matroid basis problems, which allows us to use the elegant distributed maximum matroid basis algorithm Peleg [66] to efficiently compute the required changes to this structure.

Lower bound framework. The main technical challenge here is to extend the notion of lower bound family lower bounds into the batch dynamic setting. While the relevant parameter in CONGEST is the size n of the network, in the batch dynamic model, the time complexity is (mainly) parameterised in terms of the number α of changes. For this reparameterisation, we introduce the notion of *extension properties* and a padding technique that allow us to embed a hard input graph into a larger communication graph.

3 RELATED WORK

As the dynamic aspects of distributed systems have been investigated from numerous different perspectives, giving a comprehensive survey of all prior work is outside the scope of the current work. Thus, we settle on highlighting the key differences and similarities between the questions studied in related areas and our work.

3.1 Centralised dynamic graph algorithms

Before proceeding to the distributed setting, it is worth noting that dynamic graph algorithms in the *centralised setting* have been a major area of research for several years [46]. This area focuses on designing data structures that admit efficient update operations (e.g. node/edge additions and removals) and queries on the graph.

Early work in the area investigated how connectivity properties, e.g., connected components and minimum spanning trees, can be maintained [49, 50]. Later work has investigated efficient techniques for maintaining other graph structures, such as spanners [17], emulators [47], matchings [60], maximal independent sets [7, 8]; approximate vertex covers, electrical flows and shortest paths [18, 32, 44]. Recently, conditional hardness results have been established in the centralised setting [2, 6, 48].

Similarly to our work, the input in the centralised setting is dynamic: there is a stream of update operations on the graph and the task is to efficiently provide solutions to graph problems. Naturally, the key distinction is that changes in the centralised setting arrive sequentially and are handled by a single machine. Moreover, in the distributed setting, we can provide unconditional lower bounds for various input-dynamic graph problems, as our proofs rely on communication complexity arguments.

3.2 Distributed algorithms in changing communication networks

The challenges posed by dynamic communication networks — that is, networks where communication links and nodes may appear or be removed — have been a subject of ongoing research for decades. Classic works have explored the connection between synchronous static protocols and *fault-prone* asynchronous computation under dynamic changes to communication topology [12]. Later, it was investigated how to maintain or recompute local [65] and global [33] graph structures when communication links may appear and disappear or crash. A recent line of work has investigated how to efficiently fix solutions to graph problems under various distributed settings [7, 8, 14, 20, 22, 31, 38–40, 45, 53]. Another line of research has focused on time-varying communication networks which come with temporal guarantees, e.g., that every T consecutive communication graphs share a spanning tree [19, 55, 64].

In the above settings, the input graph and the communication network are the same, i.e., the inputs and communication topology are typically coupled. However, there are exceptions to this, as discussed next.

3.3 Input-dynamic parallel and distributed algorithms

Several instance of distributed dynamic algorithms can be seen as examples of the input-dynamic approach. Italiano [51] and later Cicerone et al. [24], considered the problem of maintaining a solution all-pairs shortest paths problem when a single edge weight may change at a time. Peleg [66] considered the task of correcting a minimum-weight spanning tree after changes to the edge weights, albeit with a large cost in local storage, as the algorithm stores the entire spanning tree locally at each node.

More recently, there has been an increasing interest in developing dynamic graph algorithms for classic parallel models [3–5, 74, 75] and massively parallel large-scale systems [27, 43, 52, 63]. In the former, communication is via shared memory, whereas in the latter the communication is via message-passing in a fixed, fully-connected network, but the input is distributed among the nodes and the communication bandwidth (or local storage) of the nodes is limited. Thus, the key difference is that in these parallel models, the communication topology always forms a fully-connected graph, whereas in the batch dynamic CONGEST considered in our work, the communication topology can be arbitrary, and thus, communication also incurs a distance cost. However, we note that the dynamic congested clique model we study in Section 9 falls under this category.

3.4 Self-stabilisation

The area of self-stabilisation [28, 29] considers robust algorithms that *eventually* recover from *arbitrary* transient failures that may corrupt the state of the system. Thus, unlike in our setting where the auxiliary state and communication network are assumed to be reliable, the key challenge in self-stabilisation is coping with possibly adversarial corruption of local memory and inconsistent local states, instead of changing inputs.

3.5 Supported models

Similar in spirit to our model is the *supported* CONGEST model, a variant of the CONGEST model designed for software-defined networks [72]. In this model, the communication graph is known to all nodes and the task is to solve a graph problem on a given *subgraph*, whose edges are given to the nodes as inputs. The idea is that the knowledge of the communication graph may allow for *preprocessing*, which may potentially offer speedup for computing solutions in the subgraph. However, unlike the batch dynamic CONGEST model, the supported CONGEST model focuses on *one-shot* computation. Korhonen and Rybicki [54] studied the complexity of subgraph detection problems in supported CONGEST. Later, somewhat surprisingly, Foerster et al. [37] showed that in many cases knowing the communication graph does not help to circumvent CONGEST lower bounds. Lower bounds were also studied in the supported LOCAL model, for maximum independent set approximation [36].

4 BATCH DYNAMIC CONGEST MODEL

In this section, we formally define the the batch dynamic CONGEST model.

4.1 Communication graph and computation

The communication graph is an undirected, connected graph $G = (V, E)$ with n nodes and m edges. We use the short-hands $E(G) = E$ and $V(G) = V$. Each node has a unique identifier of size $O(\log n)$ bits. In all cases, n and m denote the number of vertices and edges, respectively, in G, and D denotes the diameter of G.

All computation is performed using the graph G for communication, as in the case of the standard CONGEST model [67]: in a single synchronous round, all nodes in lockstep

(1) send messages to their neighbours,
(2) wait for messages to arrive, and
(3) update their local states.

We assume $O(\log n)$ bandwidth per edge per synchronous communication round. To simplify presentation, we assume that any $O(\log n)$-bit message can be sent in one communication round. Clearly, this only affects constant factors in the running times of the algorithms we obtain.

4.2 Graph problems

A *graph problem* Π is given by sets of input labels Σ and output labels Γ. For each graph $G = (V, E)$, unique ID assignment $\mathsf{ID}\colon V \to \{1, \dots, \mathrm{poly}(n)\}$ for V and input labelling of edges $\ell\colon E \to \Sigma$, the problem Π defines a set $\Pi(G, \ell)$ of valid output labellings of form $f\colon V \to \Gamma$. We assume that input labels can be encoded using $O(\log n)$ bits, and that the set $\Pi(G, \ell)$ is finite and computable. We focus on the following problem categories:

- *Subgraph problems:* The input label set is $\Sigma = \{0, 1\}$, and we interpret a labelling as defining a subgraph $H = (V, \{e \in E\colon \ell(e) = 1\})$. Note that in this case, the diameter of the input graph H can be much larger than the diameter D of the communication graph G, but we still want the running times of our algorithms to only depend on D.
- *Weighted graph problems:* The input label set is $\Sigma = \{0, 1, 2, \dots, n^C\}$ for a constant C, i.e., the labelling defines weights on edges. We can also allow negative weights of absolute value at most n^C, or allow some weights to be infinite, denoted by ∞.

4.3 Batch dynamic algorithms

We define batch dynamic algorithms via the following setting: assume we have some specified input labels ℓ_1 and have computed a solution for input ℓ_1. We then change α edge labels on the graph to obtain new inputs ℓ_2, and want to compute a solution for ℓ_2. In addition to seeing the local input labellings, each node can store auxiliary information about the previous labelling ℓ_1 and use it in the computation of the new solution.

More precisely, let Π be a problem. Let Λ be a set of local auxiliary states; we say that a (global) auxiliary state is a function $x\colon V \to \Lambda$. A *batch dynamic algorithm* is a pair $(\xi, \mathcal{A})$ defined by a set of valid auxiliary states $\xi(G, \ell)$ and a CONGEST algorithm $\mathcal{A}$ that satisfy the following conditions:

- For any G and ℓ, the set $\xi(G, \ell)$ is finite and computable. In particular, this implies that there is a (centralised) algorithm that computes some $x \in \xi(G, \ell)$ from G and ℓ.
- There is a computable function $s\colon \Lambda \to \Gamma$ such that for any $x \in \xi(G, \ell)$, outputting $s(x(v))$ at each node $v \in V$ gives a valid output labelling, that is, $s \circ x \in \Pi(G, \ell)$.
- The algorithm $\mathcal{A}$ is a CONGEST algorithm such that
  (a) all nodes v receive as local input the labels on their incident edges in both an old labelling ℓ_1 and a new labelling ℓ_2, as well as their own auxiliary state $x_1(v)$ from $x_1 \in \xi(G, \ell_1)$,

(b) all nodes v will halt in finite number of steps and upon halting produce a new auxiliary state $x_2(v)$ so that together they satisfy $x_2 \in \xi(G, \ell_2)$.

Note that we do not require all nodes to halt at the same time. We assume that halted nodes have to announce halting to their neighbours, and will not send or receive any messages after halting.

We define the running time of $\mathcal{A}$ as the maximum number of rounds for all nodes to halt; we use the number of label changes between ℓ_1 and ℓ_2 as a parameter and denote this by α. The (per node) space complexity of the algorithm is the maximum number of bits needed to encode any auxiliary state $x(v)$ over $x \in \xi(G, \ell)$.

While all of our algorithms are deterministic, one can also consider randomised batch dynamic algorithms. Here one can adopt different correctness and complexity measures. The most common one in centralised and parallel dynamic algorithms (e.g. [4, 17, 49, 75]) is to consider *Las Vegas* algorithms. In our setting, this means requiring that, upon halting, nodes always output a valid new auxiliary state; the running time can be measured either (a) by the *expected* running time of the algorithm, or (b) by establishing running time bounds that hold with high probability. Alternatively, one can also consider *Monte Carlo* algorithms, succeeding with high probability within a fixed running time (e.g. [70]); however, these have the disadvantage that the algorithm is likely to fail at some point over an arbitrarily long sequence of batches. Our lower bounds hold for all of these variants, as we discuss in Section 8.

REMARK 4.1. *Allowing nodes to halt at different times is done for technical reasons, as we do not assume that nodes know the number of changes α and thus we cannot guarantee simultaneous halting in general. Note that with additive $O(D)$ round overhead, we can learn α globally.*

4.4 Notation

Finally, we collect some notation used in the remainder of this paper. For any set of nodes $U \subseteq V$, we write $G[U] = (U, E')$, where $E' = \{e \in E : e \subseteq U\}$, for the subgraph of G induced by U. For any set of edges $F \subseteq E$, we write $G[F] = (V', F)$, where $V' = \bigcup F$. When clear from the context, we often resort to a slight abuse of notation and treat a set of edges $F \subseteq E$ interchangeably with the subgraph (V, F) of G. Moreover, for any $e = \{u, v\}$ we use the shorthand $e \in G$ to denote $e \in E(G)$. For any $v \in V$, the set of edges incident to v is denoted by $E(v) = \{\{u, v\} \in E\}$. The neighbourhood of v is $N^+(v) = \bigcup E(v)$. We define

$$\dot{E} = \{e \in E : \ell_1(e) \neq \ell_2(e)\}$$

to be the set of at most α edges whose labels were changed during an update.

5 UNIVERSAL UPPER BOUNDS

As a warmup, we establish the following easy baseline result showing that *any* problem Π has a dynamic batch algorithm that uses $O(\alpha + D)$ time and $O(m \log n)$ bits of auxiliary space per node: each node simply stores previous input labelling ℓ_1 as the auxiliary state and broadcasts all changes to determine the new labelling ℓ_2. First, we recall some useful primitives that follow from standard techniques [67].

LEMMA 5.1. *In the* CONGEST *model:*

(a) A rooted spanning tree T of diameter D of the communication graph G can be constructed in $O(D)$ rounds.

(b) Let M be a set of $O(\log n)$-bit messages, each given to a node. Then all nodes can learn M in $O(|M| + D)$ rounds.

With the above routing primitives, it is straightforward to derive the following universal upper bound.

Theorem 5.2. *For any problem* Π*, there exists a dynamic batch algorithm that uses* $O(\alpha + D)$ *time and* $O(m \log n)$ *space.*

Proof. Define $\xi(G, \ell) = \{\ell\}$, that is, the only valid auxiliary state is a full description of the input. Define the algorithm $\mathcal{A}$ as follows:

(1) Let $\dot{E} \subseteq E$ be the set of edges that changed. Define
$$M = \{(u, v, \ell_2(\{u, v\})) : \{u, v\} \in \dot{E}\} .$$
The set M encodes the α changes and each message in M can be encoded using $O(\log n)$ bits. By Lemma 5.1b, all nodes can learn the changes in $O(\alpha + D)$ rounds.
(2) Given M, each node $v \in V$ can locally construct ℓ_2 from M and ℓ_1. Set $x_2(v) = \ell_2$.
(3) Each node $v \in V$ locally computes a solution $s \in \Pi(G, \ell_2)$ and outputs $s(v)$.

