Noname manuscript No.
(will be inserted by the editor)

Constant-Time Local Computation Algorithms

Yishay Mansour · Boaz Patt-Shamir · Shai Vardi

Abstract Local computation algorithms (LCAs) produce small parts of a single (possibly approximate) solution to a given search problem using time and space sublinear in the size of the input. In this work we present LCAs whose time complexity (and usually also space complexity) is independent of the input size. Specifically, we give (1) a $(1-\epsilon)$-approximation LCA to the maximum weight acyclic edge set, (2) LCAs for approximating multicut and integer multicommodity flow on trees, and (3) a local reduction of weighted matching to any unweighted matching LCA, such that the running time of the weighted matching LCA is d times (where d is the maximal degree) the running time of the unweighted matching LCA, (and therefore independent of the edge weight function).

1 Introduction

Local computation algorithms (LCAs) provide a solution to situations in which we require fast and space-efficient access to part of a solution to a large computational problem, but we never need the entire solution at once. Consider, for instance, a database describing a network with millions of nodes and edges, on which we would like to compute a maximal matching. At any point in time, the algorithm may be queried about an edge, and is expected to reply "yes"

Y. Mansour
Tel Aviv University and Microsoft Research, Herzliya. E-mail: mansour@post.tau.ac.il. Supported in part by a grant from the Israel Science Foundation, by a grant from United States-Israel Binational Science Foundation (BSF), by a grant from the Israeli Ministry of Science (MoS) and the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11).

B. Patt-Shamir
Tel Aviv University. E-mail: boaz@tau.ac.il. Supported in part by the Israel Science Foundation (grant No. 1444/14) and by the Israel Ministry of Science and Technology.

S. Vardi
California Institute of Technology. E-mail: shaivardi@gmail.com. Supported in part by the Google Europe Fellowship in Game Theory.

or "no", depending on whether the edge is part of a maximal matching. The algorithm may never be required to compute the entire solution. However, replies to queries are expected to be consistent with a single matching.

LCA measures. Typically, LCAs use polylog(n) space, and are required to reply to each query in polylog(n) time.[1] Some papers on LCAs (e.g., [21]) give three criteria for measuring LCAs: running time per query, the total space required, and failure probability. Others (e.g., [4]) consider only the number of times the input is probed, and the amount of information the LCA needs to store between queries.

In this paper we propose a more comprehensive model that unifies the two approaches. The idea is to distinguish between computational time and probe complexities, and between enduring and transient memory. More specifically, a *probe* to the input graph consists of asking a vertex for a list of its neighbors.[2] Regarding space, we assume that there is an *enduring memory*, which is written only once by the algorithm, before the first query is presented. We think about it as an augmentation of the input. Enduring memory is useful in randomized LCAs (e.g., [2,13,21]), where the LCA must use the same randomness each time it is invoked to ensure consistency. This can be done by storing a random seed in the enduring memory. *Transient memory* is simply the memory required to compute a reply to each query. Note that as the enduring memory is read-only, the algorithm's reply to a query depends only on the input and the enduring memory, and not on the history, and hence is oblivious to the order in which it receives queries. The failure probability of an LCA A (cf. [13]) is the probability, taken over coin flips of A, that the running time of A for any query exceeds its stated running time. Formal definitions are provided in Section 2.

New LCAs. We give the first non-trivial constant-time, constant-probe LCAs to several graph problems, assuming graphs have constant maximal degree. The high-level technique behind all of our LCAs is the following: we take an existing algorithm for a graph problem that does not appear to be implementable as an LCA, and "weaken" it in some sense (running time, approximation ratio, etc.) The resulting algorithm, while weaker, *can* be implemented (usually in a straightforward fashion) as an LCA. We focus on the following problems.

- *Maximum weight forest.* Given an undirected graph with edge weights, the task is to find an acyclic edge set of (approximately) maximum weight. In the corresponding LCA, a query specifies an edge, and the algorithm says whether the given edge is in the solution forest. We present a deterministic $(1-\epsilon)$-approximate LCA for this problem, whose running time and space are independent of the size of the graph.

[1] We assume the standard uniform-cost RAM model [1], in which the word size is $O(\log n)$ bits, where n is the input size.

[2] We typically assume that vertex degrees are bounded by a constant.

- *Integer Multi-Commodity Flow (IMCF) and Multicut on Trees.* Given a tree with capacitated edges and source-destination pairs, the goal of IMCF is to route the greatest possible total flow where each pair represents a different commodity, subject to edge capacity constraints. Multicut is the dual problem where the goal is to pick an edge set of minimal total capacity so that no source can be connected to its destination without using a selected edge. We give a deterministic LCA for IMCF and multicut on trees that runs in constant time and gives a (1/4)-approximation to the optimal IMCF and a 4-approximation to the minimum multicut. We also give a randomized LCA to IMCF, with constant running time, very little enduring memory (less than a word), and approximation ratio $\frac{1}{2} - \epsilon$ (for any constant $\epsilon > 0$). We note that because these problems are global in nature, we need to make strong assumptions about the input graph in order to obtain LCAs for them.
- *Weighted Matching.* Given a weighted graph, we would like to approximate the maximum weight matching. We design a deterministic reduction from any (possibly randomized) LCA A for unweighted matching with approximation ratio α to weighted matching with approximation ratio $\alpha/8$. Our reduction invokes A a constant number of times. Both the running time and approximation ratio are independent of the magnitude of the edge weights.

We note that there cannot exist any LCA for maximum cardinality matching [25], and that any LCA for *maximal* cardinality matching requires $\Omega(\log^* n)$ probes [5]. We do not know whether there exists an LCA that gives a constant approximation to the maximum cardinality matching in constant time.

Related Work. LCAs were introduced by Rubinfeld et al. [21]. Alon et al. [2] described LCAs for hypergraph 2-coloring and maximal independent set (MIS) on graphs of bounded degree, using a reduction from parallel and distributed algorithms. Mansour et al. [12], showed how to convert a large class of online algorithms to LCAs, and recently, Reingold and Vardi [20] extended these results to a much wider class of graphs, and obtained stronger bounds. The foundation of the results of [2,12,20] is the technique of Nguyen and Onak [16], in which a random ordering is generated over the vertices, and this ordering is used to simulate an online algorithm: in order to reply to a query about a vertex, we need to simulate the online algorithm on all vertices that come before it in the ordering. The main challenge is to bound the number of probes that one needs to make per query.

