Input-dynamic distributed graph algorithms for congested networks

Klaus-Tycho Foerster · klaus-tycho.foerster@univie.ac.at · University of Vienna

Janne H. Korhonen · janne.korhonen@ist.ac.at · IST Austria

Ami Paz · ami.paz@univie.ac.at · University of Vienna

Joel Rybicki · joel.rybicki@ist.ac.at · IST Austria

Stefan Schmid · stefan_schmid@univie.ac.at · University of Vienna

Abstract. Consider a distributed system, where the topology of the *communication network* remains fixed, but *local inputs* given to nodes 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, where the communication network $G = (V, E)$ remains fixed 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.

We give a general picture of the complexity landscape in this model, including a general framework for lower bounds. In particular, we prove non-trivial upper bounds for two selected, contrasting problems: maintaining a minimum spanning tree and detecting cliques.

1 Introduction

Distributed systems are inherently dynamic. Especially in large and long-lived networks, change is inevitable: nodes may join and leave, new links may appear or existing links fail, new requests may increase congestion, and network properties such as link weights may be updated. As distributed systems often need to maintain data structures and other information related to the operation of the network, it is often essential 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 [8, 45]: 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 [14] 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 [8, 10, 11, 15, 17, 26, 37, 45]; and self-stabilisation considers recovery from arbitrary transient faults that may corrupt an existing solution [23, 24].

Input-dynamic distributed algorithms. Given the difficulty of 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 [38], 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.

While the input-dynamic distributed setting is of theoretical interest, it is also highly relevant in practice. 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 dynamic. For example, network operators perform link weight updates for dynamic traffic engineering [33] or to update spanning trees in local area networks [58, 62], content distribution providers modify cache assignments [34], and traffic patterns evolve over time [6, 34]. In all these cases, the input, at the form of edge weights or communication demand, change over time, while the network topology is remains unchanged.

1.1 Batch dynamic CONGEST model

To model input-dynamic distributed graph algorithms, we introduce the *batch dynamic* CONGEST model. In brief, the model is a dynamic variant of the standard CONGEST model with the following characteristics:

(1) The communication network is represented by a static graph $G = (V, E)$. The nodes can communicate with each other over the edges, with $O(\log n)$ bandwidth per round.

(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

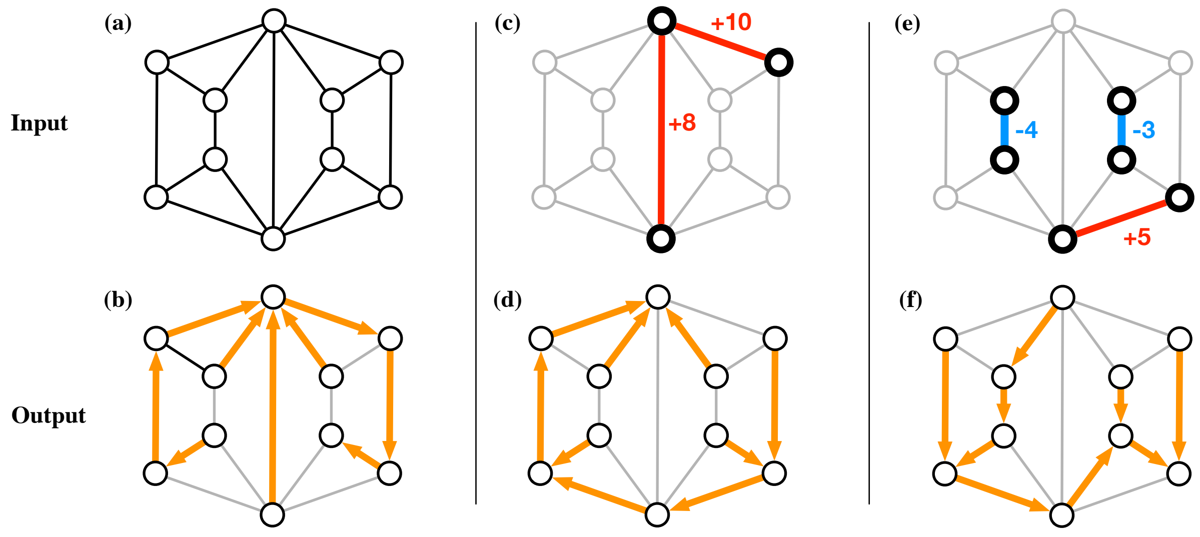

Figure 1: Examples of input-dynamic minimum-weight spanning tree. (a) The underlying communication graph, with all edges starting with weight 1. (b) A feasible minimum-weight spanning tree. (c) A batch of two edge weight increments. (d) Solution to the new input labelling. (e) A new batch of three changes: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

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 in a simultaneously, and the nodes should react to these changes. The nodes may maintain *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 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. A natural starting point for studying communication-efficient solutions for graph problems in networks is the CONGEST model, 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 in the next time the algorithm starts. Indeed, this parallels centralised dynamic algorithms, where the processing of changes is not interfered 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 [57] 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 [22, 35, 53].

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 7 for details.