The claim follows by observing that the update algorithm $\mathcal{A}$ takes $O(\alpha + D)$ rounds and that $\xi(G, \ell) = \{\ell\}$ can be encoded using $O(m \log n)$ bits. □

As a second baseline, we consider problems that are strictly local in the sense that there is a constant r such that the output of a node v only depends on the radius-r neighbourhood of v. Equivalently, this means that the problem belongs to the class of problems solvable in $O(1)$ rounds in the LOCAL model, denoted by LOCAL(1).

Theorem 5.3. *For any* LOCAL(1) *problem, there exists a dynamic batch algorithm that uses* $O(\alpha)$ *time and* $O(m \log n)$ *space.*

Proof. Let r be the constant such that the output of a node v only depends on the radius-r neighbourhood of v. For each node v, the auxiliary state is the full description of the input labelling in radius-r neighbourhood of v. Define the algorithm $\mathcal{A}$ as follows:

(1) Let $\dot{E} \subseteq E$ be the set of edges that changed. Define
$$M_{v,1} = \{(u, v, \ell_2(\{u, v\})) : u \in N^+(v), \{u, v\} \in \dot{E}\} .$$
The set $M_{v,1}$ encodes the label changes of edges incident to v, and each message in $M_{v,1}$ can be encoded using $O(\log n)$ bits.
(2) For phase $i = 1, 2, \ldots, r$, node v broadcasts $M_{v,i}$ to all of its neighbours, and then announces it is finished with phase i. Let $R_{v,i}$ denote the set of messages node v received in phase i. Once all neighbours have announced they are finished with phase i, node v sets $M_{v,i+1} = R_{v,i} \setminus \bigcup_{j=1}^{i} M_{v,j}$ and moves to phase $i + 1$.
(3) Once all neighbours of v are finished with phase r, node v can locally reconstruct ℓ_2 in it's radius-r neighbourhood and set the new local auxiliary state $x_2(v)$.
(4) Node v locally computes output $s(v)$ from $x_2(v)$ and halts.

The claim follows by observing that each set $M_{i,v}$ can have size at most α, and a node can be in any of the $r = O(1)$ phases for $O(\alpha)$ rounds. In the worst case, the radius-r neighbourhood of a node is the whole graph, in which case encoding the full input labelling takes $O(m \log n)$ bits. □

6 BATCH DYNAMIC CLIQUE ENUMERATION

In this section, we show that we can do better than the trivial baseline of $O(\alpha)$ rounds for the fundamental local subgraph problem of enumerating cliques.

We consider a setting where the input is a subgraph of the communication graph, represented by label for each edge indicating its existence in the subgraph. We show that for any $k \geq 3$, there is a sublinear-time (in α) batch dynamic algorithm for *enumerating* k-cliques. More precisely, we give an algorithm that for each node v maintains the *induced* subgraph of its radius-1 neighbourhood. This algorithm runs in $O(\alpha^{1/2})$ rounds and can be used to maintain, at each node, the list all cliques the node is part of.

To contrast this upper bound, Section 8 shows that even the easier problem of *detecting* k-cliques requires $\Omega(\alpha^{1/4}/\log \alpha)$ rounds. While this does not settle the complexity of the problem, it shows that this central problem has non-trivial, *intermediate* complexity: more than constant or poly $\log \alpha$, but still sublinear in α.

6.1 Acyclic orientations

An orientation of a graph $G = (V, E)$ is a map σ that assigns a direction to each edge $\{u, v\} \in E$. For any $d > 0$, we say that σ is a d-orientation if

(1) every node $v \in V$ has at most d outgoing edges,
(2) the orientation σ is acyclic.

We use $\text{outdeg}_\sigma(v)$ to denote the number of outgoing edges from v in the orientation σ.

A graph G has *degeneracy* d ("is d-degenerate") if every non-empty subgraph of G contains a node with degree at most d. It is well-known that a graph G admits a d-orientation if and only if G has degeneracy of at most d. We use the following graph theoretic observation.

Lemma 6.1. *Let G be a d-degenerate graph with n nodes and m edges. Then $d \le \sqrt{2m}$ and $m \le nd$.*

Proof. For the first claim, suppose that $d > \sqrt{2m}$. Then there is a subset of nodes U such that $G[U]$ has minimum degree $\delta > \sqrt{2m}$. It follows that U has at least $\delta + 1$ nodes, and thus the number of edges incident to nodes in U in G is at least

$$\frac{1}{2}\sum_{v \in U} \deg_G(v) \ge \frac{1}{2}\delta(\delta + 1) > \frac{1}{2}\sqrt{2m}(\sqrt{2m} + 1) > m,$$

which is a contradiction. The second claim follows by considering a d-orientation σ of G and observing that

$$m = \sum_{v \in V} \text{outdeg}_\sigma(v) \le nd. \qquad \square$$

Let $\dot{E} \subseteq E$ be the set of α edges that are changed by the batch update. We show that the edges of $G[\dot{E}]$ can be quickly oriented so that each node has $O(\sqrt{\alpha})$ outgoing edges despite nodes not knowing α. This orientation serves as a routing scheme for efficiently distributing relevant changes in the local neighbourhoods.

Lemma 6.2. *An $O(\sqrt{\alpha})$-orientation of $G[\dot{E}]$ can be computed in $O(\log^2 \alpha)$ rounds.*

Proof. Recall that m is the number of edges in the communication graph G. Let $H = (U, \dot{E})$ and note that $|U| \le 2\alpha$ and $\alpha \le m$. For an integer d, define

$$f(d) = 3 \cdot \sqrt{2^{d+1}} \quad \text{and} \quad T(d) = \left\lceil \log_{3/2} 2^{d+1} \right\rceil.$$

The orientation of H is computed iteratively as follows:

(1) Initially, each edge $e \in \dot{E}$ is unoriented.
(2) In iteration $d = 1, \dots, \lceil \log m \rceil$, repeat the following for $T(d)$ rounds:
   - If node v has at most $f(d)$ unoriented incident edges, then v orients them outwards and halts. In case of conflict, an edge is oriented towards the node with the higher identifier.
   - Otherwise, node v does nothing.

Clearly, if node v halts in some iteration d, then v will have outdegree at most $f(d)$.

Fix $\hat{d} = \lceil \log \alpha \rceil \leq \lceil \log m \rceil$. We argue that by the end of iteration $\hat{d}$, all edges of H have been oriented. For $0 \leq i \leq T(\hat{d})$, define $U_i \subseteq U$ to be the set of vertices that have unoriented edges after $i \geq 0$ rounds of iteration $\hat{d}$, i.e.,

$$U_{i+1} = \{v \in U_i : \deg_i(v) > f(\hat{d})\},$$

where $\deg_i(v)$ is the degree of node v in subgraph $H_i = H[U_i]$ induced by U_i.

Note that every $u \in U \setminus U_0$ has outdegree at most $f(\hat{d})$. We now show that each node in U_0 halts with outdegree at most $f(\hat{d})$ within $T(\hat{d})$ rounds. First, observe that $|U_{i+1}| < \frac{2}{3}|U_i|$. To see why, notice that by Lemma 6.1 each H_i has degeneracy at most $\sqrt{2\alpha}$ and thus at most $|U_i| \cdot \sqrt{2\alpha}$ edges. If $|U_{i+1}| \geq \frac{2}{3}|U_i|$ holds, then H_{i+1} has at least

$$\frac{1}{2} \cdot \sum_{v \in U_{i+1}} \deg_i(v) > \frac{1}{2} \cdot |U_{i+1}| \cdot f(\hat{d}) \geq \frac{2}{3} \cdot |U_i| \cdot f(\hat{d})$$
$$= |U_i| \cdot \sqrt{2^{\hat{d}+1}} > |U_i| \cdot \sqrt{2\alpha}$$

edges, which is a contradiction. Thus, we get that $|U_{i+1}| < (2/3)^i \cdot |U|$ and

$$|U_{T(\hat{d})}| < (2/3)^{T(\hat{d})} \cdot 2\alpha \leq \frac{2\alpha}{2^{\hat{d}+1}} \leq 1.$$

Therefore, each edge of H is oriented by the end of iteration $\hat{d} = \lceil \log \alpha \rceil$ and each node has at most $f(\hat{d}) = O(\sqrt{\alpha})$ outgoing edges. As a single iteration takes at most $O(\log \alpha)$ rounds, all nodes halt in $O(\log^2 \alpha)$ rounds, as claimed. □

6.2 Algorithm for clique enumeration

Let $G^+[v]$ denote the subgraph induced by the radius-1 neighbourhood of v; note that this includes all edges between neighbours of v. Let $H_1 \subseteq G$ and $H_2 \subseteq G$ be the subgraphs given by the previous input labelling ℓ_1 and the new labelling ℓ_2, respectively. The auxiliary state $x(v)$ of the batch dynamic algorithm is a map $x(v) = y_v$ such that $y_v : E(G^+[v]) \to \{0, 1\}$. The map y_v encodes which edges in $G^+[v]$ are present in the input subgraph.

The dynamic algorithm computes the new auxiliary state x_2 encoding the subgraph $H_2^+[v]$ as follows:

(1) Each node v runs the $O(\alpha^{1/2})$-orientation algorithm on $G[\dot{E}]$ until all nodes in its radius-1 neighbourhood $N^+(v)$ have halted (and oriented their edges in $\dot{E}$).
(2) Let $\dot{E}_{\text{out}}(v) \subseteq \dot{E}$ be the set of outgoing edges of v in the orientation. Node $v \in V$ sends the set
$$A(v) = \{(e, \ell_2(e)) : e \in \dot{E}_{\text{out}}(v)\}$$
to each of its neighbours $u \in N(v)$.
(3) Define $R(v) = \bigcup_{u \in N^+(v)} A(u)$ and the map $y'_v \colon E(G^+[v]) \to \{0, 1\}$ as
$$y'_v(e) = \begin{cases} \ell_2(e) & \text{if } (e, \ell_2(e)) \in R(v) \\ y_v(e) & \text{otherwise,} \end{cases}$$
where y_v is the map encoded by the auxiliary state $x_1(v)$.
(4) Set the new auxiliary state to $x_2(v) = y'_v$.

First, we show that the computed auxiliary state of each node v encodes the subgraph $H_2^+[v]$ induced by the radius-1 neighbourhood of v in the new input graph H_2.

Lemma 6.3. *Let $v \in V$ and $e \in G^+[v]$. Then we have $y'_v(e) = 1$ if and only if $e \in H_2^+[v]$.*

Proof. There are two cases to consider. First, suppose $e = \{u, w\} \in \dot{E}$. After Step (1), the edge $\{u, w\}$ is w.l.o.g. oriented towards u. Hence, in Step (2), if $w \neq v$, then w sends $(e, \ell_2(e)) \in A(w)$ to v, as $w \in N(v)$, and if $w = v$ then v knows $A(v)$. Thus, $e \in G^+[v] \cap \dot{E} \subseteq R(v)$. By definition of y'_v it holds that $y'_v(e) = \ell_2(e) = 1$ if and only if $e \in H_2^+[v]$ holds.

For the second case, suppose $e \notin \dot{E}$. Then, as $H_1^+[v] \setminus \dot{E} = H_2^+[v] \setminus \dot{E}$, and by definition of y'_v, we have that $y'_v(e) = y_v(e) = 1$ if and only if $e \in H_2^+[v] \setminus \dot{E}$ holds. □

Next, we upper bound the running time of the above algorithm.

Lemma 6.4. *Each node v computes $H_2^+[v]$ in $O(\alpha^{1/2})$ rounds.*

Proof. By Lemma 6.2, Step (1) completes in $O(\log^2 \alpha)$ rounds and $|A| = O(\alpha^{1/2})$. Since each edge in A can be encoded using $O(\log n)$ bits, Step (2) completes in $O(\alpha^{1/2})$ rounds. As no communication occurs after Step (2), the running time is bounded by $O(\alpha^{1/2} + \log^2 \alpha)$. By Lemma 6.3, node v learns $H_2^+[v]$ in Step (3). □

Note that if a node v is part of a k-clique, then all the edges of this clique are contained in $H_2^+[v]$. Thus, node v can enumerate all of its k-cliques by learning $H_2^+[v]$, and hence, we obtain the following result.

Theorem 6.5. *There exists an algorithm for clique enumeration in the batch dynamic* CONGEST *model that runs in $O(\alpha^{1/2})$ rounds and uses $O(m \log n)$ bits of auxiliary state.*

7 MINIMUM-WEIGHT SPANNING TREES

In this section, we construct an algorithm that computes a minimum-weight spanning tree in the dynamic batch model in $O(\alpha + D)$ rounds and using $O(\log n)$ bits of auxiliary state between batches. For the dynamic minimum spanning tree, we assume that the input label $w(e) \in \{0, 1, 2, \dots, n^C\} \cup \{\infty\}$ encodes the weight of edge $e \in E$, where C is a constant, and that the output defines a rooted minimum spanning tree, with each node v outputting the identifier of their parent.