The family of graphs of constant bounded degree has been extremely well studied in the context of distributed algorithms. Naor and Stockmeyer [14] investigate the question of what can be computed on these networks in a constant number of rounds in the LOCAL[3] model. They consider locally checkable labeling (LCL) problems, where the legality of a labeling can be checked in

[3] In the LOCAL model [19], at the beginning of the algorithm's execution, each vertex knows only its ID and the IDs of its neighbors. In each round, each vertex is allowed to send an unbounded message to all of its neighbors and perform an unbounded amount of

constant time (e.g., coloring). They conclude that there are non-trivial LCL algorithms with constant-time distributed algorithms (an example is weak coloring on graphs of odd degree, where weak coloring means that every vertex has at least one neighbor colored differently from it). Cole and Vishkin [3] showed that it is possible to obtain a 3-coloring of an n-cycle in $O(\log^* n)$ rounds (assuming that there are unique node identifiers). This was shown to be tight by Linial [10], and Göös et. al [5] showed that this lower bound holds for LCAs as well. The LOCAL model has received considerable interest over the past three decades, e.g., [7,8,9,18]; see [19] for an introductory book and [22] for a survey from 2013.

Even et al. [4] investigate the connection between local distributed algorithms and LCAs. They show how to color the vertices in a small neighborhood of a graph to obtain an acyclic orientation of the neighborhood. Their coloring algorithm is an adaptation of the techniques of Panconesi and Rizzi [18] and Linial [10], and produces a coloring in $O(\log^* n)$ rounds. This orientation defines an order on the vertices, and an online algorithm can be simulated on this order, similarly to the technique of [12]. This generates a deterministic LCA for MIS on constant degree graphs that requires $O(\log^* n)$ probes. They also give algorithms with similar running times for approximate maximum cardinality (MCM) and maximum weight matchings (MWM). Their results for MWM depend logarithmically on the ratio of the maximum to the minimum weight.[4]

We give more problem-specific related work in the relevant sections.

2 Preliminaries

We denote the set $\{1, 2, \ldots n\}$ by $[n]$.

Graph concepts. Let $G = (V, E)$ be a simple undirected graph. The neighborhood of a vertex v, denoted $N(v)$, is the set of vertices that share an edge with v: $N(v) = \{u : (u, v) \in E\}$. The *degree* of a vertex v, is $|N(v)|$. The *distance* between two vertices u and v, denoted $\mathrm{dist}(u, v)$, is the minimal number of edges on any path from one to the other. For any vertex v, its k-neighborhood, denoted $N^k(v)$ is the set of all vertices at distance at most k from v. (Note that $N(v) = N^1(v) \setminus N^0(v)$.) For any edge $e = (u, v)$, its k-neighborhood is defined as $N^k(e) = N^k(u) \cup N^k(v)$. Given a non-empty vertex set $S \neq V$, a *cut* $(S, V \setminus S)$ is the set of edges with exactly one endpoint in S.

Throughout this paper, we assume that the maximal degree in G is upper bounded by some constant d. For simplicity of presentation, we assume that G is d-regular and that $|V|$ and d are powers of 2. All our results hold without these assumptions.

computation. The goal is to minimize the maximal number of rounds a vertex requires to compute its own portion of the output.

[4] We note that while our algorithm for MWM runs in constant time, independently of the size of the graph and of the edge weights, its approximation guarantee is much worse than that of [4], whose approximation factor is $(1 - \epsilon)$.

Approximation algorithms. We define approximation algorithms as follows.

Definition 1 Given a maximization problem over graphs and a real number $0 \leq \alpha \leq 1$, a (possibly randomized) α-approximation algorithm $\mathcal{A}$ is guaranteed, for any input graph G, to output a feasible solution whose (possibly expected) value is at least an α fraction of the value of an optimal solution.[5] The definition of approximation algorithms to minimization problems is analogous, with $\alpha \geq 1$.

LCAs. We extend the model of [21] for local computation algorithms (LCAs) to distinguish between time and probe complexity, and between enduring and transient memory. The measures by which the complexity of LCAs is quantified are:

– *Number of probes*. An LCA can access a vertex in the input graph and ask for a list of its neighbors, or the ID of a neighbor. This is called a "probe".
– *Running time*. The running time of an LCA, per query, is the number of word operations that the LCA performs in order to output a reply to the query. In the running time calculation, a probe to the graph is assumed to take $O(1)$ time. Note, however, that if we wish to perform an operation on the entire list of neighbors and the list is of size ℓ, (such as committing the list to memory or finding the maximal ID neighbor), this will take $O(\ell)$ operations.
– *Enduring memory*. As a preprocessing step, before it is given its first query, the LCA can be allocated some memory to which it is allowed to write. Once the first query is given to the algorithm, it can only read from this memory and can never modify it. This can be viewed as the LCA being allowed to augment the input with some small number of bits.
– *Transient memory*. This is simply the amount of memory (measured in the number of words) that an LCA requires in order to reply to a query. It does not include the enduring memory.
– *Failure probability*. The LCA *fails* if it deviates from its prescribed complexity (the number of probes, running time or transient memory). We require that, even if the LCA is queried on all vertices, the probability that it will fail on any of them is still very low. Note that the algorithm is never allowed to reply incorrectly - the failure is only a function of its complexity. Note further that the LCA is never allowed to use more enduring memory than it requested.

We define LCAs as follows.

Definition 2 (LCA) A *$(t(n), p(n), em(n), tm(n), \delta(n))$-local computation algorithm* $\mathcal{A}$ for a computational problem is a randomized algorithm that receives an input of size n and a query x. Before the first query, $\mathcal{A}$ is allowed to write $em(n)$ bits to the *enduring memory*, and may only read from it thereafter. Algorithm $\mathcal{A}$ makes at most $p(n)$ probes to the input in order to reply to any

[5] In case of a randomized algorithm, expectation is over its random choices.

query x, and does so in time at most $t(n)$ using at most $tm(n)$ bits of *transient memory* (in addition to the enduring memory). The probability that $\mathcal{A}$ deviates from the probe, time or transient memory bounds[6] (i.e., uses more than the prescribed amounts) is at most $\delta(n)$, which is called $\mathcal{A}$'s *failure probability.* Algorithm $\mathcal{A}$ must be *consistent*, that is, the algorithm's replies to all possible queries conform to a single feasible solution to the problem.

Remark 1 We define LCAs as randomized algorithms, but they can be deterministic. In that case, we require that the failure probability is zero. All algorithms in this paper, except Algorithm 5, are deterministic, and although Algorithm 6 is deterministic, it uses a (possibly) randomized algorithm as a subroutine.

Remark 2 All the LCAs of this paper (except Algorithm 6)[7] have failure probability 0; we include it in the model for compatibility with previous models, but omit it from the statements for brevity. Furthermore, they all have a constant running time and the transient memory is guaranteed to be of constant size; we omit this measure from the statements as well.

