A Cache-Efficient $(1-\epsilon)$-Approximation Algorithm for the Maximum Flow Problem on Undirected Graphs

Bachelor Thesis

Henrik Laxhuber

Supervised by

Lukas Gianinazzi   Niels Gleinig

Prof. Torsten Hoefler

30th November 2021

Abstract

This thesis presents a cache-efficient probabilistic algorithm for approximating maximum flows on undirected graphs using state-of-the-art methods. It draw on general algorithmic advances in the maximum flow problem as well as recent research on related problems in the parallel and distributed setting. These methods escape the combinatorial formulation of the maximum flow problem and work instead using gradient descent methods from convex optimisation. The thesis presents a comprehensive survey of these techniques, highlights open problems, and develops novel cache-efficient counterparts to the elementary algorithms involved in the construction.

The algorithm presented here requires $\mathcal{O}(\frac{m^{1+o(1)}}{DB}\epsilon^{-3})$ I/Os in the external memory model and succeeds with high probability. Here D is the number of disks, B is the number of records transferred per disk per I/O, and ϵ measures the quality of the approximation. Note that this becomes almost linear asymptotically. In anticipation of future improvements to the super-linear overhead, the thesis is structured as a general framework in which individual components can be easily substituted.

Contents

1 Introduction **3**
- 1.1 The External Memory Model . . . 4
- 1.2 Related Models . . . 5
- 1.3 Related Work on Maximum Flow . . . 6
- 1.4 Overview of this Thesis . . . 8

2 Preliminaries **12**
- 2.1 Undirected Maximum Flow and Edge Congestion . . . 12
- 2.2 Max-Flow Min-Cut Theorem for Congestion Minimisation . . . 14
- 2.3 Maximally Congested Cut . . . 16
- 2.4 Approximate Flow Packing . . . 17
- 2.5 Working with Trees . . . 21
- 2.6 Undirected Breadth-First Search . . . 23
- 2.7 Other Graph Algorithms . . . 26

3 Graph Sparsification **28**

4 Low Average Stretch Spanning Trees **33**
- 4.1 The Algorithm of Alon et al. . . . 35
- 4.2 A Bucket-Partitioning Oracle . . . 39
- 4.3 The Algorithm of Elkin et al. . . . 45
- 4.4 Towards a Practical Cache-Efficient Algorithm . . . 52

5 Congestion Approximators **54**
- 5.1 Embedding into Trees . . . 58
- 5.2 Using Multiple Trees . . . 62
- 5.3 Slicing Trees . . . 66
- 5.4 From Sliced Trees to j-Trees . . . 70
- 5.5 Decomposing Recursively . . . 75
- 5.6 Expander Graphs . . . 79

6 Sherman's Approximate Maximum Flow Algorithm **81**
- 6.1 The Algorithm `Route` . . . 82
- 6.2 The Algorithm `AlmostRoute` . . . 84
- 6.3 Discussion . . . 91

7 Conclusion and Future Work **92**
- 7.1 Towards a Practical Algorithm . . . 92
- 7.2 Decomposing into Non-Trees . . . 94
- 7.3 Handling Directed Graphs . . . 95

References **97**

Chapter 1

Introduction

With data sizes increasing rapidly, research has for many years focused on parallelising algorithms across multiple nodes. However, individual machines have also greatly grown in capability, and it was recently demonstrated that simple graph problems can now be solved on a single multicore node in practice, even for enormous graphs [DBS21], by leveraging cache-aware implementations of modern parallel algorithms. This bachelor thesis surveys the extent to which this approach could become applicable for more complex problems by studying the modern approximation algorithms for the maximum flow problem from the perspective of theoretically cache-efficient algorithm design.

To do so, this thesis translates current state-of-the-art algorithms from the parallel and distributed setting into efficient implementations in the *external memory model*, a computational model that studies the design of cache-efficient algorithms. Exact techniques for the maximum flow problem have been notoriously hard to parallelise efficiently with provable performance guarantees,[1] and similar issues plague the design of a cache-efficient exact algorithm. Approximate algorithms that achieve a solution within $(1-\epsilon)$ of optimal have on the other hand seen much progress in recent years after the work of Christiano et al. [Chr+11], and some parallel or distributed versions are known [Ble+13]. This thesis follows the work of Sherman [She13], who devises a gradient-descent based solver for the $(1-\epsilon)$-approximate maximum flow problem, which was also used by Ghaffari et al. [Gha+18] for an algorithm in the distributed setting.

In doing so, the thesis serves two purposes: (i) as a self-contained survey of existing methods (translated to the external memory model that will be defined in the next section), with proofs (loosely) taken either from the original work and if necessary augmented with extra steps, or reproduced entirely when simpler methods suffice. Later improvements to sub-problems etc. are incorporated where applicable, aiming to depict the current state-of-the-art as much as possible. As a result of the translation to the EM model, some new algorithms for elementary problems are developed in this thesis. Additionally, (ii) the thesis attempts to serve as a basis for future research in resolving the main remaining bottleneck, the computation of a low-stretch spanning tree. Multiple approaches to the problem are given, and it is discussed what would be necessary for omitting this problem entirely.

[1] In fact, the problem is shown to be P-complete [GHR95], and thus admits no deterministic polylogarithmic-depth exact PRAM solution unless $P = NC$. Moreover, the maximum flow problem is log-space complete [GSS82], i.e. any exact and deterministic algorithm requires more than $\mathcal{O}(\mathsf{polylog}(n))$ space, hence no 'trivially' cache-efficient algorithm can exist.

This work is one of few resources, if not the only resource, that collects all the results that combine to form Sherman's maximum flow algorithm into a single document. While most of the theorems and lemmas presented here are not novel themselves, great care has been taken to introduce a consistent and unified notation, making the thesis a self-contained reference for the topic. Proofs are augmented with additional details necessary for those new to the rather specific field. Folklore results that are often only alluded to in original works are formalised and proved in detail.

1.1 The External Memory Model

The external memory model, or short EM model, is an alternative computational model to the RAM model for analysing the runtime of computations on a single machine. Instead of concerning itself with the number of computational steps, the model recognises that large problems are often I/O-bound on modern hardware, and instead considers the number of I/O accesses to some external memory required to perform the computation. In the initial publication due to Aggarwal and Vitter [AV88], the authors argued that the internal memory (i.e. RAM) of machines was too small to hold large problems entirely, and hence solving large problems would require frequent accesses to a set of disks for fetching parts of the input and storing intermediate results. The model has remained relevant even today: In some contexts, such as embedded systems, memory can still be a limiting factor. But even large, modern machines with ample of internal memory have CPUs with only limited cache sizes. Aggressive pipelining has made computations themselves incredibly performant – but only up to the point when data must be fetched from main memory, which throws a wrench into the pipelined execution, and costs not only the hundreds of CPU cycles of fetching data from main memory, but also the tens of cycles introduced by a pipeline stall. This implies that algorithms which perform frequent random accesses to main memory will not benefit from most of the gains introduced by modern hardware and perform considerably worse than algorithms that are carefully designed to make good use of processor caches and *prefetchers* that predict and load data from memory even before it is used.

Formally, the external memory model views memory as containing M *records*, such as a machine word or a vertex of a graph,[2] that can in a single I/O request transfer B consecutive records to or from external memory, and do so concurrently for D *blocks* within the same I/O request [AV88]. For example, iterating over an n-record list in external memory would require $\frac{n}{DB} =: \mathsf{scan}(n)$ I/Os. Sorting this list requires $\Theta(\frac{n}{DB} \log_{M/DB} \frac{n}{DB}) =: \mathsf{sort}(n)$ I/Os in the average and worst-case, as proved by Aggarwal and Vitter [AV88] (the lower bound holds for comparison-based sorts).

Note that this model carries over well to the modern setting of CPU caches fetching data from SDRAM, where accessing a single word must load the entire row of B words, which is then transferred to the CPU as a *burst*; moreover multiple I/O requests can be made in parallel to D different banks. For this reason, we will refer to EM algorithms also as *cache-efficient algorithms*. Some important details are however not captured: Modern architectures allow memory requests to be pipelined, and hardware prefetchers make predictable memory access patterns much more efficient. Moreover, cache eviction policies are not part of the model. In general however, designing complex

[2]Throughout the thesis, we make the practical assumption that the record size is always large enough to store unique identifiers for vertices and edges, i.e. that the record size is at least $1 + 2\log_2 n$ bits.

EM-efficient algorithms typically relies only on a few well-structured data access primitives such as sorts and scans, which play well with all of these considerations.

It is also worth noting that many EM algorithms rely on sorting to structure their data accesses. For large values of n but fixed M, D, and B, this could eventually require *more* I/Os than a naive, unstructured algorithm due to the logarithmic overhead of sorting. For all practical values of n however, the algorithm will still outperform a naive algorithm, precisely due to the design of modern hardware, which in practice makes sorting very efficient [San00].

Designing EM-efficient algorithms is hard for a number of reasons. Most notably, algorithms from the RAM model seldom carry over to become efficient EM algorithms, and hence EM algorithms need to be designed *from the ground up*. Even simple routines such as depth-first-search have no efficient EM counterpart; a major design constraint for EM graph algorithms is to find ways of avoiding expensive graph traversals such as DFS. Fortunately, the ideas that lead to efficient *parallel* or *distributed* algorithms often also lead to efficient EM algorithms, and the combined body of research from these areas has by now become substantial. However, black-box techniques for translating results from one model to another are often not good enough: They generally require strong assumptions about the algorithms that need to be validated, possibly requiring modification of the algorithm to make it EM-efficient. For example, black-box simulation techniques for PRAM algorithms as efficient EM algorithms exist, but make limitations on the size of memory used by the PRAM algorithm, and perform best when this memory size decreases exponentially throughout the execution depth of the PRAM algorithm [MZ03]. These assumptions must be carefully validated for the more complex PRAM algorithms, hence it can be preferable to translate the underlying ideas to the EM model instead.

1.2 Related Models

Let us briefly remark on some of the related computational models. The *semi-external* model for graph algorithms assumes that $|V| \leq \mathcal{O}(M)$. Sometimes semi-external algorithms can be combined with external-memory reduction steps to yield efficient algorithms [ACZ12]; in the case of approximate maximum flow however, we note that *sparsification* (see Chapter 3) yields graphs where $m \leq n \,\mathsf{polylog}(n)$ and hence the semi-external assumption would almost admit an internal-memory algorithm, so studying the fully external model seems more insightful.

The *cache-oblivious* model [Fri+99] requires algorithms to be efficient independent of the cache parameters. This more accurately models the deep cache hierarchies encountered in real-world systems, where optimising for the parameters of only one cache level would neglect the other cache levels. By making the algorithm independent of cache parameters, it will automatically perform well on arbitrary hierarchies. The model assumes a fully associative cache with optimal replacement. Algorithms that rely only on sorts and scans are in general immediately cache-oblivious. We note that the maximum flow algorithm presented in this thesis is immediately cache-oblivious, but do not discuss this explicitly in the thesis.

The *parallel EM model* [Arg+08] gives each processor a private cache of M records, connected to a shared external PRAM memory, making it a natural extension of the EM model to parallel algorithms. We note here that this thesis does *not* make explicit use of the parallel EM model, because the main algorithm developed here is a high fan-out

recursive algorithm, and hence parallelises readily at least after the first recursion, with computations on individual processors being EM-efficient by design.

The *single-pass* and *multi-pass streaming models* [HRR98] study algorithms that make one or a small (polylogarithmic) number of passes over the adversarially-ordered input (the *stream*), possibly modifying it between passes. The goal is to find algorithms that require only few passes, while using only a small (polylogarithmic) amount of working memory. These algorithms yield good semi-external memory algorithms, but the models are otherwise not immediately comparable (since the working memory can be accessed arbitrarily often during a pass of the input). However, the techniques used to achieve good streaming algorithms can carry over to PRAM and EM algorithms [BEL20]. The *semi-streaming* model [Fei+04] increases the working memory to at most $\mathcal{O}(n \operatorname{polylog}(n))$. This is more suitable for graph traversal algorithms, but makes the approximate maximum flow problem essentially trivial after sparsification, which can be performed efficiently in this model [GKK10].

