Pruned Pivot: Correlation Clustering Algorithm for Dynamic, Parallel, and Local Computation Models

Mina Dalirrooyfard [*1] **Konstantin Makarychev** [*2] **Slobodan Mitrović** [*3]

Abstract

Given a graph with positive and negative edge labels, the correlation clustering problem aims to cluster the nodes so to minimize the total number of between-cluster positive and within-cluster negative edges. This problem has many applications in data mining, particularly in unsupervised learning. Inspired by the prevalence of large graphs and constantly changing data in modern applications, we study correlation clustering in dynamic, parallel (MPC), and local computation (LCA) settings. We design an approach that improves state-of-the-art runtime complexities in all these settings. In particular, we provide the first fully dynamic algorithm that runs in an expected amortized constant time, without any dependence on the graph size. Moreover, our algorithm essentially matches the approximation guarantee of the celebrated PIVOT algorithm.

1. Introduction

We study algorithms for the Correlation Clustering problem, which has many applications in Machine Learning and Data Mining (Bansal et al., 2004; Becker, 2005; Kalashnikov et al., 2008; Arasu et al., 2009; Firman et al., 2013; Bonchi et al., 2013; Li et al., 2017). Among the most prominent applications is clustering products into categories or detecting communities based on product co-purchasing (Wang et al., 2013; Veldt et al., 2020; Shi et al., 2021). In this problem, we are given a set of objects with "similar" or "dissimilar" labels between every pair of objects, and the goal is to cluster these objects such that similar objects are in the same cluster and dissimilar objects are in different clusters. Formally, given a complete graph with edge weights in $\mathbb{R}$, correlation clustering with the *minimum disagreement* objective asks to cluster the nodes such that the sum of the weights of positive edges between clusters plus the sum of the weights of negative edges inside clusters is minimized.[1] This paper focuses on the unweighted setting, where weights are in $\{-1, +1\}$.

Correlation Clustering is APX hard (Charikar et al., 2005). There has been a long line of work on approximation algorithms for correlation clustering; see e.g., Bansal et al. (2004); Charikar et al. (2005); Demaine et al. (2006); Chawla et al. (2015); Jafarov et al. (2021); Cohen-Addad et al. (2022); Behnezhad et al. (2022); Chakrabarty & Makarychev (2023); Cohen-Addad et al. (2023). The best known approximation factor is 1.437 due to Cao, Cohen-Addad, Lee, Li, Newman, and Vogl (2024).[2] However, all known algorithms with approximation factors less than 3 use linear programming (LP), which makes most of them impractical for dealing with massive data.

In their seminal work, Ailon, Charikar, and Newman (2008) introduced an elegant 3-approximation algorithm called PIVOT, which runs in linear time (time proportional to the number of positive *edges* in the graph). It is the algorithm of choice in practice. The algorithm has been adapted for various computational models including semi-streaming (Behnezhad et al., 2023; Cambus et al., 2022; Chakrabarty & Makarychev, 2023), parallel algorithms (MPC) (Cambus et al., 2022; Behnezhad et al., 2022), local computation algorithms (LCA) (Behnezhad et al., 2022), and dynamic algorithms (Behnezhad et al. (2022); see also Chechik & Zhang (2019)).

We study correlation clustering for massive and dynamic graphs. Such graphs are used to represent social networks (Tantipathananandh & Berger-Wolf, 2011; Hafiene

[*]Equal contribution [1]Morgan Stanley [2]Department of Computer Science, Northwestern University, Evanston, USA [3]Department of Computer Science, UC Davis, Davis, USA. Correspondence to: Mina Dalirrooyfard <mina.dalirrooyfard@morganstanley.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

[1]Correlation clustering has also been studied on weighted graphs and with other objectives such as *maximum agreement* or *minimum ℓ_p norm* (Bansal et al., 2004; Swamy, 2004; Charikar et al., 2005; Demaine et al., 2006; Puleo & Milenkovic, 2016; Charikar et al., 2017; Ahmadi et al., 2019; Jafarov et al., 2021; Kalhan et al., 2019).

[2]The proof of 1.437 approximation is computer-assisted, but the same publication also presents an analytical proof of 1.49 approximation.

et al., 2020), knowledge graphs (Fang et al., 2020; Yan et al., 2021), and user-product interactions (Ding et al., 2019). We design a PIVOT-like algorithm that can be easily implemented in the fully dynamic regime and LCA and MPC models. Our algorithm is inspired by the recent works by Behnezhad, Charikar, Ma, and Tan (2022) and Chakrabarty and Makarychev (2023). Behnezhad et al. (2022) presented a $(3+\varepsilon)$-approximate algorithm, called R-pivot, that runs in $O(\frac{1}{\varepsilon})$ MPC rounds, for any $\varepsilon > 0$. This algorithm can be implemented in LCA with $\Delta^{O(1/\varepsilon)}$-probe complexity, where Δ is the maximum node degree in the graph (consisting of positive edges). Chakrabarty & Makarychev (2023) give a $(3+\varepsilon)$-approximate semi-streaming algorithm that uses $O(n/\varepsilon)$ words of memory. Behnezhad et al. (2019) show how to maintain the lexicographically first maximal matching in a fully dynamic graph. Their result can be used to implement PIVOT in the fully dynamic setting. The expected update time for relabelling edges is $O(\log^2 n \log^2 \Delta)$ per operation. Chechik & Zhang (2019) provide a similar result for maximal matching with expected worst-case running time $O(\log^4 n)$ per update.

1.1. Our Contributions

In this paper, we provide a new variant of PIVOT, which we call PRUNED PIVOT, that gives a $(3+\varepsilon)$-approximation for Correlation Clustering (see Theorem 4.1). Our algorithm is local and parallelizable by design: Given a node u and common randomness, it returns the cluster of u after exploring only $O(1/\varepsilon)$ nodes of the entire graph. This makes it easy to implement PRUNED PIVOT in various computational models, including dynamic algorithms, MPC, and LCA.

Our first result is an efficient algorithm for dynamically maintaining a clustering. This is the first dynamic algorithm for Correlation Clustering, whose expected running time does not depend on the graph size.

Theorem 1.1 (Fully-dynamic correlation clustering)**.** *For any $\varepsilon > 0$, there is a data structure that maintains a $3+\varepsilon$ approximation of correlation clustering in a fully-dynamic setting with an oblivious adversary. The expected update time is $O(1/\varepsilon)$ per operation.*

Theorem 1.1 gives an almost 3 approximation fully dynamic algorithm with update time $O(1)$, which answers an open question posed by Behnezhad et al. (2022).

Theorem 1.2 (Correlation clustering in MPC)**.** *For any $\varepsilon > 0$, there is a randomized $O(\log \frac{1}{\varepsilon})$-round MPC algorithm that achieves a $3+\varepsilon$ approximation for Correlation Clustering. This bound holds even when each machine has a memory sublinear in the node-set size.*

The previously best-known MPC algorithm by Behnezhad et al. (2022) requires $O(1/\varepsilon)$ rounds. Hence, our approach improves the dependence on $1/\varepsilon$ exponentially.

Theorem 1.3 (Correlation clustering in LCA)**.** *For any $\varepsilon > 0$, there is a randomized $O(\Delta/\varepsilon)$-probe complexity local computation algorithm that achieves a $3+\varepsilon$ approximation of correlation clustering on graphs with maximum degree Δ.*

Theorem 1.3 gives an almost 3 approximation LCA in, essentially, $O(\Delta)$ probes, thus answering another question posed by Behnezhad, Charikar, Ma, and Tan (2022). The previously best-known algorithms for LCA were the algorithm by Behnezhad et al. (2022) giving $3+\varepsilon$ approximation in $\Delta^{O(1/\varepsilon)}$ probes, and the work by Behnezhad et al. (2023) providing a 5-approximate algorithm in $O(\text{poly}\log n)$ space and $O(\Delta \cdot \text{poly}\log n)$ probes.[3]

We provide empirical evaluations on synthetic graphs in Appendix A. They show that exploring only 4 nodes to obtain a node's clustering suffices for the cost of PRUNED PIVOT to be within 1% of the cost of PIVOT.

Moreover, in Appendix G we describe how to implement our algorithm in the CRCW PRAM model in $O(1/\varepsilon)$ rounds.

We finally note that our main technical result (Theorem 4.10) is of independent interest. A seminal work by Yoshida, Yamamoto, and Ito (2009) shows how to learn whether a randomly chosen node is in MIS using the average-degree LCA probes. Our work essentially recovers that claim using a significantly different analysis.

1.2. Comparison to prior work

Several closely related works have introduced variants of the PIVOT algorithm. In PIVOT, nodes are processed according to a predefined and random order, and each node queries its neighbors that have already been processed to determine its cluster (for a formal description of PIVOT see Section 3). Hence, to determine the cluster of each node, multiple "query paths" are made, and the collection of these query paths makes a "query tree". (For a more formal definition of query paths and query trees, refer to Section 4.1.)

First, given a parameter R, the R-PIVOT algorithm by Behnezhad et al. (2022) runs the PIVOT algorithm but only considers "query paths" of depth at most R. If, to obtain the cluster of a node, one needs to consider query paths of length more than R, then this node is put into a singleton cluster. Behnezhad et al. (2022) show that this algorithm has approximation factor $3 + O(\frac{1}{R})$.

Second, the semi-streaming algorithm by Chakrabarty & Makarychev (2023) is another modified version of PIVOT. In this approach, given a parameter R, every node only queries at most its R top-ranked neighbors when deciding on their cluster. This algorithm yields an approximation factor of $3 + O(\frac{1}{R})$.

[3] Our LCA algorithm uses $O(\Delta/\varepsilon \cdot \text{poly}\log n)$ space.

In PRUNED PIVOT, instead of only limiting the number of neighbors each node queries or the depth of the query paths, our algorithm limits *the total size* of the query tree. Our analysis of the algorithm uses a new approach to counting query and *expensive* paths, which is very different from the approach of Behnezhad et al. (2022).

Our main technical contribution is a proof that by limiting the total size of the query tree, the existence/non-existence of each edge only influences at most a constant number of other nodes. This crucial point allows us to achieve low probe complexity in LCA and MPC, and constant update time in the dynamic setting.

2. Preliminaries

An instance of the correlation clustering problem receives an unweighted graph $G = (V, E)$ on input. We consider E representing positive and $(V \times V) \setminus E$ representing negative labels between the nodes of V. This problem aims to cluster V to minimize the number of positive between-cluster and negative within-cluster labels. The neighbors of a node $u \in V$ are denoted by $N(u)$. We let $u \in N(u)$, i.e., u is a neighbor of itself. Next, we formally define the MPC and LCA models.

The MPC model. Massively Parallel Computation (MPC) is a theoretical model of real-world parallel computation such as MapReduce (Dean & Ghemawat, 2008). It was introduced in a sequence of works by Dean & Ghemawat (2008); Karloff et al. (2010); Goodrich et al. (2011). In MPC, computation is performed in synchronous rounds, where in each round every machine locally performs computation on the data that resides locally and then sends and receives messages to any other machine. Each machine has a memory of size S, and can send and receive messages of total size S. As the local computations frequently run in linear or near-linear time, they are ignored in the analysis of the complexity of the MPC model, and so the efficiency of an algorithm in this model is measured by the number of rounds it takes for the algorithm to terminate where the memory S plays a key role. We focus on the *sublinear memory* regime, where $S = n^\alpha$ for some constant $\alpha \in (0, 1)$.

The LCA model. Local Computation Algorithms (LCAs) were introduced by Rubinfeld et al. (2011) for tasks where the input and output are too large to be stored in the memory. An LCA is not required to output the entire solution but should answer queries about a part of the output by examining only a small portion of the input. In Correlation Clustering, the query is a node v, and the output is the cluster ID of v. Formally, an LCA A is given access to the adjacency list oracle for the input graph G, a tape of random bits, and local read-write computation memory. When given an input query x, A must compute an answer for x depending only on x, G and the random bits. The answers given by A to all possible queries must be consistent, meaning that they must constitute some valid solution to the computation problem.

We use *probe* to refer to accessing a node in an adjacency list. The LCA complexity of an algorithm is measured by the number of probes the algorithm makes per single query.

3. Recursive and Pruned Pivot

This section describes our variant of the PIVOT algorithm that we call PRUNED PIVOT. We remind the reader how the standard PIVOT algorithm works. First, it picks a random ordering $\pi : V \to \{1, \ldots, n\}$. We say that $\pi(u)$ is the rank of node u. If $\pi(u) < \pi(v)$, then u has a higher rank than v. Therefore, the node with rank 1 is the highest-ranked, and the node with rank n is the lowest-ranked node. The algorithm maintains a list of not yet clustered nodes. Initially, all nodes are not clustered. At every step, the algorithm picks the highest not yet clustered node, marks it as a pivot, and assigns itself and all its not yet clustered neighbors to a new cluster. The algorithm labels all nodes in this new cluster as *clustered* and proceeds to the next step. Each cluster created by the PIVOT algorithm contains a unique pivot node. We say that the cluster is represented by that pivot. If node u belongs to the cluster represented by pivot v, we say that u is assigned to pivot v.