To do this, we will use a distributed variant of an *Eulerian tour tree*, a data structure familiar from classic centralised dynamic algorithms. In the distributed setting, it allows us to make inferences about the relative positions of edges with regard to the minimum spanning tree *without* full information about the tree. Moreover, the Eulerian tour tree can be compactly encoded into the auxiliary state using only $O(\log n)$ bits per node.

In the following, we first describe a distributed variant of this structure and then how to use it in conjunction with a the *minimum-weight matroid basis* algorithm of Peleg [66] to compute the minimum spanning tree in the dynamic batch CONGEST model efficiently.

7.1 Distributed Eulerian tour trees

We now treat $G = (V, E)$ as a directed graph, where each edge $\{u, v\}$ is replaced with (u, v) and (v, u). As before, we treat a subgraph (V, F) of G interchangeably with the edge set $F \subseteq E$ and further abuse the notation by taking a subtree T of G to mean a directed subgraph of G containing all directed edges corresponding to an undirected tree on G. In particular, we use $|T|$ to denote the number of directed edges in T.

Let $H \subseteq G$ be a subgraph of G. The bijection $\tau\colon E(H) \to \{0, \dots, |E(H)| - 1\}$ is an *Eulerian tour labelling* of H if the sequence of directed edges $\tau^{-1}(0), \tau^{-1}(1), \dots, \tau^{-1}(|E(H)| - 1)$ gives an Eulerian tour of H. We say that u is the root of τ if there is some edge (u, v) such that $\tau(u, v) = 0$. For a map $f\colon A \to B$ and a set $C \subseteq A$, the restriction of f to domain C is the map $f \restriction_C\colon C \to A$ given by $f(c) = f \restriction_C (c)$ for all $c \in C$.

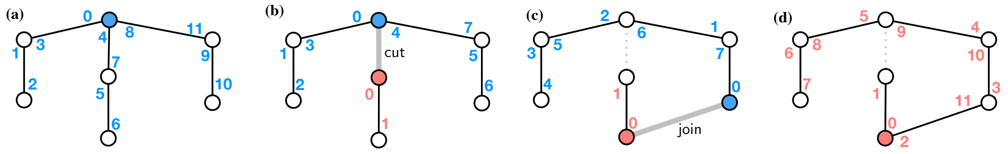

Fig. 2. Eulerian tour trees. (a) An example of Eulerian tour labelling, with root node marked in blue. (b) The updated Eulerian tour labellings after applying a cut operation. The roots of the new blue and red trees are marked with respective colours. (c) To apply a join operation, we root the red and blue tree to the endpoints of the join edge. (d) Eulerian tour labelling after the join operation.

Eulerian tour forests. An *Eulerian tour forest on* $\mathcal{F}$ is a tuple $\mathcal{L} = (L, r, s, a)$ such that

(1) $\mathcal{F} = \{T_1, \dots, T_h\}$ is a spanning forest of G,
(2) $L\colon E \longrightarrow \{0, \dots |E| - 1\} \cup \{\infty\}$ is a mapping satisfying the following conditions:
   - for each $T \in \mathcal{F}$ the map $L \restriction_T$ is an Eulerian tour labelling of T, and
   - if $(u, v) \notin \bigcup \mathcal{F}$, then $L(u, v) = \infty$,
(3) $r\colon V \longrightarrow V$ is a mapping such that for each $T \in \mathcal{F}$ and node $v \in V(T)$, we have that $r(v)$ is the root of the Eulerian tour labelling $L \restriction_T$ of T,
(4) $s\colon V \longrightarrow \{0, \dots |E|\}$ is a mapping satisfying $s(v) = |T|$ for each $T \in \mathcal{F}$ and a node $v \in V(T)$,
(5) $a\colon V \longrightarrow \{0, \dots |E| - 1\}$ is a mapping satisfying for each $T \in \mathcal{F}$ and node $v \in V(T)$ the following conditions:
   - if T contains at least one edge, then $a(v) = \min\{L(e)\colon e \text{ is an outgoing edge from } v\}$,
   - if T consists of only node v, then $a(v) = 0$.

We define distributed operations which allow us to merge any two trees or cut a single tree into two trees, given that all nodes know which edges the operations are applied to. This data structure is then used to efficiently maintain a minimum spanning tree of G under edge weight changes.

Eulerian tour forest operations. Let $\mathcal{L}$ be an Eulerian tour forest of G. For any $\mathcal{L} = (L, r, s, a)$ and $E' \subseteq E$, we define the restricted labelling $\mathcal{L} \restriction_{E'} = (L \restriction_{E'}, r \restriction_U, s \restriction_U, a \restriction_U)$, where $U = \bigcup E'$ is the set of nodes incident to edges in E'. We implement two operations for manipulating $\mathcal{L}$ (illustrated by Figure 2): a join operation that merges two trees and a cut operation that removes an edge from a tree and creates two new disjoint trees. To implement the two basic operations join and cut, we also use an auxiliary operation root that is used to reroot a tree.

For brevity, let $T(u)$ to denote the tree node u belongs to in the Eulerian tour forest. We use $|T|$ to denote the number of directed edges in T. The three operations are as follows:

- root$(\mathcal{L}, u)$: Node u becomes the root of the tree $T(u)$.
  Implementation: Set
$$\begin{aligned} L(w, v) &\leftarrow L(w, v) - a(u) \bmod s(u) \\ &\quad \text{for each } (w, v) \in T(u), \text{ and} \\ a(v) &\leftarrow a(v) - a(u) \bmod s(u) \\ &\quad \text{for each } v \in V(T(u)). \end{aligned}$$
  Otherwise, L and a remain unchanged. Moreover, $r(v) \leftarrow u$ if $r(u) = r(v)$ and otherwise r remains unchanged. All tree sizes remain unchanged.
- join$(\mathcal{L}, e)$: If $e = \{v_i, v_j\}$, where $v_i \in V(T_i)$ and $v_j \in V(T_j)$ for $i \neq j$, then merge T_i and T_j and create an Eulerian tour labelling of $T' = T_i \cup T_j \cup \{e\}$. The root of T' will be the endpoint of e with the smaller identifier.

Implementation: Let $e = \{v_i, v_j\}$, where $v_i \in V(T_i)$ and $v_j \in V(T_j)$ for $i \neq j$. Without loss of generality, suppose $v_i < v_j$. The operation is implemented by the following steps:

(1) Run $\mathsf{root}(v_i)$ and $\mathsf{root}(v_j)$.
(2) Set $L(v_i, v_j) \longleftarrow s(v_i)$ and $L(v_j, v_i) \longleftarrow s(v_i) + s(v_j) + 1$.
For each $(u, v) \in T_j$, set $L(u, v) \longleftarrow L(u, v) + s(v_i) + 1$.
(3) For each $u \in V(T_j)$, set $a(u) \longleftarrow a(u) + s(v_i) + 1$.
(4) For each $u \in T'$, set $s(u) \longleftarrow s(v_i) + s(v_j) + 2$ and $r(u) \longleftarrow v_i$.

– $\mathsf{cut}(\mathcal{L}, e)$: For an edge $e = \{v_1, v_2\}$ in some tree T, create two new disjoint trees T_1 and T_2 with Eulerian tour labellings rooted at v_1 and v_2 such that $T_1 \cup T_2 = T \setminus \{e\}$.
Implementation: Let $e = \{v_1, v_2\}$. Without loss of generality, assume that $a(v_1) < a(v_2)$. Let $z_1 = L(v_1, v_2)$, $z_2 = L(v_2, v_1)$, and $x = z_2 - z_1$. The edge labels are updated as follows:

(1) Set $L(v_1, v_2) \longleftarrow \infty$ and $L(v_2, v_1) \longleftarrow \infty$.
(2) If $a(v) \in [0, \dots, z_1]$, then set $s(v) \longleftarrow s(v) - x - 1$ and $a(v) \longleftarrow a(v)$.
If $a(v) \in (z_1, \dots, z_2]$, then set $s(v) \longleftarrow x - 1$ and $a(v) \longleftarrow a(v) - z_1 - 1$.
Otherwise, set $s(v) \longleftarrow s(v) - x - 1$ and $a(v) \longleftarrow a(v) - x - 1$.
(3) If $L(u, v) \in [0, \dots, z_1)$, then set $L(u, v) \longleftarrow L(u, v)$.
If $L(u, v) \in (z_1, \dots, z_2)$, then set $L(u, v) \longleftarrow L(u, v) - z_1 - 1$.
Otherwise, set $L(u, v) \longleftarrow L(u, v) - x - 1$.
(4) Run $\mathsf{root}(v_1)$ and $\mathsf{root}(v_2)$.

The next lemma shows that the above operations result in a new Eulerian tour forest, i.e., the operations are correct.

Lemma 7.1. *Given an Eulerian tour forest $\mathcal{L}$, each of the above three operations produce a new Eulerian tour forest $\mathcal{L}'$.*

Proof. For the $\mathsf{root}(u)$ operation, observe that only labels in the subtree $T(u)$ change by being shifted by $a(u)$ modulo $|T(u)|$. Hence, the updated labelling of $T(u)$ remains an Eulerian tour labelling. Since the smallest outgoing edge of u will have label $a(u) - a(u) \mod |T(u)| = 0$, node u will be the root of $T(u)$ in the new Eulerian tour labelling of $T(u)$.

For the $\mathsf{join}(e)$ operation, we observe that after the first step, v_i and v_j are the roots of their respective trees. In particular, after the root operations, the largest incoming edge of v_i will have label $|T_i| - 1$ and the smallest outgoing edge of v_i will have label 0. Hence v_i becomes the root of T'. Moreover, in the new Eulerian labelling any edge in $T(v_i)$ will have a valid Eulerian tour labelling, as the labels for T_i remain unchanged. In T_j the labels are a valid Eulerian tour labelling shifted by $|T_i| + 1$. As, in the new labeling the the edge (v_i, v_j) will have label $|T_i|$ and the smallest outgoing label of v_j will be $|T_i| + 1$, and the largest incoming label will of v_j will be $|T_i| + |T_j|$. The label of (v_j, v_i) will therefore be $|T_i| + |T_j| + 1 = |T_i \cup T_j \cup \{e\}| - 1$ and this is the largest label of the new Eulerian tour labelling. Hence, the new labelling is an Eulerian tour forest.

Finally, consider the $\mathsf{cut}(e)$ operation. Let T_1 and T_2 be the trees created by removing the edge e from T. Note that $x = z_2 - z_1 = |T_2| + 1$ and $|T| = |T_1| + |T_2| + 2$, since we are counting directed edges. Clearly, after cutting the edge e from T, the a node v belongs to subtree T_2 if $a(v) \in [z_1, \dots, z_2)$ and otherwise to T_1. Thus, in the latter case $s(v)$ is set to $|T| - x - 1 = |T| - |T_2| - 2 = |T_1|$, and in the former, $s(v)$ is set to $x - 1 = |T_2|$.

Suppose an edge $L(u, v) < z_1$. Then the edge (u, v) belongs to T_1 and its label will remain unchanged. Now suppose $L(u, v) > z_2$. Then (u, v) will be part of T_1 and its new label will be $L(u, v) - x - 1 = L(u, v) - |T_2| - 1$. In particular, the edge of u_1 with the smallest outgoing label $z_2 + 1$ will have the label z_1 in the new labeling. Thus, L restricted to T_1 will be a valid Eulerian tree tour labelling of T_1. It remains to consider the case that $L(u, v) \in (z_1, z_2)$. However, it is easy to check that now the root of T_2 will be v_2 and the new labelling restricted to T_2 will be a valid

Eulerian tree tour labelling of T_2. Finally, the root operations ensure that the endpoints of e become the respective roots of the two trees, updating the variables $r(\cdot)$. □

A key property of the Eulerian tour forest structure is that any node that knows the labels of a set $E' \subseteq E$ can locally deduce the new labels of all edges in E' after either join or cut operation is applied to a given edge in E'.