3 Maximum Weight Forest

In this section we consider the problem of finding a maximum weight forest (MWF), also known as the maximum weight basis of a graphic matroid [17]. We are given a graph $G = (V, E)$, with edge weights, $w : E \to \mathbb{R}$; we would like to find an acyclic set of edges of (approximately) maximum weight. That is, we seek a forest whose weight is close to the weight of a MWF. Without loss of generality, we assume that edge weights are distinct, as it is always possible to break ties by ID. Recall that when the edge weights are distinct, there is a unique MWF (e.g., [23]).

We describe a parallel algorithm for finding a MWF in a graph, analyze its correctness and approximation ratio, and explain how to adapt it to an LCA. We note that our parallel algorithm is less efficient than others (say, the Borůvka's algorithm [15]), when applied as parallel algorithms. In fact, there is no instance on which it would out-perform Borůvka's algorithm. Nevertheless, it is useful as adapting it to an LCA and analyzing its local properties are simple.

We first need a few definitions. Define the distance between a vertex v and an edge $e = (u, w)$, denoted $\mathrm{dist}(v, e)$, to be $\min\{\mathrm{dist}(v, u), \mathrm{dist}(v, w)\}$.

Definition 3 (Connected component, Truncated CC) Let $G = (V, E)$ be a simple undirected graph. For a vertex $v \in V$, and a subset of edges $\mathcal{S} \subseteq E$, the connected component of v with respect to $\mathcal{S}$ is $\mathrm{CC}_{\mathcal{S}}(v) \subseteq \mathcal{S}$ which includes the edges $e \in \mathcal{S}$ that have a path from e to v using only edges in

[6] Note that the LCA is *not* allowed to deviate from the enduring memory bound.

[7] Algorithm 6 inherits it running time, space complexity and failure probability from the LCA it uses as a subroutine.

$\mathcal{S}$, and their endpoints. (Note that $\mathrm{CC}_{\mathcal{S}}(\cdot)$ defines a partition of $\mathcal{S}$.) The *k-truncated connected component* of v (denoted by $\mathrm{TCC}^k_{\mathcal{S}}(v)$) is the set of all vertices in $\mathrm{CC}_{\mathcal{S}}(v)$ at distance at most k from v (w.r.t. G).

Algorithm 1 works as follows. We maintain a forest $\mathcal{S}$, initially empty. In round k, vertex v considers the cut $(\mathrm{TCC}^k_{\mathcal{S}}(v), V \setminus \mathrm{TCC}^k_{\mathcal{S}}(v))$ and adds the heaviest edge of the cut, say e, to $\mathcal{S}$. (Note that k is both the round number and the radius parameter of the truncated connected component.) In contrast to many MWF algorithms (such as Prim's and Kruskal's algorithms), an edge can be considered more than once, and it is possible that an edge e is considered—and even added—when it is already in $\mathcal{S}$ (if we add e to $\mathcal{S}$ when $e \in \mathcal{S}$, $\mathcal{S}$ remains the same).

Algorithm 1: Parallel (CREW) MWF Approximation Algorithm.

```
Input  : $G = (V, E)$ with weight function $w : E \to \mathbb{R}^+, \epsilon > 0$
Output: a forest $\mathcal{S}$
// assume all edge weights are distinct
For all $v \in V$, $\mathcal{S}(v) = \emptyset$;
for round $k = 1$ to $1/\epsilon$ do
    for all vertices $v \in V$ in parallel do
        if $e$ is the heaviest edge of the cut $(\mathrm{TCC}^k_{\mathcal{S}}(v), V \setminus \mathrm{TCC}^k_{\mathcal{S}}(v))$ then
            $\mathcal{S}(v) = \mathcal{S}(v) \cup \{e\}$;
Return $\mathcal{S} = \bigcup_{v \in V} \mathcal{S}(v)$.
```

3.1 Correctness and approximation ratio

The correctness of Algorithm 1 relies on the so-called "blue rule" [23].

Lemma 1 ([23]) *Let C be any cut of the graph. Then the heaviest edge in C belongs to the* MWF*.*

Corollary 1 establishes the correctness of Algorithm 1.

Corollary 1 *All edges added to $\mathcal{S}$ by algorithm 1 are in the* MWF*.*

We now turn to analyze the approximation ratio of Algorithm 1. To this end, define $\mathcal{S}_k$ to be the set of edges of MWF that were added to $\mathcal{S}$ in rounds $1, 2, \ldots, k$ (implying that $\mathcal{S}_k \subseteq \mathcal{S}_{k+1}$). Let $\mathcal{R}_k = \mathrm{MWF} \setminus \mathcal{S}_k$.

Consider the *component tree* of MWF, defined as follows: the node set is $\{CC_{\mathcal{S}_k}(v) \mid v \in V\}$, and the edge set is $\{(CC_{\mathcal{S}_k}(v), CC_{\mathcal{S}_k}(u)) \mid (v, u) \in \mathcal{R}_k\}$. In words, there is a node in the component tree for each connected component of $\mathcal{S}_k$, and there is an edge in the component tree iff there is an edge in $\mathcal{R}_k$ connecting nodes in the corresponding components. For the analysis, we choose

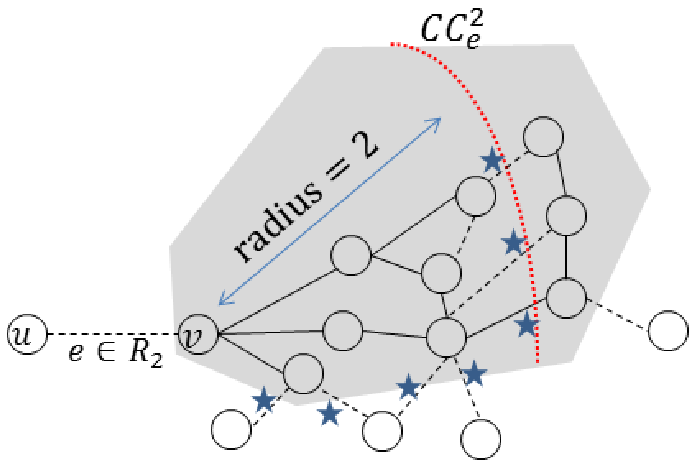

Fig. 1 The situation considered in the proof of Proposition 1, for $k = 2$. The edge $e = (u, v)$ is in $\mathcal{R}_k$. Solid edges are in $\mathcal{S}$, dashed edges are in $E \setminus \mathcal{S}$. The shaded area represents CC_e^2, and the dotted red arc represents the distance 2 range. Edges marked by $\star$ are considered by v in round 2.

an arbitrary component as the root of the component tree, and direct all the edges towards it; this way, each edge $e \in \mathcal{R}_k$ is outgoing from exactly one connected component of $\mathcal{S}_k$. We denote this component by CC_e^k. Note that $CC_e^i \subseteq CC_e^j$ for $i < j$ (for $e \in R_j$) because components only grow. For any set of edges S, let $w(S) = \sum_{e \in S} w(e)$.