To describe our variant of the PIVOT algorithm, we first rewrite the standard PIVOT as a recursive or top-down dynamic programming algorithm. The algorithm relies on the recursive function CLUSTER (see Algorithm 1). For a given node u and random permutation π, this function returns the pivot node to which u is assigned, along with a flag indicating if u is a pivot. Note that u is a pivot if and only if it is assigned to itself.

Algorithm 1 RECURSIVE PIVOT

```
function CLUSTER(u, π):
    Sort all neighbors of u (including u itself) by their rank π(v). Denote the sorted list by N_π.
    for all v in N_π:
        if v = u:
            return u belongs to the cluster of u; u is a pivot.
        CLUSTER(v, π)
        if v is a pivot:
            return u is in the cluster of v; u is not a pivot.
```

To reduce the running time, we can cache (memoize) the values returned by the function CLUSTER. We want to use this recursive function in our local computation algorithm (LCA). The problem is, however, that to cluster some nodes, the algorithm may need to make as many as $\Omega(n)$ calls to

CLUSTER (for instance, if node u is connected to all nodes in the left part of the complete bipartite graph $K_{n,n}$). That is why we propose a crucial change: execute only k recursive calls of PIVOT. If the status of the node is not determined by then, mark that node as *unlucky* and make it a singleton. The algorithm is given below.

Algorithm 2 PRUNED PIVOT

```
Initialize a global variable REC-CALLS to 0.
function PRUNED-CLUSTER (u,π):
  If REC-CALLS ≥ k:
    terminate this recursion
  Sort all neighbors of u (including u itself) by their rank π(v). Denote the sorted list by N_π.
  for all v in N_π:
    if v = u:
      return u belongs to the cluster of u; u is a pivot.
    REC-CALLS ← REC-CALLS + 1
    PRUNED-CLUSTER (v,π)
    if v is a pivot:
      return u is in the cluster of v; u is not a pivot.
```

The recursion tree for the modified PRUNED-CLUSTER function contains at most k edges. Consequently, if k is a constant, the running time of function PRUNED-CLUSTER is also constant. We show how to implement this algorithm as a Local Computation (LCA), Massively Parallel Computation (MPC), and Dynamic Graph Algorithm. In the next section, we prove that the approximation factor of PRUNED PIVOT is $3 + O(1/k)$.

4. Sequential Implementation

In the previous section, we described PRUNED PIVOT algorithm. *For the sake of analysis*, we now examine a sequential algorithm that produces the same clustering as the recursive algorithm above and, moreover, marks the same set of nodes as unlucky. First, we consider the standard PIVOT implemented as a bottom-up dynamic programming algorithm (see Algorithm 3).

Algorithm 3 SEQUENTIAL PIVOT

```
Pick a random ordering π : V → {1,...,n}.
Let V_π be the list of all nodes u ∈ V sorted by the rank π(u).
for each u in V_π:
  Sort all neighbors of u by their rank π(v). Denote the sorted list by N_π.
  while u is not assigned to a cluster:
    Pick the next neighbor v ∈ N_π(u).
    if v is a pivot: place u in the cluster of v.
    if v = u: mark u as a pivot; create a new cluster for u; and place u in that cluster.
```

In the main loop (see the **for each** loop above), the algorithm iterates over all nodes in V. At iteration $i \in \{1, \dots, n\}$, the algorithm processes node u with rank i, i.e., $u = \pi^{-1}(i)$. It checks all neighbors v of u with rank higher than that of u. If one of these neighbors is a pivot, the algorithm assigns u to the highest-ranked pivot neighbor of u. If none of these neighbors are pivots, the algorithm marks u as a pivot and assigns u to itself.

Let us set up some notation. Consider a neighbor v of u. It is processed at iteration $i = \pi(v)$. Suppose that no other neighbor of u (including u itself) is marked as a pivot before iteration i. Then, we know that u will be assigned to the cluster of v, since it is the highest-ranked pivot neighbor of u. Thus, we will say that u is *settled* at step i. In other words, u is settled when the first neighbor of u is marked as a pivot. We denote the iteration when u is settled by $\sigma(u)$. Note that node u is assigned to the node processed at iteration $\sigma(u)$, i.e., node $\pi^{-1}(\sigma(u))$. In particular, if u is a pivot, then it is settled at the iteration $i = \pi(u)$, the same iteration as it is processed. We always have $\sigma(u) \le \pi(u)$, because if u is not settled before iteration $\pi(u)$, then it is marked as a pivot and assigned to itself at iteration $\pi(u)$; thus, if $\sigma(u) \ge \pi(u)$, then $\sigma(u) = \pi(u)$.

If neighbor v of u is considered in the **while**-loop of the SEQUENTIAL PIVOT algorithm, then we say that u queries v. We denote by $Q(u)$ the set of all neighbors queried by u, except for u itself, and call this set the set of *queried neighbors of* u. Observe that $Q(u) = \{v \in N(u) \setminus \{u\} : \pi(v) \le \sigma(u)\}$. That is, $Q(u)$ is the set of all neighbors of u, excluding u, whose rank is higher than the rank of the pivot to which u is assigned. Finally, we formally define the recursion tree $\mathcal{T}_u$ for node u. The definition is recursive: If $Q(u)$ is empty, then $\mathcal{T}_u$ only contains node u. Otherwise, $\mathcal{T}_u$ is the tree with root u and $|Q(u)|$ subtrees $\widetilde{\mathcal{T}}_v$ attached to it – one tree for every $v \in Q(u)$. Each $\widetilde{\mathcal{T}}_v$ is a copy of the recursive tree $\mathcal{T}_v$. We stress that the recursive tree may contain multiple copies of the same node v. One can think of nodes of $\mathcal{T}_u$ as being "stack traces" or "execution paths" for the recursive function CLUSTER.

Sequential Pivot with Pruning. We now describe how to modify the bottom-up algorithm to make it equivalent to PRUNED PIVOT algorithm. First, we run the bottom-up algorithm as is and record its trace. We then define the recursive call count for every node u. The recursive call count of u equals the number of edges in the recursive tree $\mathcal{T}_u$. It can be computed using the following recurrence relation:

$$\text{call-count}(u) = \sum_{v \in Q(u)} (1 + \text{call-count}(v)). \qquad (1)$$

If $Q(u)$ is empty, then the recursive call count of u equals 0, by definition. We mark node u as *unlucky* if its recursive call

count is at least k. Note that if one of the queried neighbors of u is unlucky, then u is also unlucky.

Algorithm 4 PRUNING

1: Compute the recursive call count of every node u using recurrence relation (1).
2: Mark all nodes u with call-count$(u) \geq k$ as unlucky.
3: Create a new cluster for each unlucky node u, remove u from its current cluster, and place u in the new cluster.

The pruning step puts all unlucky nodes into singleton clusters. We refer to the standard PIVOT algorithm as PIVOT without pruning or simply PIVOT. We refer to the PIVOT algorithm that runs the pruning as PIVOT WITH PRUNING. Note that the PIVOT WITH PRUNING algorithm produces the same clustering output as the PRUNED PIVOT algorithm described in the previous section. The key difference between these algorithms lies in their structure: PIVOT WITH PRUNING consists of two distinct steps – a PIVOT step followed by a PRUNING step – whereas PRUNED PIVOT combines both steps together. We show that the expected cost of the PIVOT WITH PRUNING is $(3+O(1/k))$OPT and obtain the following theorem.

Theorem 4.1. *The expected cost of the clustering produced by the* PRUNED PIVOT *is* $(3 + O(1/k))$ OPT.

Ailon, Charikar, and Newman (2008) showed that the approximation factor of PIVOT is 3. By Lemma F.1, the PIVOT step (see Algorithm 3) is equivalent to the PIVOT algorithm. Hence, its approximation factor is also 3. The pruning step of PIVOT WITH PRUNING removes some nodes (namely, unlucky nodes) from their original clusters and puts them into singleton clusters. This pruning step can increase the number of pairs of nodes (u, v) disagreeing with the clustering. Note, however, that if u and v are dissimilar (i.e., not connected with an edge), then the pruning step will never make them disagree with the clustering if they agreed with the original clustering. Thus, the pruning step can increase the objective function only by separating pairs of similar nodes $(u, v) \in E$. In such case, we say that the pruning step cuts edge (u, v). Specifically, edge (u, v) is cut by the pruning step of PIVOT WITH PRUNING if u and v are in the same cluster after the PIVOT step of the algorithm, but are separated by the pruning step, because u, v, or both u and v are unlucky nodes. We say that an edge $(u, v) \in E$ is cut by PIVOT (without pruning), if PIVOT places u and v in distinct clusters. In the next sections, we show Lemma 4.2 that states that the expected number of edges cut by the pruning step of PIVOT WITH PRUNING is upper bounded by the expected number of edges cut by PIVOT divided by $\lceil (k-1)/2 \rceil /2$. The "triangle-based" analysis of PIVOT by Ailon, Charikar, and Newman (2008) shows that PIVOT cuts at most 2OPT edges in expectation. Thus, the pruning step cuts at most $4\text{OPT}/\lceil (k-1)/2 \rceil$ edges in expectation. We conclude that the expected cost of PIVOT WITH PRUNING is at most $(3 + 4/\lceil (k-1)/2 \rceil)$OPT.

Lemma 4.2. *The expected number of edges (u, v) cut by the pruning step of* PIVOT WITH PRUNING *is upper bounded by the expected number of edges cut by* PIVOT *divided by $\lceil (k-1)/2 \rceil /2$.*

4.1. Query Paths

Our goal now is to prove Lemma 4.2.

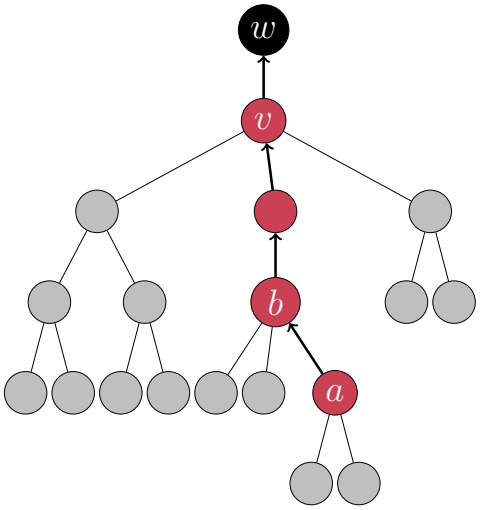

Figure 1. This figure shows an *extended query path* in the recursion tree $\mathcal{T}_v$ for node v. The path starts with edge (a, b) goes to the root of the tree, node v, and then proceeds to node w. The path from a till v is a *query path*. The path from a to w *extends* the path from a to v. If edge (a, b) is cut by the pivot step of PIVOT but edge (v, w) is not cut, then this path is *expensive*. We call it expensive because if v is unlucky, then (v, w) is cut by the pruning step of PIVOT WITH PRUNING and the cost of (v, w) is partially charged to this path.

In this section, we define *query paths*, *extended query paths*, and *expensive extended query paths*. We then show that on the one hand, the number of edges cut by the pruning step of the PIVOT WITH PRUNING algorithm is upper bounded by the number of expensive extended query paths divided by $\lceil (k-1)/2 \rceil$ (see Corollary 4.9); and, on the other hand, the expected number of expensive extended query paths is upper bounded by two times the expected number of edges cut by the PIVOT algorithm (see Theorem 4.10). This will imply Lemma 4.2.

Definition 4.3 (Query Paths)**.** *A path $(u_0, u_1, \ldots, u_L)$ is a* query path *if each u_i $(i > 0)$ queries u_{i-1}.*

Definition 4.4 (Extended Query Paths)**.** *A path $(u_0, u_1, \ldots, u_L)$ of length $L \geq 2$ is an* extended query path *(EQ-path) if the following two conditions hold: (1) $(u_0, u_1, \ldots, u_{L-1})$ is a query path; and (2)*

$\pi(u_{L-2}) \leq \sigma(u_L)$. *We say that EQ-path* $(u_0, u_1, \ldots, u_L)$ *is an extension of the query path* $(u_0, u_1, \ldots, u_{L-1})$. *We also call every path consisting of one edge* (u_0, u_1) *an extended query path.*

Note that a proper prefix of a query or extended query path is a query path.

Recall, that for every u, we have $\sigma(u) \leq \pi(u)$. Also, a node u queries its neighbor v ($v \neq u$) if and only if u is not settled before v is processed, i.e., $\sigma(u) \geq \pi(v)$. Thus, $(u_0, u_1, \ldots, u_L)$ is a query path if and only if

$$\sigma(u_0) \leq \pi(u_0) \leq \sigma(u_1) \leq \pi(u_1) \leq \ldots \\ \leq \sigma(u_{L-1}) \leq \pi(u_{L-1}) \leq \sigma(u_L) \leq \pi(u_L). \quad (2)$$

Similarly, a path $(u_0, u_1, \ldots, u_L)$ of length $L \geq 2$ is an EQ-path if and only if

$$\sigma(u_0) \leq \pi(u_0) \leq \sigma(u_1) \leq \pi(u_1) \leq \ldots \\ \leq \sigma(u_{L-2}) \leq \pi(u_{L-2}) \leq \min(\sigma(u_{L-1}), \sigma(u_L)). \quad (3)$$

We will charge all edges cut by the pruning step of PIVOT WITH PRUNING to $\Theta(k)$ *expensive EQ-paths* which are defined as follows.