1.3 Related Work on Maximum Flow

We now switch our focus towards the maximum flow problem. Since this problem was originally formulated only for directed graphs, let $G = (V, E)$ for now be directed and consider the standard formulation of finding the maximum flow from some source vertex $s \in V$ to the sink $t \in V$, subject to the usual flow constraints. The perhaps most famous algorithm solving this problem was given by Ford and Fulkerson: Any given flow can be iteratively improved up to reaching the maximum flow by finding augmenting s-t paths in the *residual network*. Many faster algorithms have been developed from this idea: Dinitz's algorithm (sometimes also referred to as Dinic's algorithm) restructures the problem to maintain more information from previous BFS paths between iterations, and achieves a running time of $\mathcal{O}(n^2 m)$ [Din06] (where $n = |V|$ and $m = |E|$). Taken further, one can find entire augmenting DAGs of increasing BFS distance from s to t [Tar84], achieving a running time of time $\mathcal{O}(n^3)$ or $\mathcal{O}(n \cdot (\mathsf{BFS}(n, m) + n^2/B))$ I/Os on general graphs.[3] While asymptotically inefficient compared to modern algorithms, it is conceivable that this algorithm performs very well in practice on moderate input sizes due to its simplicity.

Breaking the cubic barrier with an augmenting-path algorithm seems to require the use of special data structures to accelerate path queries in the dynamically changing residual network, such as the use of dynamic trees [ST81]. An I/O efficient construction of such data structures is not necessarily impossible, however previous methods for achieving I/O-efficient tree-like datastructures don't appear to be applicable. Many related algorithms based on similar path-query datastructures are known, all of which are plagued by this same difficulty.

A more promising approach for a cache-efficient algorithm might be the push-relabel paradigm: Instead of globally increasing the flow while respecting all constraints, the push-relabel algorithms locally augment the flow without respecting flow conservation, but then converge to a valid maximum flow. Parallel and distributed versions of this method are known [GT86], and some heuristics are used in practice for increased performance, such as in the empirically cache-aware implementations due to

[3] On directed, acyclic graphs, the performance improves to $\mathcal{O}(n \operatorname{sort}(m))$ I/Os. The idea is to employ the $\mathcal{O}(n \operatorname{sort}(m))$-I/O algorithm for topological sorting due to Ajwani, Cosgaya-Lozano and Zeh [ACZ12] in conjunction with a version of Karzanov's $\mathcal{O}(n^2)$-time blocking flow algorithm due to Tarjan [Tar84].

Bader and Sachdeva [BS05] and Delong and Boykov [DB08], or the GPU-accelerated implementation due to He and Hong [HH10]. However, none of these come with good, provable worst-case asymptotic bounds. Without using dynamic trees, the best provable bound for this paradigm of algorithms in the RAM model is time $\mathcal{O}(n^2\sqrt{m})$ [CM89], which is achieved when globally ordering the local operations in a suitable way, making it difficult to achieve the same bound in a cache-efficient algorithm. In any case, the bound falls short of the $\tilde{\mathcal{O}}(nm)$ time achieved by modern algorithms such as those of Orlin and Gong [OG21].

Finally, a new set of techniques has been built around the use of *electrical flows*, which are flows induced by treating the reciprocal capacities $1/c_e$ as 'electrical' resistances and iteratively finding vertex potentials ϕ_v, inducing a flow $\frac{1}{c_e}(\phi_u - \phi_v)$ on the edge (u, v), such that a maximum flow can be built from a combination of electrical flows [Chr+11]. Electrical flows posses desirable linear-algebraic and spectral properties, allowing for asymptotically efficient algorithms based on advances in graph sparsification and Laplacian system solving [ST14]. These algorithms are generally $(1 - \epsilon)$-approximate [Chr+11; LRS13], but an exact algorithm for directed, unit-capacity graphs is also known [Mad13; KLS20]. When combined with interior-point methods, algorithms for integer-capacity graphs with runtime $\tilde{\mathcal{O}}((m + n^{3/2}) \log U)$ and better can be achieved [Bra+21; GLP21], where U is the capacity ratio $U = \max_e c_e / \min_e c_e$. This constitutes the current state of the art for *exact* maximum flows on integer-capacity graphs with small capacity ratio.

The $(1 - \epsilon)$-approximate algorithm of Sherman [She13] that this thesis follows is loosely inspired by the electrical flow approach, although it abandons it entirely and an alternative explanation (given in the next section) is perhaps more approachable. As already noted, a distributed variant of Sherman's algorithm is known [Gha+18]. This thesis incorporates some later improvements to some of the problems involved in the construction, while replacing the parts specific to the distributed model with their external-memory counterparts. It is difficult to determine exactly the current state-of-the art in $(1 - \epsilon)$-approximate algorithms, because many related approaches have been published, differing only in polylogarithmic runtime factors, whose exponents evolve rapidly in the presence of later incremental improvements to perhaps superficially unrelated problems. The design of Sherman [She13] is however the basis of most of these algorithms, and falls within the general category of current state-of-the-art almost-linear-time approximate algorithms.

Other candidates for a basis of this work could be the work of Kelner et al. [Kel+14], who independently derive a similar algorithm to that of Sherman. They however require the construction of an *oblivious routing scheme*, which is effectively an oracle for finding approximate maximum flows with low accuracy. The same object can be used in the algorithm of Sherman. In a sense, if we can implement the approach of Kelner et al. efficiently, then we can also implement the algorithm of Sherman efficiently, but not vice-versa.

Peng [Pen16] improves Sherman's algorithm from $\mathcal{O}(m^{1+o(1)}\epsilon^{-3})$ to $\tilde{\mathcal{O}}(m\epsilon^{-3})$ (where $\tilde{\mathcal{O}}$ hides substantial polylogarithmic terms). His process involves a circular reduction of a modification of Sherman's algorithm to itself. Although the algorithms differ, they suffer from the same bottleneck of computing a *low-stretch spanning tree* (see Chapter 4).

Recent work due to Kyng et al. [Kyn+19] also presents a strong generalisation of the maximum-flow problem that can be solved with a closely related algorithm. Their algorithm again reduces to the low-stretch spanning tree problem that we will find to

be hard to solve cache-efficiently. Hence studying Sherman's algorithm is also a first step towards a cache-efficient version of the more general algorithm due to Kyng et al., while being less complex.

1.4 Overview of this Thesis

By translating the work of Sherman [She13] to the EM model, we improve on the previous best-known bounds for approximate maximum flow in the EM model (c.f. Table 1.1) – note that the problem has not been previously studied in the EM model, and hence previous bounds are based on naive implementation of state-of-the-art algorithms.

To motivate the remainder of this thesis, we must first introduce some basic notation. We will write flows as vector $\boldsymbol{f} \in \mathbb{R}^m_{\geq 0}$ index by the edges E; f_e is therefore a flow amount on the edge e. Likewise, capacities are vectors $\boldsymbol{c} \in \mathbb{R}^m_{\geq 0}$, and the capacity on an edge e is thus c_e.

Let us now take a step back and divide the algorithms from the previous section into three major categories: (i) The augmenting-path algorithms that iteratively find paths (or even subgraphs) in the residual network, (ii) algorithms based on the push-relabel approach that locally improve the solution in the residual network, and (iii) modern approximation algorithms that use iterative solver techniques. All of these algorithms have in common that they iteratively improve the solution, either in terms of improving the flow, or in terms of reducing violation of the flow constraints to converge to an optimal solution. In somewhat more precise terms, virtually all algorithms can be expressed as starting in iteration i with some (almost-)flow $\boldsymbol{f}^{(i)}$, and producing a flow $\boldsymbol{f}^{(i+1)} = \boldsymbol{f}^{(i)} + \tilde{\boldsymbol{f}}^{(i)}$. This perspective allows us to escape the combinatorial formulation of the problem: If we are able to take large, global steps $\tilde{\boldsymbol{f}}^{(i)}$ *as vectors*, then we can hope to construct a fast algorithm. The challenge is to find large steps that, after taking all steps to arrive at a final flow $\boldsymbol{f}$, still guarantee $\boldsymbol{f}$ to uphold the flow constraints. At a high level, the idea then is to construct an algorithm to compute large steps $\tilde{\boldsymbol{f}}^{(i)}$ while also providing some mechanism to uphold the flow constraints for the final flow $\boldsymbol{f}$. This is the approach taken by Sherman [She13], and it is the approach that this thesis studies, so we begin by making the idea somewhat more precise and providing a rough intuition.

A convenient reformulation of the maximum flow problem for this purpose is the equivalent *congestion minimisation* problem: Instead of seeking to maximise the value of the flow, we seek to minimise the *congestion* $\mathsf{cong}^c_f(e) = \frac{f_e}{c_e}$ on the edges while achieving a unit-valued flow from s to t. Then by scaling up the flow so that

Graph	Previous best bound	This thesis
Capacitated and undirected	$\tilde{\mathcal{O}}(m\epsilon^{-3})$ [Pen16]	$\mathcal{O}(\frac{m^{1+o(1)}}{DB}\epsilon^{-3})$
Expander graphs	$\tilde{\mathcal{O}}(m\lambda_2^{-2}\epsilon^{-3})$ [She13]	$\tilde{\mathcal{O}}(\frac{m}{DB}\lambda_2^{-2}\epsilon^{-3})$

Table 1.1: I/O Bounds of the approximate maximum flow algorithms developed in this thesis. Note that previous algorithms were not designed to be cache-efficient. We use $\tilde{\mathcal{O}}$ to hide lower-order terms. For expander graphs, λ_2 is the second-largest eigenvalue of the graph Laplacian. The dependence on ϵ can be improved in some cases, see Section 6.3.

the maximally-congested edges have congestion one, we recover the solution to the maximum flow problem (this is made precise in Section 2.2). In fact, we can understand any of the iterative-improvement algorithms from this perspective: Simply divide every edge's flow by the current value of the flow after every iteration; if a maximum flow is achieved then the congestion of the unit-valued flow is minimised.

Our requirement for the step-finding algorithm in the congestion-minimisation formulation is that the final flow $\boldsymbol{f}$ should (i) route one unit of flow from s to t, and (ii) satisfy flow conservation at every other vertex. A key insight is that both these constraints can be combined elegantly by reformulating the problem as *demand routing*: Given a demand b_v for each $v \in V$, demand routing asks to find a flow that has excess exactly b_v at vertex v. For the unit-flow problem, we have $b_s = -1, b_t = 1$, and $b_v = 0$ everywhere else. In an iterative algorithm that works its way towards routing the unit flow, every step tries to satisfy the *residual demands* left by the flow excesses of the previous steps, thus 'cleaning up' after the violated flow conservation constraints and working towards a unit-valued flow.

Intuitively then, we want every step $\tilde{\boldsymbol{f}}^{(i)}$ of the algorithm to respect two goals: (i) to reduce the maximum congestion of the flow $\boldsymbol{f}^{(i)} + \tilde{\boldsymbol{f}}^{(i)}$ in order to solve the congestion minimisation problem, and (ii) to reduce congestion required to route the remaining demands after taking the step, so that after everything is said and done, the flow constraints can be satisfied: Once the remaining demands become small enough, we will route them in some trivial way to satisfy the constraints exactly, while taking care not to increase the congestion induced by the final flow too much. Let us make these desiderata for the step $\tilde{\boldsymbol{f}}^{(i)}$ somewhat more precise as a potential function that we aim to (approximately) minimise to find $\tilde{\boldsymbol{f}}^{(i)}$:

Define $\boldsymbol{C} = \mathrm{diag}(c_{e_1}, \ldots, c_{e_m})$ to be the $m \times m$ diagonal matrix of edge capacities and note that $\|\boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty$ measures the maximum edge congestion of a flow vector $\boldsymbol{f}$. Let $\boldsymbol{B}$ be the $n \times m$ incidence matrix of the directed graph G given by

$$(\boldsymbol{B})_{v,e} = \begin{cases} -1 & \text{if } e = (v, \cdot) \\ +1 & \text{if } e = (\cdot, v) \\ 0 & \text{otherwise} \end{cases}$$

For a flow $\boldsymbol{f}$, $(\boldsymbol{B}\boldsymbol{f})_v$ is the flow excess at v. A naive potential function $\Phi(\tilde{\boldsymbol{f}})$ that implements the intuition discussed above for some demands $\boldsymbol{b}$ is then

$$\Phi(\tilde{\boldsymbol{f}}) = \|\boldsymbol{C}^{-1}\tilde{\boldsymbol{f}}\|_\infty + \mathsf{opt}_G(\boldsymbol{b} - \boldsymbol{B}\tilde{\boldsymbol{f}})$$

where $\mathsf{opt}_G(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f})$ is the maximum congestion incurred by an optimal solution for routing the residual demands $\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f}$. Indeed, observe that $\Phi(\tilde{\boldsymbol{f}})$ is optimised when $\tilde{\boldsymbol{f}}$ is an optimal flow, and thus by optimising $\Phi(\tilde{\boldsymbol{f}})$ to within some factor $(1 + \delta)$, we achieve that (i) $\tilde{\boldsymbol{f}}$ has low congestion, and that (ii) the residual demands $\boldsymbol{b} - \boldsymbol{B}\tilde{\boldsymbol{f}}$ can be routed with little additional congestion.