	Upper bound		Lower bound	
Problem	Time	Space	Time	Ref.
any problem	$O(\alpha + D)$	$O(m \log n)$	—	§4
any LOCAL(1) problem	$O(\alpha)$	$O(m \log n)$	—	§4
minimum spanning tree	$O(\alpha + D)$	$O(\log n)$	$\Omega(\alpha/\log^2 \alpha + D)$	§5, §7.4
k-clique	$O(\alpha^{1/2})$	$O(m \log n)$	$\Omega(\alpha^{1/4}/\log \alpha)$	§6, §7.3
4-cycle	$O(\alpha)^\dagger$	$O(m \log n)^\dagger$	$\Omega(\alpha^{2/3}/\log \alpha)$	§7.3
k-cycle, $k \geq 5$	$O(\alpha)^\dagger$	$O(m \log n)^\dagger$	$\Omega(\alpha^{1/2}/\log \alpha)$	§7.3
diameter, $(3/2 - \varepsilon)$-apx.	$O(\alpha + D)^\dagger$	$O(m \log n)^\dagger$	$\Omega(\alpha/\log^2 \alpha + D)$	§7.3
APSP, $(3/2 - \varepsilon)$-apx.	$O(\alpha + D)^\dagger$	$O(m \log n)^\dagger$	$\Omega(\alpha/\log^2 \alpha + D)$	§7.3

1.2 Contributions

In this work, we focus on the following 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 needs 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.

Universal upper bounds. We show 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 LOCAL – 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 4.)

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 for storing the auxiliary state; this improves the storage requirements of a previous distributed dynamic algorithm of Peleg [57], 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 5.)

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.)

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. (Section 7.)

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 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 8.)

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 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. Indeed, many related subfields have recently emerged as vibrant research areas of their own. Thus, we settle on highlighting the key differences and similarities between the questions studied in these areas and our work.

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 [38]. This area focuses on designing data structures that and 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 [41, 42]. Later work has investigated efficient techniques for maintaining other graph structures, such as spanners [12], emulators [40], matchings [51], maximal independent sets [4, 5]; approximate vertex covers, electrical flows and shortest paths [13, 27, 36]. Recently, many conditional hardness results have been established in the centralised setting [1, 3, 39].

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 are arrive sequentially and 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.

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 [7]. Later, it was investigated how to maintain or recompute local [55] and global [28] 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 [4, 5, 10, 15, 17, 26, 29, 30, 37, 45]. 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 [14, 47, 54].

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.

Input-dynamic distributed algorithms. Several instance of distributed dynamic algorithms can be seen as examples of the input-dynamic approach. Italiano [43] and later Cicerone et al. [19], considered the problem of maintaining a solution all-pairs shortest paths problem when a single edge weight may change at a time. Peleg [57] 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 massively parallel large-scale systems [22, 35, 44, 53]. In these models, the communication is assumed to be fixed and fully-connected, but 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 the massively 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.

Self-stabilisation. The vibrant area of self-stabilisation [23, 24] 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.

Supported models. Similar in spirit to our model is the *supported* CONGEST model, a variant of the CONGEST model designed for software-defined networks [60]. 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 [46] studied the complexity of subgraph detection problems in *supported* CONGEST. Later, somewhat surprisingly, Foerster et al. [32] 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 [31].

3 Batch dynamic CONGEST

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

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. 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 will only affect constant factors in the running times of the algorithms we obtain.

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 $s\colon V \to \Gamma$. We assume that input labels can be encoded using $O(\log n)$ bits, and that the set $\Pi(G, \ell)$ is finite and computable.

In particular, we will consider the two 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 can be much larger than the diameter D of the communication graph, but we still want the running times of our algorithms to only depend on D.

– *Weighted graph problems:* The input label set $\Sigma = \{0, 1, 2, \dots, n^C\}$ for a constant C, that is, 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 label ∞.

Dynamic batch 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 input labellings, each node can store auxiliary information about previous labelling ℓ_1 and use it in 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 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 $x_1 \in \xi(G, \ell_1)$, and

  (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 encode any auxiliary state $x(v)$ over $x \in \xi(G, \ell)$.

Remark 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.

Remark 2. We only consider deterministic algorithms in batch dynamic CONGEST. Properly defining randomised algorithms for the model involves some subtle details, as we would ideally want algorithms that can handle an arbitrarily long sequence of patch updates, and simply requiring algorithms to compute the new auxiliary state correctly with high probability does not suffice. Possible solutions include considering Las Vegas, or requiring the algorithm to produce a locally checkable proof for the correctness of the new auxiliary states.

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.

4 Universal upper bounds

As a warmup, we show 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 ℓ_1 as the auxiliary state and broadcasts all changes to compute ℓ_2. We recall some useful primitives that follow from standard techniques [56].

Lemma 3. *In the* CONGEST *model:*

(a) A rooted spanning tree T of diameter D of the communication graph G can be found 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.

Theorem 4. *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 3a, 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. □

We now 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. *For any* LOCAL(1) *problem* Π*, 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

$$M_{v,1} = \{(u, v, \ell_2(\{u, v\})) : u \in N^+(v) \text{ and } \{u, v\} \in \dot{E}\}\,.$$

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. □

5 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 $w(e) \in \{0, 1, 2, \ldots, n^C\} \cup \{\infty\}$ 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. In the following, we first describe how to implement a distributed variant of this structure and then how to use it in conjunction with a the *minimum-weight matroid basis* algorithm of Peleg [57] to compute the minimum spanning tree in the dynamic batch CONGEST model.