Lemma 7.2. *Let $\mathcal{L}$ be an Eulerian tour forest and $e, f \in E$ be edges. Suppose $\mathcal{L}'$ is obtained by applying either the* $\mathsf{join}(\mathcal{L}, e)$ *or the* $\mathsf{cut}(\mathcal{L}, e)$ *operation. Then $\mathcal{L}' \restriction_{\{e,f\}}$ can be computed from $\mathcal{L} \restriction_{\{e,f\}}$.*

Proof. Let $e = \{u_1, u_2\}$ and $f = \{v_1, v_2\}$. Let $f \neq e$ be an edge whose labels we need to compute after an operation on e. We show that after applying any one of the three operations on $\mathcal{L}$, the labels $\mathcal{L}' \restriction_f$ can be computed from $\mathcal{L} \restriction_{\{e \cup f\}}$. There are three cases to consider:

(1) $\mathcal{L}' = \mathsf{root}(\mathcal{L}, u)$: If $f \notin T(u)$, then $\mathcal{L} \restriction_f = \mathcal{L}' \restriction_f$, as the labels of f do not change. If $f \in T(u)$, then $\mathcal{L}' \restriction_f$ depends only on $a(u)$ and $s(u)$.
(2) $\mathcal{L}' = \mathsf{join}(\mathcal{L}, e)$: If $f \notin T_1 \cup T_2$, then $\mathcal{L} \restriction_f = \mathcal{L}' \restriction_f$, as the labels of f do not change after the joining these two trees. Hence suppose $f \in T_1 \cup T_2$. From the previous case, we know that the two root operations depend on $a(u_i)$ and $s(u_i)$ for $i \in \{1, 2\}$. The latter two steps depend only on $s(u_i)$. As these values are contained in $\mathcal{L} \restriction_{\{e,f\}}$, the restriction $\mathcal{L}' \restriction_f$ is a function of $\mathcal{L} \restriction_{\{e,f\}}$.
(3) $\mathcal{L}' = \mathsf{cut}(\mathcal{L}, e)$: If $f \notin T$, then the labels of f do not change. Hence, suppose $f \in T$. One readily checks that the update operations in Steps 1-3 depend on $z_1 = L(u_1, u_2)$, $z_2 = L(u_2, u_1)$, $a(v_i)$ and $s(v_i)$ for $i \in \{1, 2\}$. Therefore, $\mathcal{L}' \restriction_f$ is a function of $\mathcal{L} \restriction_{\{e,f\}}$.

Thus, in all cases $\mathcal{L}' \restriction_f$ is a function of $\mathcal{L} \restriction_{\{e,f\}}$, and the claim follows. □

Storing the Eulerian tour tree of a minimum-weight spanning tree. Suppose $\mathcal{L}$ is an Eulerian tour forest on the minimum-weight spanning tree of G. Later, our algorithm will in fact always maintain such a Eulerian tour forest after a batch of updates.

The auxiliary state x is defined as follows. For each node $v \in V$, the auxiliary state $x(v)$ consists of the tuple $(r(v), p(v), \lambda(v))$, where

- $r(v)$ is the identifier of the root of the spanning tree,
- $p(v)$ points to the parent of v in the spanning tree,
- $\lambda(v) = \big(L(p(v), v), L(v, p(v))\big)$, respectively.

These variables can be encoded in $O(\log n)$ bits. Moreover, each node v can reconstruct $\mathcal{L} \restriction_{E(v)}$ from the auxiliary state x in $O(1)$ rounds.

Lemma 7.3. *Given the auxiliary state $x_1(v)$ that encodes $\mathcal{L}$ on a spanning tree of G, each node v can learn in $O(1)$ communication rounds $\mathcal{L} \restriction_{E(v)}$. Likewise, given $\mathcal{L} \restriction_{E(v)}$, node v can compute the corresponding auxiliary state $x_1(v)$ locally.*

Proof. Since $\mathcal{L}$ is an Eulerian tour forest on a spanning tree, every node v knows s (the size of the spanning tree) and r (the root of the tree), as both are constant functions. As $\lambda(v)$ can be encoded using $O(\log n)$ bits, each node v can send $\lambda(v)$ to all of its neighbours in $O(1)$ communication rounds. Thus, after $O(1)$ rounds node v knows $L \restriction_{E(v)}$. The second part follows directly from the definition of $x_1(v)$. □

7.2 Maximum matroid basis algorithm

We use an algorithm of Peleg [66] as a subroutine for finding minimum and maximum weight *matroid bases* in distributed manner. We first recall the definition of matroids.

Definition 7.4. A *matroid* is a pair $\mathcal{M} = (A, \mathcal{I})$, where A is a set and $\mathcal{I} \subseteq 2^A$ satisfies the following:

(1) The family $\mathcal{I}$ is non-empty and closed under taking subsets.
(2) For any $I_1, I_2 \in \mathcal{I}$, if $|I_1| > |I_2|$, then there is an element $x \in I_1 \setminus I_2$ such that $I_2 \cup \{x\} \in \mathcal{I}$. This is called the *augmentation property* of a matroid.

We say that a set $I \subseteq A$ is *independent* if $I \in \mathcal{I}$. A maximal independent set is called a *basis.*

In the *maximum matroid basis problem*, we are given a matroid $\mathcal{M} = (A, \mathcal{I})$ with a weight function $w\colon A \to \{-n^C, \dots, n^C\}$ giving unique weights for all elements, and the task is to a find a basis B of $\mathcal{M}$ with maximum weight $w(B) = \sum_{x \in B} w(x)$. In more detail, the input is specified as follows:

– Each node receives a set $A_v \subseteq A$ as input, along with the associated weights. We have a guarantee that $\bigcup_{v \in V} A_v = A$, and the sets A_v may overlap.
– Each element $x \in A$ is decorated with additional data $M(x)$ of $O(\log n)$ bits, and given $M(A')$ for $A' \subseteq A$, a node v can locally compute if A' is independent in $\mathcal{M}$.

As output, all nodes should learn the maximum-weight basis B. Note that since negative weights are allowed and all bases have the same size, this is equivalent to finding a *minimum-weight* matroid basis.

Theorem 7.5 ([66]). *The distributed maximum matroid basis problem over $\mathcal{M}$ can be solved in $O(\alpha + D)$ rounds, where α is the size of bases of $\mathcal{M}$.*

7.3 Cycle and cut properties

We make use of the following well-known cycle and cut properties of spanning trees.

Lemma 7.6. *Suppose the weights of the graph G are unique. Then the following hold:*

– *Cycle property: For any cycle C, the heaviest edge of C is not in minimum-weight spanning tree of G.*
– *Cut property: For any set $X \subseteq V$, the lightest edge between X and $V \setminus X$ is in the minimum-weight spanning tree of G.*

7.4 Maintaining a minimum spanning tree

Let $G_1 = (V, E, w_1)$ and $G_2 = (V, E, w_2)$ be the graph before and after the α edge weight changes. Since each edge is uniquely labelled with the identifiers of the end points, we can define a global total order on all the edge weights, where edges are ordered by weight and any equal-weight edges are ordered by the edge identifiers. Let T_1^* and T_2^* be the unique minimum-weight spanning trees of G_1 and G_2, respectively.

Communicated messages. We now assume that each *communicated* edge e is decorated with the tuple $M(e) = \big(\mathcal{L} \restriction_{\{e\}}, w_1(e), w_2(e)\big)$. For a set E, we write $M(E) = \{M(e)\colon e \in E\}$. Note that for any edges $e, e' \in E$, the information $M(e)$ and $M(e')$ suffice to compute $\mathcal{L}' \restriction_{e'}$ after either a $\mathsf{join}(\mathcal{L}, e)$ or $\mathsf{cut}(\mathcal{L}, e)$ operation on $\mathcal{L}$, by Lemma 7.2. Since $M(e)$ can be encoded in $O(\log n)$ bits, the message encoding $M(e)$ can be communicated via an edge in $O(1)$ rounds.

Overview of the algorithm. The algorithm heavily relies on using a BFS tree $\mathcal{B}$ of the communication graph G as a broadcast tree, given by Lemma 5.1. Without loss of generality, observe that we can first process at most α weight *increments* and then up to α weight *decrements* afterwards. On a high-level, the algorithm is as follows:

(1) Let $E^+ = \{e : w_2(e) > w_1(e)\}$ and $E^- = \{e : w_2(e) < w_1(e)\}$.
(2) Solve the problem on the graph G_1' obtained from G_1 by changing only the weights in E^+.
(3) Solve the problem on the graph G_2 obtained from G_1' by changing the weights in E^-.

We show that Steps (2)–(3) can be done in $O(\alpha + D)$ rounds, which yields the following result.

Theorem 7.7. *There is an algorithm for minimum-weight spanning trees in the batch dynamic* CONGEST *model that runs in $O(\alpha + D)$ rounds and uses $O(\log n)$ bits per node to store the auxiliary state.*

7.5 Handling weight increments

We now design an algorithm that works in the case $|E^+| \leq \alpha$ and $E^- = \emptyset$. That is, the new input graph G_2 differs from G_1 by having only the weights of edges in E^+ incremented. Let T_1^* and T_2^* be the minimum spanning trees of G_1 and G_2, respectively. Note that $F = T_1^* \setminus E^+$ is a forest on G_1 and G_2 and $w_1(F) = w_2(F)$, splitting the graph into connected components. Let $A^* \subseteq E \setminus F$ be the *lightest* set of edges connecting the components of F under weights w_2.

Lemma 7.8. *The spanning tree $F \cup A^*$ is the minimum-weight spanning tree of G_2.*

Proof. Suppose there exists some edge $e \in F \setminus T_2^*$. Let u be a node incident to e and let $S \subseteq V$ be the set of nodes in the connected component of u in $F \setminus \{e\}$. By the *cut property* given in Lemma 7.6, the lightest edge f (with respect to w_2) in the cut between S and $V \setminus S$ is in the minimum spanning tree T_2^*. Since $e \notin T_2^*$ and $f \in T_2^*$, we have that $w_2(f) < w_2(e)$. By definition, $e \in F$ implies that $e \notin E^+$, and hence,

$$w_1(f) \leq w_2(f) < w_2(e) = w_1(e).$$

Thus, there exists a spanning tree $T' = (T_1^* \setminus \{e\}) \cup \{f\}$ such that $w_1(T') < w_1(T_1^*)$. But by definition of F, we have $e \in F \subseteq T_1^*$, which is a contradiction. Hence, $F \subseteq T_2^*$. Since $F \subseteq T_2^*$ is a forest and A^* is the lightest set of edges that connects the components of F, the claim follows. □

We show that the set A^* can be obtained as a solution to a minimum matroid basis problem, and thus can be computed in $O(\alpha + D)$ communication rounds. In the following, we assume that the auxiliary state encodes an Eulerian tour forest $\mathcal{L}$ on T_1^*. We first show that A^* is a minimum-weight basis of an appropriately chosen matroid. Let A be the set of *all* edges that connect components of F in G_2.

Lemma 7.9. *Let $\mathcal{I} = \{I \subseteq A \colon F \cup I$ is acyclic on $G_2\}$. Then $\mathcal{M} = (A, \mathcal{I})$ is a matroid and the minimum-weight basis of $\mathcal{M}$ is A^*.*

Proof. We note that $\mathcal{M}$ is matroid, as it's the contraction of the graphical matroid on G (see e.g. [73, Part IV: Matroids and Submodular Functions]). Moreover, for any basis $B \in \mathcal{I}$, the set $F \cup B$ is a spanning tree on G_2 with weight $w_2(F) + w_2(B)$. Since $A^* \in \mathcal{I}$ and $F \cup A^*$ is the unique minimum spanning tree on G_2, it follows that A^* is the minimum-weight basis for $\mathcal{M}$. □

To apply the minimum matroid basis algorithm of Theorem 7.7, we next show that nodes can locally compute whether a set is independent in the matroid $\mathcal{I}$, given appropriate information.

Lemma 7.10. *Assume a node v knows $M(E^+)$ and $M(X)$ for a set $X \subseteq A$. Then v can locally determine if X is independent in $\mathcal{M}$.*

Proof. Recall that $\mathcal{L}$ is the fixed Eulerian tour forest on T_1^* encoded by the auxiliary data of the nodes and messages $M(e)$. By definition, node v can obtain $\mathcal{L} \restriction_{E^+ \cup X}$ from $M(E^+)$ and $M(X)$. Let $X = \{e_1, e_2, \ldots, e_k\}$. To check that X is independent, i.e. $F \cup X$ is a forest, node v uses the following procedure:

(1) Let $\mathcal{L}_0 \restriction_{E^+ \cup X}$ be the Eulerian tour forest on F obtained from $\mathcal{L} \restriction_{E^+ \cup X}$ by applying the cut operation for each $e \in E^+$ in sequence.
(2) For $i \in \{1, \ldots, k\}$ do the following:

(a) Determine from $\mathcal{L}_{i-1} \upharpoonright_{e_i}$ if the endpoints u and w of e_i have the same root, i.e. $r(u) = r(w)$. If this is the case, then $F \cup \{e_1, e_2, \ldots, e_i\}$ has a cycle, and node v outputs that X is not independent and halts.

(b) Compute $\mathcal{L}_i \upharpoonright_X = \mathsf{join}(\mathcal{L}_{i-1} \upharpoonright_X, e_i)$.

(3) Output that X is independent.

If X is not independent, then $F \cup X$ has a cycle and algorithm will terminate in Step 2(a). Otherwise, $F \cup X$ is a forest, and the algorithm will output that X is independent. □

Algorithm for handling weight increments. The algorithm for maintaining minimum spanning trees under weight increments is now as follows:

(1) Each node v computes its local Euler tour forest labelling $\mathcal{L} \upharpoonright_{E(v)}$ from the auxiliary state $x_1(v)$.
(2) Broadcast $M(e)$ for each $e \in E^+$ using the broadcast tree $\mathcal{B}$ given by Lemma 5.1.
(3) Use the minimum matroid basis algorithm over $\mathcal{M}$ to compute A^*.
(4) Each node v locally computes $\mathcal{L}_1 \upharpoonright_{E(v)}$ by applying the cut operation on each edge in $E^+ \setminus A^*$ in lexicographical order, starting from $\mathcal{L} \upharpoonright_{E(v)}$.
(5) Each node v locally computes $\mathcal{L}_2 \upharpoonright_{E(v)}$ by applying the join operation on each edge in $A^* \setminus E^+$ in lexicographical order, starting from $\mathcal{L}_1 \upharpoonright_{E(v)}$.
(6) Each node v outputs local auxiliary state $x_2(v)$ corresponding to $\mathcal{L}_2$.