In practice, we cannot hope to compute $\mathsf{opt}_G(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f})$ exactly, and thus we will make due with an approximation operator $\boldsymbol{R}$ for G that satisfies, for any demands $\tilde{\boldsymbol{b}}$ and some constant α,

$$\|\boldsymbol{R}\tilde{\boldsymbol{b}}\|_\infty \le \mathsf{opt}_G(\tilde{\boldsymbol{b}}) \le \alpha\|\boldsymbol{R}\tilde{\boldsymbol{b}}\|_\infty$$

and call it an *α-congestion-approximator*. In other words, the approximator $\boldsymbol{R}$ is constructed such that it underestimates the congestion by at most an α-factor, allowing us to later bound the number of iterations to (approximately) minimise $\Phi(\tilde{\boldsymbol{f}})$.

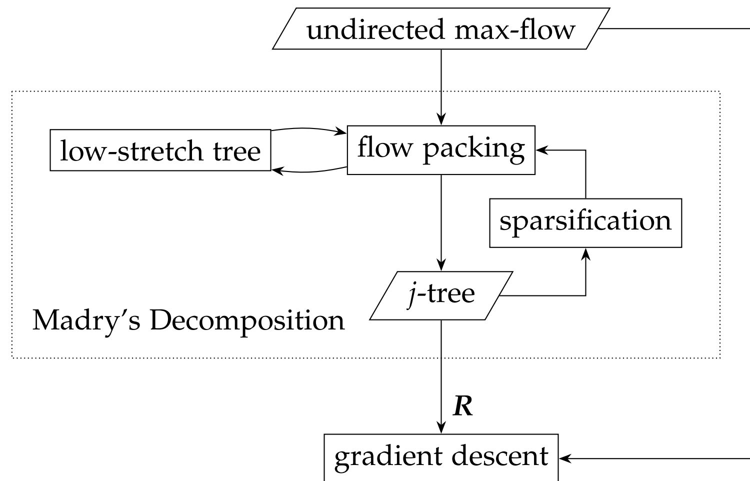

Figure 1.1: An overview of the reduction process for Sherman's algorithm.

The minimisation of $\Phi(\tilde{f})$ will be based on gradient descent. We require the operator $\boldsymbol{R}$ to be linear to facilitate this process.[4] As such, $\boldsymbol{R}$ can be thought of as matrix, though we are only ever interested in computing $\boldsymbol{Rb}$ (and, for computing gradients, matrix-vector products of the form $\boldsymbol{R}^T\boldsymbol{v}$), so we will never construct $\boldsymbol{R}$ explicitly as a matrix. We will instead use as $\boldsymbol{R}$ a convex combination of graphs that approximate the cut-flow structure of G. Computing $\boldsymbol{Rb}$ under this interpretation amounts to routing $\boldsymbol{b}$ in the approximating graphs, which is simple if these are e.g. trees. Note that even if computing $\boldsymbol{R}$ turns out to be expensive in terms of constant factors, this only needs to be done once for any given G, and the same $\boldsymbol{R}$ can be used throughout the entire optimisation procedure.

We also have to ensure that upon termination, the final residual demands can be routed easily, without increasing the congestion too much. The crucial idea of Sherman is to over-emphasise the term for the residual congestion by a factor of two to accumulate a guaranteed 'slack' between $\tilde{f}$ and a $(1+\epsilon)$-optimal solution, making room for the routing of the final residual demands. This will be better understood from the analysis in Chapter 6, Section 6.1. We are left with the following potential function:

$$\Phi(\tilde{f}) = \|\boldsymbol{C}^{-1}\tilde{f}\|_\infty + 2\alpha\|\boldsymbol{R}(\boldsymbol{b} - \boldsymbol{B}\tilde{f})\|_\infty$$

A step of the algorithm requires us to minimise $\Phi(\tilde{f})$ approximately (taking care of the supremum norm using standard approximations) for the current demands, compute the residual demands for the next iteration, and iterate until some trivial solution suffices. What remains to be done is to find a good congestion approximator $\boldsymbol{R}$ for general graphs, and to prove correctness and runtime of this construction formally.

For special cases of graphs, simple and efficient constructions of $\boldsymbol{R}$ can be made. In the general case however, we will rely on a rather involved reduction process proposed by Madry [Mad11] that computes a cut-approximating distribution of trees. At a high level (c.f. Figure 1.1), this reduction involves a *flow-packing* procedure that queries a large set of graphs $\mathbb{H}$ on V through an oracle to produce a distribution over a small subset of these graphs. The querying oracle is built around the construction of *low average-stretch spanning trees*. These are spanning trees that, on average, preserve

[4] Linearity of $\boldsymbol{R}$ is useful also for other optimisation methods such as coordinate descent [ST18].

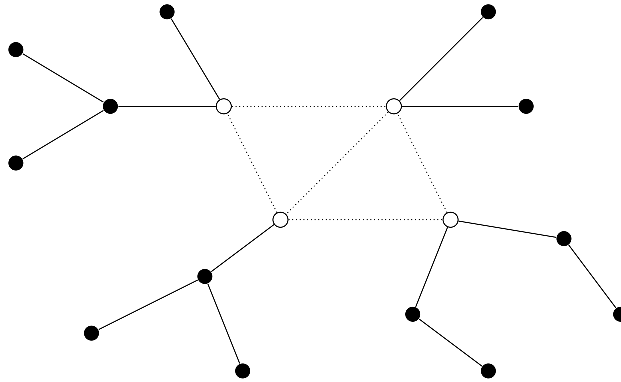

Figure 1.2: j-Tree with $j = 4$.

distances between edge endpoints from G within some multiplicative factor. The set of graphs $\mathbb{H}$ produced by the flow packing will not have a structure directly amenable to solving the demand-routing quickly, so we will transform these graphs into *j-trees* (c.f. Figure 1.2), which are disjoint trees connected to some arbitrary, smaller graph on j vertices. By recursively decomposing these j-trees, we obtain a convex combination of trees that overall approximates the cut-structure of G well. Unfortunately, this reduction process is only defined for undirected graphs.

This thesis is structured as follows: In Chapter 2, the congestion minimisation problem is stated formally, and some preliminaries are introduced: A collection of cache-efficient subroutines is presented, and the *flow packing* problem is studied. Chapter 3 is devoted to *sparsifying* graphs by removing edges while keeping the value of every cut almost unchanged; this will greatly aid the performance of later stages of the algorithm. In Chapter 4, cache-efficient constructions for low-stretch spanning trees are discussed. In Chapter 5, some congestion approximators for special cases are discussed, and the construction of a general approximator based on a decomposition by Madry [Mad11] is given. Then in Chapter 6, the maximum flow algorithm described so far is stated formally and analysed.

Chapter 2

Preliminaries

2.1 Undirected Maximum Flow and Edge Congestion

As noted in the introduction, the construction of a congestion approximator will constrain us to undirected graphs. We now make the notation for this problem precise and prove some key structural lemmas. We still interpret flows and capacities as m-dimensional vectors $\boldsymbol{f}, \boldsymbol{c}$ indexed by the edges E, but allow the flow to also take negative values:

To have well-defined notions of inflow and outflow in the undirected setting, fix an arbitrary[1] orientation $\vec{E} = \{(u,v), \ldots\}$ of the edges E, and interpret *negative* flow values $f_e < 0$ as flowing 'against' the direction of that edge. More concretely, we define inflow and outflow to entire (vertex-induced) cuts in G based on an arbitrary orientation $\vec{E}$: For any disjoint subsets $S \subseteq V$ and $T \subseteq V$ with $S \cap T = \varnothing$ and $S, T \neq \varnothing$, write the set of edges from S to T as

$$S \overset{E}{\to} T = \{(u,v) \in \vec{E} \mid u \in S, v \in T\}$$

The total signed flow across the cut $S \overset{E}{\to} T$ is then

$$f_{S \overset{E}{\to} T} = \sum_{e \in (S \overset{E}{\to} T)} f_e - \sum_{e \in (T \overset{E}{\to} S)} f_e$$

where the signs of the f_e depend on the orientation $\vec{E}$. For singleton sets $\{v\}$, we will sometimes abbreviate $\{v\} \to V \setminus \{v\}$ as just '$v \to$' and write e.g. $f_{v\to}$. With this, the 'inflow' minus 'outflow' at a vertex v, or *excess* at v, is given by $f_{v\to}$.

The undirected maximum s-t flow problem asks to find an assignment of possibly negative flows $f_e \in \mathbb{R}$ to capacitated and arbitrarily oriented edges $e \in E$ with capacities $c_e \geq 0$ on an *undirected* graph $G = (V, E, \boldsymbol{c})$, such that (i) the flow from s to t given by $f_{s\to}$ is maximised, (ii) all capacities are respected in the sense that $|f_e| \leq c_e$ for all e, and (iii) the flow is conserved at all vertices except s and t, i.e. for any $v \in V \setminus \{s,t\}$, $f_{v\to} = 0$.

The s-t flow problem can be generalised to the *demand routing* problem, where we have multiple sources and sinks by asking that the flow excess at vertex v is exactly

[1] The orientation itself is irrelevant, however it is important to consider it to be *fixed* for any given graph so that the signs of any two distinct flows $\boldsymbol{f}^{(1)}, \boldsymbol{f}^{(2)}$ are defined in terms of the same orientation. Moreover, the orientation should be preserved for subgraphs. We could define a fixed orientation by for example mapping vertices to integers and using the total order on the integers to orient the edges, i.e. $(u,v) \in \vec{E} \iff \{u,v\} \in E \wedge u \leq v$.

$f_{v\to} = b_v$ for some *demand* vector $\boldsymbol{b} \in \mathbb{R}^n$ indexed by the vertices V. Similar to the notation above, for a subset $S \subseteq V$, write

$$b_S = \sum_{v \in S} b_v$$

We say that the demands are *valid* when $b_V = 0$, i.e. when there exists some flow (possibly exceeding capacities) that *routes* $\boldsymbol{b}$. The objective in this formulation is not to find the maximum flow (the flow value is already defined by $\boldsymbol{b}$), but to find the flow $\boldsymbol{f}$ that minimises the maximum *edge congestion* $\mathsf{cong}_f(e) = \frac{|f_e|}{c_e}$. The flow $\boldsymbol{f}$ is said to be *feasible* when it respects all capacities, i.e. when it has maximum congestion at most one.

Finally, the capacity matrix $\boldsymbol{C} \in \mathbb{R}^{m\times m}$ and incidence matrix $\boldsymbol{B} \in \mathbb{R}^{m\times n}$ are as defined in the introduction; recall that $\boldsymbol{C} = \operatorname{diag}(c_{e_1}, \ldots, c_{e_m})$ is the diagonal matrix of edge capacities, such that $\boldsymbol{C}^{-1}\boldsymbol{f}$ is the vector of (signed) edge congestions under $\boldsymbol{f}$, and for the incidence matrix $\boldsymbol{B}$, $(\boldsymbol{B}\boldsymbol{f})_v$ measures the demand satisfied by a flow $\boldsymbol{f}$ on the vertex v.