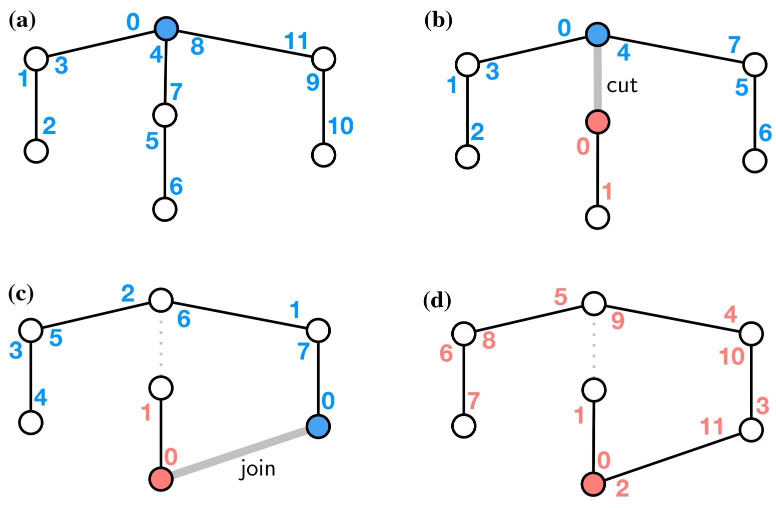

Figure 2: Eulerian tour trees. (a) An example of Eulerian tour labelling, with root node marked in blue. (b) The updated Eulerian tour labellings after applying a cut operation. The roots of the new blue and red trees are marked with respective colours. (c) To apply a join operation, we root the red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

5.1 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 F interchangeably with the edge set $F \subseteq E$. We further abuse the notation by taking a subtree T of G to mean a directed subgraph of G that contains all edges corresponding an undirected tree on G, and use $|T|$ to denote the number of directed edges in T.

Let H be some subgraph of G. A bijection $\tau \colon E \to \{0, \ldots, |E| - 1\}$ is an *Eulerian tour labelling* of the subgraph H if $(e_1, \ldots, e_m)$, where $\tau(e_i) = i$, is an Eulerian tour of H. See Figure 2(a) for an example. We say that u is the root of τ if there is some edge (u, v) such that $\tau(u, v) = 0$.

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

(1) $\mathcal{F} = \{T_1, \ldots, T_h\}$ is a spanning forest of G,

(2) $L \colon E \to \{0, \ldots |E| - 1\} \cup \{\infty\}$ 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) $r \colon V \to V$ is a mapping such that for each $T \in \mathcal{F}$ and node $v \in V(T)$, we have that $s(v)$ is the root of the Eulerian tour labelling $L \restriction T$.

(4) $s \colon V \to \mathbb{N}$ is a mapping satisfying $s(v) = |T|$ for each $T \in \mathcal{F}$ and a node $v \in V(T)$.

(5) $a \colon V \to \{0, \ldots |E| - 1\}$ 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 next define and implement 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 will then be 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 three primitive operations for manipulating $\mathcal{L}$. For brevity, let $T(u)$ to denote the tree node u belongs to in the Eulerian tour forest. The operations are as follows:

– $\mathsf{root}(\mathcal{L}, u)$: Node u becomes the root of the tree $T(u)$.

Implementation: Set

$$\begin{aligned} L(u,v) &\leftarrow L(u,v) - a(u) \bmod s(u) && \text{for each } (u,v) \in T(u), \text{ and} \\ a(v) &\leftarrow a(v) - a(u) \bmod s(u) && \text{for each } v \in V(T(u)). \end{aligned}$$

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.

– $\mathsf{join}(\mathcal{L}, e)$: 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 $L(v_i, v_j) \leftarrow s(v_i)$ and $L(v_j, v_i) \leftarrow s(v_i) + s(v_j) + 1$.
For each $(u,v) \in T_j$, set $L(u,v) \leftarrow L(u,v) + s(v_i) + 1$.

(3) For each $u \in V(T_j)$, set $a(u) \leftarrow a(u) + s(v_i) + 1$.

(4) For each $u \in T'$, set $s(u) \leftarrow s(v_i) + s(v_j) + 2$ and $r(u) \leftarrow v_i$.

– $\mathsf{cut}(\mathcal{L}, e)$: 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 $L(v_1, v_2) \leftarrow \infty$ and $L(v_2, v_1) \leftarrow \infty$.

(2) If $a(v) \in [0, \dots, z_1]$, then set $s(v) \leftarrow s(v) - x - 1$ and $a(v) \leftarrow a(v)$.
If $a(v) \in (z_1, \dots, z_2]$, then set $s(v) \leftarrow x - 1$ and $a(v) \leftarrow a(v) - z_1 - 1$.
Otherwise, set $s(v) \leftarrow s(v) - x - 1$ and $a(v) \leftarrow a(v) - x - 1$.

(3) If $L(u,v) \in [0, \dots, z_1)$, then set $L(u,v) \leftarrow L(u,v)$.
If $L(u,v) \in (z_1, \dots, z_2)$, then set $L(u,v) \leftarrow L(u,v) - z_1 - 1$.
Otherwise, set $L(u,v) \leftarrow L(u,v) - x - 1$.