The following proposition is the key to the analysis. The intuition is that even if edges can be added to $\mathcal{S}$ more than once, they will never be added more than once by any specific vertex.

Proposition 1 *For any $k \geq 1$, $\forall e \in \mathcal{R}_k$, $w(e) \leq \frac{w(CC_e^k)}{k}$.*

Proof If $\mathcal{R}_k$ is empty, the claim holds trivially. Let $e = (v, u)$ be any edge in $\mathcal{R}_k$ (directed from v to u). Edge e was not chosen by vertex v in rounds $1, \ldots, k$. For $i \in [k]$, let e_i be the edge chosen by v in round i. It suffices to show that (1) all edges e_i are heavier than e, i.e. $\forall i \in [k], w(e_i) > w(e)$, and that (2) the edges e_i are distinct, i.e., $\forall i, j \in [k], i \neq j \Rightarrow e_i \neq e_j$.

The proof of (1) is straightforward: e was in the cut $(\text{TCC}_{\mathcal{S}}^i(v), V \setminus \text{TCC}_{\mathcal{S}}^i(v))$ in all rounds $i \in [k]$, but it was never chosen. This must be because v chose a heavier edge in each round.

To prove (2), we show by induction that e_i is distinct from $\{e_1, e_2, \ldots, e_{i-1}\}$. The base of the induction is trivial. For the inductive step, consider the two possible cases. If $e_i \notin CC_e^{i-1}$, then, by the definition of CC_e^{i-1}, e_i cannot be any edge that had already been added by v (and is now being added again), hence it is distinct from $\{e_1, e_2, \ldots, e_{i-1}\}$. If $e_i \in CC_e^{i-1}$, then e_i must be at distance exactly i from v, otherwise it would not have been in the cut $(\mathrm{TCC}^i_{\mathcal{S}}(v), V \setminus \mathrm{TCC}^i_{\mathcal{S}}(v))$ and could not have been added. But $e_1, \ldots, e_{i-1}$ are all at distance at most $i-1$ (w.r.t. G). In other words, if $e_i \in CC_e^{i-1}$, e_i was previously added by some vertex $w \neq v$ to CC_e^i. □

Corollary 2 *For $k \geq 1$, $w(\mathcal{R}_k) \leq \frac{w(\mathcal{S}_k)}{k}$.*

Proof

$$\begin{aligned} w(\mathcal{R}_k) &= \sum_{e \in \mathcal{R}_k} w(e) \\ &\leq \sum_{e \in \mathcal{R}_k} \frac{w(CC_e^k)}{k} && \text{by Prop. 1} \\ &= \frac{w(\mathcal{S}_k)}{k}. && e \neq e' \Rightarrow CC_e^k \neq CC_{e'}^k, \text{ and } \bigcup_{e \in \mathcal{R}_k} CC_e^k = \mathcal{S}_k \end{aligned}$$

□

This enables us to prove our approximation bound. Denote the weight of MWF by OPT.

Lemma 2 *$w(\mathcal{S}_k) \geq (1 - \frac{1}{k+1})OPT$.*

Proof As $\mathcal{R}_k = \mathrm{MWF} \setminus \mathcal{S}_k$,

$$\frac{w(\mathcal{S}_k)}{\mathrm{OPT}} = \frac{w(\mathcal{S}_k)}{w(\mathcal{S}_k) + w(\mathcal{R}_k)} \geq \frac{w(\mathcal{S}_k)}{w(\mathcal{S}_k) + w(\mathcal{S}_k)/k} = \frac{k}{k+1},$$

where the inequality is due to Corollary 2. □

This concludes the analysis of Algorithm 1. We now describe the LCA we derive from it.

3.2 Adaptation to an LCA and complexity analysis

Given a graph $G = (V, E)$ and a query $e = (u, v) \in E$, the implementation of Algorithm 1 as an LCA is as follows. Set $k^* = 1/\epsilon$. Probe G to discover $N^{2k^*}(u)$ and $N^{2k^*}(v)$.[8] Simulate Algorithm 1 on all vertices in $N^{k^*}(u) \cup N^{k^*}(v)$ for k^* rounds: In each round $i \in [k^*]$, for each node $s \in N^{k^*}(u) \cup N^{k^*}(v)$, the algorithm computes $\mathrm{TCC}^i_{\mathcal{S}}(v)$, finds the heaviest edge e in the cut $(\mathrm{TCC}^i_{\mathcal{S}}(v), V \setminus \mathrm{TCC}^i_{\mathcal{S}}(v))$, and adds it to the solution.

[8] In order to simulate Algorithm 1 on a vertex at distance i from v for j rounds, we need to discover vertices at distance $i + j$ from v.

Lemma 3 *The time required to simulate the execution of Algorithm 1 for k rounds as an LCA is $kd^{O(k)}$, and the probe complexity is $d^{O(k)}$.*

Proof The time to discover $N^{2k}(v) \cup N^{2k}(u)$ by probing the graph is bounded by $d^{O(k)}$. Each vertex $z \in N^k(v) \cup N^k(u)$ executes Algorithm 1 for k rounds: in round j, it explores $N^j(z) \subseteq N^{2k}(v) \cup N^{2k}(u)$ and updates $\mathcal{S}(z)$ (as defined in Algorithm 1). Overall, the time complexity is at most $d^{O(k)} + \sum_{z \in N^k(v) \cup N^k(u)} k|N^k(z)| = kd^{O(k)}$. □

Combining Lemmas 2 and 3 gives our first main result.

Theorem 1 *There exists a deterministic LCA, that for any graph G whose degree is bounded by d and every $\epsilon > 0$, computes a forest whose weight is a $(1-\epsilon)$-approximation to the maximal weighted forest of G in time $t(n) = \frac{1}{\epsilon} d^{O(1/\epsilon)}$, probe complexity $p(n) = d^{O(1/\epsilon)}$, and enduring memory $em(n) = 0$.*

4 Multicut and Integer Multicommodity Flow in Trees

In this section we consider the integer multicommodity flow (IMCF) and multicut problems in trees. While simple, our LCAs demonstrate how, under some circumstances, one can find constant-time LCAs for apparently global problems.