Definition 4.5 (Expensive Extended Query Paths)**.** *An extended query path* $(u_0, u_1, \ldots, u_L)$ *is expensive if* $\sigma(u_0) < \sigma(u_1)$ *but* $\sigma(u_{L-1}) = \sigma(u_L)$. *We denote the set of all expensive query paths by* $\mathcal{X}$.

Note that in every expensive EQ-path, the first edge is cut by PIVOT (because $\sigma(u_0) < \sigma(u_1)$) but the last edge is not cut (because $\sigma(u_{L-1}) = \sigma(u_L)$). A path $(u_0, u_1, \ldots, u_L)$ is an EQ-path if and only if condition (3) holds, thus $(u_0, u_1, \ldots, u_L)$ is an expensive EQ-path if and only if

$$\sigma(u_0) < \sigma(u_1) \text{ and } \sigma(u_0) \leq \pi(u_0) \leq \sigma(u_1) \leq \pi(u_1) \leq \\ \cdots \leq \pi(u_{L-2}) \leq \sigma(u_{L-1}) = \sigma(u_L). \quad (4)$$

The first condition $\sigma(u_0) < \sigma(u_1)$ in (4) can be replaced with $\sigma(u_0) \neq \sigma(u_1)$, because we always have $\sigma(u_0) \leq \sigma(u_1)$ if $u_0, \ldots, u_{L-1}$ is a query path.

4.1.1. CHARGING CUT EDGES TO EXPENSIVE PATHS

We now prove a lemma that establishes a connection between edges cut by the pruning step of PIVOT WITH PRUNING and expensive EQ-paths.

Lemma 4.6. *For every unlucky node* v *and every edge* (v, w) *with* $\sigma(v) = \sigma(w)$*, there exist at least* $\lceil (k-1)/2 \rceil$ *expensive extended query paths that end with edge* (v, w).

Proof. Let $\mathcal{T}_v$ be the recursion tree for node v. We first show that $\mathcal{T}_v$ contains at least $\lceil (k-1)/2 \rceil$ edges cut by PIVOT (formally, $\mathcal{T}_v$ contains copies of edges cut by PIVOT). Consider an edge (u', u'') in T. Since (u', u'') is an edge in the recursion tree, u' queries u''. Thus, u' is assigned to u'' (if u'' is a pivot) or some neighbor of u' which is lower ranked than u'' (if u'' is not a pivot). Vertex u'' is assigned to itself (if u'' is a pivot) or one of its neighbors ranked higher than u'' (if u'' is not a pivot). Thus, if edge (u', u'') is not cut by PIVOT, then u'' is a pivot and u' is assigned to u''. This means that u'' is the highest-ranked pivot neighbor of u'. Consequently, for every u', there is at most one child node u'' such that (u', u'') is not cut. Moreover, if one such u'' exists, then u' is not a pivot, and hence the edge from u' to its parent is cut (unless u' is the root). We get the following claim.

Claim 4.7. *For every node* u *in the recursion tree* $\mathcal{T}_v$*, at most one edge incident on* u *is not cut by* PIVOT.

Node v is unlucky. Hence, the recursion tree $\mathcal{T}_v$ must have at least k edges. Therefore, by Claim 4.7 and Lemma 4.8 (see below) there are at least $\lceil (k-1)/2 \rceil$ cut edges in $\mathcal{T}_v$. In Lemma 4.8, red edges are cut edges, and blue edges are not cut edges.

Lemma 4.8. *Consider a tree* T *with* k *edges colored red or blue. Suppose that at most one blue edge is incident on each node in* T*. Then,* T *contains at least* $\lceil (k-1)/2 \rceil$ *red edges.*

Proof. Tree T has k edges and $k+1$ nodes. At most one blue edge is incident on each node. So, blue edges form a matching. The size of this matching is at most $\lfloor (k+1)/2 \rfloor$. The number of edges not in the matching is at least $k - \lfloor (k+1)/2 \rfloor = \lceil (k-1)/2 \rceil$. All of them are red. □

Now, for every edge (b, a) in T such that b queries a and (b, a) is cut by PIVOT, we construct an expensive EQ-path. This path starts with edge (a, b), then goes to the root of tree T – node v – along the edges of T, and, finally, proceeds to node w (see Figure 1). Observe that the subpath from a to v is a query path since each node on the path queries the preceding node. We know that $\sigma(v) = \sigma(w)$. Hence, by (3), the path $(a, b, \ldots, v, w)$ is an expensive query path. □

The immediate corollary of this lemma gives us a bound on the number of edges cut by the pruning step.

Corollary 4.9. *The number of edges cut by the pruning step of* PIVOT WITH PRUNING *is at most* $|\mathcal{X}| / \lceil (k-1)/2 \rceil$.

Proof. Every edge $(v, w) \in E$ cut by the pruning step of PIVOT WITH PRUNING is not cut by PIVOT. Hence, $\sigma(v) = \sigma(w)$. Moreover, if (v, w) is cut by the pruning step, then v, w, or both v and w must be unlucky. Thus, by Lemma 4.6, there are at least $\lceil (k-1)/2 \rceil$ expensive EQ-paths that end with (v, w) or (w, v). Therefore, there

exists at least $\lceil (k-1)/2 \rceil$ unique expensive EQ-paths for each edge (v, w) cut by the pruning step. □

4.1.2. Expected Number of Expensive EQ-Paths

We now prove that the expected number of expensive EQ-paths is at most $4OPT$ and the expected number of query paths that start with a fixed directed edge (a, b) – we denote these paths by $\mathcal{Q}(a, b)$ – is at most 2.

Theorem 4.10. *For every ordered pair (a, b) with $(a, b) \in E$, we have $\mathbf{E}_\pi|\mathcal{Q}(a, b)| \le 2$, and*

$$\mathbf{E}_\pi|\mathcal{X}| \le 2\mathbf{E}\Big[\sum_{(u,v)\in E} \mathbf{1}(\sigma(u) \ne \sigma(v))\Big].$$

We will refer to the time when iteration t of Pivot occurs as time t. For the sake of analysis, we shall assume that the ordering π is initially (at time 0) hidden from us and is revealed one node at a time. At the beginning of iteration t, we learn the value of $\pi^{-1}(t)$, or, in other words, the identity of the node processed at time t. Note that the state of the algorithm after the first t iterations is completely determined by the nodes $\pi^{-1}(1), \ldots, \pi^{-1}(t)$. In particular, at time t, for every node u, we can tell if u is settled by time t and, if it is settled, then we know the value of $\sigma(u)$; otherwise, we know that $\sigma(u) > t$. Let $\mathcal{F}_t$ be the filtration generated by $\pi^{-1}(1), \ldots, \pi^{-1}(t)$. We will use the standard notation $\Pr[\,\cdot \mid \mathcal{F}_t]$ and $\mathbf{E}[\,\cdot \mid \mathcal{F}_t]$ to denote the conditional probability and conditional expectation given the state of the algorithm after iteration t. Note that each $\pi(u)$ and $\sigma(v)$ is a *stopping time* with respect to $\mathcal{F}_t$.

Let $\mathcal{P}(a, b)$ be the set of all paths that start with edge (a, b). As we run the Pivot algorithm, we add paths to sets $\mathcal{Q}_t(a, b)$ and $\mathcal{X}_t(a, b)$. Loosely speaking, we add a path from $\mathcal{P}(a, b)$ to $\mathcal{Q}_t(a, b)$ if we can verify that this path is a query path using condition (2) at time t; we add a path from $\mathcal{P}(a, b)$ to $\mathcal{X}_t(a, b)$ if we can verify that this path is an expensive EQ-path using condition (4) at time t. Formally, we add path $(u_0 = a, u_1 = b, \ldots, u_L)$ to $\mathcal{Q}_t(a, b)$ at time $\pi(u_{L-1})$ if condition (2) holds; and we add path $(u_0 = a, u_1 = b, \ldots, u_L)$ to $\mathcal{X}_t(a, b)$ at time $\sigma(u_{L-1}) = \sigma(u_L)$ if condition (4) holds. Thus, $\mathcal{Q}_t(a, b)$ is the set of all query paths $P \in \mathcal{P}(a, b)$ for which $\pi(u_{L-1}) \le t$; and $\mathcal{X}_t(a, b)$ is the set of all expensive EQ-paths $P \in \mathcal{P}(a, b)$ for which $\sigma(u_{L-1}) = \sigma(u_L) \le t$. Note that at the times $\pi(u_{L-1})$ and $\sigma(u_{L-1}) = \sigma(u_L)$, we can check conditions (2) and (4), respectively. We also define a set of *dangerous* paths at time t, denoted by $\mathcal{D}_t(a, b)$, as follows.

Definition 4.11 (Dangerous EQ-path)**.** *An extended query path $(u_0, \ldots, u_L)$ ($L \ge 1$) is* dangerous *at iteration t if $\pi(u_{L-2}) \le t$, $\pi(u_{L-1}) > t$, and $\sigma(u_L) > t$. We omit the first condition ($\pi(u_{L-2}) \le t$) for paths of length 1. Denote the set of all extended query paths that start with edge (a, b) and are dangerous at iteration t by $\mathcal{D}_t(a, b)$.*

Note that a path $P \in \mathcal{P}(a, b)$ may become dangerous at some iteration t, stay dangerous for some time, but eventually it will become non-dangerous. After that, it will remain non-dangerous until the end of the algorithm. The definition of dangerous paths is justified by the following lemma, which, loosely speaking, says that every query path and every expensive EQ-path is created from a dangerous path.

Lemma 4.12. *Consider a path $P = (u_0, u_1, \ldots, u_L) \in \mathcal{P}(a, b)$. Let $P' = (u_0, u_1, \ldots, u_{L-1})$. Then, the following claims hold for every $t \ge 0$:*

- *If $P \in \mathcal{Q}_{t+1}(a, b) \setminus \mathcal{Q}_t(a, b)$, then $P \in \mathcal{D}_t(a, b)$ but $P \notin \mathcal{D}_{t+1}(a, b)$.*
- *If $P \in \mathcal{D}_{t+1}(a, b) \setminus \mathcal{D}_t(a, b)$, then $P' \in \mathcal{D}_t(a, b)$ but $P' \notin \mathcal{D}_{t+1}(a, b)$.*
- *If $P \in \mathcal{X}_{t+1}(a, b) \setminus \mathcal{X}_t(a, b)$, then $P \in \mathcal{D}_t(a, b)$ or $P' \in \mathcal{D}_t(a, b)$ but $P \notin \mathcal{D}_{t+1}(a, b)$ and $P' \notin \mathcal{D}_{t+1}(a, b)$.*

We prove this lemma in Appendix C.1.

Our approach to bounding $\mathbf{E}|\mathcal{Q}_t(a, b)|$ and $\mathbf{E}|\mathcal{X}_t(a, b)|$ is based on the following idea: At time $t = 0$, the path (a, b) is dangerous, and there are no query or expensive EQ-paths that start with (a, b). If P is a dangerous EQ-path at time t, then at the next iteration, it may be extended to a longer dangerous path, replaced with a query path, and/or created one or more expensive EQ-paths. A dangerous path may also disappear without producing any new dangerous, query, or expensive EQ-paths. For every EQ-path P dangerous at iteration t, we will compute the probabilities of creating new paths and derive the desired bounds on $\mathbf{E}|\mathcal{Q}_t(a, b)|$ and $\mathbf{E}|\mathcal{X}_t(a, b)|$. To make our argument formal, we define two random processes:

$$\Phi_t(a, b) = 2|\mathcal{D}_t(a, b)| + |\mathcal{Q}_t(a, b)|;$$
$$\Psi_t(a, b) = 2|\mathcal{D}_t(a, b)| + |\mathcal{X}_t(a, b)|.$$

We claim that $\Phi_t(a, b)$ and $\Psi_t(a, b)$ are supermartingales. That is, $\mathbf{E}[\Phi_{t+1}(a, b) \mid \mathcal{F}_t] \le \Phi_t(a, b)$; and $\mathbf{E}[\Psi_{t+1}(a, b) \mid \mathcal{F}_t] \le \Psi_t(a, b)$.

