Approximating the Held-Karp Bound for Metric TSP in Nearly-Linear Time

Chandra Chekuri Kent Quanrud
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL
{chekuri,quanrud2}@illinois.edu

***Abstract*—We give a nearly linear-time randomized approximation scheme for the Held-Karp bound [22] for Metric-TSP. Formally, given an undirected edge-weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ on m edges and $\epsilon > 0$, the algorithm outputs in $O(m \log^4 n/\epsilon^2)$ time, with high probability, a $(1 + \epsilon)$-approximation to the Held-Karp bound on the Metric-TSP instance induced by the shortest path metric on $\mathcal{G}$. The algorithm can also be used to output a corresponding solution to the Subtour Elimination LP. We substantially improve upon the $O(m^2 \log^2(m)/\epsilon^2)$ running time achieved previously by Garg and Khandekar.**

I. Introduction

The *Traveling Salesman Problem* (TSP) is a central problem in discrete and combinatorial optimization, and has inspired fundamental advances in optimization, mathematical programming and theoretical computer science. Cook's recent book [11] gives an introduction to the problem, its history, and general appeal. See also Gutin and Punnen [21], Applegate, Bixby, Chvatal, and Cook [4], and Lawler, Lenstra, Rinnooy-Kan, and Shmoys [29] for book-length treatments of TSP and its variants.

Formally, the input to TSP is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ equipped with positive edge costs $c : \mathcal{E} \to \mathbb{R}_{>0}$. The goal is to find a minimum cost Hamiltonian cycle in $\mathcal{G}$. In this paper we focus on TSP in undirected graphs. Checking whether a given graph has a Hamiltonian cycle is a classical NP-Complete decision problem, and hence TSP is not only NP-Hard but also inapproximable. For this theoretical reason, as well as many practical applications, a special case of TSP called Metric-TSP is extensively studied. In Metric-TSP, $\mathcal{G}$ is a complete graph K_n and c obeys the triangle inequality $c_{uv} \leq c_{uw}+c_{wv}$ for all $u, v, w \in \mathcal{V}$. An alternative interpretation of Metric-TSP is to find a minimum-cost *tour* of an edge-weighted graph $\mathcal{G}$; where a tour is a closed walk that visits all the vertices. In other words, Metric-TSP is a relaxation of TSP in which a vertex can be visited more than once. The graph-based view of Metric-TSP allows one to specify the metric on $\mathcal{V}$ implicitly and *sparsely*.

Unlike TSP, which is inapproximable, Metric-TSP admits a constant factor approximation. The classical algorithm of Christofides [10] yields a $3/2$-approximation. On the other hand it is known that Metric-TSP is APX-Hard and hence does not admit a PTAS (Lampis [28] showed that there is no $\frac{185}{184}$-approximation unless $P = NP$). An outstanding open problem is to improve the bound of $3/2$. A well-known conjecture states that the worst-case integrality gap of the *Subtour-Elimination LP* formulated by Dantzig, Fulkerson, and Johnson [12] is $4/3$ (see [17]). There has been exciting recent progress on this conjecture and several related problems; we refer the reader to an excellent survey by Vygen [37]. The Subtour Elimination LP for TSP is described below and models the choice to take an edge $e \in \mathcal{E}$ with a variable $y_e \in [0, 1]$. In the following, let $\mathcal{C}(U)$ (resp. $\mathcal{C}(v)$) denotes the set of edges crossing the set of vertices $U \subseteq \mathcal{V}$ (resp. the vertex $v \in \mathcal{V}$).

$$\begin{aligned} \mathrm{SE}(\mathcal{G}, c) = \min \quad & \langle c, y\rangle \\ \text{s.t.} \quad & \textstyle\sum_{e\in\mathcal{C}(v)} y_e = 2 && \text{for all } v \in \mathcal{V}, \\ & \textstyle\sum_{e\in\mathcal{C}(U)} y_e \geq 2 && \text{for all } \emptyset \subsetneq U \subsetneq \mathcal{V}, \\ \text{and} \quad & y_e \in [0,1] && \text{for all } e \in \mathcal{E}. \end{aligned}$$

The first set of constraints require each vertex to be incident to exactly two edges (in the integral setting); these are referred to as degree constraints. The second set of constraints force connectivity, hence the name "subtour elimination". The LP provides a lower bound for TSP, and in order to apply it to an instance of Metric-TSP defined by $\mathcal{G}$, one needs to apply it to the metric completion of $\mathcal{G}$.

A problem closely related to Metric-TSP is the 2-*edge-connected spanning subgraph problem* (2ECSS). In 2ECSS the input is an edge-weighted graph $(\mathcal{G} = (\mathcal{V}, \mathcal{E}), c)$, and the goal is to find a minimum cost subgraph of $\mathcal{G}$ that is 2-edge-connected. We focus on the simpler version where an edge is allowed to be used more than once. A natural LP relaxation for 2ECSS is the following.

$$\begin{aligned} \mathsf{2ECSS}(\mathcal{G}, c) = \min & \langle c, y\rangle \\ \text{s. t.} \sum_{e\in C} y_e \geq 2 & \qquad \forall C \in \mathcal{C} \\ \text{and } y_e \geq 0 & \qquad \forall e \in \mathcal{E} \end{aligned}$$

We have a variable y_e for each edge $e \in \mathcal{E}$, and constraints which ensure that each cut has at least two edges crossing it. The dual LP below corresponds to a maximum packing of cuts into the edge costs. In the following, let $\mathcal{C} \subseteq 2^{\mathcal{E}}$ denote the family of all cuts in $\mathcal{G}$.

$$\begin{aligned} \mathsf{2ECSSD}(\mathcal{G}, c) = \max\ & 2\langle \mathbb{1}, x\rangle \\ \text{s. t.}\ & \sum_{C \ni e} x_C \le c_e && \forall e \in \mathcal{E} \\ \text{and}\ & x_C \ge 0 && \forall C \in \mathcal{C} \end{aligned}$$

Cunningham (see [30]) and Goemans and Bertsimas [18] observed that for any edge-weighted graph $(\mathcal{G}, c)$, the optimum value of the Subtour Elimination LP for the metric completion of $(\mathcal{G}, c)$ coincides with the optimum value of the 2ECSS LP for $(\mathcal{G}, c)$. The advantage of this connection is twofold. First, the 2ECSS relaxation is a pure covering LP, and its dual is a pure packing LP. Second, the 2ECSS formulation works directly with the underlying graph $(\mathcal{G}, c)$ instead of the metric completion.

On the importance of solving the Subtour-LP: The subtour elimination LP is extensively studied in mathematical programming both for its application to TSP as well as the many techniques its study has spawned. It is a canonical example in many books and courses on linear and integer programming. The seminal paper of Dantzig, Fulkerson and Johnson proposed the cutting plane method based on this LP as a way to solve TSP exactly. Applegate, Bixby, Chvátal, and Cook [3] demonstrated the power of this methodology by solving TSP on extremely large real world instances; the resulting code named Concorde is well-known [4]. The importance of solving the subtour elimination LP to optimality has been recognized since the early days of computing. The Ellipsoid method can be used to solve the LP in polynomial time since the separation oracle required is the global mincut problem, but is not practical. One can also write polynomial-sized extended formulations using flow variables, but the number of variables and constraints is cubic in n. Held and Karp [22] provided an alternative lower bound for TSP via the notion of *one-trees*. They showed, via Lagrangian duality, that their lower bound coincides with the one given by $\mathrm{SE}(\mathcal{G}, c)$. The advantage of the Held-Karp bound is that it can be computed via a simple iterative procedure relying on minimum spanning tree computations. In practice, this iterative procedure provides good estimates for the lower bound. However, there is no known polynomial-time implementation with guarantees on the convergence rate to the optimal value.

In the rest of the paper we focus on Metric-TSP. For the sake of brevity, we refer to the Held-Karp bound for the metric completion of $(\mathcal{G}, c)$ as simply the Held-Karp bound for $(\mathcal{G}, c)$. How fast can one compute the Held-Karp bound for a given instance? Is there a strongly polynomial-time or a combinatorial algorithm for this problem? These questions have been raised implicitly and are also explicitly pointed out, for instance, in [6] and [18]. A fast algorithm has several applications ranging from approximation algorithms to exact algorithms for TSP.

Plotkin, Shmoys, and Tardos [33], in their influential paper on fast approximation schemes for packing and covering LPs via Lagrangian relaxation methods, showed that a $(1+\epsilon)$-approximation for the Held-Karp bound for Metric-TSP can be computed in $O\big(n^4 \log^6 n/\epsilon^2\big)$ randomized time. They relied on an algorithm for computing the global minimum cut[1]. Subsequently, Garg and Khandekar obtained a $(1+\epsilon)$-approximation in $O(m^2 \log^2 m/\epsilon^2)$ time and they relied on algorithms for minimum-cost branchings (see [26]).

The main result: We obtain a near-linear running time for a $(1+\epsilon)$-approximation, substantially improving the best previously known running time bound.