With this out of the way, we are ready to formally state the congestion minimisation problem:

Definition 2.1 (Minimum Congestion Flow Problem)**.** Given an undirected, capacitated graph G with valid demands $\boldsymbol{b} \in \mathbb{R}^n$ and edge capacities $\boldsymbol{c}$, the *minimum congestion flow problem* asks to find an assignment of flow $\boldsymbol{f} \in \mathbb{R}^m$ to the edges such that the excess at every vertex $v \in V$ is b_v, i.e. $\boldsymbol{B}\boldsymbol{f} = \boldsymbol{b}$, and the maximum edge congestion $\|\boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty$ is minimised, i.e. the problem asks to solve

$$\min \quad \|\boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty \qquad \text{subject to } \boldsymbol{B}\boldsymbol{f} = \boldsymbol{b}$$

Denote by $\mathsf{opt}_G(\boldsymbol{b})$ the optimal *value* of this problem on the graph G with demands $\boldsymbol{b}$.◇

Somewhat surprisingly, an immediate reduction from the directed to the undirected flow problem on the same graph size (up to constant factors) exists [Mad11]. However, the construction falls apart when using approximate max-flow algorithms, as will be discussed in Section 7.3.

The other direction, reducing the undirected to the directed case, is much easier:

Lemma 2.1. *Let $G = (V, E, \boldsymbol{c})$ be some undirected, capacitated graph, and let $\boldsymbol{b}$ be valid demands on G. Consider an arbitrary orientation $\vec{E}$ of the edges E, and denote for $e = (u, v)$ by $\overleftarrow{e}$ the edge (v, u). Let $\vec{G} = (V, A, \vec{\boldsymbol{c}})$ be the directed graph resulting from adding arcs e and $\overleftarrow{e}$ for each $e \in \vec{E}$ with capacity c_e each. If $\vec{\boldsymbol{f}}$ is a routing of $\boldsymbol{b}$ in $\vec{G}$, then $\boldsymbol{f}$ where $f_e = \vec{f}_e - \vec{f}_{\overleftarrow{e}}$ is a routing of $\boldsymbol{b}$ in G.*

Proof. We prove that flow excess of $\boldsymbol{f}$ at every vertex $v \in V$ is exactly b_v. Compute

$$\begin{aligned}
f_{v\to} &= \sum_{e \in (\{v\} \xrightarrow{\vec{E}} V\setminus\{v\})} f_e \quad - \sum_{e \in (V\setminus\{v\} \xrightarrow{\vec{E}} \{v\})} f_e \\
&= \sum_{e \in (\{v\} \xrightarrow{\vec{E}} V\setminus\{v\})} \left(\vec{f}_e - \vec{f}_{\overleftarrow{e}}\right) \quad - \sum_{e \in (V\setminus\{v\} \xrightarrow{\vec{E}} \{v\})} \left(\vec{f}_e - \vec{f}_{\overleftarrow{e}}\right) \\
&= \sum_{e \in (\{v\} \xrightarrow{A} V\setminus\{v\})} \vec{f}_e \quad - \sum_{e \in (V\setminus\{v\} \xrightarrow{A} \{v\})} \vec{f}_e \\
&= b_v
\end{aligned}$$

where we use that if $e \in (\{v\} \xrightarrow{\vec{E}} V \setminus \{v\})$, then $\overleftarrow{e} \notin (V \setminus \{v\} \xrightarrow{\vec{E}} v)$, but $\overleftarrow{e} \in (V \setminus \{v\} \xrightarrow{A} v)$. □

2.2 Max-Flow Min-Cut Theorem for Congestion Minimisation

The congestion minimisation formulation is fully equivalent to the maximum s-t flow problem, in the sense that both problems easily reduce to a single invocation of each other. An optimal routing of demands $b_s = -1, b_t = 1$, and $b_v = 0$ otherwise, has maximum congestion exactly $1/\nu$, where ν is the value of a maximum s-t flow: If the congestion were any less, then we could scale up the routing to arrive at an s-t flow of value greater than ν. If the congestion were any more, then we could instead use the s-t flow, but scaled down by $1/\nu$.

Likewise, a routing of any valid demands $\boldsymbol{b}$ can be obtained using a single maximum s-t flow computation. This is made more precise in Lemma 2.3, but at a high level, one can attach an artificial source s and sink t, with an edge (s, v) of capacity $-b_v$ for all v with negative demands $b_v < 0$, and an edge (v, t) of capacity b_v for all v with positive demands $b_v > 0$. Then computing a maximum s-t flow and removing s, t yields a routing of the demands $\boldsymbol{b}$. A minor caveat is that the capacities need to be scaled appropriately such that the s-t flow computation does not run into capacity limits inside the original graph.

This reduction leads to a direct analogue of the well-known max-flow min-cut theorem for the congestion minimisation formulation, which will play a crucial role throughout the thesis. To ease notation in preparation for the theorem, denote the set of edges crossing the cut between $S \subseteq V$ and $T \subseteq V$, where again $S \cap T = \varnothing$ and $S, T \neq \varnothing$, as

$$S \overset{E}{\leftrightarrow} T = \{\{u, v\} \mid u \in S, v \in T\}$$

and write the capacity of the cut $S \overset{E}{\leftrightarrow} T$ as

$$c_{S \overset{E}{\leftrightarrow} T} = \sum_{e \in (S \overset{E}{\leftrightarrow} T)} c_e$$

Whenever E is implied from context, write only $S \leftrightarrow T$.

Theorem 2.1 (Max-Flow Min-Cut Theorem for Congestion Minimisation)**.** *Let $\boldsymbol{b}$ be arbitrary valid demands and $\boldsymbol{c}$ be arbitrary capacities for a connected, undirected graph G. Then for any cut $S \subseteq V$,*

$$\frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} \leq \mathsf{opt}_G(\boldsymbol{b})$$

The bound is tight in the sense that

$$\mathsf{opt}_G(\boldsymbol{b}) = \frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} \iff S \in \arg\max_{S \subseteq V} \frac{|b_S|}{c_{S \leftrightarrow V \setminus S}}$$

The theorem follows from bounding $\mathsf{opt}_G(\boldsymbol{b})$ from below and above as in the standard max-flow min-cut theorem:

Lemma 2.2. *Let $S \subseteq V$ be arbitrary. Then for any valid demands $\boldsymbol{b}$ and capacities $\boldsymbol{c}$,*

$$\frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} \leq \mathsf{opt}_G(\boldsymbol{b})$$

Proof. Assume that every edge has congestion strictly less than $\frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$. Any routing of $\boldsymbol{b}$ must flow the demands $|b_S|$ across the cut S, hence

$$|b_S| = |f_{S\to V\setminus S}| \leq \sum_{e\in(S\leftrightarrow V\setminus S)} c_e \mathsf{cong}_f(e) < \frac{|b_S|}{c_{S\leftrightarrow V\setminus S}} \sum_{e\in(S\leftrightarrow V\setminus S)} c_e = |b_S|$$

which is a contradiction. Thus there must exist an edge of congestion at least $\frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$.□

The bound holds with equality if and only if S maximises the cut congestion:

Lemma 2.3. *Let $S \subseteq V$ be an arbitrary cut that maximises the cut congestion $\frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$. Then, and only then,*

$$\mathsf{opt}_G(\boldsymbol{b}) \leq \frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$$

Proof. Let $S \subseteq V$ be an arbitrary maximally-congested cut and assume w.l.o.g. by rescaling that $\min c_e \geq 1$. We now describe a reduction of the demand routing problem to the maximum s-t flow problem using an artificial source and sink.

Let $\lambda = \frac{c_{S\leftrightarrow V\setminus S}}{|b_S|}$ and let $\vec{G}$ be the directed graph corresponding to G as in Lemma 2.1. Write $V^- = \{v \in V \mid b_v < 0\}$ and $V^+ = \{v \in V \mid b_v > 0\}$. Attach to $\vec{G}$ two new vertices s, t (c.f. Figure 2.1), with an edge (s, v) for any $v \in V^-$ of capacity $c_{(s,v)} = -\lambda b_v > 0$. Likewise, for any $v \in V^+$, add an edge (v, t) of capacity $c_{(v,t)} = \lambda b_v > 0$ to $\vec{G}$.

The proof proceeds by constructing an s-t flow of value $c_{S\leftrightarrow V\setminus S}$. First, contract all vertices $\{s\} \cup V^-$ into v^- and $\{t\} \cup V^+$ into $v+$. Since S maximises the cut congestion in G, and V^-, V^+ maximise the demand in G, the minimum cut between v^- and v^+ is at least $c_{S\leftrightarrow V\setminus S}$ (otherwise we could find a cut S' with greater cut congestion than S). In particular, there exists some flow from v^- to v^+ of value $c_{S\leftrightarrow V\setminus S}$. Now undo all contractions and observe that the cuts $\{s\}$ and $\{t\}$ have capacity exactly $\sum_{v\in V^-} -\lambda b_v = \sum_{v\in V^+} \lambda b_v = c_{S\leftrightarrow V\setminus S}$, hence together with the uncontracted flow from the previous step, there exists a flow $\tilde{f}$ from s to t of value $c_{S\leftrightarrow V\setminus S}$. All edges incident to s or t have congestion one, and hence the inflow resp. outflow to any $v \in V^-$ resp. V^+ is exactly $\lambda|b_v|$.

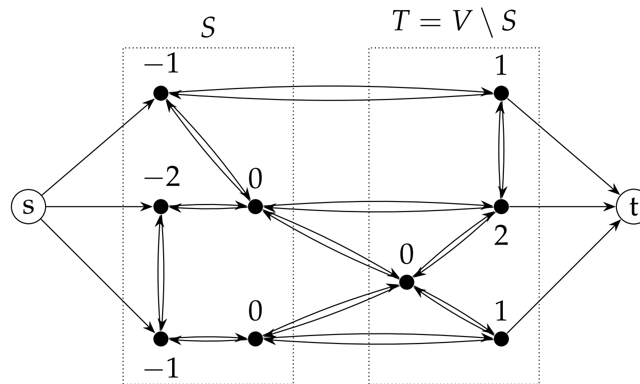

Figure 2.1: An artificial source-sink construction for routing the indicated demands, and a hypothetical corresponding minimum cut S.

Remove the vertices s and t, scale $\tilde{\boldsymbol{f}}$ by λ^{-1}, and obtain the resulting flow $\boldsymbol{f}$ in the undirected graph G as in Lemma 2.1. Any vertex $v \in V$ now has flow excess (w.r.t. $\boldsymbol{f}$) exactly b_v, i.e. $\boldsymbol{f}$ routes $\boldsymbol{b}$ in G: If v is neither in V^- nor V^+, then it has excess $0 = b_v$ by construction of the original $\tilde{\boldsymbol{f}}$. Else, removal of s or t means that v has no inflow resp. outflow, and thus after scaling $\boldsymbol{f}$ has flow excess exactly b_v. Moreover, since $\tilde{\boldsymbol{f}}$ originally had congestion at most one because it respected all edge capacities, $\boldsymbol{f}$ now has congestion at most $\lambda^{-1} = \frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$. More pointedly, $\boldsymbol{f}$ is certainly a solution to the congestion minimisation problem, and thus $\mathsf{opt}_G(\boldsymbol{b}) \leq \lambda^{-1} = \frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$, which proves the 'if' direction.

By Lemma 2.2, the bound can only hold for cuts that maximise the congestion $\frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$, and the lemma follows. □

2.3 Maximally Congested Cut

In order to efficiently generate certificates in the form of $(1+\epsilon)$-minimum cuts from our solution to the minimum congestion flow problem, we will rely on a dual problem:

Definition 2.2 (Maximally Congested Cut Problem; [She13])**.** Given an undirected, capacitated graph G with valid demands $\boldsymbol{b} \in \mathbb{R}^n$, the *maximally congested cut problem* asks to find *vertex potentials* $\boldsymbol{\psi} \in \mathbb{R}^n$ achieving

$$\max \quad \boldsymbol{b}^T\boldsymbol{\psi} \qquad \text{subject to } \|\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1 \leq 1 \qquad \diamond$$

As the name suggests, this program measures the congestion of the maximally-congested cut (which by Theorem 2.1 is $\mathsf{opt}_G(\boldsymbol{b})$). While it is possible to derive this duality in a mechanical way (see e.g. [GT04]), we will require some more structural insight to extract the actual cut from $\boldsymbol{\psi}$.

We want to prove that for any feasible potentials $\boldsymbol{\psi}$, there exists a cut S with $\boldsymbol{b}^T\boldsymbol{\psi} \leq \frac{|b_S|}{c_{S\leftrightarrow V\setminus S}} \leq \mathsf{opt}_G(\boldsymbol{b})$. This weak duality is sufficient for using the dual problem to efficiently generate solution certificates, but the problem satisfies even strong duality, i.e. its optimum value is also $\mathsf{opt}_G(\boldsymbol{b})$.