The input is an undirected tree $G = (V, E)$ with a positive integer capacity $c(e)$ for each $e \in E$, and a set of pairs of vertices $\{(s_1, t_1), \ldots, (s_k, t_k)\}$. (The pairs are distinct, but the vertices are not necessarily distinct.)

In the *Integer Muticommodity Flow Problem*, the goal is to route commodity i from s_i to t_i so as to maximize the sum of the commodities routed, subject to edge capacity constraints. (There is no a priori upper or lower bound on the amount of flow for each commodity.) Note that in a tree, the only question is how much to route: the route is uniquely determined. In the dual *Multicut Problem*, the goal is to find a minimum capacity *multicut*, where a multicut is an edge set that separates s_i from t_i for all $i \in [k]$.

We make the following assumptions about the input: The tree is rooted and along with its ID, each vertex knows its distance from the root. The degree of the tree is bounded by $d = O(1)$. Furthermore, the distances from s_i to t_i are bounded by some given parameter ℓ; i.e., $\forall i, \mathrm{dist}(s_i, t_i) \leq \ell$. Our bounds will be a function of ℓ, so that if ℓ is independent of tree size, then so are the time and space complexity of our algorithms.

In the local version of IMCF (multicut), we are queried on an edge, and are required to output how much of each resource is routed on it (whether it is part of the cut). As before, we adapt a classical algorithm to an LCA. This time we use the known algorithm of Garg et al. [6] as a subroutine (denoted Algorithm 2, and presented below for completeness). Note that this is a primal-dual algorithm, that solves IMCF and multicut simultaneously.

Theorem 2 ([26]) *Algorithm 2 achieves approximation factors of* 2 *for the multicut problem and* 1/2 *for the IMCF problem on trees.*

```
Algorithm 2: Multicut and IMCF in trees [6,26]
Input  : A rooted tree $T = (V, E)$, a function $w : E \to \mathcal{R}^+$, which
         represents the weights for multicut and the capacities for
         IMCF
Output: a flow $f$ and a cut $D$
Initialize $f = 0, D = \emptyset$;
for each vertex $v$ in nonincreasing order of depth do
    for each pair $(s_i, t_i)$ let $v$ be the lowest common ancestor of $s_i$ and
     $t_i$;
    do
        Greedily route flow from $s_i$ to $t_i$ if possible;
        Add all saturated edges to $D$ in arbitrary order;
Let $e_1, \ldots, e_k$ be the ordered list of edges in $D$;
for $j = k$ down to 1 do
    If $D - \{e_j\}$ is a multicut, then remove $e_j$ from $D$;
```

Our deterministic LCA for multicut is detailed in Algorithm 3[9]. It finds a $(4+\epsilon)$-approximation to the multicut problem. In trees with minimum capacity $c_{\min} \geq 2$, a similar algorithm (Algorithm 4) finds an IMCF with approximation factor $\frac{\lfloor c_{\min}/2 \rfloor}{2c_{\min}} \geq \frac{1}{6}$.[10] We also present a randomized Algorithm (Algorithm 5), that gives an approximation factor of $(\frac{1}{2} - \epsilon)$ to IMCF for any desired $\epsilon > 0$ (the running time depends on $1/\epsilon$). All three algorithms are similar, in that they partition the tree to subtrees and apply Algorithm 2 to each subtree. The randomized algorithm requires a very small amount of enduring memory, namely $O(\log(\ell/\epsilon))$ bits.

4.1 Multicut on trees

Similarly to Section 3, we first describe and analyze a global algorithm and then show how to implement it as an LCA.

An edge is said to be at depth z if it connects vertices at depths $z - 1$ and z. The deterministic (global) algorithm (Algorithm 3) for multicut is as follows. We consider two overlapping decompositions of the tree into subtrees of height 2ℓ: the first decomposition is obtained by removing edges at depth $2\ell, 4\ell, \ldots$ After removing these edges, the tree is decomposed a forest consisting

[9] Technically, this is a global algorithm, which we later show how to implement as an LCA

[10] The ratio is $\frac{1}{4}$ when all capacities are even, and it tends to $\frac{1}{4}$ as $c_{\min} \to \infty$. For $c_{\min} = 1$ the approximation ratio is 0.

of subtrees of depths at most 2ℓ with root nodes being the nodes at depth $0, 2\ell, 4\ell, \ldots$ in the original tree. The second decomposition is similar, only we remove all edges at depth $\ell, 3\ell, \ldots$ We then run Algorithm 2 on both forests. Let the output of the "even" instance be D^e, and the output of the "odd" instance be D^o. The output of the global algorithm is $D^e \cup D^o$.

For the LCA implementation, we are given an edge e as a query, and execute Algorithm 2 on the subtrees that contain e (there is at least one such tree and at most two).The output of the LCA is "yes" (i.e., the edge is in the cut) if it is in $D^o \cup D^e$. We discuss this implementation in more detail in Subsection 4.3.

Algorithm 3: Deterministic algorithm for Multicut in trees

```
Input   : A graph $G = (V, E)$, with $c : E \to \mathbb{Z}^+$, and $\ell > 0$
Output  : A multicut $D$
Step 1.
    Delete all edges at depth $k\ell$, for odd $k$;
    On each remaining subtree $T_i$, run Algorithm 2;
    Let $D_i$ be the multicut returned on $T_i$;
    Let $D^o = \cup_i D_i$;
Step 2.
    Delete all edges at depth $k\ell$, for even $k$;
    On each remaining subtree $T_j$, run Algorithm 2;
    Let $D_j$ be the multicut returned on $T_j$;
    Let $D^e = \cup_j D_j$;
Step 3.
    Return $D = D^o \cup D^e$.
```

Let us call the subtrees created in **Step 1** of Algorithm 3 *odd subtrees*, and the subtrees created in **Step 2** *even subtrees.*

Lemma 4 *Algorithm 3 outputs a multicut, whose capacity is at most* 4 *times the capacity of the minimum multicut of T.*

Proof Each vertex is contained in exactly two subtrees, and every edge is contained in at least one and at most two subtrees. As the subtrees are of depth 2ℓ, each path (s_i, t_i) must be fully contained within either an odd subtree or an even subtree (or both), as we assume that the paths are of length at most ℓ. Therefore, as D^o and D^e are multicuts of their respective forests, at least one edge in $D^o \cup D^e$ is on the path between s_i and t_i; hence $D = D^o \cup D^e$ is a multicut. From Theorem 2, D^o and D^e are 2-approximations to the minimal capacity multicut of T, hence their union is at most a 4-approximation. □

4.2 IMCF on trees

The global algorithm for IMCF on trees (Algorithm 4) splits the tree into subtrees, similarly to Algorithm 3. It then decreases the capacity of each edge e from c_e by a factor of 2 (rounding down to ensure feasibility and integrality).