Theorem 1. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with $|\mathcal{E}| = m$ edges and $|\mathcal{V}| = n$ vertices, and positive edge weights $c : \mathcal{E} \to \mathbb{R}_{>0}$. For any fixed $\epsilon > 0$, there exists a randomized algorithm that computes a $(1+\epsilon)$-approximation to the Held-Karp lower bound for the* Metric-TSP *instance on $(\mathcal{G}, c)$ in $O(m \log^4 n/\epsilon^2)$ time. The algorithm succeeds with high probability.*

The algorithm in the preceding theorem can be modified to return a $(1+\epsilon)$-approximate solution to the 2ECSS LP within the same asymptotic time bound. For fixed ϵ, the running time we achieve is asymptotically faster than the time to compute or even write down the metric completion of $(\mathcal{G}, c)$. Our algorithm can be applied to low-dimensional geometric point sets to obtain a running-time that is near-linearly in the number of points.

In typical approximation algorithms that rely on mathematical programming relaxations, the bottleneck for the running time is solving the relaxation. Surprisingly, for algorithms solving Metric-TSP via the Held-Karp bound, the bottleneck is no longer solving the relaxation (albeit we only find a $(1+\epsilon)$-approximation and do not guarantee a basic feasible solution). We mention that the recent approaches towards the $4/3$ conjecture for Metric-TSP are based on variations of the classical Christofides heuristic (see [37]). The starting point is a near-optimal feasible solution x to the 2ECSS LP on $(\mathcal{G}, c)$. Using a well-known fact that a scaled version of x lies in the spanning tree polytope of $\mathcal{G}$, one generates one or more (random) spanning trees T of $\mathcal{G}$. The tree T is then augmented to a tour via a min-cost matching M on its odd degree nodes. Genova and Williamson [16] recently evaluated some of these *Best-of-Many Christofides' algorithms* and demonstrated their effectiveness. A key step in this scheme, apart from solving the LP, is to decompose a given point y in the spanning tree polytope of $\mathcal{G}$ into a convex combination of spanning trees. Our recent work [7] shows how to achieve a $(1-\epsilon)$-approximation for this task in near-linear time; the algorithm implicitly stores the decomposition in near-linear space.

[1]Their scheme can in fact be implemented in randomized $O(m^2 \log^4 m/\epsilon^2)$ time using subsequent developments in minimum cut algorithms and width reduction techniques.

One remaining bottleneck to achieve an overall near-linear running time is to compute an approximate min-cost perfect matching on the odd-degree nodes of a given spanning tree T.

A. Integrated design of the algorithm

Our algorithm is based on the multiplicative weight update framework (MWU), and like Plotkin, Shmoys, and Tardos [33], we approximate the pure packing LP 2ECSSD. Each iteration requires an oracle for computing the global minimum cut in an undirected graph. A single minimum cut computation takes randomized near-linear-time via the algorithm of [25], and the MWU framework requires $\tilde{\Omega}(m/\epsilon^2)$ iterations. Suprisingly, the whole algorithm can be implemented to run in roughly the same time as that required to compute one global mincut.

While the full algorithm is fairly involved, the high-level design is directed by some ideas developed in recent work by the authors [7] that is inspired by earlier work of Mądry [31] and Young [39]. We accelerate MWU-based algorithms for some implicit packing problems with the careful interplay of two data structures. The first data structure maintains a minimum cost object of interest (here the global minimum cut) by (partially) dynamic techniques, rather than recompute the object from scratch in every iteration. The second data structure applies the multiplicative weight update in a lazy fashion that can be amortized efficiently against the weights of the constraints. The two data structures need to be appropriately meshed to obtain faster running times, and this meshing depends very much on the problem specifics as well as the details of the dynamic data structures.

While we do inherit some basic ideas and techniques from this framework, the problem here is more sophisticated than those considered in [7]. The first component in this paper is a fast dynamic data structure for maintaining a $(1+\epsilon)$-approximate global minimum cut of a weighted graph whose edge weights are only increasing. We achieve an amortized poly-logarithmic update time by a careful adaptation of the randomized near-linear-time minimum cut algorithm of Karger [25] that relies on approximate tree packings. This data structure is developed with careful consideration of the MWU framework; for example, we only need to compute an approximate tree packing $\tilde{O}(1/\epsilon^2)$ times (rather than in every iteration) because of the monotonicity of the weight updates and standard upper bounds on the total growth of the edge weights.

The second technical ingredient is a data structure for applying a multiplicative weight update to each edge in the approximately minimum cuts selected in each iteration. The basic difficulty here is that we cannot afford to touch each edge in the cut. While this task suggests a lazy weight update similar to [7], these techniques require a compact representation of the minimum cuts. It appears difficult to develop in isolation a data structure that can apply multiplicative weight updates along any (approximately) minimum cut. However, additional nice properties of the cuts generated by the first dynamic data structure enable a clean interaction with lazy weight updates. We develop an efficient data structure for weight updates that is fundamentally inextricable from the data structure generating the approximately minimum cuts.

Remark 2. If we extract the data structure for minimum cuts from the MWU framework, then we obtain the following.

> *Given an edge-weighted G graph on m edges there is a dynamic data structure for the weighted incremental maintenance of a $(1+\epsilon)$-approximate global minimum cut that updates and queries in constant time plus $O(m \operatorname{polylog}(n) \log(W_1/W_0)/\epsilon)$ total amortized time, where W_0 is the weight of the minimum cut of the initial graph and W_1 is the weight of the minimum cut of the final graph (after all updates). Here, updates in the weighted incremental setting consist of an edge e and a positive increment α to its weight. The edges of the approximate minimum cut can be reported in constant time per edge reported.*

However, a data structure for dynamic maintenance of minimum cuts is not sufficient on its own, as there are several subtleties to be handled both appropriately and efficiently. For instance, Thorup [35] developed a *randomized fully-dynamic* data structure for global mincut with a worst-case update time of $\tilde{O}(\sqrt{n})$. Besides the slower update time, this data structure assumes a model and use case that clashes with the MWU framework at two basic points. First, the data structure does not provide access to the edges of the mincut in an implicit fashion that allows the edge weights to be updated efficiently, and updating each of the edges of the minimum cut individually leads to quadratic running times. Second, the randomization in the data structure assumes an oblivious adversary, which is not suitable for our purposes where the queries of the MWU algorithm to the data structure are *adaptive*.

B. Related Work

There has been a surge of interest in faster approximation algorithms for classical problems in combinatorial optimization including flows, cuts, matchings and linear programming to name a few. These are motivated by not only theoretical considerations and techniques but also practical applications with very large data sizes. Many of these are based on the interplay between discrete and continuous techniques and there have been several breakthroughs, too many to list here. Our work adds to the list of basic problems that admit a near-linear-time approximation scheme.

In another direction there has been exciting work on approximation algorithms for TSP and its variants including Metric-TSP, Graphic-TSP, Asymmetric-TSP, TSP-Path to name a few. Several new algorithms, techniques, connections and problems have arisen out of this work. Instead of recapping the many results we refer the reader to the survey [37]. Our result adds to this literature and suggests that one can obtain not only improved approximation but also much faster approximations than what was believed possible. As we already mentioned, some other ingredients in the rounding of a solution such as decomposing a fractional point in a spanning tree polytope into convex combination of spanning trees can also be sped up using some of the ideas in our prior work.

In terms of techniques, our work falls within the broad framework of improved implementations of iterative algorithms. Our recent work showed that MWU based algorithms, especially for implicit packing and covering problems, have the potential to be substantially improved by taking advantage of data structures. The key, as we described already, is to combine multiple data structures together while exploiting the flexibility of the MWU framework. Although not a particularly novel idea (see the work of Mądry [31] for applications to multicommodity flow and the work of Agarwal and Pan [1] for geometric covering problems), our work demonstrates concrete and interesting problems to highlight the sufficient conditions under which this is possible. Lagrangian-relaxation based approximation schemes for solving special classes of linear programs have been studied for several decades with a systematic investigation started in Plotkin, Shmoys, and Tardos [33] and Grigoriadis and Khachiyan [20] following earlier work on multicommodity flows. Since then there have been many ideas, refinements and applications. We refer the reader to [27, 39, 9] for some recent papers on MWU-based algorithms which supply further pointers, and to the survey [5] for the broader applicability of MWU in theoretical computer science. Accelerated gradient descent methods have recently resulted in fast randomized near-linear-time algorithms for explicit fractional packing and covering LPs; these algorithms improved the dependence of the running time on ϵ to $1/\epsilon$ from $1/\epsilon^2$. See [2] and [38].

Our work here builds extensively on Karger's randomized nearly-linear-time mincut algorithm [25]. As mentioned already, we adapt his algorithm to a partially dynamic setting informed by the MWU framework. [35] also builds on Karger's tree packing ideas to develop a *fully* dynamic mincut algorithm. Thorup's algorithm is rather involved and is slower than what we are able to achieve for the partial dynamic setting; he achieves an update time of $\widetilde{O}(\sqrt{n})$ while we achieve polylogarithmic time. There are other obstacles to integrating his ideas and data structure to the needs of the MWU framework as we already remarked. For unweighted incremental mincut, Goranci, Henzinger, and Thorup [19] recently developed a deterministic data structure with a polylogarithmic amortized update time.