Lemma 4.13. *Random processes $\Phi_t(a, b)$ and $\Psi_t(a, b)$ are supermartingales.*

We prove this lemma in Appendix C.2. We now use it to finish the proof of Theorem 4.10. We first upper-bound $E_\pi|\mathcal{Q}(a, b)|$. Fix a directed edge (a, b). At time 0, $\Phi_0(a, b) = 2$, since (a, b) is a dangerous EQ-path at time 0 but $(a, b) \notin \mathcal{Q}_0(a, b)$. Process $\Phi_0(a, b)$ is a supermartingale. Hence, $\mathbf{E}[\Phi_n(a, b)] \le 2$. Note that at time n, there

are no dangerous EQ-paths because by time n all nodes are processed and settled. Hence,

$$\mathbf{E}[\mathcal{Q}(a,b)] = \mathbf{E}[\mathcal{Q}_n(a,b)] = \mathbf{E}[\Phi_n(a,b)] \le 2.$$

We now upper-bound $\mathbf{E}|\mathcal{X}|$. Every expensive EQ-path $P = (u_0, \dots, u_L)$ starts with a directed edge (u_0, u_1) and at some iteration t is added to the set $\mathcal{Q}_t(u_0, u_1)$. We have (in the sum below, we have two terms, $\mathbf{E}|\mathcal{X}_n(a,b)|$ and $\mathbf{E}|\mathcal{X}_n(b,a)|$, for every edge $(a,b) \in E$):

$$\begin{aligned}\mathbf{E}|\mathcal{X}| &= \sum_{a,b:(a,b)\in E} \mathbf{E}|\mathcal{X}_n(a,b)| \\ &= \sum_{a,b:(a,b)\in E} \mathbf{E}\big[|\mathcal{X}_n(a,b)| \cdot \mathbf{1}(\sigma(a) < \sigma(b))\big].\end{aligned}$$

Here, we used the definition of expensive PQ-paths: In every expensive path P in $\mathcal{P}(a,b)$, $\sigma(a) < \sigma(b)$. Thus, if $\sigma(a) \ge \sigma(b)$, then $\mathcal{X}_n(a,b) = \varnothing$. Observe that $|\mathcal{X}_n(a,b)| = \Psi_n(a,b)$ and $\mathbf{E}\big[\Psi_n(a,b) \mid \mathcal{F}_{\sigma(a)}\big] \le \Psi_{\sigma(a)}(a,b)$, because Ψ_n is a supermartingale. Moreover,

$$\begin{aligned}\mathbf{E}\big[\Psi_n(a,b) \cdot \mathbf{1}(\sigma(a) < \sigma(b)) \mid \mathcal{F}_{\sigma(a)}\big] \le \\ \le \Psi_{\sigma(a)}(a,b) \cdot \mathbf{1}(\sigma(a) < \sigma(b)),\end{aligned}$$

because the event $\{\sigma(a) < \sigma(b)\}$ is in $\mathcal{F}_{\sigma(a)}$, or, in other words, at time $\sigma(a)$, we already know the value of $\mathbf{1}(\sigma(a) < \sigma(b))$, and this value does not change over time. Thus,

$$\begin{aligned}\mathbf{E}|\mathcal{X}_n(a,b)| &= \mathbf{E}\mathbf{E}\big[\Psi_n(a,b) \cdot \mathbf{1}(\sigma(a) < \sigma(b)) \mid \mathcal{F}_{\sigma(a)}\big] \\ &\le \mathbf{E}\big[\Psi_{\sigma(a)}(a,b) \cdot \mathbf{1}(\sigma(a) < \sigma(b))\big].\end{aligned}$$

It remains to compute $\Psi_{\sigma(a)}(a,b) \cdot \mathbf{1}(\sigma(a) < \sigma(b))$. If $\sigma(a) < \sigma(b)$, then a is not a pivot (otherwise, we would have $\sigma(a) = \sigma(b) = \pi(a)$). Thus, $\pi(a) > \sigma(a)$, and the only EQ-path in $\mathcal{P}(a,b)$ dangerous at time $\tau = \sigma(a)$ is the path (a,b). Hence, $|\mathcal{D}_{\sigma(a)}(a,b)| = 1$. Similarly, there are no expensive PQ-paths in $\mathcal{X}_{\sigma(a)}(a,b)$, because $\sigma(b) > \tau = \sigma(a)$. Therefore, $\Psi_{\sigma(a)}(a,b) \cdot \mathbf{1}(\sigma(a) < \sigma(b)) = 2 \cdot \mathbf{1}(\sigma(a) < \sigma(b))$, and

$$\mathbf{E}|\mathcal{X}| \le 2\mathbf{E}\Big[\sum_{(a,b)\in E} \mathbf{1}(\sigma(a) \ne \sigma(b))\Big].$$

5. MPC Algorithm

Now we discuss how to simulate PRUNED PIVOT in MPC. To that end, observe that PRUNED PIVOT has depth at most k. Moreover, for each node v, there are at most k neighbors of v, which the computation for v depends on. To find those k neighbors of v, we sort all the neighbors of v and select the top k ones. Sorting can be done in $O(1)$ MPC rounds (Goodrich et al., 2011) and $O(m)$ total memory for all the nodes simultaneously.

Define a directed graph H on V such that H contains an edge (u,v) if and only if v is among the top k neighbors of u. The k-hop neighborhood of u in H contains all the edges and nodes needed to process u by PRUNED PIVOT. Given this, our MPC algorithm simultaneously gathers k-hop neighborhood for each u. This can be done in $O(\log k)$ MPC rounds and $O(n \cdot k^{k+1})$ total memory via graph exponentiation (see the paper by Lenzen & Wattenhofer (2010)). The output for u of PRUNED PIVOT is computed on a single machine using its relevant k-hop neighborhood.

6. Fully Dynamic Algorithm under Oblivious Adversary

This section explains how to implement PRUNED PIVOT in the fully dynamic setting with an oblivious adversary. On a high level, when an edge e is updated, our dynamic algorithm simply recomputes the clustering for each node that queries e. As Theorem 4.10 states, there are only $O(1)$ such nodes in expectation. This property is the key to enabling us to obtain only $O(k)$ expected amortized update time.

In addition, our algorithm updates the neighbor list of the endpoints of e, which is needed to implement PRUNED PIVOT. To obtain the desired running time, we use that PRUNED PIVOT visits at most the top k neighbors of a node. Therefore, instead of maintaining the entire neighborhood list of a node in a sorted manner, we do it only for the top k neighbors of each node. We show how to dynamically maintain this list in expected amortized $O(\log k)$ time.

We now provide details. Our algorithm maintains the following information for each node u:

- $N_k(u)$: Top k neighbors of u kept in an ordered balanced binary tree.
- $Q_P^{-1}(u)$: The nodes that query u, except from u itself, kept in a double-linked list.
- $Q_P(u)$: The set of nodes queried by PRUNED-CLUSTER(π, u), which is maintained during the recursive calls. Together with every node w queried by u, in $Q_P(u)$ is also stored the pointer to where u is in the double-linked list $Q_P^{-1}(w)$. We use these pointers to efficiently update $Q_P^{-1}(\cdot)$.

Consider an update of edge $\{a,b\}$, i.e., an edge insertion or removal. This update triggers several updates in the information we maintain for each node. Without loss of generality, assume that $\pi(a) < \pi(b)$. Note that among all $N_k(\cdot)$, only $N_k(b)$ changes. We show how to update $N_k(b)$ in $O(\log k)$ expected time in Appendix E.1.

If $N_k(b)$ is changed, but b queries (b,a) neither before nor after this update, then Q_P and Q_P^{-1} structures remain the

same as before the update. However, if whether b queries (b, a) changes after the update, then all the nodes querying b, i.e., those in $Q_P^{-1}(b)$ might change their structures. Moreover, only those nodes that query b before the update might change their structures. To see that, assume that w does not query b before the edge update. First, if w queries a, it will not query (a, b) since $\pi(a) < \pi(b)$. Second, and taking into account that w never queries (a, b), if w never reaches b in the invocation of PRUNED-CLUSTER(w, π) before the update, then $\{a, b\}$ cannot be queried by w regardless of the update. This yields the following observation.

Observation 6.1. *Consider an edge update $\{a, b\}$ with $\pi(a) < \pi(b)$. Then, only the nodes in $Q_P^{-1}(b) \cup \{b\}$ are those whose maintained data structures can potentially change on the update.*

We remind the reader that PRUNED-CLUSTER(w, π) does not use memoization. In PRUNED PIVOT-UPDATE$(\cdot, \cdot)$ (see Algorithm 6) we provide our update procedure, which uses Algorithm 5 as a subroutine. We analyze its running time in Appendix E.2.

Algorithm 5 A-NODE-UPDATE(w)

for every $u \in Q_P(w)$:
  Remove w from $Q_P^{-1}(u)$ by using the corresponding pointer from $Q_P(w)$.
Invoke PRUNED-CLUSTER (w,π) (Algorithm 2) by using $N_k(\cdot)$ as the neighborhood list. During the execution, update $Q_P(w)$.
for every $u \in Q_P(w)$:
  Append w to $Q_P^{-1}(u)$ and record in $Q_P(w)$ the pointer where w is appended to.

Algorithm 6 PRUNED PIVOT-UPDATE(a, b)

Update $N_k(b)$ as described in Appendix E.1.
A-NODE-UPDATE(b)
if b queries (b, a):
  for every $w \in Q_P^{-1}(b)$:
    A-NODE-UPDATE(w)

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

Acknowledgements

We are grateful to anonymous reviewers for their valuable feedback and for suggesting to discuss the implementation of our algorithm in PRAM. K.Makarychev was supported by the NSF Awards CCF-1955351 and EECS-2216970. S. Mitrović was supported by the Google Research Scholar and NSF Faculty Early Career Development Program #2340048.

References

Ahmadi, S., Khuller, S., and Saha, B. Min-max correlation clustering via multicut. In *International Conference on Integer Programming and Combinatorial Optimization*, pp. 13–26. Springer, 2019.

Ailon, N., Charikar, M., and Newman, A. Aggregating inconsistent information: ranking and clustering. *Journal of the ACM (JACM)*, 55(5):1–27, 2008.

Akl, S. G. *The design and analysis of parallel algorithms*. Prentice-Hall, Inc., 1989.

Arasu, A., Ré, C., and Suciu, D. Large-scale deduplication with constraints using dedupalog. In *2009 IEEE 25th International Conference on Data Engineering*, pp. 952–963. IEEE, 2009.

Bansal, N., Blum, A., and Chawla, S. Correlation clustering. *Machine learning*, 56:89–113, 2004.

Becker, H. A survey of correlation clustering. *Advanced Topics in Computational Learning Theory*, pp. 1–10, 2005.

Behnezhad, S., Derakhshan, M., Hajiaghayi, M., Stein, C., and Sudan, M. Fully dynamic maximal independent set with polylogarithmic update time. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 382–405. IEEE, 2019.

Behnezhad, S., Charikar, M., Ma, W., and Tan, L.-Y. Almost 3-approximate correlation clustering in constant rounds. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 720–731. IEEE, 2022.

Behnezhad, S., Charikar, M., Ma, W., and Tan, L.-Y. Single-pass streaming algorithms for correlation clustering. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 819–849. SIAM, 2023.

Bonchi, F., Gionis, A., and Ukkonen, A. Overlapping correlation clustering. *Knowledge and information systems*, 35:1–32, 2013.

Cambus, M., Pai, S., and Uitto, J. A parallel algorithm for $(3+\varepsilon)$-approximate correlation clustering. *arXiv preprint arXiv:2205.07593*, 2022.

Cao, N., Cohen-Addad, V., Lee, E., Li, S., Newman, A., and Vogl, L. Understanding the cluster lp for correlation clustering. In *STOC*, 2024.

Chakrabarty, S. and Makarychev, K. Single-Pass Pivot Algorithm for Correlation Clustering. Keep it simple! In *NeurIPS 2023*, 2023.

Charikar, M., Guruswami, V., and Wirth, A. Clustering with qualitative information. *Journal of Computer and System Sciences*, 71(3):360–383, 2005.

Charikar, M., Gupta, N., and Schwartz, R. Local guarantees in graph cuts and clustering. In *International Conference on Integer Programming and Combinatorial Optimization*, pp. 136–147. Springer, 2017.

Chawla, S., Makarychev, K., Schramm, T., and Yaroslavtsev, G. Near optimal lp rounding algorithm for correlation clustering on complete and complete k-partite graphs. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pp. 219–228, 2015.

Chechik, S. and Zhang, T. Fully dynamic maximal independent set in expected poly-log update time. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 370–381. IEEE, 2019.

Cohen-Addad, V., Lee, E., and Newman, A. Correlation clustering with sherali-adams. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 651–661. IEEE, 2022.

Cohen-Addad, V., Lee, E., Li, S., and Newman, A. Handling correlated rounding error via preclustering: A 1.73-approximation for correlation clustering. *arXiv preprint arXiv:2309.17243*, 2023.

Dean, J. and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Demaine, E. D., Emanuel, D., Fiat, A., and Immorlica, N. Correlation clustering in general weighted graphs. *Theoretical Computer Science*, 361(2-3):172–187, 2006.

Ding, L., Han, B., Wang, S., Li, X., and Song, B. User-centered recommendation using us-elm based on dynamic graph model in e-commerce. *International Journal of Machine Learning and Cybernetics*, 10:693–703, 2019.

Fang, Y., Wang, H., Zhao, L., Yu, F., and Wang, C. Dynamic knowledge graph based fake-review detection. *Applied Intelligence*, 50:4281–4295, 2020.