Algorithm 2 is then executed on each subtree. This gives flows f^e for the "even" subtrees and f^o for the "odd" subtrees. The algorithm outputs $f^e + f^o$. This algorithm is very similar in structure to Algorithm 3; we include its pseudocode for completeness.

Algorithm 4: Deterministic algorithm for IMCF in trees

```
Input  : A graph $G = (V, E)$, with $c : E \to \mathbb{Z}^+$, and $\ell > 0$
Output : A flow $f$
for every $e \in E$ do
    Set $c_e = \lfloor \frac{c_e}{2} \rfloor$;
Step 1.
    Delete all edges at depth $k\ell$, for odd $k$;
    On each remaining subtree $T_i$, run Algorithm 2;
    Let $f_i$ be the flow returned on $T_i$;
    Let $f^o = \sum_i f_i$;
Step 2.
    Delete all edges at depth $k\ell$, for even $k$;
    On each remaining subtree $T_j$, run Algorithm 2;
    Let $f_j$ be the flow returned on $T_j$;
    Let $f^e = \sum_j f_j$;
Step 3.
    Return $f = f^o + f^e$.
```

Lemma 5 *Algorithm 4 outputs a feasible flow, whose value is at least $(\frac{1}{4} - \frac{1}{4c_{\min}})$ fraction of the maximal multicommodity flow on T.*

Proof Let f^o and f^e denote the flows computed on odd and even trees, respectively. Their sum is a feasible flow because each of them uses at most half of the capacity of each edge. Moreover, the flow that the algorithm returns for subtree T_i is a 1/2 approximation to the optimal flow of T_i. Denote the value of the maximal IMCF on T by f^*. Let f_i^* be the optimal flow on the subtree T_i. As in Algorithm 4, we denote indices of odd subtrees by i, and even subtrees by j. Then

$$\begin{aligned} f^* &\le \sum_i f_i^* + \sum_j f_j^* \\ &\le \sum_i 2f_i + \sum_j 2f_j \\ &\le (2f^e + 2f^o) \cdot \frac{c_{\min}}{\lfloor c_{\min}/2 \rfloor} \\ &\le \frac{4c_{\min}}{c_{\min} - 1}(f^e + f^o) \ . \end{aligned}$$

□

4.3 Implementation as LCAs

The implementation of Algorithms 3 and 4 as LCAs is straightforward: when queried on an edge, determine to which subtrees it belongs, and then execute Algorithm 3 or 4 on the two subtrees. To determine to which subtrees it belongs, we need to find the root of the subtree, which is done by repeatedly finding the parent of each vertex on the path to the root. We then perform a BFS from the root vertex, stopping when we reach the leaves of the subtree, which are the vertices 2ℓ levels down from the root. Finding the parent of a vertex takes $O(d)$ probes. The maximal size of a subtree is $d^{2\ell}$, hence our probe complexity is $O(d^{2\ell+1})$. We do not attempt to optimize the running time, but simply note that it is trivial to implement the algorithm on subtree T_j in time $O(|T_j|^3)$, as there can be at most $|T_j|^2$ pairs (s_i, t_i) in T_j. This gives

Theorem 3 *Given a rooted tree T with depth information at the nodes, maximal degree d, integer edge capacities at least $c_{\min}$ and source-destination pairs with maximal distance at most $\ell > 0$, there are LCAs with $t(n) = d^{O(\ell)}$, $p(n) = d^{O(\ell)}$ and $em(n) = 0$, for 4-approximate multicut and $(\frac{1}{4} - \frac{1}{4c_{\min}})$-approximate IMCF. If all capacities are even, the approximation ratio to IMCF is $\frac{1}{4}$.*

4.4 Randomized LCA for IMCF on trees

We now turn to the randomized setting. Our randomized algorithm (detailed in Algorithm 5) is very similar to the deterministic one; instead of an overlapping decomposition, we use a random one as follows. Let $H = \lceil \frac{\ell}{\epsilon} \rceil$. We pick an integer j uniformly at random from $[H]$, and remove all edges whose depth modulo H is $j - 1$. The result is a collection of subtrees of depth at most $H - 1$ each. Now, given an edge e, we run Algorithm 2 on the subtree that contains e and output the output of Algorithm 2 (with probability $1/H$, the edge queried, e, is not in any tree; in this case, e carries 0 flow).

Algorithm 5: IMCF in trees

```
Input  : A graph $G = (V, E)$, with $c : E \to \mathbb{Z}^+$, and $\ell > 0$
Output: A flow $f$
Let $H = \lceil \frac{\ell}{\epsilon} \rceil$;
Uniformly sample an integer $j$ from $\{0, \ldots, H\}$;
Delete all edges at depth $j + kH$, for all $k \in \mathbb{N}$ such that $j + kh \leq \text{depth}(T)$;
On each remaining subtree, run Algorithm 2;
Return the union of the flows on all subtrees.
```

Theorem 4 *Algorithm 5 achieves an approximation ratio of* $1/2 - \epsilon$ *to the maximum integer multicommodity flow on trees.*

Proof For any i, the probability that the path (s_i, t_i) is not fully contained within a subtree is at most $\frac{\ell}{H} \leq \epsilon$. Fix an optimal solution f^* with value $|f^*|$. After deleting edges, the expected amount of remaining flow is at least $(1-\epsilon)|f^*|$. By Theorem 2, Algorithm 2 outputs at least a half of that amount. The result follows.

The implementation of Algorithm 5 as an LCA is almost identical to that of Algorithm 3; in order to achieve a $1/2 - \epsilon$ approximation to the IMCF, the enduring memory has to hold an integer whose value is at most $\lceil \frac{\ell}{\epsilon} \rceil$, i.e., $O(\log(\ell/\epsilon))$ bits. We therefore have

Theorem 5 *Given a rooted tree T with depth information at the nodes, maximal degree d, integer edge capacities and vertex pairs with maximal distance at most $\ell > 0$, there is an LCA with $t(n) = d^{O(\ell/\epsilon)}$, $p(n) = d^{O(\ell/\epsilon)}$, and $em(n) = O(\log \ell/\epsilon)$, that achieves an approximation ratio of* $(1/2-\epsilon)$ *to IMCF.*

5 Weighted Matchings

In this section we present a different kind of an LCA: a reduction. Specifically, we consider the task of computing a maximum weight matching (MWM), and show how to locally reduce it to maximum cardinality matching (MCM). Our construction, given any graph of maximal degree d and a t-time α-approximation LCA for MCM, yields an $O(td)$-time, $\frac{\alpha}{8}$-approximation LCA for MWM.