Some of the data structures in this paper rely on Euler

```
held-karp-1($\mathcal{G} = (\mathcal{V}, \mathcal{E}), c, \epsilon$)
  $x \leftarrow \mathbb{0}^{\mathcal{C}}$, $w \leftarrow 1/c$, $t \leftarrow 0$, $\eta \leftarrow \epsilon / \ln m$
  while $t < 1$
    $C \leftarrow \arg\min_{C \in \mathcal{C}} \overline{w}(C)$ // min-cut
    $y \leftarrow \frac{\langle w, c \rangle}{\overline{w}(C)} \cdot 1_C$, $\gamma \leftarrow \min_{e \in C} c_e$, $\delta \leftarrow \frac{\epsilon \gamma}{\eta}$, $x \leftarrow x + \delta y$
    for all $e \in C$
      $w_e \leftarrow w_e \exp(\delta \eta / c_e) = w_e \exp(\epsilon \gamma / c_e)$
    $t \leftarrow t + \delta$
  return $x$
```

Figure 1. An exact (and slow) implementation of the MWU framework applied to packing cuts.

tour representations of spanning trees. The Euler tour representation imposes a particular linear order on the vertices by which the edges of the underlying graph can be interpreted as intervals. The Euler tour representation was introduced by Tarjan and Vishkin [34] and has seen applications in other dynamic data structures, such as the work by Henzinger and King [23] among others.

Organization: Our main result is a combination of some high-level ideas and several data structures, and the implementation details take up considerable space. Section II gives a high-level overview of a first MWU-based algorithm, highlighting some important properties of the MWU framework that arise in more sophisticated arguments later. Section III summarizes the key properties of tree packings that underlie Karger's mincut algorithm, and explains their use in our MWU algorithm via the notion of epochs. Section IV describes an efficient subroutine, via appropriate data structures, to find all approximate mincuts (in the partial dynamic setting) induced by the spanning trees of a tree packing. Section V describes the data structure that implement the weights of the edges in a lazy fashion, building on Euler tours of spanning trees, range trees, and some of our prior work. In Section VI, we outline a formal proof of Theorem 1. Due to space constraints, many details and figures have been omitted. We encourage the reader to read the full version of the paper [8] instead.

II. MWU based Algorithm and Overview

Given an edge-weighted graph ($\mathcal{G} = (\mathcal{V}, \mathcal{E}), c$) our goal is to find a $(1-\epsilon)$-approximation to the optimal value of the LP 2ECSSD. We later describe how to obtain an approximate solution to the primal LP 2ECSS. Note that the LP is a packing LP of the form $\max\langle \mathbb{1}, x \rangle : Ax \leq c$ with an exponential number of variables corresponding to the cuts, but only m non-trivial constraints corresponding to the edges. The MWU framework can be used to obtain a $(1-\epsilon)$-approximation for fractional packing. Here we pack cuts into the edge capacities c, which MWU reduces to finding global minimum cuts, as follows. The framework starts with an empty solution $x = \mathbb{0}^{\mathcal{C}}$ and maintains edge weights $w : \mathcal{E} \rightarrow \mathbb{R}_{\geq 0}$, initialized to $1/c$ and *non-decreasing* over the course of the algorithm. Each iteration, the framework finds the minimum cut $C = \arg\min_{C' \in \mathcal{C}} \overline{w}(C')$ w/r/t w, where $\overline{w}(C') \overset{\text{def}}{=} \sum_{e \in C'} w_e$ denotes the total weight of a cut. The framework adds a fraction of C to the solution x, and updates the weight of every edge in the cut in a multiplicative fashion such that the weight of an edge is exponential in its load $x(e)/c(e)$. The weight updates steer the algorithm away from reusing highly loaded edges. The MWU framework guarantees that the output x will have objective value $\langle \mathbb{1}, x \rangle \geq$ OPT while satisfying $\sum_{C \ni e} x_C \leq (1+O(\epsilon))c_e$ for every edge e. Scaling down x by a $(1+O(\epsilon))$-factor gives a $(1-O(\epsilon))$-approximation satisfying all packing constraints.

An actual implementation of the above high-level idea needs to specify the step size in each iteration and the precise weight-update. The non-uniform increments idea of Garg and Könemann [15] gives rise to a width-independent bound on the number of iterations, namely $O(m \log m/\epsilon^2)$ in our setting. Our algorithms follow the specific notation and scheme of Chekuri et al. [9], which tracks the algorithm's progress by a "time" variable t from 0 to 1 increasing in non-uniform steps. A step of size δ in an iteration corresponds to adding $\delta\beta 1_C$ to the current solution x where 1_C is the characteristic vector of the mincut C found in the iteration, and $\beta = \langle w, c \rangle / \overline{w}(C)$ greedily takes as much of the cut as can fit in the budget $\langle w, c \rangle$. The process is controlled by a parameter η, which when set to $O(\ln m/\epsilon)$ balances the width-independent running time with a $(1-\epsilon)$-approximation factor.

Both the correctness and the number of iterations are derived by a careful analysis of the edge weights w. We review the argument that bounds the number of iterations. At the beginning of the algorithm, when every edge e has weight $w_e = 1/c_e$, we have $\langle w, c \rangle = m$. The weights monotonically increase, and standard proofs show that this inner product is bounded above by $\langle w, c \rangle \leq \tilde{O}\big(m^{O(1/\epsilon)}\big)$. The edge weight increases are carefully calibrated so that at least one edge increases by a $(1+\epsilon)$-multiplicative factor in each iteration. As the upper bound on $\langle w, c \rangle$ is an upper bound on $w_e c_e$ for any edge e, and the initial weight of an edge e is $1/c_e$, an edge weight w_e can increase by a $(1+\epsilon)$-factor at most $\log_{(1+\epsilon)}\big(m^{O(1/\epsilon)}\big) = O\big(\log(m)/\epsilon^2\big)$ times. Charging each iteration to an edge weight increased by a $(1+\epsilon)$-factor, the algorithm terminates in $\tilde{O}\big(m/\epsilon^2\big)$ iterations.

A direct implementation of the MWU framework, `held-karp-1`, is given in Fig. 1. Each iteration requires the minimum global cut w/r/t the edge weights w, which can be found in $\tilde{O}(m)$ time (with high probability) by the randomized algorithm of Karger [25]. Calling Karger's algorithm in each iteration gives a quadratic running time of $\tilde{O}\big(m^2/\epsilon^2\big)$.

Approximation in the framework

The MWU framework is robust to approximation, and we exploit this slack at two points in particular.

```
held-karp-2($\mathcal{G} = (\mathcal{V}, \mathcal{E}), c, \epsilon$)
  $x \leftarrow 0^{\mathcal{C}}$, $w \leftarrow 1/c$, $t \leftarrow 0$, $\eta \leftarrow \epsilon/\ln m$
  $\lambda \leftarrow$ size of minimum cut w/r/t edge weights $w$
  while $t < 1$
    while (a) $t < 1$ and
          (b) $\exists$ cut $C \in \mathcal{C}$ s.t. $\overline{w}(c) \leq (1+\epsilon)\lambda$
      $y \leftarrow \frac{\langle w, c \rangle}{\overline{w}(C)} \cdot 1_C$, $\gamma \leftarrow \min_{e \in C} c_e$, $\delta \leftarrow \frac{\epsilon\gamma}{\eta}$, $x \leftarrow x + \delta y$
      for all $e \in C$, $w_e \leftarrow \exp(\delta\eta/c_e) \cdot w_e = \exp(\epsilon\gamma/c_e) \cdot w_e$
      $t \leftarrow t + \delta$
    $\lambda \leftarrow (1+\epsilon)\lambda$ // start new epoch
  return $x$
```

Figure 2. A second implementation of the MWU framework applied to packing cuts that divides the iterations into epochs and seeks only approximately minimum cuts at each iteration.

Approximate mincuts: First, we only look for cuts that are within a $(1+O(\epsilon))$-approximate factor of the true minimum cut. To this end, we maintain a target cut value $\lambda > 0$, and maintain the invariant that there are no cuts of value strictly less than λ. We then look for cuts of value $\leq (1+O(\epsilon))\lambda$ until we are sure that there are no more cuts of value $\leq (1+\epsilon)\lambda$. When we have certified that there are no cuts of value $\leq (1+\epsilon)\lambda$, we increase λ to $(1+\epsilon)\lambda$, and repeat. Each stretch of iterations with the same target value λ is called an *epoch*. Epochs were used by [13] for approximating fractional multicommodity flow. We show that *all* the approximately minimum cuts in a single epoch can be processed in $\tilde{O}(m)$ total time (plus some amortized work bounded by other techniques). If κ is the weight of the initial minimum cut (when $w = 1/c$), then the minimum cut size increases monotonically from κ to $\tilde{O}\big(m^{1/\epsilon}\kappa\big)$. If the first target cut value is $\lambda = \kappa$, then there are only $O(\ln m/\epsilon^2)$ epochs over the course of the entire algorithm.