Firman, M., Thomas, D., Julier, S., and Sugimoto, A. Learning to discover objects in rgb-d images using correlation clustering. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1107–1112. IEEE, 2013.

Goodrich, M. T., Sitchinava, N., and Zhang, Q. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*, pp. 374–383. Springer, 2011.

Hafiene, N., Karoui, W., and Romdhane, L. B. Influential nodes detection in dynamic social networks: A survey. *Expert Systems with Applications*, 159:113642, 2020.

Jafarov, J., Kalhan, S., Makarychev, K., and Makarychev, Y. Local correlation clustering with asymmetric classification errors. In *International Conference on Machine Learning*, pp. 4677–4686. PMLR, 2021.

JáJá, J. *Parallel algorithms*. 1992.

Kalashnikov, D. V., Chen, Z., Mehrotra, S., and Nuray-Turan, R. Web people search via connection analysis. *IEEE Transactions on Knowledge and Data Engineering*, 20(11):1550–1565, 2008.

Kalhan, S., Makarychev, K., and Zhou, T. Correlation clustering with local objectives. *Advances in Neural Information Processing Systems*, 32, 2019.

Karloff, H., Suri, S., and Vassilvitskii, S. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pp. 938–948. SIAM, 2010.

Lenzen, C. and Wattenhofer, R. Brief announcement: Exponential speed-up of local algorithms using non-local communication. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pp. 295–296, 2010.

Li, P., Dau, H., Puleo, G., and Milenkovic, O. Motif clustering and overlapping clustering for social network analysis. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pp. 1–9. IEEE, 2017.

Puleo, G. and Milenkovic, O. Correlation clustering and biclustering with locally bounded errors. In *International Conference on Machine Learning*, pp. 869–877. PMLR, 2016.

Rubinfeld, R., Tamir, G., Vardi, S., and Xie, N. Fast local computation algorithms. *arXiv preprint arXiv:1104.1377*, 2011.

Shi, J., Dhulipala, L., Eisenstat, D., Łăcki, J., and Mirrokni, V. Scalable community detection via parallel correlation clustering. *Proceedings of the VLDB Endowment*, 14(11): 2305–2313, 2021.

Swamy, C. Correlation clustering: maximizing agreements via semidefinite programming. In *SODA*, volume 4, pp. 526–527. Citeseer, 2004.

Tantipathananandh, C. and Berger-Wolf, T. Y. Finding communities in dynamic social networks. In *2011 IEEE 11th international conference on data mining*, pp. 1236–1241. IEEE, 2011.

Veldt, N., Wirth, A., and Gleich, D. F. Parameterized correlation clustering in hypergraphs and bipartite graphs. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1868–1876, 2020.

Wang, Y., Xu, L., Chen, Y., and Wang, H. A scalable approach for general correlation clustering. In *Advanced Data Mining and Applications: 9th International Conference, ADMA 2013, Hangzhou, China, December 14-16, 2013, Proceedings, Part II 9*, pp. 13–24. Springer, 2013.

Yan, Y., Liu, L., Ban, Y., Jing, B., and Tong, H. Dynamic knowledge graph alignment. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pp. 4564–4572, 2021.

Yoshida, Y., Yamamoto, M., and Ito, H. An improved constant-time approximation algorithm for maximum˜ matchings. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 225–234, 2009.

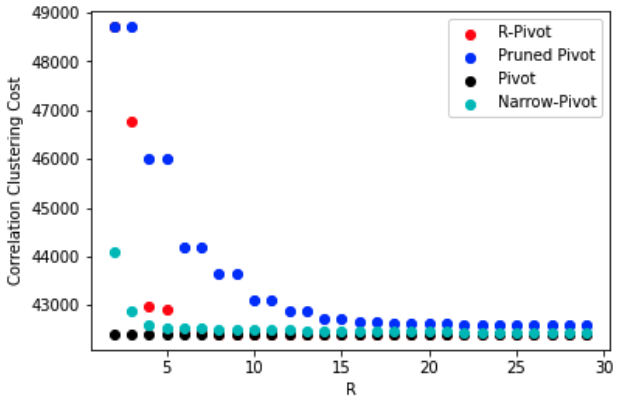

Figure 2. Comparison of correlation clustering cost for PIVOT, R-PIVOT, Narrow-PIVOT and PRUNED PIVOT. The optimal clustering has an expected cost of less than 17970.

A. Empirical Evaluation

In this section, we conduct a simple empirical assessment of our algorithm. We compare the correlation clustering cost of PRUNED PIVOT to that of PIVOT, R-PIVOT of Behnezhad et al. (2022), and Narrow-PIVOT of Chakrabarty & Makarychev (2023) on synthetic graphs.

Set-Up. We use stochastic block model graphs generated as follows: Each sample graph has three partitions, each with 200 nodes. The probability of the appearance of an edge inside each partition is 0.9, and between partitions is 0.1.

We generate 100 graphs and run PIVOT, R-PIVOT, Narrow-PIVOT, and PRUNED PIVOT on these graphs with parameter R ranging from 2 to 30 (Here R is the parameter k for PRUNED PIVOT). Note that PIVOT does not depend on R. For each R, we take the mean error of these runs for each algorithm (see Figure 2). We remove the standard deviation of the error for figure readability; the std is very similar for all the algorithms for $R > 13$ and is around 2300.

Results. First note that the clustering that puts each partition in one cluster achieves an expected cost of 17970, which is an upper bound on the average optimum value. We selected edge probabilities 0.1 and 0.9 partly to approximate the optimal clustering cost, as computing the exact value is NP-hard. So, PIVOT's approximation factor is at least 2.35 in this case. Furthermore, the trivial clustering that puts each node in one cluster results in 65730 average error which is significantly more than the error of the other algorithms.

We observe that all three PIVOT variants converge fast to the cost of PIVOT. Even though, as a function of R, PRUNED PIVOT has the steadiest improvement in cost, it still converges to the cost of PIVOT exponentially. PRUNED PIVOT queries significantly fewer nodes compared to the other algorithms, albeit at the expense of a negligible increase in its approximation factor. In fact, for $R \geq 4$, the increase in the cost of PRUNED PIVOT compared to PIVOT is less than $\%1$. This property of PRUNED PIVOT makes it flexible in adapting to parallel and dynamic settings while losing a small factor in the approximation guarantee.

B. Illustrations for Query, Extended Query, Expensive, and Dangerous Paths

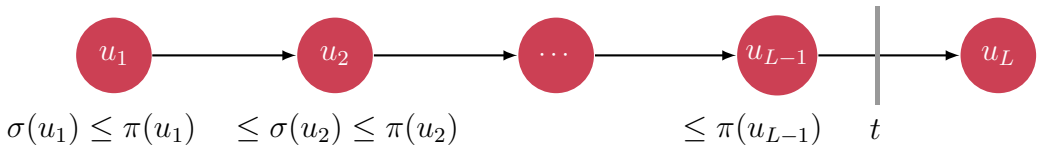

Figure 3. Path $(u_1, \ldots, u_{L-1}, u_L)$ is a query path if each u_i $(i > 1)$ queries u_{i-1}. This condition is equivalent to $\sigma(u_1) \le \pi(u_1) \le \cdots \le \sigma(u_L) \le \pi(u_L)$. Note that $\sigma(u) \le \pi(u)$ for every node $u \in V$. Set $\mathcal{Q}_t(u_1, u_2)$ contains this path for $t \ge \pi(u_{L-1})$.

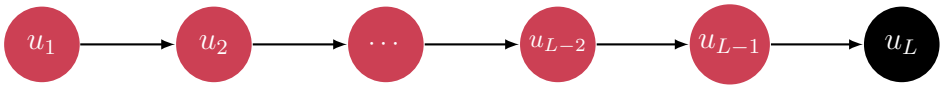

Figure 4. Path $(u_1, \ldots, u_{L-1}, u_L)$ is an extended query path (EQ-path) if each vertex on the path, except for the first and last one, queries the previous vertex and $\pi(u_{L-2}) \le \sigma(u_L)$. This condition is equivalent to $\sigma(u_1) \le \pi(u_1) \le \cdots \le \pi(u_{L-2}) \le \min(\sigma(u_{L-1}), \sigma(u_L))$. The last inequality says that neither u_{L-1} nor u_L is settled before u_{L-2} is processed.

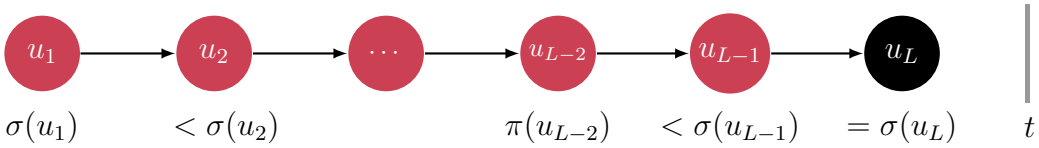

Figure 5. Path $(u_1, \ldots, u_{L-1}, u_L)$ is an expensive extended query path (expensive EQ-path) if (1) each vertex on the path, except for the first and last one, queries the previous vertex, (2) $\sigma(u_1) < \sigma(u_2)$, and (3) $\pi(u_{L-2}) \le \sigma(u_{L-1}) = \sigma(u_L)$. This condition is equivalent to $\sigma(u_1) \le \pi(u_1) < \sigma(u_2) \le \cdots \le \pi(u_{L-2}) \le \sigma(u_{L-1}) = \sigma(u_L)$. Inequality $\sigma(u_1) \le \pi(u_1) < \sigma(u_2)$ tells us that PIVOT places u_1 and u_2 in distinct clusters (i.e., edge (u_1, u_2) is cut by PIVOT). Condition $\sigma(u_{L-1}) = \sigma(u_L)$ tells us that PIVOT places u_{L-1} and u_L in the same cluster. Set $\mathcal{X}_t(u_1, u_2)$ contains this path for $t \ge \sigma(u_{L-1}) = \sigma(u_L)$.

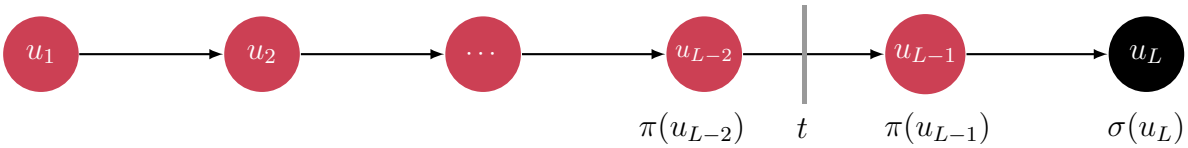

Figure 6. Path $(u_1, \ldots, u_{L-1}, u_L)$ is dangerous at time t if it is an extended query path and $\pi(u_{L-2}) \le t$ but $\pi(u_{L-1}) > t$ and $\sigma(u_L) > t$.

C. Omitted proofs

C.1. Proof of Lemma 4.12

Proof of Lemma 4.12. We remind the reader the definitions of query, extended query, expensive extended query, and dangerous paths in Figures 3, 4, 5, and 6.

I. We show that if P is included in the set of query paths at time $t+1$, then P is dangerous at time t but not at time $t+1$. If $P \in \mathcal{Q}_{t+1}(a,b) \setminus \mathcal{Q}_t(a,b)$, then P is a query path, and $\pi(u_{L-1}) = t+1$. Using condition (2), we get $\pi(u_{L-2}) < \pi(u_{L-1}) = t+1$; $\sigma(u_L) \geq \pi(u_{L-1}) = t+1$. We have $\pi(u_{L-2}) \leq t$ and $\sigma(u_L) \geq \pi(u_{L-1}) > t$. Also, as every query path, P is an extended query path. Thus, P is a dangerous extended query path at time t. However, it is no longer dangerous at time $t+1$, because $\pi(u_{L-1}) = t+1$.

II. Now, we show that if P becomes dangerous at time $t+1$, then its prefix P' is dangerous at time t but not time $t+1$. We first note that the length of L cannot be 1 because paths (a,b) of length 1 are dangerous from time 0 till time $\min(\pi(a), \sigma(b))$ i.e., they are dangerous from the very first iteration of the algorithm and thus cannot become dangerous at some later time $t+1$.

If $P \in \mathcal{D}_{t+1}(a,b) \setminus \mathcal{D}_t(a,b)$, then the following conditions hold: P is an extended query path; $\pi(u_{L-2}) = t+1$; $\pi(u_{L-1}) > t+1$; and $\sigma(u_L) > t+1$. To see how we derived $\pi(u_{L-2}) = t+1$, observe that $P \in \mathcal{D}_{t+1}(a,b)$ implies $\pi(u_{L-2}) \leq t+1$, but the fact that $P \notin \mathcal{D}_t(a,b)$, together with $\pi(u_{L-1}) > t+1$ and $\sigma(u_L) > t+1$, implies $\pi(u_{L-2}) > t$. Since P' is a query path by Definition 4.4, we have $\sigma(u_{L-1}) \geq \pi(u_{L-2}) = t+1$ (see (2)). This shows that $P' \in \mathcal{D}_t(a,b) \setminus \mathcal{D}_{t+1}(a,b)$ if the length of P' is 1, i.e., if $L = 2$. For $L \geq 3$, we need to additionally check that $\pi(u_{L-3}) \leq t$. This is the case because $\pi(u_{L-3}) < \pi(u_{L-2}) = t+1$.