Formally, in the MWM problem, we are given a graph $G = (V, E)$ with a weight function $w : E \to \mathbb{N}$, and we need to output a set of disjoint edges of (approximately) maximum total *weight*. In MCM, the task is to find a set of disjoint edges of (approximately) the largest possible *cardinality*.

The main idea of our reduction is a variant of the well-known technique of *scaling* (e.g., [11,24,27]): partition the edges into classes of more-or-less uniform weight, run an MCM instance for each class, and somehow combine the MCM outputs. Motivated by local computation, however, we use a very crude combining rule that lends itself naturally to LCAs.

Specifically, the algorithm is as follows (the "global" algorithm is presented as Algorithm 6). Let $\gamma > 2$ be a constant whose value will be optimized later. Partition the edges by weight to sets E_i, such that $E_i = \{e : w(e) \in [\gamma^{i-1}, \gamma^i)\}$. The *level* of a class E_i is i, and we denote the level of an edge e by level(e); that is, level(e) $= i : e \in E_i$. For each level i, find a maximum cardinality matching M_i on the graph $G_i = (V, E_i)$, using any LCA for MCM. Let $M = \cup_i M_i$. Given an edge e, our LCA for MWM returns "yes" iff e is a local maximum in M, i.e., iff (1) e is in M, and (2) for any edge e' in M which shares a node with e, $w(e') < w(e)$ (no ties can occur).

Algorithm 6: Reduction of MWM to MCM

```
Input  : A graph $G = (V, E)$, with $w : E \to \mathbb{N}$, and $\gamma > 2$
Output: A matching $M$
Partition the edges into classes $E_i = \{e : w(e) \in [\gamma^{i-1}, \gamma^i)\}$ for
  $i = 1, 2, \ldots$
In parallel, compute an unweighted matching $M_i$ for each level $i$;
$M = \bigcup_i M_i$;
for each edge $e \in M$ do
    if $e$ has a neighbor $e' \in M$, with level($e'$) > level($e$) then
        Remove $e$ from $M$;
Return $M$.
```

5.1 Correctness and approximation ratio

The correctness of the reduction follows from the fact that edges in different classes have different weights, and hence no pair of adjacent edges can be selected to be in M. The more interesting part is the approximation ratio analysis. We require the following definition.

Definition 4 [M-shrub] For any edge $(u, v) = e \in M$, recursively define an edge set T_e as follows. If e is lighter than all of its neighbors in M, then $T_e = \{e\}$. Otherwise, let f_u be the heaviest edge in M that touches u and is lighter than e. Define f_v similarly, and let $T_e = \{e\} \cup T_{f_u} \cup T_{f_v}$. (If f_u does not exist, let $T_{f_u} = \emptyset$; similarly f_v.) We call T_e the *M-shrub* of e. See Figure 2 for an example.

In other words an M-shrub is an edge e and all of its neighbors that are lighter than e and are in their respective matchings, their neighbors that are lighter than them and are in their respective matchings, and so on. Intuitively, it is the set of edges that are in $\bigcup_i M_i$ but not in M that we can "charge" to e.

Define a new weight function on G, $\hat{w}$: $\hat{w}(e) = \gamma^{\text{level}(e)-1}$; i.e., $\hat{w}(e)$ is $w(e)$ rounded down to the nearest power of γ. Note that the choices made by Algorithm 6 are identical under w and $\hat{w}$. The main argument in the analysis of the approximation ratio of Algorithm 6 is the following.

Proposition 2 *Let $e = (v, u)$ be any edge in M, such that $\hat{w}(e) = \gamma^k$ and let T_e be the M-shrub of e. Then $\hat{w}(T_e) \leq \sum_{i=0}^{k} 2^{k-i}\gamma^i$.*

Proof The proof is by induction on k. For the base of the induction, $k = 0$, we have $\hat{w}(T_e) = 2^0\gamma^0$. For the inductive step, assume that the proposition holds for all integers up to $k - 1$. Let $e = (u, v)$. The heaviest edge that is

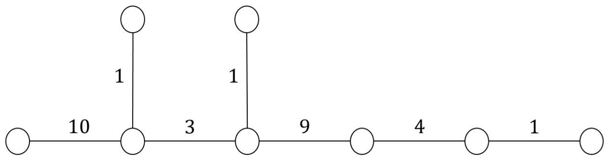

Fig. 2 A simple example. Here $\gamma = 3$ and that $M = \cup_i M_i$ is the entire graph. The M-shrub (see Definition 4) of the edge of weight 3 consists of the edge itself and its two neighbors of weight 1. The M-shrub of the edge of weight 10 includes itself and the M-shrub of the edge of weight 3. The M-shrub of the edge of weight 9 is the entire graph, except the edge of weight 10.

lighter than e and touches u (denoted f_u) weighs at most γ^{k-1}. Similarly for f_v. Therefore, by the inductive hypothesis,

$$\hat{w}(T_e) \leq \gamma^k + 2\sum_{i=0}^{k-1} 2^{k-i-1}\gamma^i = \sum_{i=0}^{k} 2^{k-i}\gamma^i.$$

□

Corollary 3 *Let $e = (v, u)$ be any edge in M, such that $\hat{w}(e) = \gamma^k$ and let T_e be the M-shrub of e. Then $\hat{w}(e) \geq \frac{\gamma-2}{\gamma}\hat{w}(T_e)$.*

Proof As $\gamma > 2$,

$$\begin{aligned}
\hat{w}(T_e) &\leq \sum_{i=0}^{k} 2^{k-i}\gamma^i \\
&= \gamma^k \sum_{i=0}^{k} \frac{2^{k-i}}{\gamma^{k-i}} \\
&= \gamma^k \sum_{j=0}^{k} \frac{2^j}{\gamma^j} \\
&< \gamma^k \sum_{j=0}^{\infty} \frac{2^j}{\gamma^j} \\
&= \hat{w}(e)\frac{\gamma}{\gamma-2}.
\end{aligned}$$

□

Lemma 6 *Using any α-approximate MCM algorithm, Algorithm 6 finds a matching of total weight at least $\alpha\frac{\gamma-2}{\gamma^2}OPT$.*