Lemma 7.11. *The above algorithm solves batch dynamic minimum-weight spanning trees under edge weight increments in $O(\alpha + D)$ rounds.*

Proof. By Lemma 7.3, Step (1) of the algorithm can be done in $O(1)$ rounds, and by Lemma 5.1, Step (2) can be done in $O(\alpha + D)$ rounds. Step (3) can be implemented in $O(\alpha + D)$ rounds by Theorem 7.5 and Lemma 7.9, and after Step (3) all nodes have learned the set A^*. Since all nodes apply the same operations to the Eulerian tour forest in the same order in Steps (4) and (5), all nodes produce compatible auxiliary states in Step (6). □

7.6 Handling weight decrements

We now consider the dual case, where $|E^-| \leq \alpha$ and $E^+ = \emptyset$. Let $B = T_1^* \cup E^-$ and $\mathcal{C}$ be the set of cycles in B. Let $B^* \subseteq B$ be the heaviest edge set such that $B \setminus B^*$ is a spanning tree.

Lemma 7.12. *The spanning tree $B \setminus B^*$ is the minimum spanning tree of G_2.*

Proof. Let $e \in T_2^*$ and suppose $e \notin B = T_1^* \cup E^-$. Since $e \notin T_1^*$ the edge e creates a unique cycle C in T_1^*. The edge e is the heaviest edge on cycle C under weights w_1, as otherwise we would obtain a spanning tree lighter than T_1^* by replacing the heaviest edge on C by e. Since we assume no weight increments and $e \notin E^-$, edge e remains the heaviest edge on the cycle C also under the new edge weights w_2. Hence, $e \notin T_2^*$ by the cycle property, which contradicts our initial assumption. Thus, $T_2^* \subseteq B$.

Now consider any spanning tree $T \subseteq B$. All spanning trees have the same number of edges, and we have $w_2(T) = w_2(B) - w_2(B \setminus T)$. Thus, for the minimum spanning tree T the weight $w_2(B \setminus T)$ is maximised. Since the complement of any spanning tree cuts all cycles in B, we have $T_2^* = B \setminus B^*$. □

Lemma 7.13. *Let*

$$\mathcal{J} = \{J \subseteq B \colon B \setminus J \text{ contains a spanning tree of } B\}.$$

Then $\mathcal{N} = (B, \mathcal{J})$ is a matroid and the maximum-weight basis of $\mathcal{N}$ is B^*.

Proof. We have that $\mathcal{N}$ is the dual of the graphical matroid on (V, B), and thus a matroid (see e.g. [73, Part IV: Matroids and Submodular Functions]). Moreover, B^* is the complement of the minimum spanning tree and thus maximum-weight basis of $\mathcal{N}$. □

Lemma 7.14. *Assume a node v knows $M(E^-)$ and $M(X)$ for a set $X \subseteq B$. Then v can locally determine if X is independent in $\mathcal{N}$.*

Proof. We observe that X is independent in $\mathcal{N}$ if and only if the edge set $B \setminus X$ spans the graph G_2, directly by definitions. Thus, we implement the independence check by using local Eulerian tour forest operations to check if we can obtain a spanning tree $T \subseteq B \setminus X$, by starting from the old minimum spanning tree T_1^*, deleting all edges from X, and then adding edges from E^- to complete the tree if possible.

In more detail, the algorithm works as follows. Recall that by definition, node v can compute $\mathcal{L} \restriction_{E^- \cup X}$ from $M(E^-)$ and $M(X)$. Let $E^- \setminus X = \{e_1, e_2, \dots, e_k\}$.

(1) Let $\mathcal{L}_0 \restriction_{E^- \cup X}$ be the Eulerian tour forest on B obtained from $\mathcal{L} \restriction_{E^- \cup X}$ by applying the cut operation for each $e \in X \cap T_1^*$ in sequence. Note that node can check directly from $\mathcal{L} \restriction_{E^- \cup X}$ which edges in X are in the minimum spanning tree T_1^*.
(2) For $i \in \{1, \dots, k\}$ do the following:
   (a) Determine from $\mathcal{L}_{i-1} \restriction_{e_i}$ if the endpoints u and v of e_i have the same root, i.e. $r(v) = r(u)$.
   (b) If they have the same root, skip this edge and set $\mathcal{L}_i \restriction_{E^- \cup X} = \mathcal{L}_{i-1} \restriction_{E^- \cup X}$.
   (c) If they have different roots, compute $\mathcal{L}_i \restriction_{E^- \cup X} = \mathsf{join}(\mathcal{L}_{i-1} \restriction_{E^- \cup X}, e_i)$.
(3) Check from labels how many connected components $\mathcal{L}_k \restriction_{E^- \cup X}$ has. If the number of roots is one, output that X is independent, otherwise output that X is not independent.

Note that since T_1^* is connected, the final edge set $B \setminus X$ can only have multiple connected components due to removal of edges in X. Thus, the node v will locally see all connected components of $B \setminus X$ from $\mathcal{L}_k \restriction_{E^- \cup X}$. □

Algorithm for handling weight decrements. The algorithm for batch dynamic minimum spanning tree under weight decrements is as follows:

(1) Each node v computes $\mathcal{L} \restriction_{E(v)}$ from the auxiliary state.
(2) Broadcast $M(e)$ for each $e \in E^-$ using the broadcast tree $\mathcal{B}$.
(3) Use the maximum matroid basis algorithm over $\mathcal{N}$ to compute B^*.
(4) Each node v locally computes $\mathcal{L}_1 \restriction_{E(v)}$ by applying the cut operation on each edge in $B^* \cap T_1^*$ in lexicographical order, starting from $\mathcal{L} \restriction_{E(v)}$.
(5) Each node v locally computes $\mathcal{L}_2 \restriction_{E(v)}$ by applying the join operation on each edge in $E^- \cap B^*$ in lexicographical order, starting from $\mathcal{L}_1 \restriction_{E(v)}$.
(6) Each node v outputs local auxiliary state $x_2(v)$ corresponding to $\mathcal{L}_2$.

Lemma 7.15. *There is an algorithm that solves batch dynamic minimum-weight spanning trees under edge weight decrements in $O(\alpha + D)$ rounds.*

Proof. By Lemma 7.3, Step (1) of the algorithm can be done in $O(1)$ rounds, and by Lemma 5.1, Step (2) can be done in $O(\alpha + D)$ rounds. Step (3) can be implemented in $O(\alpha + D)$ rounds by Theorem 7.5 and Lemma 7.14, and after Step (3) all nodes have learned the set B^*. Since all nodes apply the same operations to the Eulerian tour forest in the same order in Steps (4) and (5), all nodes will produce compatible auxiliary states in Step (6). □

8 LOWER BOUNDS

In this section, we investigate lower bounds for the batch dynamic CONGEST model. We start with some necessary preliminaries in Section 8.1 on two-party communication complexity [56], followed by our lower bound framework in Section 8.2, which we instantiate in Section 8.3. Finally, we give a lower bound for the minimum spanning tree problem in Section 8.4 by adapting arguments from Das Sarma et al. [26].

8.1 Two-party communication complexity

Let $f\colon \{0,1\}^k \times \{0,1\}^k \to \{0,1\}$ be a Boolean function. In the two-party communication game on f, there are two players who receive a private k-bit strings x_0 and x_1 as inputs, and their task is to have at least one of the players compute $f(x_0, x_1)$. The players follow a predefined protocol, and the complexity of a protocol is the maximum, over all k-bit inputs, of number of bits the parties exchange when executing the protocol on the input. The *deterministic communication complexity* $\mathrm{CC}(f)$ of a function f is the minimal complexity of a protocol for computing f. Similarly, the *randomised communication complexity* $\mathrm{RCC}(f)$ is the worst-case complexity of protocols, which compute f with probability at least $2/3$ on all inputs, even if the players have access to a source of shared randomness.

While our framework is generic, all the reductions we use are based on *set disjointness* lower bounds. In set disjointness over universe of size k, denoted by DISJ_k, both players inputs are $x_0, x_1 \in \{0,1\}^k$, and the task is to decide whether the inputs are disjoint, i.e. $\mathsf{DISJ}_k(x_0, x_1) = 1$ if for all $i \in \{1, 2, \ldots, k\}$ either $x_0(i) = 0$ or $x_1(i) = 0$, and $\mathsf{DISJ}_k(x_0, x_1) = 0$ otherwise. It is known [56, 71] that

$$\mathrm{CC}(\mathsf{DISJ}_k) = \Omega(k) \qquad \text{and} \qquad \mathrm{RCC}(\mathsf{DISJ}_k) = \Omega(k)\,.$$

8.2 Lower bound framework

For proving lower bounds for batch dynamic algorithms, we use the standard CONGEST lower bound framework of *lower bound families* (e.g. [1, 30]). This allows us to translate existing CONGEST lower bound constructions to batch dynamic CONGEST; however, we need a slightly different definition of lower bound families to account for our setting.

Definition 8.1. For $\alpha \in \mathbb{N}$, let $f_\alpha\colon \{0,1\}^{2k(\alpha)} \to \{0,1\}$ and $s, C\colon \mathbb{N} \to \mathbb{N}$ be functions and Π a predicate on labelled graphs. Suppose that there exists a constant α_0 such that for all $\alpha > \alpha_0$ and $x_0, x_1 \in \{0,1\}^{k(\alpha)}$ there exists a labelled graph $(G(\alpha), \ell(\alpha, x_0, x_1))$ satisfying the following properties:

(1) $(G(\alpha), \ell(\alpha, x_0, x_1))$ satisfies Π iff $f(x_0, x_1) = 1$,
(2) $G(\alpha) = (V_0 \cup V_1, E)$, where
    (a) V_0 and V_1 are disjoint and $|V_0 \cup V_1| = s(\alpha)$,
    (b) the cut between V_0 and V_1 has size at most $C(\alpha)$,
(3) $\ell(\alpha, x_0, x_1)\colon E \to \Sigma$ is an edge labelling such that
    (a) there are at most α edges whose labels depend on x_0 and x_1,
    (b) for $i \in \{0,1\}$, all edges whose label depend on x_i are in $E \cap V_i \times V_i$, and
    (c) labels on all other edges do not depend on x_0 and x_1.

We then say that $\mathcal{F} = (\mathcal{G}(\alpha))_{\alpha > \alpha_0}$ is a *family of lower bound graphs for* Π, where

$$\mathcal{G}(\alpha) = \left\{ (G(\alpha), \ell(\alpha, x_0, x_1))\colon x_0, x_1 \in \{0,1\}^{k(\alpha)} \right\}.$$

Extensions. Since our aim is to prove lower bounds that depend on number of input changes α independently of the number of nodes n, we need to construct lower bounds where α can be arbitrarily small compared to n. We achieve this by embedding the lower bound graphs into a larger graph; this requires that the problem we consider has the following property.

Definition 8.2. Let Π be a problem on labelled graphs. We say that Π has the *extension property with label* γ if $\gamma \in \Gamma$ is an input label such that for any labelled graph (G, ℓ), attaching new nodes and edges with label γ does not change the output of the original nodes.

Lower bound theorems. We now present our lower bound framework, which we will instantiate in the next Section 8.3. We first show the following general version of the lower bound result.