III. Finally, we show that if P is included in the set of expensive paths at time $t+1$, then P or its prefix P' is dangerous at time t but neither of them is dangerous at time $t+1$.

If $P \in \mathcal{X}_{t+1}(a,b) \setminus \mathcal{X}_t(a,b)$, then P is an expensive extended query path and $\sigma(u_{L-1}) = \sigma(u_L) = t+1$. Note that $\pi(u_{L-1}) \geq \sigma(u_{L-1}) = t+1$. Since P' is a query path, we have $\pi(u_{L-2}) \leq \sigma(u_{L-1}) = t+1$. Consider two cases. If $\pi(u_{L-2}) \leq t$, then P is dangerous at time t but not at time $t+1$ (because $\sigma(u_L) = t+1$). If $\pi(u_{L-2}) = t+1$, then P' is dangerous at time t (because $\pi(u_{L-3}) < \pi(u_{L-2}) = t+1$). However, P' is no longer dangerous at time $t+1$ (because $\sigma(u_{L-1}) = t+1$). □

C.2. Proof of Lemma 4.13

Proof of Lemma 4.13. We first analyze process $\Phi_t(a,b)$. By Lemma 4.12, if path P is added to set $\mathcal{Q}_{t+1}(a,b)$ or path Pw becomes dangerous at step $t+1$, then P is dangerous at step t. Hence,

$$\begin{aligned}\Phi_{t+1}(a,b) - \Phi_t(a,b) &= -2|\mathcal{D}_t(a,b) \setminus \mathcal{D}_{t+1}(a,b)| + 2|\mathcal{D}_{t+1}(a,b) \setminus \mathcal{D}_t(a,b)| + |\mathcal{Q}_{t+1}(a,b) \setminus \mathcal{Q}_t(a,b)| = \\ &= \sum_{P \in \mathcal{D}_t(a,b)} -2 \cdot \underbrace{\mathbf{1}\{P \notin \mathcal{D}_{t+1}(a,b)\}}_{\text{no longer dangerous paths}} + \underbrace{\mathbf{1}\{P \in \mathcal{Q}_{t+1}(a,b)\}}_{\text{new query paths}} + 2 \sum_{w \in V} \underbrace{\mathbf{1}\{Pw \in \mathcal{D}_{t+1}(a,b)\}}_{\text{new dangerous paths}}.\end{aligned}$$

We show that the conditional expectation of every term in the sum above given $\mathcal{F}_t$ is non-positive. Consider a path $P = (u_0, \ldots, u_L) \in \mathcal{P}(a,b)$. Let

$$\Delta_{t+1}(P) = -2 \cdot \mathbf{1}\{P \notin \mathcal{D}_{t+1}(a,b)\} + \mathbf{1}\{P \in \mathcal{Q}_{t+1}(a,b)\} + 2 \sum_{w \in V} \mathbf{1}\{Pw \in \mathcal{D}_{t+1}(a,b)\}.$$

We need to show that

$$\mathbf{E}\Big[\Delta_{t+1}(P) \mid \mathcal{F}_t, P \in \mathcal{D}_t(a,b)\Big] \leq 0.$$

Note that if $P \in \mathcal{D}_t(a,b)$ and $P \in \mathcal{D}_{t+1}(a,b)$, then $\Delta_{t+1}(P) = 0$, because $\mathbf{1}\{P \in \mathcal{Q}_{t+1}(a,b)\} = 0$ and $\mathbf{1}\{Pw \in \mathcal{D}_{t+1}(a,b)\}$ (for all w) by Lemma 4.12; while $\mathbf{1}\{P \notin \mathcal{D}_{t+1}(a,b)\} = 0$ since $P \in \mathcal{D}_{t+1}(a,b)$. Thus, it suffices to prove that $\mathbf{E}\Big[\Delta_{t+1}(P) \mid \mathcal{F}_t, P \in \mathcal{D}_t(a,b), P \notin \mathcal{D}_{t+1}(a,b)\Big] \leq 0$. Let $W_t = \big\{w \in N(u_L) : \sigma(w) > t\big\}$ be the set of neighbors of u_L that are not settled by iterations t. Note that P stops being dangerous at iteration $t+1$ only if u_{L-1} is processed or u_L is

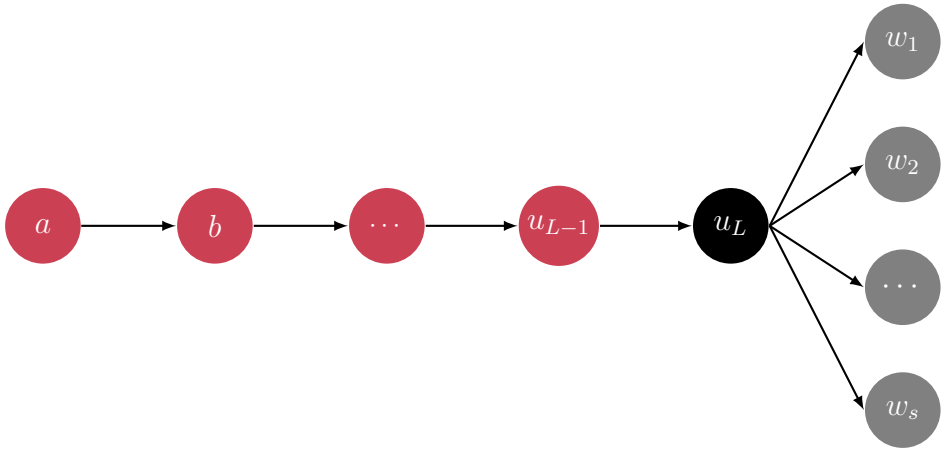

Figure 7. Illustration for the proof of Theorem 4.13. Path $(a, b, \ldots, u_{L-1}, u_L)$ is a dangerous EQ-path at iteration t. At iteration $t+1$, it may become a query path and/or an expensive EQ path. It may also get extended to EQ-paths Pw, where $w \in W_t \setminus \{u_{L-1}, u_L\}$. These extended paths Pw may be dangerous or expensive at iteration $t+1$, but they also may be non-dangerous and non-expensive at iteration $t+1$.

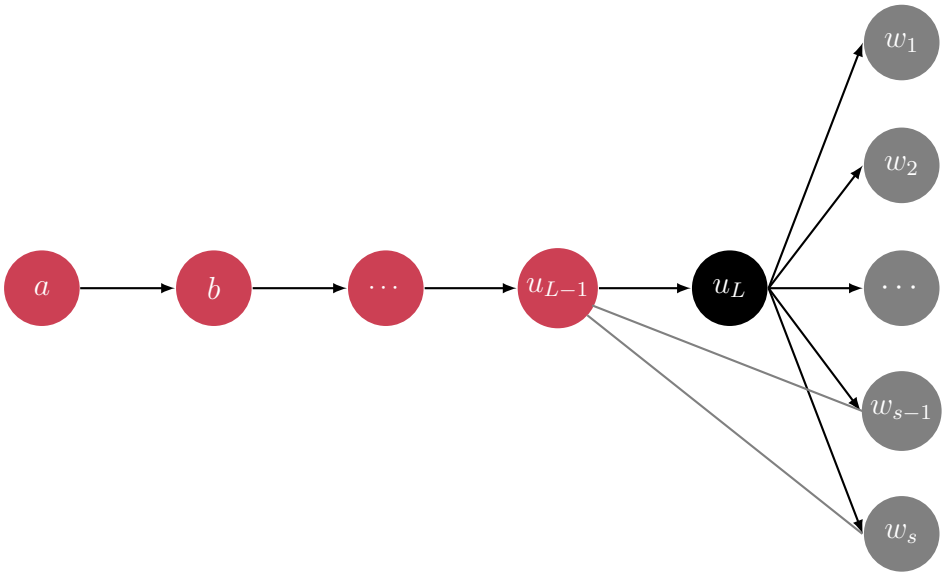

Figure 8. Illustration for the proof of Theorem 4.13. Path $(a, b, \ldots, u_{L-1}, u_L)$ is a dangerous EQ-path at iteration t. Set $W_t^{(1)}$ contains nodes in W_t that are not neighbors of u_{L-1}. $W_t^{(2)}$ contains nodes in W_t that are neighbors of u_{L-1}.

settled at iteration $t+1$. The latter event occurs if and only if one of the nodes in W_t is processed at iteration $t+1$. Hence,

$$\mathbf{E}\Big[\Delta_{t+1}(P) \mid \mathcal{F}_t, P \in \mathcal{D}_t(a,b), P \notin \mathcal{D}_{t+1}(a,b)\Big] = \mathbf{E}\Big[\Delta_{t+1}(P) \mid \mathcal{F}_t, P \in \mathcal{D}_t(a,b), \pi^{-1}(t+1) \in W_t \cup \{u_{L-1}\}\Big].$$

Here, $\pi^{-1}(t+1)$ is the node processed at iteration $t+1$. Observe that $\pi^{-1}(t+1)$ is uniformly distributed in $W_t \cup \{u_{L-1}\}$ given that $\pi^{-1}(t+1) \in W_t \cup \{u_{L-1}\}$. Now, if $\pi^{-1}(t+1) = u_{L-1}$, then $\pi(u_{L-1}) = t+1$, and path P is added to $\mathcal{X}_{t+1}(a,b)$ (it is an expensive path because condition (2) is satisfied). Moreover, if $\pi^{-1}(t+1) = u_{L-1}$, then some or all paths Pw, where $w \in W_t \setminus \{u_{L-1}, u_L\}$, are added to the set of dangerous paths $\mathcal{D}_{t+1}(a,b)$. Note that no path Pw' with $w' \notin W_t \setminus \{u_{L-1}, u_L\}$ is added to $\mathcal{D}_{t+1}(a,b)$ since for $w' \in N(u_L) \setminus W_t$ and $w' \notin \{u_{L-1}, u_L\}$, we have $\sigma(w') \le t$, but for Pw in $\mathcal{D}_{t+1}(a,b)$, we must have $\sigma(w) > t$. Thus, if $\pi^{-1}(t+1) = u_{L-1}$, then $\Delta_{t+1}(P) \le -2 + 1 + (|W_t| - 1)$ (here, we use that u_L always belongs to W_t if $P \in \mathcal{D}_t(a,b)$). If, however, $\pi^{-1}(t+1) \in W_t \setminus \{u_{L-1}\}$, then $\Delta_t(a,b) = -2$, because (1) P does not become a query path, and (2) P is not extended to any dangerous paths at iteration $t+1$. We obtain the following bound

$$\mathbf{E}\Big[\Delta_{t+1}(P) \mid \mathcal{F}_t, P \in \mathcal{D}_t(a,b), \pi^{-1}(t+1) \in W_t \cup \{u_{L-1}\}\Big] \le -2 + \frac{1 + 2(|W_t| - 1)}{|W_t \cup \{u_{L-1}\}|} < 0.$$

This proves that $\Phi_t(a,b)$ is a supermartingale. We now show that $\Psi_t(a,b)$ is also a supermartingale. Using Lemma 4.12, we get

$$\Psi_{t+1}(a,b) - \Psi_t(a,b) = \sum_{P \in \mathcal{D}_t(a,b)} \Delta'(P),$$

where

$$\Delta'_t(P) = -2 \underbrace{\mathbf{1}\{P \notin \mathcal{D}_{t+1}(a,b)\}}_{\text{no longer dangerous paths}} + 2 \underbrace{\sum_{w \in V} \mathbf{1}\{Pw \in \mathcal{D}_{t+1}(a,b)\}}_{\text{new dangerous paths}} + \underbrace{\mathbf{1}\{P \in \mathcal{X}_{t+1}(a,b)\} + \sum_{w \in V} \mathbf{1}\{Pw \in \mathcal{X}_{t+1}(a,b)\}}_{\text{new expensive paths}}.$$

As before, it suffices to show that for every $P \in \mathcal{P}(a,b)$, we have

$$\mathbf{E}\Big[\Delta'_{t+1}(P) \mid \mathcal{F}_t, P \in \mathcal{D}_t(a,b), P \notin \mathcal{D}_{t+1}(a,b)\Big] \le 0.$$

From the previous argument, we know that $\pi^{-1}(t+1)$ is uniformly distributed in $W_t \cup \{u_{L-1}\}$ given $\mathcal{F}_t$, $P \in \mathcal{D}_t(a,b)$, and $P \notin \mathcal{D}_{t+1}(a,b)$. We now consider two cases: u_{L-1} is already settled by iteration t (then $u_{L-1} \notin W_t$) and u_{L-1} is not yet settled (then $u_{L-1} \in W_t$).