Assuming weak duality, strong duality is easy to prove:

Lemma 2.4. *For every vertex-induced cut $S \subseteq V$, there exists $\boldsymbol{\psi}$ with $\|\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1 \leq 1$ and $\boldsymbol{b}^T\boldsymbol{\psi} = \frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$. In particular, there exists $\boldsymbol{\psi}$ satisfying the constraint such that $\boldsymbol{b}^T\boldsymbol{\psi} = \mathsf{opt}_G(\boldsymbol{b})$.*

Proof. Fix any S and let $\psi_v = \frac{\operatorname{sign}(b_S)}{c_{S\leftrightarrow V\setminus S}}$ for all $v \in S$ and 0 otherwise. It is not hard to see that $\boldsymbol{b}^T\boldsymbol{\psi} = \frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$. Now write $\|\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1 = \sum_{\{u,v\}\in E} c_{u,v}|\psi_u - \psi_v|$. All terms in the sum are zero except for those corresponding to edges that cross the cut, thus the sum collapses to $\sum_{\{u,v\}\in(S\leftrightarrow V\setminus S)} c_{u,v}\frac{1}{c_{S\leftrightarrow V\setminus S}} = 1$.

Choosing S as the maximally-congested cut in the sense of Theorem 2.1 implies the lemma. □

With this insight into the structure of the problem, let us proceed to proving weak duality:

Lemma 2.5. *for every $\boldsymbol{\psi}$ with $\|\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1 \leq 1$, there exists a vertex-induced cut $S \subseteq V$ such that $\boldsymbol{b}^T\boldsymbol{\psi} \leq \frac{|b_S|}{c_{S\leftrightarrow V\setminus S}} \leq \mathsf{opt}_G(\boldsymbol{b})$.*

Proof. First, note that the problem is shift-invariant because (i) validity of $\boldsymbol{b}$ implies $\boldsymbol{b}^T(\boldsymbol{\psi} + \alpha\mathbf{1}) = \boldsymbol{b}^T\boldsymbol{\psi} + 0$, and (ii)

$$\|\boldsymbol{C}\boldsymbol{B}^T(\boldsymbol{\psi} + \alpha\mathbf{1})\|_1 = \sum_{(u,v)\in E} c_{u,v}\Big|\psi_u + \alpha - (\psi_v + \alpha)\Big| = \|\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1$$

Assume therefore by shifting that the smallest vertex potential is zero, and let $\mu = \max_v \psi_v$ be the maximum vertex potential.

Let $S_\lambda = \{v \in V \mid \psi_v \geq \lambda\}$ be the cut resulting from thresholding ψ at λ. Imagine selecting $\lambda \in_{\text{uar}} [0, \mu]$.[2] Then $\frac{1}{\mu}\|\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1 \leq \frac{1}{\mu}$ measures the expected capacity of such a random threshold cut, and $\frac{1}{\mu}\boldsymbol{b}^T\boldsymbol{\psi}$ measures the expected demand of the cut. In other words, if C is the random variable modelling the cut capacity, and B is the random variable for the cut demand, we have

$$\frac{\mathbb{E}[B]}{\mathbb{E}[C]} \geq \boldsymbol{b}^T\boldsymbol{\psi}$$

and one can prove that there thus must exist a joint outcome, i.e. a single cut S_λ, that attains congestion $\boldsymbol{b}^T\boldsymbol{\psi}$.[3] Note that since $\boldsymbol{b}$ is valid, $b_\varnothing = b_V = 0$ and hence if $\boldsymbol{b}^T\boldsymbol{\psi} > 0$ (otherwise any cut suffices), S must be a proper cut, i.e. $\varnothing \neq S_\lambda \neq V$. □

Lemma 2.5 gives rise to a simple and efficient algorithm for computing the cut S_λ from a feasible dual solution $\boldsymbol{\psi}$: Sort all vertices descending by their potential ψ_v, sort all edges descending by the larger potential of either endpoint, and scan through these lists until the cut reaches congestion at least $\boldsymbol{b}^T\boldsymbol{\psi}$ – the lemma guarantees that such a cut exists. The algorithm is given in pseudocode form in Algorithm 2.1.

Algorithm 2.1 Compute cut of congestion at least $\boldsymbol{b}^T\boldsymbol{\psi}$ from $\boldsymbol{\psi}$

```
1: procedure CongestedCut(G, b, ψ)
2:     Sort vertices by ψ_v, sort edges (u, v) by max{ψ_u, ψ_v} tiebroken arbitrarily
3:     Let μ = max_v ψ_v and initialise λ ← μ, b ← 0, c ← 0, S ← ∅
4:     repeat
5:         Continue scanning vertices v until ψ_v < λ, add these v to S
6:         For all vertices v scanned above, increment b by b_v
7:         Continue scanning edges until both endpoints have potential less than λ
8:         For all edges e scanned above, if both endpoints have potential at least λ,
           decrement c by c_e, else increment c by c_e
9:         Let λ be the potential of the next vertex in the list
10:    until |b|/c ≥ b^T ψ
11:    return S
```

2.4 Approximate Flow Packing

The set of feasible flows (i.e. flows in G with $|f_e| \leq c_e$ for all e), disregarding the demands that they satisfy, is exactly the set of flow vectors $\boldsymbol{f}$ with $\|\boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty \leq 1$

[2] This elegant technique was inspired by [Chr+11].

[3] If X, Y are discrete random variables with $\mathbb{E}[X] \leq \lambda\mathbb{E}[Y]$, then there must exist a joint outcome (x, y) with $x \leq \lambda y$: If we would have for all (x, y) that $x > \lambda y$, then $\mathbb{E}[X] = \sum_{x,y} p(x, y)x > \sum_{x,y} p(x, y)\lambda y = \mathbb{E}[Y]$, which is a contradiction.

and hence a convex polytope. If we want to find a particular feasible flow, say one that routes some demands $\boldsymbol{b}$, it might be hard to find a feasible flow directly, but we can instead hope to find a flow as a convex combination of infeasible flows $\mathbb{F}$, whose convex hull intersects the polytope of feasible flows (c.f. Figure 2.2). This is accomplished by *flow packing*, which will prove to be a central component of this thesis. The following description is based on Madry [Mad11], section 2.8, although the underlying multiplicative weights update method has general applications [AHK12]. Formally, define

Definition 2.3 (($\mathbb{F}, G$)-system; [Mad11])**.** An $(\mathbb{F}, G)$-*system* is a set of possibly infeasible flows $\mathbb{F}$ in G. The system is *feasible* if there exists a convex combination of the flows that is feasible, i.e. if there exist $\lambda_f \geq 0, \sum_{f \in \mathbb{F}} \lambda_f = 1$ such that for all $e \in E$, $\sum_{f \in \mathbb{F}} \lambda_f \mathsf{cong}_f(e) \leq 1$. $\sum_{f \in \mathbb{F}} \lambda_f f$ is called a *flow packing* of $\mathbb{F}$ in G. An *α-relaxed flow packing* satisfies only $\sum_{f \in \mathbb{F}} \lambda_f \mathsf{cong}_f(e) \leq \alpha$ for all $e \in E$. ◇

Madry does not require an explicit representation of $\mathbb{F}$, but instead queries it using a β-oracle for the system (in fact, this is a general technique in the multiplicative weights update framework):

Definition 2.4 (β-oracle for an $(\mathbb{F}, G)$-system; [Mad11])**.** A *β-oracle for an $(\mathbb{F}, G)$-system*, with $\beta \geq 1$, is an algorithm that given weights w returns a flow $\boldsymbol{f} \in \mathbb{F}$ such that $\boldsymbol{f}$ has congestion at most β on average if the system is feasible exactly, i.e.

$$\sum_{e \in E} w_e \mathsf{cong}_f(e) \leq \beta \sum_{e \in E} w_e$$

or, if the system is infeasible, either returns an $\boldsymbol{f}$ as above or indicates that the system is infeasible.

The oracle has *tightness k* if for all weights w, the oracle returns a flow $\boldsymbol{f}$ such that the number $|\{e \in E \mid \mathsf{cong}_f(e) \geq \frac{1}{2} \max_{e'} \mathsf{cong}_f(e')\}|$ of edges with congestion at least one-half the maximum congestion is at least k. ◇

Intuitively, if the weights for an edge are small, then this edge may have large congestion in the flow returned by the oracle. This can be accounted for by choosing λ_f small enough, and increasing the weight of such an edge so that in a future query to the oracle, the edge will have much lower congestion. By scaling all λ to be smaller, more flows can be packed, which leads to greater accuracy. The final approximation ratio α thus depends on the quality β of the oracle and the scaling factor δ described in the theorem below. The tightness of the oracle ensures that if one edge has large congestion, then in fact many edges have comparably large congestion also, and thus fewer flows need to be packed to arrive at a final solution.

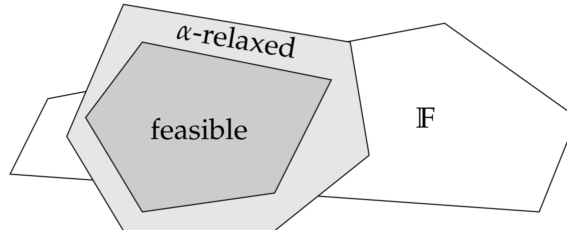

Figure 2.2: Finding feasible flows with flow packing.

Theorem 2.2 ([Mad11]). *Given an $(\mathbb{F}, G)$-system and a β-oracle $\mathcal{O}$ for it, there exists an algorithm $\texttt{RelaxedFlowPacking}^{\mathcal{O}}$ that computes an α-relaxed flow packing of the system with $\alpha = \beta(1+3\delta)$, where $0 < \delta \leq 1/2$ is an accuracy parameter. If the oracle has tightness k, then $\texttt{RelaxedFlowPacking}^{\mathcal{O}}$ makes at most $\frac{4\alpha m \log m}{k\delta^2}$ iterations, each of which performs one call to the oracle and otherwise requires $\mathcal{O}(\mathsf{scan}(m))$ I/Os.*

Algorithm 2.2 Multiplicative Weights Update for Relaxed Flow Packing [Mad11]

1: **procedure** $\texttt{RelaxedFlowPacking}^{\mathcal{O}}(G, \delta)$
2: Initialise $w_e^{(1)} \leftarrow 1$ for all $e \in E$. Define $\eta = \frac{2\log m}{\delta^2}$.
3: **for** $t = 1, \ldots$ until $\sum_{i=1}^{t} \lambda^{(i)} = 1$ **do**
4: Query the oracle $\mathcal{O}$ with weights $w^{(t)}$. If it fails, fail also.
5: Else, let $\boldsymbol{f}^{(t)}$ be the returned flow and define $\mu^{(t)} = \max_{e \in E} \mathsf{cong}_{f^{(t)}}(e)$
6: Let $\lambda^{(t)} = \min\left\{\frac{1}{\eta\mu^{(t)}}, 1 - \sum_{i=1}^{t-1} \lambda^{(i)}\right\}$ and append $(\lambda^{(t)}, \boldsymbol{f}^{(t)})$ to the output.
7: For all $e \in E$, set $w_e^{(t+1)} \leftarrow w_e^{(t)} \cdot \left(1 + \delta\eta\lambda^{(t)}\mathsf{cong}_{f^{(t)}}(e)\right)$
8: **return** the packing $\left\{\left(\lambda^{(t)}, \boldsymbol{f}^{(t)}\right)\right\}_t$

If the oracle fails, then the packing is allowed to fail also, so assume in the following that the oracle never fails. Begin by proving correctness, i.e. that the final flow packing satisfies the congestion bound $\sum \lambda_f \mathsf{cong}_f(e) \leq \beta(1+3\delta)$ upon termination.