(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 6. *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 $r(\cdot)$. □

Lemma 7. *Let $\mathcal{L} = (L, r, s)$, $e \in E$, and $E' \subseteq E$. Suppose $\mathcal{L}' = \mathsf{join}(\mathcal{L}, e)$ or $\mathcal{L}' = \mathsf{cut}(\mathcal{L}, e)$. Then $\mathcal{L}' \restriction_{e \cup E'}$ depends only on $\mathcal{L} \restriction_{e \cup E'}$.*

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.

Clearly, 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 8. *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 can compute 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 and r, 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 $x_1(v)$. □

5.2 Maximum matroid basis algorithm

We use an algorithm of Peleg [57] as a subroutine for finding minimum and maximum weight *matroid bases* in distributed manner. We first recall the basic definitions of matroids.

Definition 9. *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 $x \in I_1$ 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 equivalent with finding a *minimum-weight* matroid basis.

Theorem 10 ([57])**.** *Distributed maximum matroid basis problem over $\mathcal{M}$ can be solved in $O(\alpha + D)$ rounds, where α is the size of bases of $\mathcal{M}$.*

5.3 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. In the following we assume that each *communicated* edge e is decorated with the tuple

$$M(e) = (\mathcal{L} \restriction_e, w_1(e), w_2(e)) .$$

We abuse the notation by writing $M(E) = \{M(e) \colon e \in E\}$.

Note that for any edge $e' \in E$, 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. Clearly, 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 spanning tree $\mathcal{B}$ of diameter $O(D)$ as a broadcast tree, given by Lemma 3. We say that a node v *propagates* a message in $\mathcal{B}$ when it sends a message towards its parent in $\mathcal{B}$. 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 proceeds as follows:

(1) Let $E^+ = \{e : w_2(e) > w_1(e)\}$ and $E^- = \{e : w_2(e) < w_1(e)\}$.

(2) Broadcast the sets $M(E^+)$ and $M(E^-)$ to all nodes using the broadcast tree $\mathcal{B}$.

(3) Solve the problem the graph G_1' obtained from G_1 by changing only the weights in E^+.

(4) Solve the problem on the graph G_2 obtained from G_1' by changing the weights in E^-.

Lemma 11. *Suppose the weights of the graph G are unique. Then the following hold:*

– *Cycle property: For any cycle C in G, 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.*

5.3.1 Handling weight increments

We now design an algorithm that works in the case $|E^+| \le \alpha$ and $E^- = \emptyset$. That is, the new input graph G_2 differs from G_1 by having only the edge weights 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 12. *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 11, the lightest edge f (with respect to w_2) in the cut $X \subseteq E$ between X and $V \setminus S$ 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,

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

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 now provide an algorithm that computes A^* in $O(\alpha + D)$ communication rounds. We assume that the auxiliary state encodes an Eulerian tour forest $\mathcal{L}$ on T_1^*.

Lemma 13. *Let A be the set of all edges that connect components of F in G_2, and let*

$$\mathcal{I} = \{I \subseteq A \colon F \cup I \text{ is acyclic on } G_2\}\,.$$

Then $\mathcal{M} = (A, \mathcal{I})$ is a matroid and the minimum-weight basis of $\mathcal{M}$ is A^*.

Proof. We have $\mathcal{M}$ is matroid, as it's the contraction of the graphical matroid on G (see e.g. [61, 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 $\mathcal{M}$. □

Lemma 14. *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 $X = \{e_1, e_2, \dots, e_k\}$. 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 $i \in \{1, \dots, k\}$ 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. $r(v) = r(u)$. If this is the case, then $F \cup \{e_1, e_2, \dots, e_i\}$ has a cycle, and node v outputs that X is not independent and halts.

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

(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 full 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 minimum matroid basis algorithm over $\mathcal{M}$ to compute A^*.

(4) Each node v locally computes $\mathcal{L}_1 \restriction_{E(v)}$ by applying cut on each edge in $E^+ \setminus A^*$ in lexicographical order, starting from $\mathcal{L} \restriction_{E(v)}$.

(5) Each node v locally computes $\mathcal{L}_2 \restriction_{E(v)}$ by applying join on each edge in $A^* \setminus E^+$ 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 15. *The above algorithm solves batch dynamic minimum-weight spanning trees under edge weight increments in $O(\alpha + D)$ rounds.*

Proof. By Lemma 8, Step (1) of the algorithm can be done in $O(1)$ rounds, and by Lemma 3, Step (2) can be done in $O(\alpha + D)$ rounds. Step (3) can be implemented in $O(\alpha + D)$ rounds by Theorem 10 and Lemma 13, and after Step (3) all nodes have learnt 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 will produce compatible auxiliary states in Step (6). □

5.3.2 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 16. *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 17. *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. [61, 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 18. *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 $E^- \setminus X = \{e_1, e_2, \ldots, e_k\}$.