Lazy weight updates: Even if we can concisely identify the approximate minimum cuts quickly, there is still the matter of updating the weights of all the edges in the cut quickly. A cut can have $\Omega(m)$ edges, and updating m edge weights in each of $\tilde{O}\big(m/\epsilon^2\big)$ iterations leads to a $\tilde{O}\big(m^2/\epsilon^2\big)$ running time. Instead of visiting each edge of the cut, we have a subroutine that simulates the weight increase to all the edges in the cut, but only does work for each edge whose (true) weight increases by a full $(1+\epsilon)$-multiplicative power. That is, for each edge e, we do work proportional (up to log factors) to the number of times w_e increases to the next integer power of $(1+\epsilon)$. By the previous discussion on weights, the number of such large updates is at most $O(\log m/\epsilon^2)$ per edge. This amortized efficiency comes at the expense of approximating the (true) weight of each edge to a $(1+O(\epsilon))$-multiplicative factor. Happily, an approximate minimum cut on approximate edge weights is still an approximate minimum cut to the true weights, so these approximations are tolerable. In [7] we isolated a specific lazy-weight scheme from Young [39] into a data structure with a clean interface that we build upon here.

A. Obtaining a primal solution

Our MWU algorithm computes a $(1-\epsilon)$-approximate solution to 2ECSSD. Recall that the algorithm maintains a weight $w(e)$ for each edge e. Standard arguments show that one can recover from the evolving weights a $(1+\epsilon)$-approximate solution to the LP 2ECSS (which is the dual of 2ECCSD). A sketch is provided in the appendix of the full version [8].

III. Tree packings and epochs

Karger [25] gave a randomized algorithm that finds the global minimum cut in a weighted and undirected graph with high probability in $\tilde{O}(m)$ time. Treating Karger's algorithm as a black box in every iteration leads to a quadratic running time of $\tilde{O}(m^2/\epsilon^2)$, so we open it up and adjust the techniques to our setting.

Tree packings: Karger [25] departs from previous algorithms for minimum cut with an approach based on packing spanning trees. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with positive edge weights $w : \mathcal{E} \to \mathbb{R}_{>0}$, and let $\mathcal{T} \subseteq 2^{\mathcal{E}}$ denote the family of spanning trees in $\mathcal{G}$. A *tree packing* is a nonnegatively weighted collection of spanning trees, $p : \mathcal{T} \to \mathbb{R}_{\geq 0}$, such that for any edge e, the total weight of trees containing e is at most c_e (i.e., $\sum_{T \ni e} p(T) \leq c_e$). Classical work of Tutte [36] and Nash-Williams [32] gives an exact characterization for the value of a maximum tree packing in a graph, which as an easy corollary implies it is at least half of the value of a mincut. If $C \in \mathcal{C}$ is a minimum cut and p is a maximum packing, then a tree $T \in \mathcal{T}$ selected randomly in proportion to its weight in the packing will share ≤ 2 edges with C in expectation. By Markov's inequality, T has strictly less than 3 edges in C with constant probability.

For a fixed spanning tree T, a *one-cut* in G induced by an edge $e \in T$ is the cut $\mathcal{C}(X)$ where X is the vertex set of one of the components of $T - e$. A *two-cut* induced by two edges $e_1, e_2 \in T$ is the following. Let X, Y, Z be the vertex sets of the three components of $T - \{e_1, e_2\}$ where X is only incident to e_1 and Z is only incident to e_2 and Y is incident to both e_1, e_2. Then the two-cut induced by e_1, e_2 is $\mathcal{C}(Y) = \mathcal{C}(X \cup Z)$. Thus, if T is a spanning tree sampled from a maximum tree packing, and C is a minimum cut, then C is either a one-cut or a two-cut with constant probability.

The probabilistic argument extends immediately to $(1+\zeta)$-approximate mincuts and $(1-\zeta)$-approximate tree packings for $\zeta > 0$ sufficiently small. For constant $\zeta > 0$, a $(1-\zeta)$-approximate tree-packing can be computed in $\tilde{O}(m)$ time, either by applying $\tilde{O}(\kappa m)$-time tree packing algorithms [14, 33] to a randomly sparsified graph [24], or directly and deterministically in $\tilde{O}(m)$ time by a recent algorithm of Chekuri and Quanrud [7].

Another consequence of Karger [25] is that for $\zeta < 1/2$, the number of $(1+\zeta)$-approximate minimum cuts is at most n^2. By the union bound, if we select $O(\log n)$ trees at random from an approximately maximum tree packing, then with high probability *every* $(1+\zeta)$-approximate minimum cut is induced by one or two edges in one of the selected trees. In summary, we have the following.

Theorem 3 (25)**.** *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with edge capacities $c : \mathcal{E} \to \mathbb{R}_{>0}$, let $\delta \in (0,1)$, and let $\epsilon \in (0, 1/2)$. One can generate, in $O(m + n\log^3 n)$ time, $h = O(\log(n/\delta))$ spanning trees $T_1, T_2, \ldots, T_h$ such that with probability $\geq 1-\delta$,* every *$(1+\epsilon)$-approximate minimum cut C has $|C \cap T_i| \leq 2$ for some tree T_i.*

Karger [25] finds the minimum cut by checking, for each tree T_i, the minimum cut in G obtained by removing one or two edges from T_i in $\tilde{O}(m)$ time. (The details of this subroutine are reviewed in the following section.) Since there are $O(\log n)$ trees, this amounts to a near-linear-time algorithm. Whereas Karger's algorithm finds one minimum cut, we need to find many minimum cuts. Moreover, each time we find one minimum cut, the edge weights on that cut increase and so the underlying graph changes. We are faced with the challenge of outputting $\tilde{O}(m/\epsilon^2)$ cuts from a dynamically changing graph that in particular adapts to each cut output, all in roughly the same amount of time as a single execution of Karger's algorithm.

Epochs: We divide the problem into $O(\log m/\epsilon^2)$ epochs. Recall that at the start of an epoch we have a target value $\lambda > 0$ and are guaranteed that there are no cuts with value strictly less than λ. Our goal is to repeatedly output cuts with value $\leq (1+O(\epsilon))\lambda$ or certify that there are no cuts of value strictly less than $(1+\epsilon)\lambda$, in $\tilde{O}(m)$ time. (The reason we output a cut with value $\leq (1+O(\epsilon))\lambda$ is because the edge weights are maintained only approximately.) Each epoch is (conceptually) structured as follows.

- At the start of the epoch we invoke Theorem 3 to find $h = O(\log(n/\epsilon))$ trees $T_1, \ldots, T_h$ such that with probability $1 - \text{poly}(\epsilon/n)$, every $(1+\epsilon)$-approximate mincut of G with respect to the weights at the start of the epoch is a one-cut or two-cut of some tree T_i.
- Let $C_1, C_2, \ldots, C_r$ be an *arbitrary* enumeration of one and two-cuts induced by the trees.
- For each such cut C_j in the order, if the *current weight* of C_j is $\leq (1+O(\epsilon))\lambda$ output it to the MWU algorithm and update weights per the MWU framework (weights only increase). Reuse C_j as long as its current weight is $\leq (1+O(\epsilon))\lambda$.

Since weights of cuts only increase, and we consider every one and two-cut of the trees we obtain the following lemma.

Lemma 4. *Let $w : \mathcal{E} \to \mathbb{R}_{>0}$ be the weights at the start of the epoch and suppose mincut of G with respect to w is $\geq \lambda$; and suppose the trees $T_1, \ldots, T_h$ have the property that every $(1+\epsilon)$-approximate mincut is induced by a one or two-edge cut of one of the trees. Let $w' : \mathcal{E} \to \mathbb{R}_{>0}$ be*

```
held-karp-3($\mathcal{G} = (\mathcal{V},\mathcal{E})$, $c$, $\epsilon$)
  $x \leftarrow \mathbb{0}^{\mathcal{C}}$, $w \leftarrow 1/c$, $t \leftarrow 0$, $\eta \leftarrow \epsilon/\ln m$
  $\lambda \leftarrow$ size of minimum cut w/r/t edge weights $w$
  while $t < 1$
    $p \leftarrow O(\log(n/\epsilon))$ spanning trees per Theorem 3
    for each tree $T \in p$
      let $\mathcal{C}_T \subseteq \mathcal{C}$ be 1- and 2-cuts of $T$
      while (a) $t < 1$ and
            (b) $\exists$ cut $C \in \mathcal{C}_T$ s.t. $\overline{w}(C) < (1+\epsilon)\lambda$
        $y \leftarrow \frac{\langle w,c\rangle}{\overline{w}(C)} \cdot C$, $\gamma \leftarrow \min_{e\in C} c_e$, $\delta \leftarrow \frac{\epsilon\gamma}{\eta}$, $x \leftarrow x + \delta y$
        for all $e \in C$, $w_e \leftarrow \exp(\epsilon\gamma/c_e)\cdot w_e$
        $t \leftarrow t + \delta$
    $\lambda \leftarrow (1+\epsilon)\lambda$ // start new epoch
  return $x$
```

Figure 3. A third implementation of the MWU framework applied to packing cuts that samples a tree packing once at the beginning of each epoch and searches for approximately minimum cuts induced by one or two edges in the sampled trees.

the weights at the end of an epoch. Then, the mincut with respect to weights w' is $\geq (1+\epsilon)\lambda$.