Lemma 2.6 (Madry). *Upon termination, $\texttt{RelaxedFlowPacking}^{\mathcal{O}}$ either fails or outputs a flow packing satisfying $\sum_i \lambda^{(i)} \mathsf{cong}_{f^{(i)}}(e) \leq \beta(1+3\delta)$.*

Proof. Begin by lower-bounding $w_e^{(t)}$ during any iteration t and for any edge e. The bound relies on the fact that $1 + \delta x \geq \exp((1-\delta)\delta x)$ for $0 < \delta < 1$ and $x \in [0,1]$, which one can prove by checking equality at $x = 0$ and comparing derivatives w.r.t. x. Recall that $\lambda^{(t)} \leq \frac{1}{\eta\mu^{(t)}}$ and hence $0 \leq \eta\lambda^{(t)}\mathsf{cong}_{f^{(t)}}(e) \leq 1$ because by definition $\mathsf{cong}_{f^{(t)}}(e) \leq \mu^{(t)}$. With this, compute as follows:

$$w_e^{(t+1)} = \underbrace{w_e^{(1)}}_{=1} \prod_{i=1}^{t} \left(1 + \delta\eta\lambda^{(i)}\mathsf{cong}_{f^{(i)}}(e)\right) \geq \exp\left((1-\delta)\delta\eta \sum_{i=1}^{t} \lambda^{(i)}\mathsf{cong}_{f^{(i)}}(e)\right)$$

We can now derive an upper bound for the congestion on e in terms of $w_e^{(t+1)}$:

$$\exp\left((1-\delta)\delta\eta \sum_{i=1}^{t} \lambda^{(i)}\mathsf{cong}_{f^{(i)}}(e)\right) \leq w_e^{(t+1)} \iff \sum_{i=1}^{t} \lambda^{(i)}\mathsf{cong}_{f^{(i)}}(e) \leq \frac{\log w_e^{(t+1)}}{(1-\delta)\delta\eta}$$

To obtain an upper bound for $w_e^{(t+1)}$ that does not depend on the congestions of e, use

$w_e^{(t+1)} \leq \|\boldsymbol{w}^{(t+1)}\|_1$. To this end, compute:

$$\begin{aligned}\|\boldsymbol{w}^{(t+1)}\|_1 &= \sum_{e\in E} w_e^{(t)}(1+\delta\eta\lambda^{(t)}\mathsf{cong}_f(e)) \\ &= \|\boldsymbol{w}^{(t)}\|_1 + \delta\eta\lambda^{(t)}\sum_{e\in E} w_e^{(t)}\mathsf{cong}_f(e) \\ &\leq (1+\beta\delta\eta\lambda^{(t)})\|\boldsymbol{w}^{(t)}\|_1 \leq \exp(\beta\delta\eta\lambda^{(t)})\|\boldsymbol{w}^{(t)}\|_1 \\ \therefore \|\boldsymbol{w}^{(t+1)}\|_1 &\leq m\exp\left(\beta\delta\eta\sum_{i=1}^{t}\lambda^{(i)}\right) \qquad (\|\boldsymbol{w}^{(1)}\|_1 = m)\end{aligned}$$

Hence if the algorithm makes τ iterations in total and does not fail, then $\sum_{i=1}^{\tau}\lambda^{(i)} = 1$ by construction and thus upon termination, combining both bounds yields

$$\begin{aligned}\sum_{i=1}^{\tau}\lambda^{(i)}\mathsf{cong}_{f^{(i)}}(e) &\leq \frac{\log w_e^{(\tau+1)}}{(1-\delta)\delta\eta} \leq \frac{\log m + \beta\delta\eta\sum_{i=1}^{\tau}\lambda^{(i)}}{(1-\delta)\delta\eta} = \frac{\frac{1}{2}\eta\delta^2+\beta\delta\eta}{(1-\delta)\delta\eta} \\ &= \frac{\delta}{2(1-\delta)} + \frac{\beta}{(1-\delta)} \leq \beta(1+3\delta) \qquad (\eta = \tfrac{2\log m}{\delta^2} \text{ and } 0<\delta\leq 1/2)\end{aligned}$$

□

The number of iterations can be bounded by a potential argument relying on the correctness of the algorithm:

Lemma 2.7 ([Mad11]). *If the oracle has tightness k, then* `RelaxedFlowPacking`$^{\mathcal{O}}$ *makes at most $\frac{4\alpha m\log m}{k\delta^2}$ iterations.*

Proof. The number of iterations depends on the maximum congestion $\mu^{(t)}$ of any edge in the t-th flow, which can be bounded by combining the tightness of the oracle with the approximation factor $\alpha = \beta(1+3\delta)$ of the final solution. Concretely, the proof defines a potential function $\phi(t) = \sum_{i=1}^{t}\sum_{e\in E}\lambda^{(i)}\mathsf{cong}_{f^{(i)}}(e)$. The tightness k of the oracle dictates that at least k edges have congestion at least $\frac{1}{2}\mu^{(t)} = \frac{1}{2\eta\lambda^{(t)}}$ except in the last iteration, where $\lambda^{(\tau)}$ is bounded to fill up the convex combination. The potential thus increases up to the last iteration by at least

$$\phi(t) - \phi(t-1) = \sum_{e\in E}\lambda^{(t)}\mathsf{cong}_{f^{(t)}}(e) \geq \frac{k}{2\eta}$$

At the same time, the potential is bounded from above by Lemma 2.6:

$$\phi(t) = \sum_{e\in E}\sum_{i=1}^{t}\lambda^{(i)}\mathsf{cong}_{f^{(i)}}(e) \leq m\beta(1+3\delta) = m\alpha$$

Irrespective of the potential increase in the last iteration, this bounds the total number of iterations to at most

$$\frac{2\eta m\alpha}{k} = \frac{4\alpha m\log m}{k\delta^2}$$

□

Proof of Theorem 2.2. The number of I/Os per iteration, except execution of the oracle, is at most $\mathcal{O}(\mathsf{scan}(m))$. Combine this with the previous two lemmas, Lemma 2.6 and Lemma 2.7, to arrive at the theorem. □

2.5 Working with Trees

We now switch to presenting some elementary cache-efficient algorithms for tree processing that will be used as building blocks throughout this thesis.

Lemma 2.8 (Euler Tour; [MZ03]). *Given an undirected tree T on n vertices, an Euler tour through the arcs formed by doubling the edges as directed arcs in either direction can be computed in $\mathcal{O}(\mathsf{sort}(n))$ I/Os.*

Proof Sketch. See [MZ03] for details. The basic idea is to produce for each edge two arcs (u, v) and (v, u) during a scan of all edges, yielding the arc list $\mathcal{A}$. Now make two copies of $\mathcal{A}$, one sorted by source and one by target vertex. In a tandem scan of both lists, we can now assign for all $v \in V$ to each incoming arc (u, v) the successor outgoing arc (v, w) along the tour. A final $\mathcal{O}(\mathsf{sort}(n))$ I/Os convert this representation into a list of arcs sorted in traversal order using *list ranking* (see [MZ03]). □

Using Euler tours, it is well-known that bottom-up and top-down tree computations can be performed efficiently.

Lemma 2.9 (Processing on Rooted Trees; [MZ03]). *A suitable function f defined top-down or bottom-up on a rooted tree can be computed at all vertices v in $\mathcal{O}(\mathsf{sort}(n))$ I/Os, or $\mathcal{O}(\mathsf{scan}(n))$ if an Euler tour for the tree is already given.*

Proof Sketch. See [MZ03] for details. Intuitively, by scanning arcs in the Euler tour order while tracking whenever we are going 'up' from a leaf or 'down' towards one, we can apply f in bottom-up or top-down fashion. □

What do we mean by a *suitable* function f? Intuitively, f is a function that takes the current vertex v along with the set of values computed at its children (resp. ancestors) $\{\nu_1, \ldots\}$ to produce a value $f(v, \{\nu_1, \ldots\})$ for the vertex v, and it should do so efficiently. In particular, we cannot allow f to inspect all children in arbitrary order; it must instead perform a single pass over the children in the Euler tour order. We make this precise by defining f to be an *efficiently tree-foldable function*:

Definition 2.5 (Tree-Foldable Function). A *tree-foldable function* f is a function that can be written as $f(v, \{\nu_1, \ldots, \nu_k\}) = \tilde{f}(v, e \cdot \nu_1 \cdots \nu_k)$, where $\tilde{f} : V \times \mathcal{V} \to \mathcal{V}$ and $\cdot$ along with the set $\mathcal{V}$ forms an abelian monoid, of which e is the neutral element. Moreover, we call f *efficiently* tree-foldable when f and $\cdot$ can be evaluated without incurring any I/Os. ◇

Remark. There is some handwaving here about the set $\mathcal{V}$: Clearly, the elements of the set must have small enough encoding size (a constant number of records) such that we can keep track of the intermediate results without incurring any asymptotic I/O overhead. We can also allow the evaluations to incur some I/O overhead, as long as the I/Os amortise to at most $\mathcal{O}(\mathsf{scan}(n))$ when all evaluations take place in Euler tour order. ◇

Finally, a useful gadget when working with Euler tours is a labelling of the first incoming and last outgoing arcs to resp. from all vertices. This can be done efficiently in a single scan of the tour. We make the idea precise here to provide an example for simple operations on Euler tours.

Algorithm 2.3 Euler Tour Labelling

```
1: procedure LabelTour(T, A, (u*, v*))
2:     Initialise an empty stack S, push u* to S
3:     for all arcs (u, v) along the tour A when starting from (u*, v*) do
4:         Peek the two topmost elements of S, if they exist
5:         if u is the topmost and v the second-topmost element of S then
6:             Label (u, v) as last outgoing arc from u
7:             pop u from S
8:         else if v is not the topmost element of S then
9:             Label (u, v) as first incoming arc
10:            Push v to S
```

Lemma 2.10 (Euler Tour Labelling)**.** *Given a tree T along with an Euler tour $\mathcal{A}$ on it, we can for any starting arc (u^*, v^*) in $\mathcal{O}(\mathsf{scan}(n))$ I/Os label every arc of the tour by whether it is the first incoming or last outgoing arc of its source resp. target in the tour when starting at (u^*, v^*).*

Proof Sketch. Consider the algorithm shown in Algorithm 2.3. One can show with two simple but tedious nested inductions that (i) every vertex is pushed to the stack at most once, that (ii) vertices are pushed to the stack in the order they are visited by the tour, whenever they are first visited, and that (iii) vertices are popped from the stack exactly when they will never be visited again. We omit the details here, and instead provide only an intuition:

Since we are traversing T along an Euler tour, whenever visiting a vertex v twice, all vertices traversed between the previous and current visit to v will never be visited again. In particular, when following a path $\langle v, u, v \rangle$, u must certainly be a leaf, and the two topmost vertices on the stack are $S = u, v, \ldots$, hence u is correctly popped from the stack. This process repeats until the current vertex is no longer on top of the stack, in which case we must have discovered a previously unvisited subtree and thus push the new vertex to the stack.

The algorithm performs a single scan through the Euler tour, requiring $\mathcal{O}(\mathsf{scan}(n))$ I/Os when the tour is already given with arcs sorted in order of traversal. □

We can extend the Euler tour technique to also perform bottom-up processing on *unrooted* trees. Intuitively, this corresponds to recursively clipping off the leaves of the tree, while accumulating some value in a bottom-up fashion. To beat the naive implementation of scanning all vertices n times, we construct an Euler tour on T and show how a single scan along the Euler tour can implement the 'recursive' clipping.

Lemma 2.11 (Leaf Elimination)**.** *An efficiently tree-foldable function f can be evaluated bottom-up in a tree T in $\mathcal{O}(\mathsf{sort}(n))$ I/Os, or $\mathcal{O}(\mathsf{scan}(n))$ I/Os when an Euler tour on T is given.*

Proof Sketch. Proceed as in Algorithm 2.3, but also maintain accumulator values of the abelian monoid on the stack: Whenever visiting a vertex v for the first time, push (v, e) to S. When popping $(u, \tilde{\nu}_u)$ from the stack, output the value $\nu_u = f(u, \tilde{\nu}_u)$ for u, and then update the top of the stack $(v, \tilde{\nu}_v)$ to become $(v, \tilde{\nu}_v \cdot \nu_u)$. □

2.6 Undirected Breadth-First Search

We will later in this thesis need to devise a modified external-memory BFS where the edges incident to the source will have weights (Algorithm 4.3). In preparation of this, we devote this section to detailing a somewhat cache-efficient algorithm for BFS on undirected graphs.

Undirected graphs give more structure to traversal problems than directed graphs: Let v be some vertex with distance $d(v)$ from the root, and let u be some neighbour of v. Because $d(v) = \min_{w \in N(v)} d(w) + 1$ and $u \in N(v)$, we get $d(u) \geq d(v) - 1$, which does not hold for the directed case when e.g. there is only an arc from v to u but not vice-versa. Munagala and Ranade [MR99] exploit this fact to devise an $\mathcal{O}(n + \text{sort}(n + m))$ I/O algorithm for the BFS traversal problem.