Case 1: $\sigma(u_{L-1}) \le t$. In this case, P may never become an expensive EQ-path, since $\sigma(u_{L-1}) \le t$, but $\sigma(u_L) > t$. Moreover, if u_{L-1} is processed at iteration $t+1$, then it will not be marked as a pivot, and, consequently, no nodes will be settled at iteration $t+1$. In particular, if P is extended to some EQ-path Pw, then this path will not be added to the set of expensive EQ-paths $\mathcal{X}_{t+1}(a,b)$ (even though it may eventually be added to some set $\mathcal{X}_{t'}(a,b)$ with $t' > t+1$). However (if u_{L-1} is processed at iteration $t+1$), every path Pw with $w \in W_t \setminus \{u_L\}$ will be added to the set of dangerous EQ-paths $\mathcal{D}_{t+1}(a,b)$. If another node in W_t – not u_{L-1} – is processed at iteration $t+1$, then P is not extended to any other paths. Denote the event $\{P \in \mathcal{D}_t(a,b) \setminus \mathcal{D}_{t+1}(a,b)\}$ by $\mathcal{E}_t$. Then,

$$\mathbf{E}\Big[\Delta'_{t+1}(P) \mid \mathcal{F}_t, \mathcal{E}_t, \sigma(u_{L-1} \le t)\Big] = -2 + 2\frac{|W_t| - 1}{|W_t| + 1} < 0.$$

Case 2: $\sigma(u_{L-1}) > t$. Define two disjoint subsets of W_t: $W_t^{(1)} = W_t \setminus N(u_{L-1})$ and $W_t^{(2)} = W_t \cap N(u_{L-1}) \setminus \{u_{L-1}, u_L\}$. Then, W_t is the disjoint union of three sets: $W_t^{(1)}$, $W_t^{(2)}$, and $\{u_{L-1}, u_L\}$. If u_{L-1} is processed at iteration $t+1$, then (at this iteration) u_{L-1} is marked as a pivot and all nodes in $W_t^{(2)}$ as well as node u_L are settled. Consequently, path P and all paths Pw with $w \in W_t^{(2)} \setminus \{u_L\}$ are added to the set of expensive paths $\mathcal{X}_{t+1}(a,b)$. No other extensions of P are added to this set. All paths Pw with $w \in W_t^{(1)}$ are added to the set of dangerous paths $\mathcal{D}_{t+1}(a,b)$. However, paths Pw with $w \in W_t^{(2)}$ are not dangerous by Lemma 4.12. Hence, if u_{L-1} is processed at iteration $t+1$, then $\Delta'_{t+1}(P) = -2 + 2|W_t^{(1)}| + (|W_t^{(2)}| + 1)$. If u_L or any node in $W_t^{(2)}$ is processed at time $t+1$, then path P becomes expensive, but it is not extended to any other

EQ-path (since $\sigma(u_L) = t+1 < \pi(u_{L-1})$ and, as a result, P is not a query path). Hence, $\Delta'_{t+1} = -2+1$. Finally, if a node in $W_t^{(1)}$ is processed, then P is not extended to any other paths, and P does not become expensive. Thus, $\Delta'_{t+1} = -2$. Therefore,

$$\begin{aligned}\mathbf{E}\Big[\Delta'_{t+1}(P) \mid \mathcal{F}_t, \mathcal{E}_t, \sigma(u_{L-1}) > t)\Big] &= -2 + \frac{2|W_t^{(1)}| + |W_t^{(2)}| + 1}{|W_t|} + \frac{|W_t^{(2)}| + 1}{|W_t|} \\ &= -2 + \frac{2(|W_t^{(1)}| + |W_t^{(2)}|) + 2}{|W_t^{(1)}| + |W_t^{(2)}| + 2} < 0.\end{aligned}$$

This completes the proof of Lemma 4.13. □

D. LCA

Given a node v, our LCA algorithm simply simulates PRUNED PIVOT. The probe complexity of such an approach is almost direct. Namely, the algorithm visits at most k nodes via recursive calls. Each call scans neighbors of the corresponding node to find the k-top ones. That scan takes $O(\Delta)$ probes. Therefore, the probe complexity is $O(\Delta \cdot k)$.

It remains to analyze the space complexity and, if any, its effect on the approximation.

Random ordering. Our algorithm assumes it has access to a random node-permutation π. However, it is unclear how to obtain π in LCA. So, instead, each node v draws an integer rank $r_v \in [0, n^{10})$ uniformly at random. If values r_v are drawn independently of each other, then, with probability at least $1 - n^{-5}$, for each $u \neq v$ it holds $r_u \neq r_v$. Therefore, the values r_v implicitly define a random permutation, which is enough to simulate our algorithm. (For now, assume that $r_u \neq r_v$, and at the end of this section, we discuss how to handle the case when $r_u = r_v$ for two nodes.)

Source of randomness. If the algorithm has access to an arbitrary long tape of random bits with random access, then whenever a node wants to learn r_v, it reads $10 \log n$ bits starting at position $v \cdot 10 \log n$. If such tape is not accessible, then the corresponding local computation algorithm has to store random bits in its memory. It is obvious how to keep all the required random bits in $O(n \log n)$ memory; a node v uses $O(\log n)$ bits independent of other nodes to obtain r_v. However, we show that substantially fewer bits suffice for small Δ. To that end, we first recall the definition of w-wise independence hash functions and a folklore result about its construction.

Definition D.1 (w-wise independent hash functions)**.** *Let $w, b, N \in \mathbb{N}$ and s be a seed of independent random bits. A function $h_s : \{0,1\}^N \to \{0,1\}^b$ is a called w-wise independent hash function if for any $I \leq w$, all distinct $x_1, \ldots, x_I \in \{0,1\}^N$ and all distinct $y_1, \ldots, y_I \in \{0,1\}^b$ it holds*

$$\Pr\left(\bigwedge_{i=1}^{I} h_s(x_i) = y_i\right) = 2^{-I \cdot b}.$$

Theorem D.2 (Folklore)**.** *Let $w, b, N \in \mathbb{N}$. There exists a w-wise independent hash function $h_s : \{0,1\}^N \to \{0,1\}^b$ with a seed s of length $w \cdot \max\{N, b\}$. Moreover, h_s can be stored using $O(w \cdot (N+b))$ bits of space.*

Let $h : V \to \{0, 1, 2, \ldots, n^{10} - 1\}$ be a w-wise independent hash function; we will set w in the remainder of this section. Consider an LCA execution of our algorithm from v that, instead of randomly pre-generated r-values, uses h; let $\mathcal{A}_{\mathbf{LCA}}$ refer to that algorithm. That is, whenever $\mathcal{A}_{\mathbf{LCA}}$ needs $\pi(u)$ it uses $h(u)$ instead. So, invoking $\mathcal{A}_{\mathbf{LCA}}$ on v, the algorithm has to learn the rank-ordering of the neighbors of v. To achieve that, $\mathcal{A}_{\mathbf{LCA}}$ evaluates $h(u)$ on each $u \in N(v)$.

In the process, $\mathcal{A}_{\mathbf{LCA}}$ invokes h at most $\Delta \cdot k$ many times. Hence, if $w \geq \Delta \cdot k$, each invocation of h is entirely independent of the previous invocations of h. Therefore, *from the point of view of* v, the execution $\mathcal{A}_{\mathbf{LCA}}(v)$ is equivalent to that of executing $\mathcal{A}_{\mathbf{LCA}}$ with r values pre-generated using $O(n \log n)$ bits of randomness.

Does $\mathcal{A}_{\mathbf{LCA}}$ executed *on all* $v \in V$ resemble our algorithm? Very likely, no! Namely, for $w < n$, we should expect lots of dependencies between execution trees for some, possibly far away, nodes. Nevertheless, we will argue that, in expectation, the cost of $\mathcal{A}_{\mathbf{LCA}}$ and our main algorithm are the same for a proper value of w. To that end, in our main algorithm, let $c_{u,v}$ be the expected cost that pair $\{u, v\}$ incurs. That is, if $\{u, v\}$ is an edge, then $c_{u,v}$ is the probability that our main algorithm

cuts it. If $\{u, v\}$ is not an edge, then $c_{u,v}$ is the probability that our main algorithm does not cut. In particular, the expected cost of our main algorithm is $\sum_{u,v \in V \times V} c_{u,v}$.

Consider $\{u, v\} \in V \times V$. Then, $\mathcal{A}_{\mathbf{LCA}}$ behaves the same with respect to $\{u, v\}$ if $\mathcal{A}_{\mathbf{LCA}}(v)$ and $\mathcal{A}_{\mathbf{LCA}}(u)$ together resemble the execution of our main algorithm for v and u. This can be achieved by ensuring that the randomness used for $\mathcal{A}_{\mathbf{LCA}}(u)$ and $\mathcal{A}_{\mathbf{LCA}}(v)$ is independent, which is achieved for $w \geq 2 \cdot \Delta \cdot k$.

In this setup, we also have that in the execution tree for v, with probability $1 - n^{-5}$ at least, no two ranks are the same. Taking the union bound over all $v \in V$, with probability at least $1 - n^{-4}$, no execution tree has two same ranks.

The case when $r_u = r_v$ for $u \neq v$ in an execution tree. As argued above, this case happens with probability at most n^{-4}. We distinguish two cases: when the input graph is a union of cliques and when it is not. In the latter case, the optimum value is at least 1. Since $r_u = r_v$ happens with probability at most n^{-4} and the maximum possible cost of a clustering is n^2, we have that the expected cost in this case is at most $(3 + \varepsilon)OPT + 1/n^2 \leq (3 + \varepsilon + 1/n^2)OPT$.

Now, consider the case when the input graph is a union of cliques. In this case, it holds $OPT = 0$, and an additive cost of $1/n^2$ does not yield a $(3 + \varepsilon)$ multiplicative one. To handle this case, we add a rule for breaking ties in ranks to our algorithm. Namely, if a node x has two neighbors u and v such that $r_u = r_v$, then our algorithm ranks u before v iff $u < v$; $u < v$ means that the label of u is smaller than the label of v. Since each node sees the same neighborhood in a clique – this neighborhood includes the node itself – then each node chooses the same pivot even if $r_u = r_v$ for two distinct nodes. So, each clique is clustered correctly.

E. MPC

E.1. Maintaining $N_k(u)$

This section describes how to dynamically maintain $N_k(u)$ with $O(\log k)$ update time in expectation. To maintain $N_k(u)$, as the first step, the algorithm organizes the neighbors of u as we describe next. At each algorithm step, we associate $\tilde{d}_u$ with u. In particular, $\tilde{d}_u$ is such that the current degree of u is in the range $(1/4 \cdot \tilde{d}_u, 4 \cdot \tilde{d}_u)$. In other words, $\tilde{d}_u$ is a 4 approximation of the degree of u. Moreover, all the neighbors of u are placed in $b_u = \lceil \tilde{d}_u/(80k) \rceil$ buckets numbered $1, \ldots, b_u$. A neighbor w of u is placed in the bucket j such that $(j-1) \cdot n/b_u < \pi(w) \leq j \cdot n/b_u$. Hence, a bucket corresponds to n/b_u consecutive integers. This is convenient as if u has $d(u)$ neighbors, then under randomly chosen π it holds that in expectation $d(u)/n \cdot n/b_u = d(u)/b_u$ neighbors are in a given bucket. With at least two buckets, this ratio is in the range $(10k, 320k)$. When there is a single bucket only, it has at most $320k$ elements by definition of b_u and $\tilde{d}_u$.

Each bucket is organized as an ordered balanced binary tree. Therefore, if a bucket contains t elements, then insertion, deletion, and finding the i-th-rank node can be done in $O(\log t)$ time. We use $\mathcal{B}_u$ to refer to these buckets for node v. We note that the number of buckets might change over time. We discuss that towards the end of this section.

Edge insertion. If a new edge $\{u, x\}$ is inserted, then the algorithm adds x to the bucket in $\mathcal{B}_u$ corresponding to $\pi(x)$. In expectation, that bucket has $\Theta(k)$ nodes. Hence, this operation is done in $O(\log k)$ time in expectation. Our algorithm also checks whether x has a higher rank than the lowest rank node in $N_k(u)$. If that is the case, it removes the smallest-rank node from $N_k(u)$ and inserts x. This is done in $O(\log k)$ time.

Edge deletion. If an edge $\{u, x\}$ is deleted, then the algorithm first removes x from the bucket corresponding to $\pi(x)$. In expectation, that bucket has $\Theta(k)$ nodes. Hence, by the concavity of the $\log$ function, this operation is done in $O(\log k)$ time in expectation. Second, if x does not belong to $N_k(u)$, then the algorithm does nothing else. Otherwise, the algorithm removes x from $N_k(u)$ and finds the k-th highest-ranked element in $\mathcal{B}_u$. It does so in the following way: it visits bucket by bucket in the decreasing order of ranks until it reaches a bucket containing the desired element. We now analyze the complexity of this search.