(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 $i \in \{1, \ldots, k\}$ 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 full 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 cut 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 join 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 19. *The above algorithm solves batch dynamic minimum-weight spanning trees under edge weight decrements in $O(\alpha + D)$ rounds.*

Proof. By Lemma 8, Step (1) of the algorithm can be done in $O(1)$ rounds, and by Lemma 3, Step (2) can be done in $O(\alpha + D)$ rounds. Step (3) can be implemented in $O(\alpha + D)$ rounds by Theorem 10 and Lemma 18, and after Step (3) all nodes have learnt 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). $\square$

6 Batch dynamic algorithm for clique enumeration

In this section, 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 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 the upper bound result, Section 7 shows that even the easier problem *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 complexity — more than constant or $\operatorname{poly}\log \alpha$, and less than linear 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 $v \in V$ satisfies $\operatorname{outdeg}_\sigma(v) \leq d$,
(2) the orientation σ is acyclic.

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.

Lemma 20. *Let G be a d-degenerate graph with n nodes and m edges. Then*

(1) $d \leq \sqrt{2m}$
(2) $m \leq nd$.

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

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

which is a contradiction.

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

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

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

Lemma 21. *An $O(\sqrt{\alpha})$-orientation of $H = G[\dot{E}]$ 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 $|U| \leq 2\alpha$ and $\alpha \leq m$. 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 $d = 1, \ldots, \lceil \log m\rceil$, 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 20 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. $\square$

6.2 Dynamic batch 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 algorithm. The dynamic algorithm computes the new auxiliary state x_2 as follows:

(1) Each node v runs the $O(\alpha^{1/2})$-orientation algorithm on $G[\dot{E}]$ until all nodes in its 1-radius neighbourhood $N^+(v)$ have halted (and oriented all of their edges in $\dot{E}$).

(2) Let $\dot{E}_{\text{out}}(v) \subseteq \dot{E}$ be the set of outgoing edges of v. 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 the set $R(v)$ and the map $y'_v \colon E(G^+[v]) \to \{0, 1\}$ as

$$R(v) = \bigcup_{u \in N^+(v)} A(u) \quad \text{and} \quad y'_v(e) = \begin{cases} \ell_2(e) & \text{if } (e, \ell_2(e)) \in R(v) \\ y_v(e) & \text{otherwise,} \end{cases}$$

where y_v is the map encoded by the auxiliary state $x_1(v)$.

(4) Set the new auxiliary state to $x_2(v) = y'_v$.

Lemma 22. *Let $v \in V$ and $e \in G^+[v]$. It holds that $y'_v(e) = 1$ 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 $e \in H_2[v]$.

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 $e \in H_2[v] \setminus \dot{E}$. □

Lemma 23. *Each node v can compute $H_2[v]$ in $O(\alpha^{1/2})$ rounds.*

Proof. By Lemma 21, 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 $O(\alpha^{1/2} + \log^2 \alpha)$. □

Clearly, any k-clique node v is part of is contained in $H_2[v]$. Thus, node v can clearly enumerate all of its k-cliques by learning $H_2[v]$, and hence, we obtain the following result.

Theorem 24. *Let $k > 2$ be a constant. There exists an algorithm for k-clique enumeration problem in the batch dynamic* CONGEST *model that runs in $O(\alpha^{1/2})$ rounds.*

7 Lower bounds

In this section, we investigate lower bounds. We start with some necessary preliminaries in Section 7.1 on two-party communication complexity, followed by our lower bound framework in Section 7.2, which we instantiate in Section 7.3. We lastly go into more detail for the minimum spanning tree problem in Section 7.4, adapting arguments from Das Sarma et al. [21].

7.1 Preliminaries

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* $\mathsf{CC}(f)$ of a function f is the minimal complexity of a protocol for computing f. Similarly, the *randomised communication complexity* $\mathsf{RCC}(f)$ 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 $i \in \{1, 2, \dots, k\}$ either $x_0(i) = 0$ or $x_1(i) = 0$, and $\mathsf{DISJ}_k(x_0, x_1) = 0$ otherwise. It is known [48, 59] that

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

Note that while we are mainly interested in deterministic lower bounds, the lower bounds based on set disjointness also hold for batch dynamic algorithms that correctly update the auxiliary state with constant probability (and thus, also w.h.p.).

Lower bound families. For proving lower bounds for batch dynamic algorithms, we use the standard $\mathsf{CONGEST}$ lower bound framework of *lower bound families*. This allows us to translate existing $\mathsf{CONGEST}$ lower bound constructions to batch dynamic $\mathsf{CONGEST}$, as we will see later; however, we need a slightly different definition of lower bound families to account for our setting.

Definition 25. *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 if and only if $f(x_0, x_1) = 1$,
(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 of G 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 26. *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.*

7.2 Lower bound framework

We now present our lower bound framework, which we will instantiate in the next Section 7.3.

Theorem 27. *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 batch dynamic algorithm that solves Π in $T(\alpha, n)$ rounds for all α satisfying $n \geq L(\alpha)$ on batch dynamic* $\mathsf{CONGEST}$ *with bandwidth $b(n)$. Then we have*

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

Proof. We convert the algorithm $\mathcal{A}$ 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:
    (1) Both players construct a global auxiliary state $x \in \xi(G^*, \ell^*)$; since both players know (G^*, ℓ^*), they can do this locally.
    (2) 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 $2b\big(L(\alpha)\big)C(\alpha)T\big(\alpha, L(\alpha)\big) \geq \mathsf{CC}(f_\alpha)$, which implies

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

□

In practice, we use the following, simpler version of Theorem 27 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 28. *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 batch dynamic algorithm that solves Π in $T(\alpha)$ rounds independent of n for all $\alpha \leq n^\varepsilon$ on batch dynamic* $\mathsf{CONGEST}$ *with bandwidth $\Theta(\log n)$. Then we have*

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

7.3 Instantiations

Overview. 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 25 is a straightforward reparameterisation.