Within an epoch, we must consider all cuts induced by 1 or 2 edges in $\tilde{O}(1)$ spanning trees. We have complete flexibility in the order we consider them. For an efficient implementation we process the trees one at a time. For each tree T, we need to repeatedly output cuts with value $\leq (1+O(\epsilon))\lambda$ or certify that there are no cuts of value strictly less than $(1+\epsilon)\lambda$ induced by one or two edges of T. Although the underlying graph changes, we do not have to reconsider the same tree again because the cut values are non-decreasing. If each tree can be processed in $\tilde{O}(m)$ time, then an entire epoch can be processed in $\tilde{O}(m)$ time and the entire algorithm will run in $\tilde{O}(m/\epsilon^2)$ time. In Fig. 3, we revise the algorithm to invoke Theorem 3 once per epoch and use the trees to look for approximately minimum cuts, while abstracting out the search for approximately minimum cuts in each tree.

IV. Cuts induced by a tree

Let $T \in \mathcal{T}$ be a rooted spanning tree of $\mathcal{G}$, and let $\lambda > 0$ be a fixed value such that the minimum cut value w/r/t w is $\geq \lambda$. We want to list cuts $C \in \mathcal{C}$ induced by 1 or 2 edges in T with weight $\overline{w}(C) \leq (1+\epsilon)\lambda$, in any order, but with a twist: each time we select one cut C, we increment the weight of every edge in C, and these increments must be reflected in all subsequent cuts. We are allowed to output cuts of value $\leq (1+O(\epsilon))\lambda$. When we finish processing T, we must be confident that there are no 1-cuts or 2-cuts $C \in \mathcal{C}_T$ induced by T with weight $\overline{w}(C) \leq (1+\epsilon)\lambda$.

The algorithmic challenge here is twofold. First, we need to find a good 1-cut or 2-cut in T quickly. Second, once a good cut is found, we have to increment the weights of all edges in the cut. Either operation, if implemented in time proportional to the number of edges in the cut, can take $O(m)$ time in the worst case. To achieve a nearly-linear running time, both operations must be implemented in polylogarithmic amortized time.

In this section we focus on finding the cuts, and assume that we can update the edge weights along any 1 or 2-cut efficiently via the `lazy-inc-cuts` data structure. Since the weights are dynamically changing, we need to describe how the algorithm finding small 1-cuts and 2-cut in T interacts with the `lazy-inc-cuts` data structure.

Interacting with the lazy weight update data structure: The primary function of the `lazy-inc-cuts` data structure is `inc-cut`, which takes as input one or two edges in T, and simulates a weight update along the corresponding 1-cut or 2-cut. `inc-cut` returns a list of weight increments (e_i, δ_i), where each e_i is an edge and $\delta_i > 0$ is a positive increment that we actually apply to the underlying graph. If we let w_e denote the "true weight" of an edge e implied by a sequence of weight increments along cuts, and $\widetilde{w}_e$ the sum of δ increments for e returned by `inc-cut`, then the `lazy-inc-cuts` data structure maintains the invariant that $\widetilde{w}_e \leq w_e \leq (1+\rho\epsilon)\widetilde{w}_e$, where ρ is a fixed constant that we can adjust. In particular, a cut of weight $\leq (1+\epsilon)\lambda$ w/r/t the approximated edge weights retains weight $\leq (1+O(\epsilon))\lambda$ in the true underlying graph. Each increment (e,δ) returned by `inc-cut` reflects a $(1+\Omega(\epsilon/\log^2 n))$-multiplicative increase to w_e. As the weight of an edge is bounded above by $\tilde{O}(m^{1/\epsilon})$, the subroutine returns an increment for a fixed edge e at most $O(\log^3 n/\epsilon^2)$ times total over the entire algorithm.

A second routine, `flush()`, returns a list of increments that captures all residual weight increments not accounted for in previous calls to `inc-cut` and `flush`. The increments returned by `flush()` may not be very large, but restores $\widetilde{w}_e = w_e$ for all e. Formally, we will refer to the following.

Lemma 5. *Let $\mathcal{G} = (\mathcal{V},\mathcal{E})$ be an undirected graph with positive edge capacities $c : \mathcal{E} \to \mathbb{R}_{>0}$ and positive edge weights $w : \mathcal{E} \to \mathbb{R}_{>0}$, and T a rooted spanning tree in $\mathcal{G}$. Consider an instance of* `lazy-inc-cuts(`$\mathcal{G}$, c, w, T`)` *over a sequence of calls to* `inc-cut` *and* `flush`*. For each edge e, let w_e be the "true" weight of edge e if the MWU framework were followed exactly, and let $\widetilde{w}_e$ be the approximate weight implied by the weight increments returned by calls to* `inc-cut` *and* `flush`*.*

(a) After each call to `inc-cut`*, we have $\widetilde{w}_e \leq w_e \leq (1+O(\epsilon))\widetilde{w}_e$ for all edges $e \in \mathcal{E}$.*
(b) After each call to `flush`*, we have $\widetilde{w}_e = w_e$.*
(c) Each call to `inc-cut` *takes $O(\log^2 n + I)$, where I is the total number of weight increments returned by* `inc-cut`*.*
(d) Each call to `flush` *takes $O(m\log^2 n)$ time and returns at most m weight increments.*
(e) Each increment (e,δ) returned by `inc-cut` *satisfies $\delta \geq \Omega((\epsilon/\log^2 n)w_e)$.*

The implementation details and analysis of `lazy-inc-`

```
held-karp-4($\mathcal{G} = (\mathcal{V},\mathcal{E})$, $c$, $\epsilon$)
  $x \leftarrow \mathbb{0}^{\mathcal{C}}$, $w \leftarrow \mathbb{1}^{\mathcal{E}}$, $t \leftarrow 0$, $\eta \leftarrow \epsilon/\ln m$
  $\lambda \leftarrow$ size of minimum cut w/r/t edge weights $w$
  while $t < 1$
    $p \leftarrow O(\log(n/\epsilon))$ spanning trees per Theorem 3
    for each tree $T \in p$
      lic $\leftarrow$ lazy-inc-cuts.init($\mathcal{G}$, $c$, $T$)
      let $\mathcal{C}_T \subseteq \mathcal{C}$ be the 1- and 2-cuts of $T$
      while (a) $t < 1$ and until
            (b) there are no cuts $C \in \mathcal{C}_T$
                s.t. $\overline{w}(C) < (1+\epsilon)\lambda$
        find $C \in \mathcal{C}_T$ w/ $\overline{w}(e) < (1+O(\epsilon))\lambda$
        $y \leftarrow \left(\frac{\langle w,c\rangle}{\overline{w}(C)}\right)\cdot C$, $\gamma \leftarrow \min_{e\in C} c_e$, $\delta \leftarrow \frac{\epsilon\gamma}{\eta}$, $x \leftarrow x + \delta y$,
        $\Delta \leftarrow$ lic.inc-cut($C$)
        for each $(e,\delta) \in \Delta$, $w_e \leftarrow w_e + \delta$
        $t \leftarrow t + \delta$
      $\Delta \leftarrow$ lic.flush()
      for each $(e,\delta) \in \Delta$, $w_e \leftarrow w_e + \delta$
  return $x$
```

Figure 4. A fourth implementation of the MWU framework applied to the Held-Karp relaxation, that integrates the `lazy-inc-cuts` data structure into the subroutine that processes each tree.

`cuts` are given afterwards in Section V. An enhanced sketch of the algorithm so far, integrating the `lazy-inc-cuts` data structure but abstracting out the search for small 1- and 2-cuts, is given in Fig. 4. In this section, we assume Lemma 5 and prove the following.

Lemma 6. *Let T be a rooted spanning tree and $\lambda > 0$ a target cut value. One can repeatedly output 1-cuts and 2-cuts $C \in \mathcal{C}_T$ induced by T of weight $\overline{w}(C) \leq (1+O(\epsilon))\lambda$ and increment the corresponding edge weights (per the MWU framework) until certifying that there are no 1-cuts or 2-cuts $C \in \mathcal{C}_T$ of value $\overline{w}(C) \leq (1+\epsilon)\lambda$ in $O\big(m\log^2 n + K\log^2 n + I\log n\big)$ time, where K is the number of 1-cuts and 2-cuts of value $\leq (1+O(\epsilon))\lambda$ output and I is the total number of weight increments returned by the* `lazy-inc-cuts` *data structure.*

The proof of Lemma 6 is omitted due to space constraints. The proof carefully integrates Karger's dynamic programming approach with the `lazy-inc-cuts` data structure and is given in the full version [8].

V. Multiplicative weight updates along cuts

We address the remaining issue of implementing the `lazy-inc-cuts` data structure for a fixed and rooted spanning tree T, per Lemma 5. Recall that the MWU framework takes a cut $C \in \mathcal{C}$, computes the smallest capacity $\gamma = \arg\min_{e\in C} c_e$ in the cut, and increases the weight of each cut-edge $e \in C$ by a multiplicative factor of $\exp(\epsilon\gamma/c_e)$ (see Fig. 1). A subtle point is that the techniques of Section IV identify approximate min-cuts without explicitly listing the edges in the cut. A 1-cut $\mathcal{C}(D(s))$ is simply identified by the root s of the down-set $D(s)$, and likewise 2-cuts of the form $\mathcal{C}(D(s) \cup D(t))$ (when $s \parallel t$) and $\mathcal{C}(D(t) \setminus D(s))$ (when $s < t$) can be described by the two nodes s and t. In particular, an approximate min-cut C is identified without paying for the number of edges $O(|C|)$.