Theorem 8.3. *Let Π be a problem, assume there is a family of lower bound graphs $\mathcal{F}$ for Π and that Π has the extension property, and let $L\colon \mathbb{N} \to \mathbb{N}$ be a function satisfying $L(\alpha) \geq s(\alpha)$. Let $\mathcal{A}$ be a deterministic batch dynamic algorithm that solves Π in $T(\alpha, n)$ rounds for all α satisfying $n \geq L(\alpha)$ on batch dynamic* CONGEST *with bandwidth $b(n)$. Then we have*

$$T(\alpha, L(\alpha)) = \Omega\left(\frac{\mathrm{CC}(f_\alpha)}{C(\alpha)b\big(L(\alpha)\big)}\right).$$

If $\mathcal{A}$ is a Monte Carlo algorithm with running time $T(\alpha, n)$ rounds and success probability at least $2/3$, or a Las Vegas algorithm with running time $T(\alpha, n)$ in either expectation or with probability at least $2/3$, then we instead have

$$T(\alpha, L(\alpha)) = \Omega\left(\frac{\mathrm{RCC}(f_\alpha)}{C(\alpha)b\big(L(\alpha)\big)}\right).$$

Proof. First consider the case of deterministic $\mathcal{A}$. We convert $\mathcal{A}$ into a two-player protocol computing $f_\alpha(x_0, x_1)$. Given inputs $x_0, x_1 \in \{0,1\}^{k(\alpha)}$, the players perform the following steps:

(1) Both players construct the graph $G(\alpha)$ and a labelling ℓ such that ℓ agrees with $\ell(\alpha, x_0, x_1)$ on all labels that do not depend on x_0 and x_1, and other labels are set to some default label agreed to beforehand.
(2) Add new nodes connected to an arbitrary node with edges labelled with the extension label γ to $(G(\alpha), \ell)$ to obtain (G^*, ℓ^*) where G^* has $n = L(\alpha)$ nodes; since we assume $L(\alpha) \geq s(\alpha)$, this is possible.
(3) Simulate $\mathcal{A}$ on G^*, with player 0 simulating nodes in V_0 and player 1 simulating nodes in V_1:
   (a) Both players construct a global auxiliary state $x \in \xi(G^*, \ell^*)$; since both players know (G^*, ℓ^*), they can do this locally.
   (b) Player i constructs a new partial labelling by changing the labels on their subgraph to match $\ell(\alpha, x_0, x_1)$. This defines a global labelling ℓ_1^*, which differs from ℓ^* by on at most α edges. Players now simulate $\mathcal{A}(G^*, \ell^*, \ell_1^*, x)$ to obtain a new auxiliary state x_1; players locally simulate their owned nodes and messages between them, and send the messages that would cross the cut between V_0 and V_1 to each other.
(4) Players infer from x_1 whether Π is satisfied, and produce the output $f_\alpha(x_0, x_1)$ accordingly.

Each round, the algorithm $\mathcal{A}$ sends at most $2b(n) = 2b\big(L(\alpha)\big)$ bits over each edge, so the total number of bits players need to send to each other during the simulation is at most $2b\big(L(\alpha)\big)C(\alpha)T\big(\alpha, L(\alpha)\big)$. Since the above protocol computes f_α, we have for $\alpha > \alpha_0$ that $2b\big(L(\alpha)\big)C(\alpha)T\big(\alpha, L(\alpha)\big) \geq \mathrm{CC}(f_\alpha)$, which implies

$$T\big(\alpha, L(\alpha)\big) \geq \frac{\mathrm{CC}(f_\alpha)}{2C(\alpha)b\big(L(\alpha)\big)}.$$

For randomised algorithms, we can directly apply same argument. If $\mathcal{A}$ is a Monte Carlo algorithm with success probability at least $2/3$, then the simulation gives correct result with probability at least $2/3$. If $\mathcal{A}$ is a Las Vegas algorithm that terminates in $T(\alpha, n)$ rounds with probability at least $2/3$, we can simulate $\mathcal{A}$ for $T(\alpha, n)$ rounds and give a random output if it does not terminate by that point; this succeeds in solving set disjointness with probability at least $2/3$. Likewise, if $\mathcal{A}$ has expected running time $T(\alpha, n)$, it suffices to simulate it for $3T(\alpha, n)$ rounds. In all cases, we get

$$T\big(\alpha, L(\alpha)\big) \geq \frac{\mathrm{RCC}(f_\alpha)}{2C(\alpha)b\big(L(\alpha)\big)}$$

as desired. □

In practice, we use the following, simpler version of Theorem 8.3 for our lower bounds. Specifically, we assume the standard $\Theta(\log n)$ bandwidth and no dependence on n in the running time; however, one can easily see that allowing e.g. $\operatorname{poly}\log n$ factor in the running time will only weaker the lower bound by $\operatorname{poly}\log \alpha$ factor.

Corollary 8.4. *Let Π be a problem, assume there is a family of lower bound graphs $\mathcal{F}$ for Π and that Π has the extension property, and let $\varepsilon > 0$ be a constant such that $s(\alpha) \leq \alpha^{1/\varepsilon}$. Let $\mathcal{A}$ be a deterministic batch dynamic algorithm that solves Π in $T(\alpha)$ rounds independent of n for all $\alpha \leq n^{\varepsilon}$ on batch dynamic* CONGEST. *Then we have*

$$T(\alpha) = \Omega\left(\frac{\mathsf{CC}(f_\alpha)}{C(\alpha)\log \alpha}\right).$$

If $\mathcal{A}$ is a Monte Carlo algorithm with running time $T(\alpha)$ rounds and success probability at least $2/3$, or a Las Vegas algorithm with running time $T(\alpha)$ in either expectation or with probability at least $2/3$, then we instead have

$$T(\alpha) = \Omega\left(\frac{\mathsf{RCC}(f_\alpha)}{C(\alpha)\log \alpha}\right).$$

Note the role of ε and s in the claim; the lower bounds in terms of α only work in a regime where α is sufficiently small compared to n. The limit where the lower bound stops working usually corresponds to the complexity of computing the solution from scratch, that is, if α is sufficiently large, then recomputing everything is cheap in terms of the parameter α. On the other hand, we can make ε arbitrarily small, so the lower bound holds even under a promise of small batch size, e.g, $\alpha \leq n^{1/1000}$.

8.3 Instantiations

We now obtain concrete lower bounds by plugging in prior constructions for lower bound families into our framework. These constructions, originally used for CONGEST lower bounds, are parameterised by the number of nodes n, but transforming them to the form used in Definition 8.1 is a straightforward reparameterisation.

Clique detection. In k-clique detection for fixed k, the input labelling $\ell\colon V \longrightarrow \{0,1\}$ defines a subgraph H of G, and each node has to output 1 if they are part of a k-clique in H, and 0 otherwise. The corresponding graph property is k-clique freeness, and k-clique detection has the extension property with label 0.

– *Lower bound family.* For fixed $k \geq 4$, Czumaj and Konrad [25] give a family of lower bound graphs with parameters

$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \quad s(\alpha) = \Theta(\alpha^{1/3}), \quad C(\alpha) = \Theta(\alpha^{3/4}).$$

The lower bound given by Corollary 8.4 is $\Omega(\alpha^{1/4}/\log \alpha)$ for any α.

Cycle detection. Next we consider k-cycle detection for fixed k: the input labelling $\ell\colon V \longrightarrow \{0,1\}$ defines a subgraph H of G, and each node has to output 1 if they are part of a k-cycle in H, and 0 otherwise. The corresponding graph property is k-cycle freeness, and k-cycle detection clearly has the extension property with label 0. For different parameters k, we obtain the lower bounds from prior constructions as follows.

– For 4-cycle detection, Drucker et al. [30] give a family of lower bound graphs with parameters

$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \quad s(\alpha) = \Theta(\alpha^{2/3}), \quad C(\alpha) = \Theta(\alpha^{2/3}).$$

The lower bound given by Corollary 8.4 is $\Omega(\alpha^{1/3}/\log \alpha)$ for $\alpha = O(n^{3/2})$.

– For $(2k+1)$-cycle detection for $k \geq 2$, Drucker et al. [30] give a family of lower bound graphs with
$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \quad s(\alpha) = \Theta(\alpha^{1/2}), \quad C(\alpha) = \Theta(\alpha^{1/2}) .$$
The lower bound given by Corollary 8.4 is $\Omega(\alpha^{1/2}/\log \alpha)$ for $\alpha = O(n^2)$.

– For $2k$-cycle detection for $k \geq 3$, Korhonen and Rybicki [54] give a family of lower bound graphs with
$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \quad s(\alpha) = \Theta(\alpha), \quad C(\alpha) = \Theta(\alpha^{1/2}) .$$
The lower bound given by Corollary 8.4 is $\Omega(\alpha^{1/2}/\log \alpha)$ for $\alpha = O(n)$.

Diameter and all-pairs shortest paths. In diameter computation, the input labelling $\ell\colon V \to \{0,1\}$ defines a subgraph H of G, and each node has to output the diameter of their connected component in H. Again, diameter computation has the extension property with label 0. For exact and approximate diameter computation, we use the sparse lower bound constructions of Abboud et al. [1]:

– For distinguishing between graphs of diameter 4 and 5, there is a family of lower bound graphs with parameters
$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \quad s(\alpha) = \Theta(\alpha), \quad C(\alpha) = \Theta(\log \alpha) .$$
The lower bound given by Corollary 8.4 is $\Omega(\alpha/\log^2 \alpha)$ for $\alpha = O(n)$. This implies a lower bound for exact diameter computation.

– For distinguishing between graphs of diameter $4k+2$ and $6k+1$, there is a family of lower bound graphs with parameters
$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \quad s(\alpha) = O(\alpha^{1+\delta}), \quad C(\alpha) = \Theta(\log \alpha),$$
for any constant $\delta > 0$. The lower bound given by Corollary 8.4 is $\Omega(\alpha/\log^2 \alpha)$ for $\alpha = O(n^{1/(1+\delta)})$ for any constant $\delta > 0$. This implies a lower bound for $(3/2-\varepsilon)$-approximation of diameter for any constant $\varepsilon > 0$.

A trivial $\Omega(D)$ lower bound holds even for $(3/2-\varepsilon)$-approximation in the worst case (e.g. a cycle).

In all-pairs shortest paths problem, the input labelling gives a weight $w(e) \in \{0,1,2,\dots,n^C\} \cup \{\infty\}$ for each edge $e \in E$, and each node node v has to output the distance $d(v,u)$ for each other node $u \in V \setminus \{v\}$. Exact or $(3/2-\varepsilon)$-approximate solution to all-pairs shortest paths can be used to recover exact or $(3/2-\varepsilon)$-approximate solution to diameter computation, respectively, in $O(D)$ rounds, so the lower bounds also apply to batch dynamic all-pairs shortest paths.

8.4 Lower bound for minimum spanning tree

The CONGEST lower bound for minimum spanning tree does not fall under the family of lower bound graphs construction used above; indeed, one can show that it is in fact impossible to prove CONGEST lower bounds for minimum spanning tree using a *fixed-cut simulation* (see Bacrach et al. [13]). However, we can adapt the more involved simulation argument of Das Sarma et al. [26] to obtain a near-linear lower bound for batch dynamic MST; note that $\Omega(D)$ lower bound holds trivially for the problem.

Again, we first prove a general version of the lower bound theorem first.

Theorem 8.5. *Let $L\colon \mathbb{N} \to \mathbb{N}$ be a function satisfying $L(\alpha) \geq \alpha^2$. Let $\mathcal{A}$ be a deterministic batch dynamic algorithm or a randomised batch dynamic algorithm as in Theorem 8.3 that solves MST in $T(\alpha,n) + O(D)$ rounds for all α satisfying $n \geq L(\alpha)$ on batch dynamic* CONGEST *with bandwidth $b(n)$. Then we have*
$$T(\alpha, L(\alpha)) = \Omega\left(\frac{\alpha}{b\big(L(\alpha)\big)\log \alpha}\right).$$

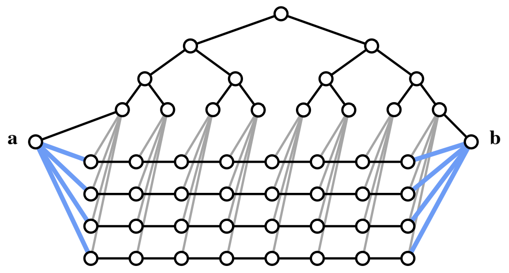

Fig. 3. Instance of the graph G_α for $\alpha = 8$ used in the minimum spanning tree lower bound. Black edges have weight 0, grey edges have weight 1 and blue edges are used to encode the set disjointness instance.

Proof. We follow the proof of Das Sarma et al. [26] with the same modifications to standard CONGEST lower bounds as in Theorem 8.3. We construct labelled graphs (G_α, ℓ_α) as follows, with ℓ_α encoding the edge weights of the graph:

- We start with two terminal nodes a and b.
- We add $\alpha/2$ paths $P_1, P_2, \dots, P_{\alpha/2}$ of length α, with all edges having weight 0. Each path P_i consists of nodes $p_{i,1}, p_{i,2}, \dots, p_{i,\alpha}$, and we refer to the set $\{p_{1,j}, p_{2,j}, \dots, p_{\alpha/2,j}\}$ as *column* j.
- We connect $p_{i,1}$ to a and and $p_{i,\alpha}$ to b for all i. These edges have weight 0.
- We add a balanced binary tree with α leaves, with all edges weight 0. We connect the first leaf to a with weight-0 edge, and the last leaf to b with weight-0 edge.
- We connect ith leaf of the tree to ith edge on each path P_j with weight-1 edge.
- Finally, we add new nodes connected by weight-0 edges to a to satisfy $n \geq L(\alpha)$; since we assume $L(\alpha) \geq \alpha^2$, this is always possible.

See Figure 3 for an example.