We point out the following subtleties regarding the lower bounds:

– The lower bounds in terms of α only work in a regime where α is sufficiently small compared to n, as can be seen in Corollary 28. 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.

– We do not explicitly consider the diameter D of the communication graph in the lower bound, as all constructions we use have constant diameter. However, for global problems, we also have a trivial $\Omega(D)$ lower bound.

Cycle detection. First we consider k-cycle detection for fixed k: the input labelling $\ell\colon V \to \{0,1\}$ 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. [25] give a family of lower bound graphs with parameters

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

The lower bound given by Corollary 28 is $\Omega(\alpha^{1/3}/\log\alpha)$ for $\alpha = O(n^{3/2})$.

– For $(2k+1)$-cycle detection for $k \geq 2$, the same work [25] gives a family of lower bound graphs with parameters

$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \qquad s(\alpha) = \Theta(\alpha^{1/2}), \qquad C(\alpha) = \Theta(\alpha^{1/2})\,.$$

The lower bound given by Corollary 28 is $\Omega(\alpha^{1/2}/\log\alpha)$ for $\alpha = O(n^2)$.

– For $2k$-cycle detection for $k \geq 3$, Korhonen and Rybicki [46] give a family of lower bound graphs with parameters

$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \qquad s(\alpha) = \Theta(\alpha), \qquad C(\alpha) = \Theta(\alpha^{1/2})\,.$$

The lower bound given by Corollary 28 is $\Omega(\alpha^{1/2}/\log\alpha)$ for $\alpha = O(n)$.

Clique detection. In k-clique detection for fixed k, the input labelling $\ell\colon V \to \{0,1\}$ 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 clearly has the extension property with label 0.

– *Lower bound family.* For fixed $k \geq 4$, Czumaj and Konrad [20] give a family of lower bound graphs with parameters

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

The lower bound given by Corollary 28 is $\Omega(\alpha^{1/4}/\log\alpha)$.

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. [2]:

- For distinguishing between graphs of diameter 4 and 5, there is a family of lower bound graphs with parameters

$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \qquad s(\alpha) = \Theta(\alpha), \qquad C(\alpha) = \Theta(\log \alpha)\,.$$

  The lower bound given by Corollary 28 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

$$f_\alpha = \mathsf{DISJ}_{\Theta(\alpha)}, \qquad s(\alpha) = O(\alpha^{1+\delta}), \qquad C(\alpha) = \Theta(\log \alpha)\,,$$

  for any constant $\delta > 0$. The lower bound given by Corollary 28 is $\Omega(\alpha/\log^2\alpha)$ for $\alpha = O(n^{1-\delta})$ for any constant $\delta > 0$. This implies a lower bound for $(3/2-\varepsilon)$-approximation of diameter for any constant $\varepsilon > 0$.

In all-pairs shortest paths problem, 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.

7.4 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. [9]). However, we can adapt the more involved simulation argument of Das Sarma et al. [21] to obtain a near-linear lower bound for batch dynamic MST.

Theorem 29. *Let $L \colon \mathbb{N} \to \mathbb{N}$ be a function satisfying $L(\alpha) \geq \alpha^2$. Let $\mathcal{A}$ be a batch dynamic algorithm 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*

$$T(\alpha, L(\alpha)) = \Omega\left(\frac{\alpha}{b\big(L(\alpha)\big)\log\alpha}\right).$$

Proof. We follow the proof of Das Sarma et al. [21] with the same modifications to standard CONGEST lower bounds as in Theorem 27. 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.

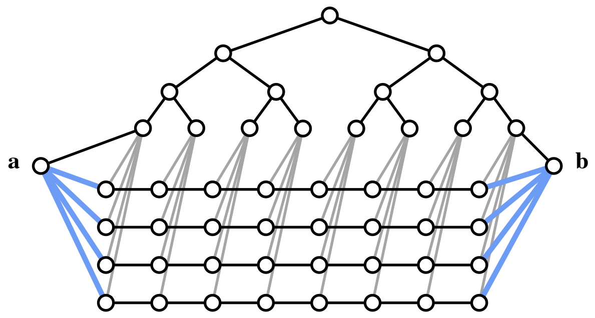

Figure 3: Instance of the graph G_α for $\alpha = 8$ used in the minimum spanning tree lower bound. Black edges have weight 0, grey edges have weight 1 and blue edges are used to encode the set disjointness instance.

– 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 $i = 1, 2, \ldots, \alpha/2$, and
– player 1 sets the weight of the edge from b to $p_{i,\alpha}$ to $x_1(i)$ for $i = 1, 2, \ldots, \alpha/2$.