While incrementing weights along a cut appears difficult to execute both quickly and exactly, we have already massaged the setting to be substantially easier. First, we can afford to approximate the edge weights w_e by a small multiplicative error. Second, we are not incrementing weights along any cut, but just the 1-cuts and 2-cuts of a fixed rooted spanning tree T. We have already seen in Section IV that restricting ourselves to 1-cuts and 2-cuts allows us to (basically) apply dynamic programming to find small cuts, and also allows us to use dynamic trees to efficiently update and scan various values in the aggregate. Here too we will see that 1-cuts and 2-cuts are simple enough to be represented efficiently in standard data structures.

Lemma 7. *Let T be a fixed rooted spanning tree of an undirected graph $\mathcal{G} = (\mathcal{V},\mathcal{E})$ with $|\mathcal{E}| = m$ edges and $|\mathcal{V}| = n$ vertices. In $O\big(m\log^2 n\big)$ time, one can construct a collection of nonempty cuts $\mathcal{D}_T \subseteq 2^{\mathcal{E}}$ such that*

(i) every edge $e \in \mathcal{E}$ appears in at most $O(\log^2 n)$ cuts $D \in \mathcal{D}_T$, and

(ii) every 1-cut or 2-cut $C \in \mathcal{C}_T$ (described succinctly by at most 2 roots of subtrees) can be decomposed into the disjoint union $C = D_1 \sqcup \cdots \sqcup D_\ell$ of $\ell = O(\log^2 n)$ cuts $D_1,\ldots,D_\ell \in \mathcal{D}_T$ in $O(\log^2 n)$ time.

By building the collection $D \in \mathcal{D}_T$ once for a tree T, Lemma 7 reduces the problem of incrementing along any 1-cut or 2-cut $C \in \mathcal{C}_T$ to incrementing along a "canonical" cut $D \in \mathcal{D}_T$ known *a priori*. This is important because multiplicative weight updates can be applied to a *static* set relatively efficiently by known techniques [39, 7]. It is also important that cuts in $\mathcal{D}_T$ are sparse, in the sense that $\sum_{D\in\mathcal{D}_T} |D| = O\big(m\log^2 n\big)$, as this sum factors directly into the running time guarantees. We first prove Lemma 7 in Section V-A, and then in Section V-B we show how to combine amortized data structures for each $D \in \mathcal{D}_T$ to increment along any 1-cut or 2-cut $C \in \mathcal{C}_T$.

A. Canonical cuts

Let T be a fixed and rooted tree on n vertices $\mathcal{V}$, and fix an Euler tour on a bidirected copy of T that replaces each edge of T with arcs in both directions, starting from the root. For each vertex v, we create two symbols: v^- means we enter the subtree T_v rooted at v, and v^+ means we leave the subtree v. Let $\mathcal{V}^{\pm} = \{v^-, v^+ : v \in \mathcal{V}(T)\}$ denote the whole collection of these $2n$ symbols. The Euler tour enters and leaves each subtree exactly once in a fixed order. Tracing the Euler tree induces a unique total ordering on $\mathcal{V}^{\pm}$.

The Euler order on the vertices endows a sort of geometry to the edges. Each edge $e = (u,v)$ (with $u^- < v^-$) can be thought of as an interval $[u^-, v^-]$. A downset $D(s)$ cuts e iff $u^- \leq s^- \leq v^- \leq s^+$ or $s^- \leq u^- \leq s^+ \leq v^-$; that is, iff $|\{u^-, v^-\} \cap [s^-, s^+]| = 1$. Alternatively, $D(s)$ cuts e iff

$|[u^-, v^-] \cap \{s^-, s^+\}| = 1$. These observations suggest that this is not a problem about graphs, but about intervals, and perhaps a problem suited for range trees.

Range trees on the Euler order: Let R be a balanced range tree with $\mathcal{V}^\pm$ at its leaves. As a balanced tree with $2n$ leaves, R has $O(n)$ nodes and height $O(\log n)$. Each range-node $a \in \mathcal{V}(R)$ induces an interval I_a on $\mathcal{V}^\pm$, consisting of all the elements of $\mathcal{V}^\pm$ at the leaves of the subtree R_a rooted at a. We call I_a a *canonical interval* of $\mathcal{V}^\pm$ (induced by a), and let $\mathcal{I} = \{I_a : a \in \mathcal{V}(R)\}$ denote the collection of all $O(n)$ canonical intervals. Each element in $\mathcal{V}^\pm$ appears in $O(\log n)$ canonical intervals because the height of R is $O(\log n)$. Moreover, every interval J on $\mathcal{V}^\pm$ decomposes into the disjoint union of $O(\log n)$ canonical intervals. The canonical intervals $\mathcal{I}$ relate to the 1-cuts and 2-cuts $\mathcal{C}_T$ induced by T as follows. For any pair of disjoint canonical intervals $I_1, I_2 \in \mathcal{I}$, let $\mathcal{C}(I_1, I_2) = \{(u,v) \in \mathcal{E} : u \in I_1, v \in I_2\}$ be the set of edges with one endpoint in I_1 and the other in I_2. We call $\mathcal{C}(I_1, I_2)$ a *canonical cut* induced by I_1 and I_2. We claim that any 1-cut or 2-cut $C \in \mathcal{C}_T$ decomposes into the disjoint union of $O(\log^2 n)$ canonical cuts.

Let $C = \mathcal{C}(D(s))$ be the 1-cut induced by the downset of a vertex s. The 1-cut $\mathcal{C}(D(s))$ consists of all edges (u,v) with one (tagged) endpoint u^- in the interval $[s^-, s^+]$ and the other endpoint v^- outside $[s^-, s^+]$. The interval $[s^-, s^+]$ decomposes to the disjoint union $[s^-, s^+] = \bigsqcup_{I_1 \in \mathcal{I}_1} I_1$ of $O(\log n)$ canonical intervals $\mathcal{I}_1 \subseteq \mathcal{I}$, and the complement $[r^-, s^-) \cup (s^+, r^+]$ also decomposes to the disjoint union $[r^-, s^-) \cup (s^+, r^+] = \bigsqcup_{I_2 \in \mathcal{I}_2} I_2$ of $O(\log n)$ canonical intervals $\mathcal{I}_2 \subseteq \mathcal{I}$. Together, the disjoint union $\mathcal{C}(D(s)) = \bigsqcup_{I_1 \in \mathcal{I}_1, I_2 \in \mathcal{I}_2} \mathcal{C}(I_1, I_2)$ of canonical cuts over the cross product $(I_1, I_2) \in \mathcal{I}_1 \times \mathcal{I}_2$ decomposes the 1-cut $\mathcal{C}(D(s))$ into $|\mathcal{I}_1| \times |\mathcal{I}_2| = O(\log^2 n)$ canonical cuts. Moreover, both decompositions $\mathcal{I}_1$ and $\mathcal{I}_2$ can be obtained in $O(\log n)$ time. Any 2-cut decomposes similarly; see the full version [8] for details.

Thus, any 1-cut or 2-cut $C \in \mathcal{C}_T$ breaks down into to the disjoint union of $O(\log^2 n)$ canonical cuts. An edge $e = (u,v)$ appears in a canonical cut $\mathcal{C}(I_1, I_2)$ iff $u \in I_1$ and $v \in I_2$. As either end point u or v appears in $O(\log n)$ canonical intervals, e appears in at most $O(\log^2 n)$ canonical cuts. In turn, there are at most $O(m \log^2 n)$ nonempty canonical cuts. The nonempty canonical cuts are easily constructed in $O(m \log^2 n)$ time by adding each edge to the $O(\log^2 n)$ canonical cuts containing it. Taking $\mathcal{D}_T$ to be this set of nonempty canonical cuts gives Lemma 7.

B. Lazy weight increments

Lemma 7 identifies a relatively small set of canonical cuts $\mathcal{D}_T$ such that any 1-cut or 2-cut $C \in \mathcal{C}_T$ is the disjoint union of $O(\log^2 n)$ canonical cuts $D_1, \ldots, D_{O(\log^2 n)} \in \mathcal{D}_T$. While we can now at least gather an implicit list of all the edges of C in $O(\log^2 n)$ time, it still appears necessary to visit every edge $e \in C$ to increment its weight. The challenge is particularly tricky because the multiplicative weight updates are not uniform. However, for a fixed and static set, recent techniques by Young [39] and Chekuri and Quanrud [7] show how to approximately apply a multiplicative weight update to the entire set in polylogarithmic amortized time. In this section, we apply these techniques to each canonical cut $D \in \mathcal{D}_T$ and show how to combine their outputs carefully to meet the claimed bounds of Lemma 5.

For every canonical cut $D \in \mathcal{D}_T$, we employ the `lazy-inc` data structure of Chekuri and Quanrud [7]. The `lazy-incs` data structure is an amortized data structure that approximates a set of counters increasing concurrently at different rates. For a fixed instance of `lazy-incs` for a particular canonical cut $D \in \mathcal{D}_T$, we will have one counter for each edge tracking the "additive part" $v_e = \ln(w_e)/\epsilon$ for each edge $e \in D$. The rate of each counter e is stored as $\texttt{rate}(e)$, and in this setting is proportional to the inverse of the capacity c_e. The primary routine is $\texttt{inc}(\rho)$, which simulates a fractional increase on each counter proportional to a single increment at the rate ρ. That is, each counter v_e increases by $\texttt{rate}(e)/\rho$.