Let Y be a random variable representing the cost of this search. Let Y_B be the time spent searching bucket B; we count 1 even if B is accessed but empty. Then, $Y = \sum_{B \in \mathcal{B}_u} Y_B$. Let B_j denote the j-th bucket in $\mathcal{B}_u$. Let Z_i be the event that the buckets $B_1 \ldots B_i$ contain less than k elements in total. We have

$$\mathbf{E}\left[Y_{B_j}\right] = \mathbf{E}\left[Y_{B_j} \mid Z_{j-1}\right] \cdot \Pr\left(Z_{j-1}\right) + \mathbf{E}\left[Y_{B_j} \mid \neg Z_{j-1}\right] \cdot \Pr\left(\neg Z_{j-1}\right).$$

Observe that $\mathbf{E}\left[Y_{B_j} \mid \neg Z_{j-1}\right] = 0$, as no search is performed on B_j if the buckets $B_1 \ldots B_{j-1}$ contain at least k elements.

This effectively implies that

$$\mathbf{E}\left[Y_{B_j}\right] = \mathbf{E}\left[Y_{B_j} \mid Z_{j-1}\right] \cdot \Pr\left(Z_{j-1}\right). \tag{5}$$

Also, we have that

$$\mathbf{E}\left[|B_j| \mid Z_{j-1}\right] \in O(\tilde{d}_u/(b_u - (j-1))).$$

Hence,

$$\mathbf{E}\left[Y_{B_j} \mid Z_{j-1}\right] \in O(1 + \log\left(\tilde{d}_u/(b_u - (j-1))\right)). \tag{6}$$

Next, we upper-bound $\Pr\left(Z_{j-1}\right)$. Definition of Z_i implies

$$\Pr\left(Z_i\right) \leq \Pr\left(|B_i| < k \mid Z_{i-1}\right) \cdot \Pr\left(Z_{i-1}\right) \tag{7}$$

$$= \prod_{t=1}^{i} \Pr\left(|B_t| < k \mid Z_{t-1}\right). \tag{8}$$

For $i = 1$, we have $\mathbf{E}\left[|B_1| \mid Z_0\right] \geq d(u)/b_u \geq 10k$; the latter inequality follows by our discussion above. For $i > 1$, we have

$$\mathbf{E}\left[|B_i| \mid Z_{i-1}\right] \geq \frac{d(u) - k}{b_u - (i-1)} \geq \mathbf{E}\left[|B_1| \mid Z_0\right].$$

The latter inequality can be easily verified algebraically, but also it is an easy observation that it holds as the first $i-1$ buckets in expectation contain at least $10k(i-1)$. Hence, $d(u) - k$ nodes distributed over $b_u - (i-1)$ buckets yield the bucket-average higher than $10k$.

Condition on Z_{i-1}. Then, let X_w be a $0/1$ random variable that equals 1 if and only if the neighbor w of u is in B_i; in particular, $\mathbf{E}\left[|B_i| \mid Z_{i-1}\right] = \mathbf{E}\left[\sum_{w \in N(u)} X_w \mid Z_{i-1}\right]$. Observe that the random variables X are negatively correlated. Hence, we can apply Chernoff bound to upper-bound the probability that $|B_i| < k$. Since $\mathbf{E}\left[|B_i| \mid Z_{i-1}\right] \geq 10k$, we have that

$$\Pr\left(|B_i| < k \mid Z_{i-1}\right) \leq e^{-k}.$$

Plugging this into Equation (7), we derive

$$\Pr\left(Z_i\right) \leq e^{-ik}. \tag{9}$$

We now turn back to computing $\mathbf{E}\left[Y\right]$. By plugging Equation (6) and Equation (9) into Equation (5), we obtain

$$\mathbf{E}\left[Y\right] = \sum_{j=1}^{b_u} \mathbf{E}\left[Y_{B_j}\right] \tag{10}$$

$$\leq \sum_{j=1}^{b_u} e^{-k(j-1)} \cdot O\left(1 + \log\left(\tilde{d}_u/(b_u - (j-1))\right)\right).$$

To upper bound Equation (10), we first let $t_u = \tilde{d}_u/b_u$. By definition, $t_u \in O(k)$. Observe that

$$b_u \leq (b_u - (j-1)) \cdot j$$

when j ranges in the interval $[1, b_u]$; the minimum is achieved for $j = 1$ and $j = b_u$. This implies that

$$\tilde{d}_u/(b_u - (j-1)) = t_u \cdot b_u/(b_u - (j-1)) \leq t_u \cdot j.$$

From Equation (10), this yields the upper-bound

$$\begin{aligned}\mathbf{E}[Y] &\leq O\left(\sum_{j=1}^{b_u} e^{-k(j-1)} \cdot (1+\log(kj))\right)\\ &= O(1) + O\left(\sum_{j=1}^{b_u} e^{-k(j-1)} \log k\right)\\ &\quad + O\left(\sum_{j=1}^{b_u} e^{-k(j-1)} \log j\right)\end{aligned}$$

$$\begin{aligned}&\leq O(1) + O(\log k) + O\left(\sum_{j=1}^{b_u} e^{-j} j\right)\\ &\leq O(1) + O(\log k) + O(1)\\ &= O(\log k),\end{aligned}$$

as desired. To upper-bound $O\left(\sum_{j=1}^{b_u} e^{-j} j\right)$ we used that it holds $\sum_{i=1}^{\infty} i/2^i = 2$.

Updating $\mathcal{B}_u$ when $d_u/\tilde{d}_u \notin (1/4, 4)$. We first describe how to address this change in $O(\log k)$ amortized expected time and then explain how to de-amortize this.

When d_u becomes $\tilde{d}_u/4$ or $4\tilde{d}_u$, then our algorithm updates the current $\tilde{d}_u$ to $\tilde{d}'_u$, and re-creates $\mathcal{B}_u$ for $\tilde{d}'_u$. If $d_u \leq \tilde{d}_u/4$, then $\tilde{d}'_u \leftarrow \tilde{d}_u/2$. Similarly, if $d_u \geq 4\tilde{d}_u$, then $\tilde{d}'_u \leftarrow 2\tilde{d}_u$. This approach is standard and typically illustrated through the example of dynamic arrays. For our problem, this technique yields amortized $O(\log k)$ expected update time.

If memory allocation takes $O(1)$ time, this technique can be de-amortized. This de-amortization is standard, but we provide a couple of sentences of explanation for the sake of completeness. Instead of creating all the buckets from scratch when $d_u = \tilde{d}_u/4$ or $d_u = 4\tilde{d}_u$, a de-amortized algorithm does that gradually. That is, as soon as it starts updating the buckets for the current $\tilde{d}_u$ value, it allocates the memory for buckets for both $\tilde{d}'_u = \tilde{d}_u/2$ and $\tilde{d}'_u = 2\tilde{d}_u$; only an allocation is performed, without any initialization. The algorithm also maintains a variable $\mathrm{ID}_{\text{so-far}}$, which, when $\tilde{d}_u$ is updated, is initialized to 0.

On a new neighbor w update, if $\pi_w \leq \mathrm{ID}_{\text{so-far}}$, the algorithm carries over that update for all three bucket structures. If $\pi_w > \mathrm{ID}_{\text{so-far}}$, the algorithm only updates the $\tilde{d}_u$-buckets. In addition, the algorithm considers 10 elements from the $\tilde{d}_u$-buckets with smallest π but greater than $\mathrm{ID}_{\text{so-far}}$ values, and copies them to the $(\tilde{d}_u/2)$- and $(2\tilde{d}_u)$-buckets. The value of $\mathrm{ID}_{\text{so-far}}$ is increased properly so to correspond to the last element copied from $\tilde{d}_u$-buckets.

E.2. Running Time for our Dynamic Algorithm

Maintaining $Q_P^{-1}(u)$. For each $u \in Q_P(w)$, updating double-linked list $Q_P^{-1}(u)$ is done in $O(1)$ time – appending takes $O(1)$ time, while the removal also takes $O(1)$ by using the pointers stored in $Q_P(w)$.

Now we upper-bound the expected running time of PRUNED PIVOT-UPDATE(a, b). In that, we use the following claim.

Lemma E.1. *The number of nodes in $Q_P^{-1}(b)$ **processed** by Algorithm 6 within the **if** condition is $O(1)$ in expectation.*

Proof. If b queries (b, a), then each node that queries b also queries (b, a). Hence, the number of nodes in $Q_P^{-1}(b)$ equals the number of nodes querying (b, a). By Theorem 4.10, that number in expectation is at most 2. Observe that Theorem 4.10 provides an upper-bound for PIVOT. Hence, PRUNED-CLUSTER might query (b, a) only less frequently. □

Lemma E.2. PRUNED PIVOT-UPDATE(a, b) *takes $O(k)$ amortized time in expectation per an edge update.*

Proof. We show that maintaining $N_k(b)$ takes $O(\log k)$ amortized time in expectation in Appendix E.1.

By Lemma E.1, in expectation, only $O(1)$ nodes w are processed within the **if** condition. In addition to them, b is also processed. For each such w or b the following is performed:

- Invocation of PRUNED-CLUSTER. This invocation visits $O(k)$ nodes. To traverse neighbors of a node u, PRUNED-CLUSTER uses $N_k(u)$. Even though $N_k(u)$ is organized as a binary balanced tree, all its nodes can be traversed in the rank-decreasing order in $O(1)$ time per node.
- Updating $Q_P^{-1}(u)$ for $u \in Q_P(w)$. As described above, this is done in $O(1)$ per u. By design of PRUNED PIVOT, we have $|Q_P(w)| \leq k$.

□

F. Equivalence between PIVOT and Sequential PIVOT

In Algorithm 7 we recall the PIVOT algorithm from (Ailon et al., 2008).

Algorithm 7 PIVOT

```
1: function PIVOT(G = (V, E))
2:     if V = ∅:
3:         terminate
4:     Pick a random pivot u ∈ V.
5:     Cluster together u and its neighbors.
6:     Let H be obtained by removing u and its neighbors from G.
7:     PIVOT(H)
```

Lemma F.1. *Algorithm 7 and Algorithm 3 are equivalent*

Proof. Instead of choosing a random pivot u each time on Line 4 of PIVOT (Algorithm 7), we assume that before the algorithm is invoked a random permutation π over the input nodes is chosen. Then, Line 4 is implemented by choosing the first, with respect to π, node available in V in that invocation of PIVOT.

We prove the lemma by induction on π. First, consider the node with rank 1, i.e., let $\pi(u) = 1$. u is a pivot in PIVOT, and since u has no higher ranked neighbors, it executes line 8 in SEQUENTIAL PIVOT and hence is a pivot there as well.

Suppose that for some $t \geq 1$, both PIVOT and SEQUENTIAL PIVOT cluster all nodes with ranks $1, \ldots, t$ the same way. Suppose $\pi(u) = t + 1$. First, suppose that u is a pivot in PIVOT. Then it must be that u does not have any pivot neighbor with a higher rank: if there is such node v, then when v is being clustered, u is put in the cluster of v in Line 5 of Algorithm 7. This means that in SEQUENTIAL PIVOT u is also a pivot.

Now, suppose that u is not a pivot in PIVOT. Let v be the pivot of u in PIVOT. Then it must be that v is the highest ranked pivot in the neighborhood of u: if there is a higher ranked pivot v' in the neighborhood of u, then when v' is being processed (before v), u is put in the cluster of v' in Line 5 of Algorithm 7. This means that in SEQUENTIAL PIVOT, when u is being processed, Line 7 is executed when the neighbor v of u is picked. □

G. CRCW PRAM Algorithm

We now describe how to execute our approach in CRCW PRAM in $O(1/\varepsilon)$ rounds using $O(n^3)$ processors. We assume that the nodes are numbered 1 through n. The same as discussed in Appendix D, instead of finding a random permutation of the nodes, each node chooses an integer rank from range $[0, n^{10})$. That is, in the shared memory, there is a designated space of n memory blocks, each of size $10 \log n$ bits, such that the i-th of those memory blocks stores the rank of node i.

Assume that each node has the sorted list of its k highest-ranked neighbors, where $k = O(1/\varepsilon)$ is the parameter used in Algorithm 2. Then, PRUNED PIVOT can be directly executed in $O(k) = O(1/\varepsilon)$ rounds in the CRCW PRAM model. It remains to discuss how to find k highest-ranked neighbors of each node in $O(k)$ rounds.

Finding top-k neighbors in $O(k)$ rounds. We assume that the neighbors of each node are given as an array. It is well-known how to find the maximum of an array in $O(1)$ CRCW PRAM rounds using $O(n^2)$ processors (JáJá, 1992; Akl, 1989); we use ARRAYMAX to refer to that algorithm. To find k highest-ranked neighbors of each node, we apply ARRAYMAX k times to each neighborhood array for all the nodes in parallel. The i-th of these invocations finds the i-th highest-ranked neighbors of the nodes. After the i-th highest-ranked neighbor w_i of a node v is found, w_i is marked with a special symbol in the neighborhood array of v, implying that in the future invocations of ARRAYMAX, w_i is considered smaller than every other non-marked neighbor of v.

Since ARRAYMAX uses $O(n^2)$ processors and is executed for each of the nodes separately, this PRAM implementation of Algorithm 2 uses $O(n^3)$ processors.