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 $t = 1, 2, \ldots, T$, 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.
    (1) 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.
    (2) 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.
    (3) 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 $\mathcal{A}$. 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'. 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. □

Corollary 30. *Let $\mathcal{A}$ be a batch dynamic algorithm 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 $T(\alpha) = \Omega(\alpha / \log^2 \alpha)$.*

8 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* [50] 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 [22, 35, 44, 53]; however, whereas the these usually consider setting where the number of node 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 [52], so recomputing from scratch is optimal also for dynamic algorithms.

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

Universal upper bound. First, we make the simple observation that the fully-connected communication topology gives faster universal upper bound that Theorem 4.

Theorem 31. *For any problem Π, there exists a batch dynamic congested clique algorithm that uses $O(\lceil \alpha/n \rceil)$ time and $O(m \log n)$ space.*

Proof. Use the same algorithm as in Theorem 4; 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 [49]. □

Matrix multiplication and batch dynamic triangle detection. As an example of a problem that has non-trivial dynamic batch 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 32 ([16, 18])**.** *There is a* CLIQUE *algorithm that computes the product $P = ST$ in*

$$O\left(\frac{(\rho_S \rho_T)^{1/3}}{n^{1/3}} + 1\right)$$

rounds.

We now show how to use Theorem 32 to obtain a non-trivial dynamic batch algorithm for matrix multiplication.

Theorem 33. *There is a batch dynamic algorithm for matrix multiplication in* CLIQUE *that uses $O\big((\alpha/n)^{1/3} + 1\big)$ time and $O(n \log n)$ space.*

Proof sketch. 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 32, this can be done in $O\big((\alpha/n)^{1/3} + 1\big)$ rounds. ☐

By a standard reduction, this implies an upper bound for triangle counting:

Corollary 34. *There is a batch dynamic algorithm for triangle counting in congested clique that uses $O\big((\alpha/n)^{1/3} + 1\big)$ time and $O(n \log n)$ space.*

Acknowledgements

We thank Jukka Suomela for discussions. 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 Unions Horizon 2020 research and innovation programme under the Marie SkodowskaCurie grant agreement No. 840605, and from WWTF project WHATIF, ICT19-045, 2020-2024.

References

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 434–443, 2014. doi:10.1109/FOCS.2014.53.

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

[3] Bertie Ancona, Monika Henzinger, Liam Roditty, Virginia Vassilevska Williams, and Nicole Wein. Algorithms and hardness for diameter in dynamic graphs. In *46th International Colloquium on Automata, Languages, and Programming, ICALP*, pages 13:1–13:14, 2019. doi:10.4230/LIPIcs.ICALP.2019.13.

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

[5] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear in n update time. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1919–1936, 2019. doi:10.1137/1.9781611975482.116.

[6] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. On the complexity of traffic traces and implications. In *Proc. ACM SIGMETRICS*, 2020.

[7] Baruch Awerbuch, Boaz Patt-Shamir, David Peleg, and Michael Saks. Adapting to asynchronous dynamic networks. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 557–570, 1992. doi:10.1145/129712.129767.

[8] Baruch Awerbuch, Israel Cidon, and Shay Kutten. Optimal maintenance of a spanning tree. *Journal of the ACM*, 55(4), September 2008. doi:10.1145/1391289.1391292.

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

[10] Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Local distributed algorithms in highly dynamic networks. In *Proc. 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2019)*, 2019. doi:10.1109/IPDPS.2019.00015.

[11] Leonid Barenboim, Michael Elkin, and Uri Goldenberg. 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)*, pages 437–446, 2018. doi:10.1145/3212734.3212769.

[12] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1899–1918, 2019. doi:10.1137/1.9781611975482.115.

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

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

[15] Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. Optimal dynamic distributed MIS. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 217–226, 2016. doi:10.1145/2933057.2933083.

[16] Keren Censor-Hillel, Michal Dory, Janne H. Korhonen, and Dean Leitersdorf. Fast approximate shortest paths in the congested clique. In *Proc. 38nd ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 74–83, 2019. doi:10.1145/3293611.3331633.

[17] Keren Censor-Hillel, Neta Dafni, Victor I. Kolobov, Ami Paz, and Gregory Schwartzman. Fast deterministic algorithms for highly-dynamic networks. *CoRR*, abs/1901.04008, 2020. URL http://arxiv.org/abs/1901.04008.

[18] Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner. Sparse matrix multiplication and triangle listing in the congested clique model. *Theoretical Computer Science*, 809:45–60, 2020. doi:10.1016/j.tcs.2019.11.006.

[19] Serafino Cicerone, Gabriele Di Stefano, Daniele Frigioni, and Umberto Nanni. A fully dynamic algorithm for distributed shortest paths. *Theoretical Computer Science*, 297(1): 83–102, 2003. ISSN 0304-3975. doi:10.1016/S0304-3975(02)00619-9.

[20] Artur Czumaj and Christian Konrad. Detecting cliques in congest networks. *Distributed Computing*, 2019. doi:10.1007/s00446-019-00368-w.

[21] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41:1235–1265, 2012. doi:10.1137/11085178X.

[22] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1300–1319, 2020. doi:10.1137/1.9781611975994.79.

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

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

[25] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing (PODC 2014)*, pages 367–376, 2014. doi:10.1145/2611462.2611493.