The output of $\texttt{inc}(\rho)$ is a list of increments $\{(e, \delta_e)\}$ over some of the edges in D. The routine does not return an increment (e, δ_e) unless δ_e is substantially large. This allows us to charge off the work required to propagate the increment (e, δ_e) to the rest of the algorithm to the maximum weight of e. In exchange for reducing the number of increments, the sum of returned increments for a fixed edge e underestimates v_e by a small additive factor. An underestimate for $v_e = \ln(w_e)/\epsilon$ within an additive factor of $O(1)$ translates to an underestimate for $w_e = \exp(\epsilon v_e)$ within a $(1 + O(\epsilon))$-multiplicative factor, as desired.

Lemma 8 (7)**.** *Let $e_1, \ldots, e_k$ be k counters with rates $\texttt{rate}(e_1) \geq \texttt{rate}(e_2) \geq \cdots \geq \texttt{rate}(e_k)$ in sorted order.*

(a) An instance of `lazy-incs` *can be initialized in $O(k)$ time.*

(b) Each `inc` *runs in $O(1)$ amortized time plus $O(1)$ for each increment returned.*

(c) `flush` *runs in $O(k)$ time.*

(d) For each counter e_i, let v_i be the true value of the counter and let $\tilde{v}_i$ be the sum of increments for counter i in the return values of `inc` *and* `flush`*.*

(i) After each call to `inc`*, we have $\tilde{v}_i \leq v_i \leq \tilde{v}_i + O(1)$ for each counter e_i.*

(ii) After each call to `flush`*, we have $\tilde{v}_i = v_i$ for each counter e_i.*

We note that `flush` is not implemented in [7], but is easy enough to execute by just reading off all the residual increments in the data structure and resetting these values to 0.

The application of `lazy-incs` to our problem bears some resemblance to its application to packing intervals in

[7], where the weights are also structured by range trees. We first invoke Lemma 7 to generate the family of canonical cuts $\mathcal{D}_T$. For each canonical cut $D \in \mathcal{D}_T$, by Lemma 8, we instantiate an instance of `lazy-incs` where each edge $e \in D$ has rate $\texttt{rate}(e) = \log^2 n / c_e$. We also compute, for each canonical cut $D \in \mathcal{D}_T$, the minimum capacity $\gamma_D = \min_{e \in D} c_e$ in the cut. These preprocessing steps take $O\big(m \log^2 n\big)$ time total.

We increment weights along a cut $C \in \mathcal{C}_T$ as follows. Lemma 7 divides C into the union of $O(\log^2 n)$ disjoint canonical cuts $\mathcal{D}'_T \subseteq \mathcal{D}_T$. The minimum capacity $\gamma = \min_{e \in C} c_e$ is the minimum over the minimum capacities of each canonical cut $D \in \mathcal{D}'_T$, which is precomputed. For each canonical set $D \in \mathcal{D}'_T$, we call $\texttt{inc}(\log^2 n/\gamma)$ on the corresponding `lazy-incs` instance. For every increment (e, δ) returned in `lazy-incs`, where $e \in C$ and $\delta > 0$ represents a increment to $v_e = \ln(w_e)/\epsilon$, we increase v_e by an additive factor of $\delta/\log^2 n$, and multiply w_e by $\exp\big(\epsilon\delta/\log^2 n\big)$ accordingly.

Fix an edge $e \in \mathcal{E}$. The edge e is a member of $O\big(\log^2 n\big)$ canonical cuts in $\mathcal{D}_T$, and receives its weight increments from $O(\log^2 n)$ instances of `lazy-incs`. The slight errors from each instance of `lazy-incs` accumulate additively (w/r/t $v_e = \ln(w_e)/\epsilon$). Although each instance of `lazy-incs` promises only a constant additive error in Lemma 8, we scaled up the rate of e from $1/c_e$ to $\log^2 n/c_e$ and divide the returned weight increments by a factor of $\log^2 n$. The scaling reduces the error from each instance of `lazy-incs` from an additive factor of $O(1)$ to $O(1/\log^2 n)$. Consequently, the sum of all $O(\log^2 n)$ instances of `lazy-incs` underestimates v_e to an $O(1)$ additive error w/r/t v_e, and underestimates w_e by at most a $(1+O(\epsilon))$-multiplicative factor, as desired.

Increasing the sensitivity of each `lazy-incs` data structure by a factor of $\log^2 n$ means every increment (e, δ) returned by `inc` may increase v_e by as little as $1/\log^2 n$. An additive increase in $1/\log^2 n$ corresponds to a multiplicative increase of $\exp\big(\epsilon/\log^2 n\big)$ in w_e, hence property (e) in Lemma 5.

VI. Proof of Theorem 1

In this section we combine the ingredients discussed so far and outline the proof of Theorem 1. Let $(\mathcal{G}, c)$ be an Metric-TSP instance and $\epsilon' > 0$ be the error parameter. We assume without loss of generality that that ϵ' is sufficiently small, and in particular that $\epsilon' < 1/2$. We will also assume that $\epsilon' > 1/n^2$ for otherwise one could use an exact algorithm and achieve the desired time bound. This implies that $\log \frac{1}{\epsilon'} = O(\log n)$. Note that it suffices to argue that the algorithm outputs a $(1 + O(\epsilon))$-approximation with high probability.

High-level MWU analysis: At a high-level, our algorithm is a standard width-independent MWU algorithm for pure packing problems with an α-approximate oracle, for $\alpha = (1 + O(\epsilon))$. Our implementation follows the "time"-based algorithm of Chekuri et al. [9]. An exact oracle in this setting repeatedly solves the global minimum cut problem w/r/t the edge weights w. To implement an α-approximate oracle, every cut output needs to be an α-approximate minimum cut.

To argue that we are indeed implementing an α-approximate minimum cut oracle, let us identify the two points at which we deviate from Karger's minimum cut algorithm [25], which would otherwise be an exact oracle that succeeds with high probability. In both Karger's algorithm and our partially dynamic extension, we sample enough spanning trees from an approximately maximum tree packing to contain all $(1 + O(\epsilon))$-approximate minimum cuts as a 1-cut or 2-cut with probability $1 - \text{poly}(1/n) = 1 - \text{poly}(\epsilon/n)$ (see Theorem 3). We will argue that all the sampled tree packings succeed with high probability later after establishing a basic correctness in the event that all the sampled packings succeed.

Karger's algorithm searches all the spanning trees for the minimum 1-cut or 2-cut. We retrace the same search, except we output any approximate minimum 1-cut or 2-cut found in the search. More precisely, we maintain a target value λ with the invariant that there are no cuts of value $< \lambda$, and output any 1-cut or 2-cut with weight $\leq (1 + O(\epsilon))\lambda$ at that moment (see Lemma 6). Since λ is no more than the true minimum cut w/r/t w at any point, any cut with weight $\leq (1 + O(\epsilon))\lambda$ is a $(1 + O(\epsilon))$-approximation to the current minimum cut.

The second point of departure is that we do not work with the "true" weights implied by the framework, but a close approximation. By Lemma 5, the approximated weights underestimate the true weights by at most a $(1 + O(\epsilon))$-multiplicative factor. In turn, we may underestimate the value of a cut by at most a $(1 + O(\epsilon))$-multiplicative factor, but crucially any 1-cut or 2-cut will still have value $\leq (1 + O(\epsilon))\lambda$, for a slightly larger constant hidden in the $O(\epsilon)$.

This establishes that the proposed algorithm in fact implements an α-approximate oracle for $\alpha = 1 + O(\epsilon)$ (with high probability). The MWU framework guarantees that the final packing of cuts output by the algorithm approximately satisfies all the constraints with approximately optimal value, as desired.

Probability of failure: We now consider the probability of the algorithm failing. Randomization enters the algorithm for two reasons. Initially, we invoke Karger's randomized minimum cut algorithm [25] to compute the value of the minimum cut w/r/t the initial weights $w = 1/c$, in order to set the first value of λ. Second, at the beginning of each epoch, we randomly sample a subset of an approximately maximum tree packing hoping that the sampled trees contain every $(1 + O(\epsilon))$-approximate minimum cut as a 1-cut or 2-cut in one of the sampled trees.

Karger [25] showed that the minimum cut algorithm fails with probability at most $\mathrm{poly}(1/n)$. He also showed (see Theorem 3) that each random sample of $O(\log(n/\epsilon)) = O(\log n)$ spanning trees of an approximately maximum tree packing (conducted at the beginning of an epoch) fails to capture all $(1+\epsilon)$-approximate minimum cuts with probability at most $\mathrm{poly}(1/n) = \mathrm{poly}(\epsilon/n)$. There are a total of $O\big(\log n/\epsilon^2\big)$ epochs over the course of the algorithm. By the union bound, the probability of either the initial minimum cut or *any* random sample of spanning trees failing is $O\big(\log n/\epsilon^2\big) \cdot \mathrm{poly}(\epsilon/n) = \mathrm{poly}(\epsilon/n)$.

Running time: It remains to bound the running time of the algorithm. The analysis is not entirely straightforward, as some operations can be bounded directly, while others are charged against various upper bounds given by the MWU framework.