Proof Let M_i^* be a maximum weighted matching on $G_i = (V, E_i)$, and let $M^* = \cup_i M_i^*$. Let $\hat{M}_i^*$ be a maximum cardinality matching (MCM) on $\hat{G}_i$. Clearly, for all i, $w(\hat{M}_i^*) \geq \frac{1}{\gamma}w(M_i^*)$, because each edge in M_i^* weighs at most γ times any edge of $\hat{M}_i^*$, and M_i^* does not contain more edges than $\hat{M}_i^*$. Also note that $w(M^*) \geq \text{OPT}$, because any restriction of an optimal MWM to edges of class i cannot have more weight than M_i^*. Call a locally heaviest edge in M an *output edge.* Note that every edge $e \in M$ is contained in the M-shrub of at least one output edge. We can therefore conclude that

$$\begin{aligned}
\sum_{e:e \text{ is an output edge}} w(e) &\geq \sum_{T_e:e \text{ is an output edge}} \frac{\gamma-2}{\gamma} w(T_e) \\
&\geq \frac{\gamma-2}{\gamma} w(M) \\
&= \frac{\gamma-2}{\gamma} \sum_i w(M_i) \\
&\geq \frac{\gamma-2}{\gamma} \sum_i \alpha w(\hat{M}_i^*) \\
&\geq \frac{\gamma-2}{\gamma^2} \sum_i \alpha w(M_i^*) \\
&= \alpha \frac{\gamma-2}{\gamma^2} w(M^*) \\
&\geq \alpha \frac{\gamma-2}{\gamma^2} \text{OPT.}
\end{aligned}$$

□

It is easy to verify, by differentiation, that the optimal value of γ is 4, yielding approximation factor $\alpha/8$.

5.2 Complexity

The simulation of Algorithm 6 as an LCA is simple, and, unlike the global algorithm, its complexity is independent of the weights on the edges. Suppose we are queried about edge e. Let N be the set of edges that includes e and all of its neighboring edges whose weight is at least $w(e)$. We invoke, for each $e' \in N$, the LCA for MCM on the edges whose weight class is level(e'). The answer for e is "yes" iff the MCM LCA replied "yes" for the query on e, and "no" for all other queries. The following theorem follows.

Theorem 6 *Let $\mathcal{A}$ be an LCA for unweighted matching, requiring $t(n)$ time, $p(n)$ probes and $em(n)$ enduring memory, and producing an α-approximation* *to the maximum matching. Then given a graph $G = (V, E)$ with maximal degree d and arbitrary weights on the edges, there is a LCA that computes a $\alpha/8$-approximation to the maximum weighted matching, requiring $O(d \cdot t(n))$ time, $O(d \cdot p(n))$ probes and $O(d \cdot em(n))$ enduring memory.*

Acknowledgements The authors would like to thank the anonymous reviewers for their useful feedback.

References

1. Aho, Alfred V., Hopcroft, John E.: The Design and Analysis of Computer Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1974)
2. Alon, N., Rubinfeld, R., Vardi, S., Xie, N.: Space-efficient local computation algorithms. In: Proc. 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1132–1139 (2012)
3. Cole, R., Vishkin, U.: Deterministic coin tossing with applications to optimal parallel list ranking. Information and Control **70**(1), 32 – 53 (1986)
4. Even, G., Medina, M., Ron, D.: Deterministic stateless centralized local algorithms for bounded degree graphs. In: 22th Annual European Symposium on Algorithms (ESA), pp. 394–405 (2014)
5. Göös, M., Hirvonen, J., Levi, R., Medina, M., Suomela, J.: Non-local Probes Do Not Help with Many Graph Problems. In: 30th 30th International Symposium, on Distributed Computing (DISC), pp.201–214 (2016)
6. Garg, N., Vazirani, V., Yannakakis, M.: Primal-dual approximation algorithms for integral flow and multicut in trees. Algorithmica **18**(1), 3–20 (1997)
7. Göös, M., Hirvonen, J., Suomela, J.: Lower bounds for local approximation. In: ACM Symposium on Principles of Distributed Computing, PODC, pp. 175–184 (2012)
8. Kuhn, F.: Local approximation of covering and packing problems. In: Encyclopedia of Algorithms (2008)
9. Kuhn, F., Moscibroda, T., Wattenhofer, R.: The price of being near-sighted. In: Proc. 17th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 980–989 (2006)
10. Linial, N.: Locality in distributed graph algorithms. SIAM J. Comput. **21**(1) (1992)
11. Lotker, Z., Patt-Shamir, B., Rosén, A.: Distributed approximate matching. SIAM J. Comput. **39**(2) (2009)
12. Mansour, Y., Rubinstein, A., Vardi, S., Xie, N.: Converting online algorithms to local computation algorithms. In: Proc. 39th International Colloquium on Automata, Languages and Programming (ICALP), pp. 653–664 (2012)
13. Mansour, Y., Vardi, S.: A local computation approximation scheme to maximum matching. In: APPROX-RANDOM, pp. 260–273 (2013)
14. Naor, M., Stockmeyer, L.J.: What can be computed locally? SIAM J. Comput. **24**(6), 1259–1277 (1995)
15. Nešetřil, J., Milková, E., Nešetřilová, H.: Otakar Borůvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. Discrete Mathematics **233**(1), 3–36 (2001)
16. Nguyen, H.N., Onak, K.: Constant-time approximation algorithms via local improvements. In: Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 327–336 (2008)
17. Oxley, J.: Matroid Theory,. Oxford University Press (1992)
18. Panconesi, A., Rizzi, R.: Some simple distributed algorithms for sparse networks. Distributed Computing **14**(2), 97–100 (2001)
19. Peleg, D.: Distributed Computing: A Locality-sensitive approach. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2000)
20. Reingold, O., Vardi, S.: New techniques and tighter bounds for local computation algorithms. J. Comput. Syst. Sci. **82**(7) pp. 1180–1200 (2016)

21. Rubinfeld, R., Tamir, G., Vardi, S., Xie, N.: Fast local computation algorithms. In: Proc. 2nd Symposium on Innovations in Computer Science (ICS), pp. 223–238 (2011)
22. Suomela, J.: Survey of local algorithms. ACM Comput. Surv. **45**(2), 24 (2013)
23. Tarjan, R.E.: Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1983)
24. Uehara, R., Chen, Z.: Parallel approximation algorithms for maximum weighted matching in general graphs. In: Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS, pp. 84–98 (2000)
25. Vardi, S.: Designing Local Computation Algorithms and Mechanisms. PhD Thesis, Tel Aviv University, Tel Aviv, Israel (2015)
26. Vazirani, V.V.: Approximation Algorithms. Springer (2001)
27. Wattenhofer, M., Wattenhofer, R.: Distributed weighted matching. In: DISC, pp. 335–348 (2004)