[26] Yuhao Du and Hengjie Zhang. Improved algorithms for fully dynamic maximal independent set. *CoRR*, abs/1804.08908, 2018. URL http://arxiv.org/abs/1804.08908.

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

[28] Michael Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 185–194, 2007. doi:10.1145/1281100.1281128.

[29] Klaus-Tycho Foerster and Stefan Schmid. Distributed consistent network updates in sdns: Local verification for global guarantees. In *18th IEEE International Symposium on Network Computing and Applications NCA*, pages 1–4. IEEE, 2019. doi:10.1109/NCA.2019.8935035.

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

[31] Klaus-Tycho Foerster, Juho Hirvonen, Jukka Suomela, and Stefan Schmid. On the power of preprocessing in decentralized network optimization. In *Proc. IEEE Conference on Computer Communications (INFOCOM 2019)*, 2019. doi:10.1109/INFOCOM.2019.8737382.

[32] Klaus-Tycho Foerster, Janne H. Korhonen, Joel Rybicki, and Stefan Schmid. Does preprocessing help under congestion? In Peter Robinson and Faith Ellen, editors, *Proc. 38nd ACM Symposium on Principles of Distributed Computing, (PODC 2019)*, pages 259–261, 2019. doi:10.1145/3293611.3331581.

[33] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *Proc. IEEE INFOCOM*, volume 2, pages 519–528. IEEE, 2000. doi:10.1109/INFCOM.2000.832225.

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

[35] Seth Gilbert and Lawrence Li. How fast can you update your mst? (dynamic algorithms for cluster computing). *CoRR*, abs/2002.06762, 2020. URL https://arxiv.org/abs/2002.06762.

[36] Gramoz Goranci, Monika Henzinger, and Pan Peng. The power of vertex sparsifiers in dynamic graph algorithms. In *25th Annual European Symposium on Algorithms, ESA*, pages 45:1–45:14, 2017. doi:10.4230/LIPIcs.ESA.2017.45.

[37] Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set and other problems. *CoRR*, abs/1804.01823, 2018. URL http://arxiv.org/abs/1804.01823.

[38] Monika Henzinger. 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*, pages 40–44, 2018. doi:10.1007/978-3-319-73117-9_3.

[39] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC*, pages 21–30, 2015. doi:10.1145/2746539.2746609.

[40] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrier and derandomization. *SIAM J. Comput.*, 45(3):947–1006, 2016. doi:10.1137/140957299.

[41] Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999. doi:10.1145/320211.320215.

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

[43] Giuseppe F. Italiano. Distributed algorithms for updating shortest paths (extended abstract). In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG)*, pages 200–211, 1991. doi:10.1007/BFb0022448.

[44] Giuseppe F. Italiano, Silvio Lattanzi, Vahab S. Mirrokni, and Nikos Parotsidis. Dynamic algorithms for the massively parallel computation model. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2019)*, page 4958, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3323165.3323202.

[45] Michael König and Roger Wattenhofer. On local fixing. In *Proceedings of the 17th International Conference on Principles of Distributed Systems (OPODIS)*, pages 191–205, 2013. doi:10.1007/978-3-319-03850-6_14.

[46] Janne H. Korhonen and Joel Rybicki. Deterministic subgraph detection in broadcast CONGEST. In *Proc. 21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, 2017. doi:10.4230/LIPIcs.OPODIS.2017.4.

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

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

[49] Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 42–50, 2013. doi:10.1145/2484239.2501983.

[50] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $o(\log \log n)$ communication rounds. *SIAM Journal on Computing*, 35 (1):120–131, 2005. doi:10.1137/S0097539704441848.

[51] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, 2016. doi:10.1145/2700206.

[52] Krzysztof Nowicki. A deterministic algorithm for the MST problem in constant rounds of congested clique, 2019. URL http://arxiv.org/abs/1912.04239. arXiv:1912.04239 [cs.DS].

[53] Krzysztof Nowicki and Krzysztof Onak. Dynamic graph algorithms with batch updates in the massively parallel computation model. *CoRR*, abs/2002.07800, 2020. URL https://arxiv.org/abs/2002.07800.

[54] Regina O'Dell and Roger Wattenhofer. Information dissemination in highly dynamic graphs. In Suman Banerjee and Samrat Ganguly, editors, *Proceedings of the DIALM-POMC Joint Workshop on Foundations of Mobile Computing, Cologne, Germany, September 2, 2005*, pages 104–110. ACM, 2005. doi:10.1145/1080810.1080828.

[55] Merav Parter, David Peleg, and Shay Solomon. Local-on-average distributed tasks. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 220–239, 2016. doi:10.1137/1.9781611974331.ch17.

[56] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 2000. ISBN 9780898714647.

[57] David Peleg. 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)*, pages 164–175. Springer, 1998. doi:10.1007/BFb0055050.

[58] Radia J. Perlman. An algorithm for distributed computation of a spanning tree in an extended LAN. In William Lidinsky and Bart W. Stuck, editors, *Proceedings of the Ninth Symposium on Data Communications (SIGCOMM)*, pages 44–53. ACM, 1985. doi:10.1145/319056.319004.

[59] Alexander A. Razborov. On the distributional complexity of disjointness. *Theor. Comput. Sci.*, 106(2):385–390, 1992. doi:10.1016/0304-3975(92)90260-M.

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

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

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