We now turn the algorithm $\mathcal{A}$ into a two-player protocol for solving $\mathsf{DISJ}_{\alpha/2}$. Given inputs $x_0, x_1 \in \{0,1\}^{\alpha/2}$, the players first construct (G_α, ℓ_α), and construct a global auxiliary state $x \in \xi(G_\alpha, \ell_\alpha)$; since both players know the (G_α, ℓ_α), they can do this locally. The players then locally change the labels according to the inputs x_0 and x_1:

- player 0 sets the weight on the edge from a to $p_{i,1}$ to weight $x_0(i)$ for $i = 1, 2, \dots, \alpha/2$, and
- player 1 sets the weight of the edge from b to $p_{i,\alpha}$ to $x_1(i)$ for $i = 1, 2, \dots, \alpha/2$.

This defines a new global labelling ℓ^*. The players now simulate the execution $\mathcal{A}(G_\alpha, \ell_\alpha, \ell^*, x)$ in a distributed manner; note in particular that players do not know the whole labelling ℓ^*.

We assume that $T \leq \alpha/2$, as otherwise we already $T > \alpha/2$ and we are happy. The simulation proceeds in steps $t = 1, 2, \dots, T$, where T is the running time of $\mathcal{A}$ on the instance.

(1) In step t of the iteration, player 0 simulates node a, columns 1 to $\alpha - t$, and the smallest subtree of the binary tree that includes children from 1 to $\alpha - t$. Dually, player 1 simulates node b, columns $i + t$ to α, and the smallest subtree of the binary tree that includes children from $t + 1$ to α.
(2) At the start of the simulation, both players know the local inputs of all the nodes they are simulating, since they are not simulating the nodes whose incident labels were changed by the other player.
(3) At step $t + 1$, players simulate one round of $\mathcal{A}$. We describe how player 0 does the simulation; player 1 acts in symmetrical way.

(a) Since the set of nodes player 0 simulates in round $t + 1$ is a subset of nodes simulated in step t, player 0 knows the full state of all the nodes it is simulating.
(b) For path nodes simulated by player 0, their neighbours were simulated in the previous round by player 0, so their incoming messages can be determined locally.
(c) For binary tree nodes, there can be neighbours that were not simulated in the previous round by player 0. However, since $T \leq \alpha/2$, these are simulated by player 1, and player 1 sends their outgoing messages to player 0. Since the height of the binary tree is $O(\log \alpha)$ and player 0 simulates a subtree of the binary tree, there are $O(\log \alpha)$ nodes that need to receive their neighbours' messages from player 1. Thus player 1 has to send $O(b(L(\alpha)) \log \alpha)$ bits to player 0 to complete one iteration of the simulation.

In total, the simulation of the execution of $\mathcal{A}(G_\alpha, \ell_\alpha, \ell^*, x)$ uses at most $CTb(L(\alpha)) \log \alpha$ bits of communication for constant C. One can verify that the minimum spanning tree in (G_α, ℓ^*) has weight 0 if x_0 and x_1 are disjoint, and weight at least 1 if they are not disjoint, so the players can determine the disjointness from the output of $\mathcal{A}$. For deterministic $\mathcal{A}$, this implies that $CTb(L(\alpha)) \log \alpha \geq \mathsf{CC}(\mathsf{DISJ}_{\alpha/2})$, and thus

$$T \geq \frac{\mathsf{CC}(\mathsf{DISJ}_{\alpha/2})}{Cb\big(L(\alpha)\big) \log \alpha} = \frac{C'\alpha}{b\big(L(\alpha)\big) \log \alpha}$$

for a constant C'. For randomised $\mathcal{A}$, we similarly get

$$T \geq \frac{\mathsf{RCC}(\mathsf{DISJ}_{\alpha/2})}{Cb\big(L(\alpha)\big) \log \alpha} = \frac{C'\alpha}{b\big(L(\alpha)\big) \log \alpha}$$

by the same argument as in the proof of Theorem 8.3. Finally, since the diameter of G_α is $O(\log n)$, we have that for sufficiently large α, we have $T(\alpha, L(\alpha)) \geq T/2$, and the claim follows. □

The general theorem implies the following simplified claim:

Corollary 8.6. *Let $\mathcal{A}$ be a deterministic batch dynamic algorithm or a randomised batch dynamic algorithm as in Corollary 8.4 that solves MST in $T(\alpha) + D$ rounds independent of n for all $\alpha \leq n^\varepsilon$ on batch dynamic* CONGEST *with bandwidth $\Theta(\log n)$, where $\varepsilon \leq 1/2$ is a constant. Then we have $T(\alpha) = \Omega(\alpha/\log^2 \alpha)$.*

9 BATCH DYNAMIC CONGESTED CLIQUE

If we set the communication graph $G = (V, E)$ to be a clique, we obtain a batch dynamic version of the *congested clique* [58] as a special case of our batch dynamic CONGEST model. This is in many ways similar to the batch dynamic versions of the k-machine and MPC models [27, 43, 52, 63]; however, whereas the these usually consider setting where the number of nodes k is much smaller than n, the setting with $k = n$ is qualitatively different. For example, a minimum spanning tree can be computed from scratch in $O(1)$ rounds in the congested clique [62], so recomputing from scratch is optimal also for input-dynamic algorithms.

In this section, we briefly discuss the batch dynamic congested clique, and in particular highlight *triangle counting* (and hence *triangle detection*) as an example of problem admitting a non-trivial batch dynamic algorithm in this setting.

9.1 Universal upper bound

First, we make the simple observation that the fully-connected communication topology gives faster universal upper bound than Theorem 5.2.

Theorem 9.1. *For any problem Π, there is a batch dynamic congested clique algorithm that runs in $O(\lceil \alpha/n \rceil)$ rounds and uses $O(m \log n)$ bits of auxiliary state.*

Proof. Use the same algorithm as in Theorem 5.2; the claim follows by observing that the message set M can be learned by all nodes in $O(\lceil \alpha/n \rceil)$ rounds using standard congested clique routing techniques [57]. □

9.2 Batch dynamic matrix multiplication and triangle detection

As an example of a problem that has non-trivial batch dynamic algorithms in congested clique, we consider the following *dynamic matrix multiplication* task. As input, we are given two $n \times n$ matrices S, T so that each node v receives row v of S and column v of T, and the task is to compute the product matrix $P = ST$ so that node v outputs row v of P. Concretely, we assume that the input label on edge $\{u, v\}$ the matrix entries $S[v, u]$, $S[u, v]$, $T[v, u]$ and $T[u, v]$. Note that in the dynamic version of the problem, the parameter α is an upper bound for changes to both matrices.

For matrix S, let *density* ρ_S of S be the smallest integer ρ such that the number of non-zero elements in S is less than ρn. We use the following result:

Theorem 9.2 ([21, 23]). *There is a congested clique algorithm that computes the product $P = ST$ in $O\big((\rho_S \rho_T)^{1/3}/n^{1/3} + 1\big)$ rounds.*

We use Theorem 9.2 to obtain a non-trivial dynamic batch algorithm for matrix multiplication. This in turn implies an upper bound for triangle counting by a standard reduction.

Theorem 9.3. *There is a batch dynamic algorithm for matrix multiplication in congested clique that runs in $O\big((\alpha/n)^{1/3} + 1\big)$ rounds and uses $O(n \log n)$ bits of auxiliary state.*

Proof. Consider input matrices S_1 and T_1 and updated input matrices S_2 and T_2. As auxiliary data $x(v)$, each node v keeps the row v of the matrix $P_1 = S_1 T_1$.

We can write

$$S_2 = S_1 + \Delta_S\,, \qquad\qquad T_2 = T_1 + \Delta_T\,,$$

where Δ_S and Δ_T are matrices with at most α non-zero elements, which implies their density is at most $\lceil \alpha/n \rceil$. Thus, we can write the product $P_2 = S_2 T_2$ as

$$\begin{aligned} P_2 &= (S_1 + \Delta_S)(T_1 + \Delta_T) \\ &= S_1 T_1 + \Delta_S T_1 + S_1 \Delta_T + \Delta_S \Delta_T \\ &= P_1 + \Delta_S T_1 + S_1 \Delta_T + \Delta_S \Delta_T\,. \end{aligned}$$

That is, it suffices to compute the products $\Delta_S T_1$, $S_1 \Delta_T$ and $\Delta_S \Delta_T$ to obtain P_2; by Theorem 9.2, this can be done in $O\big((\alpha/n)^{1/3} + 1\big)$ rounds. □

Corollary 9.4. *There is a batch dynamic algorithm for triangle counting in congested clique that runs in $O\big((\alpha/n)^{1/3} + 1\big)$ rounds and uses $O(n \log n)$ bits of auxiliary state.*

ACKNOWLEDGMENTS

We thank Jukka Suomela for discussions. We also thank our shepherd Mohammad Hajiesmaili and the reviewers for their time and suggestions on how to improve the paper. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML), from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska–Curie grant agreement No. 840605, from the Vienna Science and Technology Fund (WWTF) project WHATIF, ICT19-045, 2020-2024, and from the Austrian Science Fund (FWF) and netIDEE SCIENCE project P 33775-N.

REFERENCES

[1] Amir Abboud, Keren Censor-Hillel, and Seri Khoury. 2016. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *Proc. 30th International Symposium on Distributed Computing (DISC 2016)*. Springer, 29–42. https://doi.org/10.1007/978-3-662-53426-7_3

[2] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS*. 434–443. https://doi.org/10.1109/FOCS.2014.53

[3] Umut A. Acar, Vitaly Aksenov, and Sam Westrick. 2017. Brief Announcement: Parallel Dynamic Tree Contraction via Self-Adjusting Computation. In *Proc. 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2017)*. Association for Computing Machinery, New York, NY, USA, 275–277. https://doi.org/10.1145/3087556.3087595

[4] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. 2019. Parallel Batch-Dynamic Graph Connectivity. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2019)*. 381–392. https://doi.org/10.1145/3323165.3323196

[5] Umut A Acar, Andrew Cotter, Benoit Hudson, and Duru Türkoglu. 2011. Parallelism in dynamic well-spaced point sets. In *Proc. 23rd annual ACM symposium on Parallelism in algorithms and architectures (SPAA 2011)*. 33–42. https://doi.org/10.1145/1989493.1989498

[6] Bertie Ancona, Monika Henzinger, Liam Roditty, Virginia Vassilevska Williams, and Nicole Wein. 2019. Algorithms and Hardness for Diameter in Dynamic Graphs. In *46th International Colloquium on Automata, Languages, and Programming, ICALP*. 13:1–13:14. https://doi.org/10.4230/LIPIcs.ICALP.2019.13

[7] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. 2018. Fully dynamic maximal independent set with sublinear update time. In *Proc. 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*. 815–826. https://doi.org/10.1145/3188745.3188922

[8] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. 2019. Fully Dynamic Maximal Independent Set with Sublinear in n Update Time. In *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1919–1936. https://doi.org/10.1137/1.9781611975482.116

[9] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. 2020. On the Complexity of Traffic Traces and Implications. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 1 (2020), 20:1–20:29. https://doi.org/10.1145/3379486

[10] Daniel Awduche, Angela Chiu, Anwar Elwalid, Indra Widjaja, and XiPeng Xiao. 2002. *Overview and principles of Internet traffic engineering*. Technical Report. RFC 3272.

[11] Baruch Awerbuch, Israel Cidon, and Shay Kutten. 2008. Optimal Maintenance of a Spanning Tree. *J. ACM* 55, 4 (Sept. 2008). https://doi.org/10.1145/1391289.1391292

[12] Baruch Awerbuch, Boaz Patt-Shamir, David Peleg, and Michael Saks. 1992. Adapting to asynchronous dynamic networks. In *Proc. 24th annual ACM symposium on Theory of computing*. 557–570. https://doi.org/10.1145/129712.129767

[13] Nir Bacrach, Keren Censor-Hillel, Michal Dory, Yuval Efron, Dean Leitersdorf, and Ami Paz. 2019. Hardness of distributed optimization. In *Proc. of the 2019 ACM Symposium on Principles of Distributed Computing (PODC 2019)*. 238–247. https://doi.org/10.1145/3293611.3331597

[14] Philipp Bamberger, Fabian Kuhn, and Yannic Maus. 2019. Local Distributed Algorithms in Highly Dynamic Networks. In *Proc. 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2019)*. https://doi.org/10.1109/IPDPS.2019.00015

[15] Leonid Barenboim and Michael Elkin. 2010. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. *Distributed Computing* 22, 5-6 (2010), 363–379.