2.6.1 Graph Representations

Before we begin, let us briefly remark on suitable methods for representing the graph G in memory. While a number of *compressed* graph representations have been developed for practical high-performance computing [Bes+20], these are not feasible to work with from a theoretical standpoint when developing complex algorithms. We instead deal only with the naive representations.

When not otherwise specified, assume all graphs are represented as *edge lists*, i.e. as lists of arbitrarily ordered pairs $\{(u, v) \mid \{u, v\} \in E\}$. When the edge list is sorted appropriately, many simple graph computations can be performed efficienlty in a single scan through the list, thereby avoiding the random access cost that would be incurred otherwise. However, for some operations, the overhead of scanning the entire graph is larger than the cost of randomly accessing specific parts of the graph. For these cases, we will use an *adjacency list* representation, which is a list of n memory locations, each pointing to the list of neighbours of the given vertex. Lastly, we will sometimes make use of an *arc list*, which is an edge list with each edge represented by two arcs, one in either direction.

An edge list can be efficiently converted into an arc list on $\mathcal{O}(\mathsf{scan}(m))$ I/Os, by scanning the edge list and duplicating each tuple with the direction flipped. An arc list is then also efficiently converted to an adjacency list in $\mathcal{O}(\mathsf{sort}(m))$ I/Os by lexicographically sorting the list and then in $\mathcal{O}(\mathsf{scan}(m))$ I/Os writing the pointer list that indexes into the sorted arc list.

2.6.2 The Algorithm of Munagala and Ranade

Munagala and Ranade [MR99] give an algorithm for breadth-first search using the adjacency lists representation that improves upon a naive random-access implementation:

Theorem 2.3 ([MR99])**.** *The BFS level of every vertex of an undirected graph can be computed in $\mathcal{O}(n + \text{sort}(m))$ I/Os.*

The algorithm is given in pseudocode in Algorithm 2.4. It stores all vertices at the current BFS level in a list, then scans the adjacency list representation of the graph to capture all neighbours, excluding those which are already in the current or previous BFS level.

Since our modified BFS will rely on similar techniques, we restate the proof of the algorithm's correctness in the following lemma:

Algorithm 2.4 Munagala-Ranade External Memory BFS; [MR99]

```
1: procedure MunagalaRanade(G, s)
2:     L(0) ← {s}, L(1) ← N(s)
3:     for t = 2, . . . , until L(t) = ∅ do
4:         Scan L(t − 1) to copy the adjacency lists of all v ∈ L(t − 1) into A(t)
5:         Sort A(t) and remove duplicates
6:         for v_{t−2}, v_{t−1}, v_t in sorted tandem scan of L(t − 2), L(t − 1), A(t) do
7:             If v_t ∉ {v_{t−2}, v_{t−1}}, add v_t to L(t)
8:     return the BFS level lists {L(t)}_t
```

Lemma 2.12 ([MR99]). *After execution of Algorithm 2.4, $v \in L(t)$ if and only if the BFS distance of v from s is exactly t.*

Proof. The claim holds for $L(0)$ and $L(1)$. To proceed by induction, assume that it holds for $L(0), \dots, L(t-1)$ for some fixed t. Then $L(t)$ is built as the list of all nodes adjacent to those of $L(t-1)$ but not in $L(t-2)$, and hence $L(t)$ must include at least all nodes of distance t from s. Moreover, because the graph is undirected, all nodes adjacent to those in $L(t-1)$ have distance at least $t-2$ from s. Hence any node in $A(t)$ of distance unequal to t must have distance $t-2$ or $t-1$, and is thus excluded from $L(t)$ by the induction hypothesis. □

Lemma 2.13 ([MR99]). *Algorithm 2.4 requires $\mathcal{O}(n + \mathrm{sort}(m))$ I/Os*

Proof. The initialisation of $L(1)$ requires $\mathrm{scan}(|N(s)|)$ I/Os. In every iteration, we scan $|L(t-1)|$ lists and accumulate $|A(t)|$ total elements, requiring at least one I/O per list and thus $\mathcal{O}(|L(t-1)| + \mathrm{scan}(|A(t)|))$ in total. To remove duplicates, we sort $A(t)$ and then perform a scan, copying the unique elements into a still sorted new list; this requires $\mathcal{O}(\mathrm{sort}(|A(t)|))$ I/Os. Finally, the tandem scan requires $\mathrm{scan}(|L(t-2)| + |L(t-1)| + |A(t)|)$ I/Os.

Every vertex is in at most one list $L(t)$, and thus $\sum_t |L(t)| \le n$. Note that $A(t)$ can include duplicates, but in building $A(t)$, only $\sum_{v \in L(t-1)} \deg v$ edges are traversed, and $\sum_t |A(t)| \le \sum_t \sum_{v \in L(t-1)} \deg v \le \sum_{v \in V} \deg v \le \mathcal{O}(m)$. We can thus compute over the entire execution of the algorithm:

$$\sum_t \mathcal{O}\left(|L(t-1)| + \mathrm{scan}(|A(t)|) + \mathrm{sort}(|A(t)|)\right) \le \mathcal{O}\left(n + \mathrm{sort}(m)\right)$$

where we make use of the fact that for any values $n_1, \dots, n_k$,

$$\sum_i n_i \log n_i \le \log\left((\max\{n_1, \dots\})^{\sum_i n_i}\right) \le \left(\sum_i n_i\right) \log\left(\sum_i n_i\right)$$

□

2.6.3 From BFS Levels to BFS Tree

While one could conceiveably modify Algorithm 2.4 to also emit a BFS tree, this is much more easily described as a separate procedure. Note that the edges of the BFS tree are exactly those edges where one endpoint has a BFS level distinct from the other (in which case they must differ by exactly one). Hence to build the BFS tree as an

edge list, we must annotate all vertices in the edge list $\mathcal{E}$ of G with their BFS level as returned by the algorithm of Munagala and Ranade. We use this opportunity to demonstrate once how such tasks are achieved in a constant number of sorts and scans, so that we may omit such details in the remainder of this thesis.

First, orient the edges in $\mathcal{E}$ arbitrarily (defining source and target) and sort lexicographically by (source, target). In $\mathcal{O}(\mathsf{scan}(m))$ I/Os (assuming the level lists are stored contiguously in memory, or in an otherwise suitable format), translate the level lists of Algorithm 2.4 to the form (v, ℓ), where ℓ is the BFS level of v. Sort the resulting lists by v. Now scan the sorted $\mathcal{E}$ and level list in tandem, annotating the edges in $\mathcal{E}$ with the BFS level of their source vertex. To annotate also the target BFS levels, sort $\mathcal{E}$ lexicographically by (target, source) and repeat the tandem scan. Finally, to build the edge list $\mathcal{T}$ of the BFS tree, scan $\mathcal{E}$ for edges whose endpoints are on different BFS levels, copying these to a new list $\mathcal{T}$.

This procedure requires 2 sorts of $\mathcal{E}$ and 1 sort of the level lists, as well as a constant number of scans over lists of length m. Hence in total, it requires $\mathcal{O}(\mathsf{sort}(m))$ I/Os.

2.6.4 Simultaneous Ball Growing

Munagala and Ranade use an adjacency list representation of G so that edges are only traversed when they are needed. Somewhat surprisingly, we can in some circumstances do better by scanning the entire edge list of G in every iteration, removing all edges that have already been traversed and thereby shrinking the edge list. If for example we could guarantee that the BFS traversal terminates after only a few iterations, then the number of scans of the edge list is small, and the total number of I/Os incurred during scanning could be less than n.

This approach is particularly useful when growing 'balls' from a number of starting points 'simultaneously', partitioning the graph into a set of balls of bounded BFS *radius*. The BFS radius of a ball is the maximum BFS distance from the starting point of that ball (the *centre*) to any vertex in the same ball. If the BFS radius of all balls is at most ρ, then the number of scans over the edge list representation of G is also at most ρ. For small ρ, this improves upon running Munagala-Ranade BFS directly.

The ball-growing algorithm (inspired by Mehlhorn and Meyer's accelerated BFS for extremely sparse graphs [MM02]) is shown in Algorithm 2.5. It takes as input the graph G and a list of vertices $\mathcal{C}$ that serve as ball centres; the balls will be grown around the vertices in $\mathcal{C}$. While traversing the graph, the algorithm maintains the current BFS level list as in Algorithm 2.4, but with each vertex additionally tagged by its ball centre, i.e. $L(t)$ takes the for $\{(c_v, v), \dots\}$ where c_v is the centre of the ball to which v belongs.

Lemma 2.14. *Upon termination,* `SimultaneousBallGrowing` *outputs an edge list $\mathcal{F}$ tagged by cluster centres $c_v \in V$ such that if $(c_v, v, \cdot) \in \mathcal{F}$, then c_v is a centre with minimal hop distance to v.*

Proof. Begin by noting that once a vertex is added to some $L(t)$, all its outgoing arcs are moved from $\mathcal{E}$ to $\mathcal{F}$. Hence in line 11, only vertices from R that have not been previously captured are added to $L(t)$.

The proof proceeds by showing that $L(t)$ contains exactly the vertices v whose closest centres have hop distance t from v using strong induction on t. For $t = 0$, the statement holds by construction of $L(0)$. Fix any t and assume that the statement holds for all $t' \leq t$. In the iteration for $t+1$, R collects all outgoing arcs from $L(t)$. By the above observation, any vertex v added to $L(t+1)$ cannot have previously been

Algorithm 2.5 Simultaneous Ball Growing; based on [MM02]

```
1: procedure SimultaneousBallGrowing(G, C)
2:     Let A be the lexicographically sorted arc list of G
3:     Initialize L(0) = C × C
4:     Let F be an initially empty edge list
5:     Move all arcs incident to a c ∈ C from A to F, tagged by c
6:     for t = 1, ... until done do
7:         Let R ← ∅ be an initially empty array of requests
8:         for all (c_u, u) in L(t − 1) and (u, v) ∈ A using tandem scan of both lists do
9:             Append (c_u, u, v) to R
10:        Sort R by target, resolve duplicate targets arbitrarily
11:        Set L(t) = {(c_u, v) | (c_u, u, v) ∈ R, (v, ·) ∈ A}
12:        Move all (v, ·) ∈ A to F, tagged by c_v, for (c_v, u) ∈ L(t)
13:    Sort F lexicographically by ball tag, source, target
14:    return F
```

captured, but is incident to some vertex in $L(t)$, hence together with the induction hypothesis, v must have hop distance $t+1$ to the cluster centre assigned to it. Moreover, there cannot exist a cluster centre closer than $t+1$ hops to v, since otherwise v would have been in some $L(t')$ for $t' \leq t$ by the induction hypothesis.

Thus if $(c_v, v, \cdot) \in \mathcal{F}$ for some c_v and v, then c_v must be one of the closest clusters to v. □

Lemma 2.15 ([MM02]). *SimultaneousBallGrowing$(G, \mathcal{C})$ requires $\mathcal{O}(\rho \operatorname{scan} m + \operatorname{sort}(m))$ I/Os, where ρ is the maximum hop radius of any cluster centre.*

Proof. The initialisation up to line 6 requires $\mathcal{O}(\operatorname{sort}(m))$ I/Os to sort $\mathcal{C}, \mathcal{E}$ and extract the initial $\mathcal{F}$ from $\mathcal{E}$. In each iteration, the shrinking $\mathcal{E}$ is scanned. An edge $(v, \cdot)$ remains in $\mathcal{E}$ only for as long as v is not captured by any cluster. Since any vertex must be captured after at most ρ iterations, there are at most $\rho \operatorname{scan}(m)$ I/Os involved in scanning $\mathcal{E}$ in total. Removing captured vertices from $\mathcal{E}$ ensures that every edge is responsible for at most one request in R throughout the entire execution, hence sorting R incurs at most $\operatorname{sort}(m)$ I/Os overall. Finally, recall from Lemma 2.13 that all other operations do not require more than $\mathcal{O}(\operatorname{sort}(m))$ I/Os in total either. Hence algorithm completes after $\mathcal{O}(\rho \operatorname{scan}(m) + \operatorname{sort}(m))$ I/Os. □

We note that Mehlhorn and Meyer [MM02] show that running `SimultaneousBallGrowing` from suitably selected starting points such that $\rho \leq \sqrt{\frac{nDB}{m}}$, followed by a Munagala-Ranade BFS that uses the improved locality granted by the ball decomposition, yields a BFS algorithm that only requires $\mathcal{O}(\sqrt{\frac{nm}{DB}} + \operatorname{sort}(m))$ I/Os. This improves upon `Munagala-Ranade` when $m \leq \mathcal{O}(nDB)$, i.e. when the graph is extremely sparse.