The algorithm initializes the edge weights in $O(m)$ time, and invokes Karger's minimum cut algorithm [25] once which runs in $O\big(m \log^3 n\big)$ time. As established in Section III, the remaining algorithm is divided into $O\big(\log(n)/\epsilon^2\big)$ epochs. Each epoch invokes Theorem 3 once to sample $O(\log(n/\epsilon)) = O(\log n)$ spanning trees from an approximate tree packing in $O\big(m \log^3 n\big)$ time. We then invoke Lemma 6 for each spanning tree in the sample. Over the course of $O\big(\log(n)/\epsilon^2\big)$ epochs, each with $O(\log(n))$ spanning trees, we invoke Lemma 6 at total of $O\big(\log^2(n)/\epsilon^2\big)$ times.

Suppose we process a tree T, outputting K approximately minimum cuts and making I full edge weight increments. By Lemma 6, the total time to process T is $O\big(m \log^2 n + K \log^2 n + I \log n\big)$. The first term, $O\big(m \log^2 n\big)$, comes from following the tree-processing subroutine of Karger [25]. Over a total $O\big(\log^2(n)/\epsilon^2\big)$ trees, the total time spent mimicking Karger's subroutine is $O\big(m \log^4(n)/\epsilon^2\big)$.

The remaining terms of Lemma 6, depending on K and I, are introduced by the MWU framework and can be amortized against standard properties of the MWU framework. The quantity K represents the number of approximate minimum cuts output while processing a single tree T, and the sum of all K across all trees is the total number of approximate minimum cuts output. Each such cut corresponds to a single iteration, and the total number of iterations in the MWU framework is $O\big(m \log(n)/\epsilon^2\big)$. Thus, the sum of the second term $O\big(K \log^2 n\big)$ over all invocations of Lemma 6 is $O\big(m \log^3(n)/\epsilon^2\big)$.

The third term of Lemma 6, $O(I \log n)$ depends on the number of increments I returned by the `inc-cut` subroutine of the `lazy-inc-cuts` data structure. Each increment (e, δ) returned by `inc-cut` increases w_e by at least a $(1 + \Omega(\epsilon/\log^2 n))$-multiplicative factor (i.e., $\delta \geq \Omega(\epsilon w_e/\log^2 n)$). The MWU framework shows that a single edge can increase by a $(1 + \Omega(\epsilon/\log n))$ factor at most $O\big(\log^3(n)/\epsilon^2\big)$ times (see Section II). It follows that the total number of edge increments, over all edges, returned by `lazy-inc-cuts` is $O\big(m \log^3(n)/\epsilon^2\big)$. Thus, the sum of the quantities $O(I \log n)$ over all invocations of Lemma 6 is $O\big(m \log^4(n)/\epsilon^2\big)$. All told, the algorithm runs in $O\big(m \log^4(n)/\epsilon^2\big) = \tilde{O}\big(m/\epsilon^2\big)$ time total.

Acknowledgments

We thank Neal Young for suggesting that Metric-TSP may be amenable to the techniques from [7].

References

[1] P. K. Agarwal and J. Pan. Near-linear algorithms for geometric hitting sets and set covers. In *Proc. 30th Annu. Sympos. Comput. Geom.* (SoCG), page 271, 2014.

[2] Z. Allen-Zhu and L. Orecchia. Nearly-linear time positive LP solver with faster convergence rate. In *Proc. 47th Annu. ACM Sympos. Theory Comput.* (STOC), pages 229–236, 2015.

[3] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Implementing the dantzig-fulkerson-johnson algorithm for large traveling salesman problems. *Mathematical programming*, 97(1):91–153, 2003.

[4] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The traveling salesman problem: a computational study*. Princeton University Press, 2011.

[5] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[6] S. C. Boyd and W. R. Pulleyblank. Optimizing over the subtour polytope of the travelling salesman problem. *Mathematical programming*, 49(1-3):163–187, 1990.

[7] C. Chekuri and K. Quanrud. Near-linear time approximation schemes for some implicit fractional packing problems. To appear in Proc. 28th ACM-SIAM Sympos. Discrete Algs. *(SODA)*, 2017.

[8] C. Chekuri and K. Quanrud. Approximating the held-karp bound for metric TSP in nearly-linear time. *CoRR*, abs/1702.04307, 2017. URL http://arxiv.org/abs/1702.04307.

[9] C. Chekuri, T. Jayram, and J. Vondrák. On multiplicative weight updates for concave and submodular function maximization. In *Proc. 6th Conf. Innov. Theoret. Comp. Sci.* (ITCS), pages 201–210, 2015.

[10] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[11] W. J. Cook. *In purusit of the traveling salesman: mathematics at the limits of computation*. Princeton University Press, 2014.

[12] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2(4):393–410, 1954.

[13] L. K. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.*, 13(4):505–520, 2000. Preliminary version in Proc. 40th Annu. IEEE Sympos. Found. Comput. Sci. *(FOCS)*, 1999.

[14] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Sys. Sci.*, 50(2):259–273, 1995. Preliminary version in Proc. 23rd Annu. ACM Sympos. Theory Comput. *(STOC)*, 1991.

[15] N. Garg and J. Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM J. Comput.*, 37(2):630–652, 2007. Preliminary version in Proc. 39th Annu. IEEE Sympos. Found. Comput. Sci. *(FOCS)*, 1998.

[16] K. Genova and D. P. Williamson. An experimental evaluation of the Best-of-Many Christofides' Algorithm for the Traveling Salesman Problem. *Algorithmica*, pages 1–22, 2017. Preliminary version appeared in Proceedings of ESA 2015.

[17] M. X. Goemans. Worst-case comparison of valid inequalities for the TSP. *Math. Prog.*, 69(1-3):335–349, 1995.

[18] M. X. Goemans and D. Bertsimas. Survivable networks, linear programming relaxations and the parsimonious property. *Math. Prog.*, 60(1):145–166, June 1993.

[19] G. Goranci, M. Henzinger, and M. Thorup. Incremental exact min-cut in poly-logarithmic amortized update time. *CoRR*, abs/1611.06500, 2016. URL http://arxiv.org/abs/1611.06500. Preliminary version in Proceedings of ESA 2016.

[20] M. D. Grigoriadis and L. G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM J. Optim.*, 4(1):86–107, 1994.

[21] G. Gutin and A. P. Punnen. *The traveling salesman problem and its variations*, volume 12. Springer Science & Business Media, 2006.

[22] M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[23] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. Assoc. Comput. Mach.*, 46(4):502–516, 1999. Preliminary version in Proc. 27th Annu. ACM Sympos. Theory Comput. *(STOC)*, 1995.

[24] D. R. Karger. Random sampling and greedy sparsification for matroid optimization problems. *Math. Program.*, 82:41–81, 1998. Preliminary version in Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci. *(FOCS)*, 1993.

[25] D. R. Karger. Minimum cuts in near-linear time. *J. Assoc. Comput. Mach.*, 47(1):46–76, 2000. Preliminary version in Proc. 28rd Annu. ACM Sympos. Theory Comput. *(STOC)*, 1996.

[26] R. Khandekar. *Lagrangian relaxation based algorithms for convex programming problems*. PhD thesis, Indian Institute of Technology Delhi, Mar. 2004.

[27] C. Koufogiannakis and N. E. Young. A nearly linear-time PTAS for explicit fractional packing and covering linear programs. *Algorithmica*, 70(4):648–674, 2014. Preliminary version in Proc. 48th Annu. IEEE Sympos. Found. Comput. Sci. *(FOCS)*, 2007.

[28] M. Lampis. Improved inapproximability for TSP. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 243–253. Springer, 2012.

[29] E. L. Lawler, J. K. Lenstra, A.-G. Rinnooy-Kan, and D. B. Shmoys. *The traveling salesman problem*. John Wiley & Sons, Ltd., 1985.

[30] C. L. Monma, B. S. Munson, and W. R. Pulleyblank. Minimum-weight two-connected spanning networks. *Math. Prog.*, 46(1):153–171, Jan. 1990.

[31] A. Mądry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proc. 42nd Annu. ACM Sympos. Theory Comput.* (STOC), pages 121–130, 2010.

[32] C. S. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *J. London Math. Soc.*, 36:445–450, 1961.

[33] S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. of Oper. Res.*, 20(2):257–301, 1995.

[34] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.

[35] M. Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007. Preliminary version in Proc. 33rd Annu. ACM Sympos. Theory Comput. *(STOC)*, 2001.

[36] W. T. Tutte. On the problem of decomposing a graph into n connected components. *J. London Math. Soc.*, 36:221–230, 1961.

[37] J. Vygen. New approximation algorithms for the TSP. *OPTIMA*, 90:1–12, 2012.

[38] D. Wang, S. Rao, and M. W. Mahoney. Unified acceleration method for packing and covering problems via diameter reduction. In *Proc. 43rd Internat. Colloq. Automata Lang. Prog.* (ICALP)*, 2016*, volume 55 of *LIPIcs*, pages 50:1–50:13, 2016.

[39] N. E. Young. Nearly linear-time approximation schemes for mixed packing/covering and facility-location linear programs. *CoRR*, abs/1407.3015, 2014. URL http://arxiv.org/abs/1407.3015.