[16] Leonid Barenboim, Michael Elkin, and Uri Goldenberg. 2018. Locally-Iterative Distributed $(\Delta + 1)$-Coloring below Szegedy-Vishwanathan Barrier, and Applications to Self-Stabilization and to Restricted-Bandwidth Models. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC 2018)*. 437–446. https://doi.org/10.1145/3212734.3212769

[17] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. 2019. A Deamortization Approach for Dynamic Spanner and Dynamic Maximal Matching. In *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*. 1899–1918. https://doi.org/10.1137/1.9781611975482.115

[18] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. 2018. Dynamic algorithms via the primal-dual method. *Inf. Comput.* 261, Part (2018), 219–239. https://doi.org/10.1016/j.ic.2018.02.005

[19] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. 2012. Time-varying graphs and dynamic networks. *IJPEDS* 27, 5 (2012), 387–408. https://doi.org/10.1080/17445760.2012.668546

[20] Keren Censor-Hillel, Neta Dafni, Victor I. Kolobov, Ami Paz, and Gregory Schwartzman. 2020. Fast Deterministic Algorithms for Highly-Dynamic Networks. *CoRR* abs/1901.04008 (2020). http://arxiv.org/abs/1901.04008

[21] Keren Censor-Hillel, Michal Dory, Janne H. Korhonen, and Dean Leitersdorf. 2019. Fast Approximate Shortest Paths in the Congested Clique. In *Proc. 38nd ACM Symposium on Principles of Distributed Computing (PODC 2019)*. 74–83. https://doi.org/10.1145/3293611.3331633

[22] Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. 2016. Optimal Dynamic Distributed MIS. In *Proc. 2016 ACM Symposium on Principles of Distributed Computing (PODC 2016)*. 217–226. https://doi.org/10.1145/2933057.2933083

[23] Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner. 2020. Sparse matrix multiplication and triangle listing in the Congested Clique model. *Theoretical Computer Science* 809 (2020), 45–60. https://doi.org/10.1016/j.tcs.2019.11.006

[24] Serafino Cicerone, Gabriele Di Stefano, Daniele Frigioni, and Umberto Nanni. 2003. A fully dynamic algorithm for distributed shortest paths. *Theoretical Computer Science* 297, 1 (2003), 83–102. https://doi.org/10.1016/S0304-3975(02)00619-9

[25] Artur Czumaj and Christian Konrad. 2019. Detecting cliques in CONGEST networks. *Distributed Computing* (2019). https://doi.org/10.1007/s00446-019-00368-w

[26] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. 2012. Distributed Verification and Hardness of Distributed Approximation. *SIAM J. Comput.* 41 (2012), 1235–1265. https://doi.org/10.1137/11085178X

[27] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. 2020. Parallel Batch-Dynamic Graphs: Algorithms and Lower Bounds. In *Proc. 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA*. 1300–1319. https://doi.org/10.1137/1.9781611975994.79

[28] Edsger W. Dijkstra. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11 (1974), 643–644. https://doi.org/10.1145/361179.361202

[29] Shlomi Dolev. 2000. *Self-Stabilization.* Cambridge, MA.

[30] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. 2014. On the Power of the Congested Clique Model. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing (PODC 2014)*. 367–376. https://doi.org/10.1145/2611462.2611493

[31] Yuhao Du and Hengjie Zhang. 2018. Improved Algorithms for Fully Dynamic Maximal Independent Set. *CoRR* abs/1804.08908 (2018). http://arxiv.org/abs/1804.08908

[32] David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. 2019. Fully dynamic spectral vertex sparsifiers and applications. In *Proc. 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC*. 914–925. https://doi.org/10.1145/3313276.3316379

[33] Michael Elkin. 2007. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *Proc. 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 185–194. https://doi.org/10.1145/1281100.1281128

[34] Nick Feamster and Jennifer Rexford. 2017. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583* (2017).

[35] Nick Feamster, Jennifer Rexford, and Walter Willinger (Eds.). 2018. *Proceedings of the Afternoon Workshop on Self-Driving Networks, SelfDN@SIGCOMM 2018, Budapest, Hungary, August 24, 2018.* ACM. http://dl.acm.org/citation.cfm?id=3229584

[36] Klaus-Tycho Foerster, Juho Hirvonen, Jukka Suomela, and Stefan Schmid. 2019. On the Power of Preprocessing in Decentralized Network Optimization. In *Proc. IEEE Conference on Computer Communications (INFOCOM 2019)*. https://doi.org/10.1109/INFOCOM.2019.8737382

[37] Klaus-Tycho Foerster, Janne H. Korhonen, Joel Rybicki, and Stefan Schmid. 2019. Does Preprocessing Help under Congestion?. In *Proc. 38nd ACM Symposium on Principles of Distributed Computing, (PODC 2019)*. 259–261. https://doi.org/10.1145/3293611.3331581

[38] Klaus-Tycho Foerster, Thomas Luedi, Jochen Seidel, and Roger Wattenhofer. 2018. Local checkability, no strings attached: (A)cyclicity, reachability, loop free updates in SDNs. *Theoretical Computer Science* 709 (2018), 48–63. https://doi.org/10.1016/j.tcs.2016.11.018

[39] Klaus-Tycho Foerster, Oliver Richter, Jochen Seidel, and Roger Wattenhofer. 2017. Local Checkability in Dynamic Networks. In *Proc. of the 18th International Conference on Distributed Computing and Networking (ICDCN)*. ACM, 4:1–10. https://doi.org/10.1145/3007748.3007779

[40] Klaus-Tycho Foerster and Stefan Schmid. 2019. Distributed Consistent Network Updates in SDNs: Local Verification for Global Guarantees. In *18th IEEE International Symposium on Network Computing and Applications NCA*. IEEE, 1–4. https://doi.org/10.1109/NCA.2019.8935035

[41] Bernard Fortz and Mikkel Thorup. 2000. Internet traffic engineering by optimizing OSPF weights. In *Proc. IEEE INFOCOM*, Vol. 2. IEEE, 519–528. https://doi.org/10.1109/INFCOM.2000.832225

[42] Benjamin Frank, Ingmar Poese, Yin Lin, Georgios Smaragdakis, Anja Feldmann, Bruce Maggs, Jannis Rake, Steve Uhlig, and Rick Weber. 2013. Pushing CDN-ISP collaboration to the limit. *ACM SIGCOMM Computer Communication Review* 43, 3 (2013), 34–44. https://doi.org/10.1145/2500098.2500103

[43] Seth Gilbert and Lawrence Li. 2020. How Fast Can You Update Your MST?. In *Proc. 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 531—533. https://doi.org/10.1145/3350755.3400240

[44] Gramoz Goranci, Monika Henzinger, and Pan Peng. 2017. The Power of Vertex Sparsifiers in Dynamic Graph Algorithms. In *25th Annual European Symposium on Algorithms, ESA*. 45:1–45:14. https://doi.org/10.4230/LIPIcs.ESA.2017.45

[45] Manoj Gupta and Shahbaz Khan. 2018. Simple dynamic algorithms for Maximal Independent Set and other problems. *CoRR* abs/1804.01823 (2018). http://arxiv.org/abs/1804.01823

[46] Monika Henzinger. 2018. The State of the Art in Dynamic Graph Algorithms. In *SOFSEM 2018: Theory and Practice of Computer Science - 44th International Conference on Current Trends in Theory and Practice of Computer Science.* 40–44. https://doi.org/10.1007/978-3-319-73117-9_3

[47] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2016. Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization. *SIAM J. Comput.* 45, 3 (2016), 947–1006. https://doi.org/10.1137/140957299

[48] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *Proc. 47th Annual ACM on Symposium on Theory of Computing, STOC.* 21–30. https://doi.org/10.1145/2746539.2746609

[49] Monika Rauch Henzinger and Valerie King. 1999. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *J. ACM* 46, 4 (1999), 502–516. https://doi.org/10.1145/320211.320215

[50] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM* 48, 4 (2001), 723–760. https://doi.org/10.1145/502090.502095

[51] Giuseppe F. Italiano. 1991. Distributed Algorithms for Updating Shortest Paths. In *Proc. 5th International Workshop on Distributed Algorithms (WDAG).* 200–211. https://doi.org/10.1007/BFb0022448

[52] Giuseppe F. Italiano, Silvio Lattanzi, Vahab S. Mirrokni, and Nikos Parotsidis. 2019. Dynamic Algorithms for the Massively Parallel Computation Model. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2019).* Association for Computing Machinery, New York, NY, USA, 49–58. https://doi.org/10.1145/3323165.3323202

[53] Michael König and Roger Wattenhofer. 2013. On Local Fixing. In *Proc. 17th International Conference on Principles of Distributed Systems (OPODIS 2013).* 191–205. https://doi.org/10.1007/978-3-319-03850-6_14

[54] Janne H. Korhonen and Joel Rybicki. 2017. Deterministic Subgraph Detection in Broadcast CONGEST. In *Proc. 21st International Conference on Principles of Distributed Systems (OPODIS 2017).* https://doi.org/10.4230/LIPIcs.OPODIS.2017.4

[55] Fabian Kuhn, Nancy A. Lynch, and Rotem Oshman. 2010. Distributed computation in dynamic networks. In *Proc. 42nd ACM Symposium on Theory of Computing, STOC.* 513–522. https://doi.org/10.1145/1806689.1806760

[56] Eyal Kushilevitz and Noam Nisan. 1997. *Communication complexity.* Cambridge University Press.

[57] Christoph Lenzen. 2013. Optimal deterministic routing and sorting on the congested clique. In *Proc. 2013 ACM symposium on Principles of distributed computing (PODC 2013).* 42–50. https://doi.org/10.1145/2484239.2501983

[58] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. 2005. Minimum-Weight Spanning Tree Construction in $O(\log \log n)$ Communication Rounds. *SIAM J. Comput.* 35, 1 (2005), 120–131. https://doi.org/10.1137/S0097539704441848

[59] Oliver Michel and Eric Keller. 2017. SDN in wide-area networks: A survey. In *Proc. 4th International Conference on Software Defined Systems, (SDS 2017).* IEEE, 37–42. https://doi.org/10.1109/SDS.2017.7939138

[60] Ofer Neiman and Shay Solomon. 2016. Simple Deterministic Algorithms for Fully Dynamic Maximal Matching. *ACM Trans. Algorithms* 12, 1 (2016), 7:1–7:15. https://doi.org/10.1145/2700206

[61] Juniper Networks. 2020. Expel complexity with a Self-Driving Network. https://www.juniper.net/us/en/dm/the-self-driving-network/

[62] Krzysztof Nowicki. 2019. A Deterministic Algorithm for the MST Problem in Constant Rounds of Congested Clique. http://arxiv.org/abs/1912.04239 arXiv:1912.04239 [cs.DS].

[63] Krzysztof Nowicki and Krzysztof Onak. 2020. Dynamic Graph Algorithms with Batch Updates in the Massively Parallel Computation Model. *CoRR* abs/2002.07800 (2020). https://arxiv.org/abs/2002.07800

[64] Regina O'Dell and Roger Wattenhofer. 2005. Information dissemination in highly dynamic graphs. In *Proc. DIALM-POMC Joint Workshop on Foundations of Mobile Computing*, Suman Banerjee and Samrat Ganguly (Eds.). 104–110. https://doi.org/10.1145/1080810.1080828

[65] Merav Parter, David Peleg, and Shay Solomon. 2016. Local-on-Average Distributed Tasks. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* 220–239. https://doi.org/10.1137/1.9781611974331.ch17

[66] David Peleg. 1998. Distributed matroid basis completion via elimination upcast and distributed correction of minimum-weight spanning trees. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP 1998).* Springer, 164–175. https://doi.org/10.1007/BFb0055050

[67] David Peleg. 2000. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics.

[68] Radia J. Perlman. 1985. An algorithm for distributed computation of a spanning tree in an extended LAN. In *Proc. 9th Symposium on Data Communications (SIGCOMM)*, William Lidinsky and Bart W. Stuck (Eds.). ACM, 44–53.

https://doi.org/10.1145/319056.319004

[69] Larry L. Peterson and Bruce S. Davie. 2011. *Computer Networks, Fifth Edition: A Systems Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[70] Monika Rauch Henzinger and Mikkel Thorup. 1996. Improved sampling with applications to dynamic graph algorithms. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP 1998)*. Springer, 290–299.

[71] Alexander A. Razborov. 1992. On the Distributional Complexity of Disjointness. *Theor. Comput. Sci.* 106, 2 (1992), 385–390. https://doi.org/10.1016/0304-3975(92)90260-M

[72] Stefan Schmid and Jukka Suomela. 2013. Exploiting locality in distributed SDN control. In *Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN 2013)*. ACM Press, 121–126. https://doi.org/10.1145/2491185.2491198

[73] A. Schrijver. 2003. *Combinatorial Optimization - Polyhedra and Efficiency.* Springer.

[74] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2016. Work-efficient parallel union-find with applications to incremental graph connectivity. In *European Conference on Parallel Processing*. Springer, 561–573. https://doi.org/10.1007/978-3-319-43659-3_41

[75] Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. 2019. Batch-Parallel Euler Tour Trees. In *Proc. 21st Meeting on Algorithm Engineering and Experiments (ALENEX 2019)*. 92–106. https://doi.org/10.1137/1.9781611975499.8

[76] David Wang. 2020. Moving towards autonomous driving networks. https://www.huawei.com/en/publications/communicate/87/moving-towards-autonomous-driving-networks

[77] S. Wang, C. Wu, and C. Chou. 2015. Constructing an optimal spanning tree over a hybrid network with SDN and legacy switches. In *2015 IEEE Symposium on Computers and Communication (ISCC)*. 502–507. https://doi.org/10.1109/ISCC.2015.7405564

Received October 2020; revised December 2020; accepted January 2021