2.7 Other Graph Algorithms

Simple modifications of BFS to compute single-source shortest paths with integer edge weights $\{1, \ldots, W\}$ immediately yield the following:

Lemma 2.16 ([MM02]). *Single-source shortest paths on undirected graphs with integer edge weights* $\{1, \ldots, W\}$ *can be solved in* $\mathcal{O}(Wn + W\,\mathsf{sort}(m))$ *I/Os, or* $\mathcal{O}(\sqrt{\frac{Wnm}{DB}} + W\,\mathsf{sort}(m))$ *I/Os for sparse graphs.*

In the general case, SSSP can be solved using I/O-efficient *tournament trees* [KS96], which serve as external priority queues:

Lemma 2.17 ([ABT04]). *Single-source shortest paths on an undirected graph can be solved in* $\mathcal{O}(n + \mathsf{sort}(m)\log\frac{M}{DB})$ *I/Os.*

For extremely sparse graphs, Meyer and Zeh [MZ06] obtain a speedup by reduction to the minimum spanning tree problem (see below):

Lemma 2.18 ([MZ06]). *Single-source shortest paths on undirected graphs can be solved in* $\mathcal{O}(\sqrt{\frac{nm}{DB}}\log n + \mathsf{MST}(n,m))$ *I/Os, where* $\mathsf{MST}(n,m) \le \mathcal{O}(\mathsf{sort}(m) \cdot \max\{1, \log\log\frac{nDB}{m}\})$ *is the number of I/Os required to compute a minimum spanning tree. Note also that there exist randomised MST algorithms that require only* $\mathcal{O}(\mathsf{sort}(m))$ *I/Os with high probability.*

Cache-oblivious algorithms (c.f. Section 1.2) for finding single-source shortest paths in $\mathcal{O}(n + \mathsf{sort}(m)\log\frac{M}{DB})$ also exist [Bro+04].

The well-known PRAM algorithms for computing connected components via *pseudo-tree decomposition*, where each vertex requests to be merged with one of its neighbours such that all components are quickly reduced to a single vertex, can be translated efficiently to the EM model. Crucially, after reducing the number of vertices to $\mathcal{O}(m/DB)$, a BFS traversal for identifying the remaining components becomes efficient. The following result is due to Munagala and Ranade [MR99]:

Lemma 2.19 ([MR99]). *Connected components in undirected graphs can be identified in* $\mathcal{O}(\mathsf{sort}(m) \cdot \max\{1, \log\log\frac{nDB}{m}\})$ *I/Os.*

Minimum spanning forests can be computed using a related approach: By combining a similar vertex-reduction step followed by a 'naive' implementation of Prim's algorithm with an external priority queue, Arge, Brodal and Toma [ABT04] obtain:

Lemma 2.20 ([ABT04]). *A minimum spanning forest of an undirected graph can be computed in* $\mathcal{O}(\mathsf{sort}(m) \cdot \max\{1, \log\log\frac{nDB}{m}\})$ *I/Os.*

Randomised approaches can eliminate the $\log\log\frac{nDB}{m}$-factor with overwhelming probability. This was demonstrated by Abello, Buchsbaum and Westbrook, who give an I/O-efficient version of Karger, Klein and Tarjan's phased Borůvka-and-sampling algorithm for computing minimum spanning forests:

Lemma 2.21 ([ABW02]). *Minimum spanning forests can be computed in* $\mathcal{O}(\mathsf{sort}(m))$ *I/Os with probability at least* $1 - \exp(-\Omega(m))$.

The same algorithm can be used to compute connected components.

Chapter 3

Graph Sparsification

Graph sparsification is the process of approximating all cuts of some graph G using a graph $\tilde{G}$ on the same vertices, but with fewer edges. Recall from Theorem 2.1 that the cut structure of the graph essentially defines the maximum congestion of any optimal flow. Hence by removing edges while almost preserving the cut structure of G, we can approximate the congestion of any flow in G on a much sparser graph. Doing this early on in the construction of our congestion approximator (recall Section 1.4) will greatly improve the performance for dense graphs.

Many graph sparsification algorithms are known, perhaps most famously due to Benczúr and Karger [BK15], who give a simple and elegant algorithm for non-uniformly sampling the edges of G to yield the sparsified graph $\tilde{G}$. Other approaches rely on spectral graph theory and can provide even stronger guarantees. In the most extreme case of the celebrated result due to Batson, Spielman and Srivastava [BSS14], the spectrally-sparsified graph contains only $\lceil \frac{n-1}{\epsilon^2} \rceil$ many edges, where ϵ is the approximation quality, but requires $\mathcal{O}(n^3 m \epsilon^{-2})$ time in the RAM model to compute[1] (almost linear-time constructions with an additional polylogarithmic factor in the number of edges are known [SS11; KX16]). *Ultrasparsifiers* are a closely related line of research, where the sparsified graph is desired to have few *added* edges over a tree (as opposed to analysing the multiplicative factor), but allowed to have poorer approximation quality [ST14; Kol+10] – these algorithms internally rely on computing low-stretch spanning trees (see Chapter 4).

In the context of this thesis, we observe that the I/Os required for sparsification are hardly going to be a bottleneck, and hence favour a simple approach over the (sometimes considerably) more involved schemes that might perform better asymptotically. The sparsification algorithm presented here is that of Benczúr and Karger, suitably modified.

More concretely, we seek an algorithm that computes an *ϵ-graph sparsifier*:

Definition 3.1 (ϵ-graph-sparsifier; [BK15])**.** An *ϵ-graph-sparsifier* of some undirected, capacitated graph $G = (V, E, \boldsymbol{c})$ is a graph $\tilde{G} = (V, \tilde{E}, \tilde{\boldsymbol{c}})$ such that for any cut $S \subseteq V$,

$$(1-\epsilon) c_{S \overset{\tilde{E}}{\leftrightarrow} V \setminus S} \leq \tilde{c}_{S \overset{E}{\leftrightarrow} V \setminus S} \leq (1+\epsilon) c_{S \overset{\tilde{E}}{\leftrightarrow} V \setminus S} \qquad \diamond$$

The well-known construction due to Benczúr and Karger [BK15] is based on non-uniformly sampling the edges of E. At a high level, the idea is that if the minimum cut

[1] The existence of such sparsifiers is considered the major contribution of the work, rather than the algorithm to compute them. Hence it is conceivable that the running time may be improved upon.

that an edge e participates in has value k_e, then by sampling edges with probability $p_e \approx 1/k_e$, we expect one edge of the cut to remain as part of the sparsified graph, and can scale up its capacity by a factor of k_e to encompass the capacity of the entire cut. The formal treatment requires some terminology for the value k_e:

Definition 3.2 (k-connected; [BK15]). A graph is said to be k-connected if its minimum cut has value at least k. ◇

Definition 3.3 (k-strong component; [BK15]). A *k-strong component* of G is a maximal k-connected vertex-induced subgraph of G. ◇

Definition 3.4 (Edge strength; [BK15]). The *strength k_e* of an edge e is the minimum value of a cut that separates both its endpoints. We say that e is k-strong when its strength is at least k, and k-weak otherwise. ◇

Benczúr and Karger show a number of useful structural lemmas for edge strength and strong components. Of particular interest here will be the following result:

Lemma 3.1 ([BK15]). *The summed capacity of any graph's k-weak edges is at most $k(n-1)$.*

Armed with this, let us introduce the graph sparsification algorithm formally. We will call the sparsified graph $\tilde{G}$ that results from sampling with probabilities p_e the *compressed graph* of G under p_e:

Definition 3.5 (Compressed graph; [BK15]). The *compressed graph* $G[\boldsymbol{p}] = (V, \tilde{E}, \tilde{\boldsymbol{c}})$ of some graph $G = (V, E, \boldsymbol{c})$ with respect to sampling probabilities p_e for each $e \in E$ is the graph resulting from sampling each edge of G with probability $\min\{1, p_e\}$ and settings its capacity to $\tilde{c}_e = c_e/p_e$. ◇

Theorem 3.1 (Benczur-Karger Graph Sparsification; [BK15]). *Let $G = (V, E, \boldsymbol{c})$ be an undirected, capacitated graph with edge strengths $\boldsymbol{k}$. Then for any $0 < \epsilon \leq 1$ and integer d, with probability at least $1 - n^{-d}$, the compressed graph $G[\boldsymbol{p}]$ with $p_e = \rho_\epsilon \frac{c_e}{\epsilon^2 k_e}$, where $\rho_\epsilon = 3(d+3)\epsilon^{-2} \log n$, is an ϵ-graph-sparsifier of G containing at most $\mathcal{O}(n\rho_\epsilon)$ many edges.*

The largest computational challenge of the Benczúr and Karger sparsification approach is to compute the edge strengths k_e; the sampling itself can then be done in linear time. Fortunately, exact strengths are not required:

Corollary 3.2 (Sparsification with Approximate Edge Strengths; [BK15]). *If $\tilde{p}_e \geq p_e$ for all $e \in E$, then $G[\tilde{\boldsymbol{p}}]$ is an ϵ-graph-sparsifier of G. In particular, given edge strengths* under-estimates *$\tilde{k}_e \leq k_e$, the procedure of Theorem 3.1 yields an ϵ-graph-sparsifier of $\mathcal{O}(\rho_\epsilon \sum_e \frac{c_e}{\tilde{k}_e})$ many edges with probability at least $1 - n^{-d}$.*

In their original work, Benczúr and Karger [BK15] rely on a *sparse certificate* algorithm due to Nagamochi and Ibaraki [NI92], which they employ to identify all k-weak edges for $k = 1, 2, 4, \ldots$, thereby under-approximating the edge strengths by at most a factor of 2. This combinatorial algorithm translates poorly to the external memory model. Instead, we will rely on a result due to Goel, Kapralov and Khanna [GKK10], who observe that (in the uncapacitated case) k-weak edges can be eliminated by uniformly sampling edges with probability $\mathcal{O}(1/k)$. This is because any cut of value k or more will contain at least one edge in expectation, whereas smaller

cuts will contain none, and hence invoking a connected components algorithm on the sampled graph will allow us to identify the k-strong components in expectation. To implement this elegant idea as an algorithm that succeeds with high probability, they repeat the sampling step $\mathcal{O}(\log n)$ times, adding edges back after the sampling whenever their endpoints remain connected. This ensures that it is unlikely for k-strong components to become disconnected, while decreasing the probability that any k-weak edges remain in the final graph. Similar to Benczúr and Karger [BK15], they perform this procedure for $k = 1, 2, 4, \ldots$, which we shall refer to as the *levels* of k, and write as $k = 2^\lambda$ henceforth.

The description due to Goel, Kapralov and Khanna does not explicitly deal with capacitated graphs, and also appears to have some subtle flaws in the probability calculations. Both of these issues are remedied in this thesis.

Algorithm 3.1 Refinement Sampling for Edge Strengths; adapted from [GKK10]

```
 1: procedure EstimateStrengths(G, d)
 2:     Set τ ← ⌈77d log n⌉ and Λ ← ⌈log_2 ( 1/(n−1) Σ_{e∈E} c_e )⌉
 3:     Let G^(λ,0) ← G for λ = 1, ..., Λ
 4:     for t ← 1, ..., τ do
 5:         for all λ ← 1, ..., Λ do
 6:             Compute G^(λ,t) ← Refine(G^(λ,t−1), 2^−λ)
 7:     Compute l_e ← min ({λ | e ∉ E(G^(λ,τ))} ∪ {Λ}) for all e ∈ E
 8:     Let k̃_e ← 2^(l_e −1) and return k̃
 9: procedure Refine(G, p)
10:     Sample Ẽ ⊆ E with probabilities p̃_e = c_e p, let G̃ = (V, Ẽ)
11:     Compute ConnectedComponents(G̃)
12:     Let G′ be G with all cross-component edges removed
13:     return G′
```

Lemma 3.3 (Adapted from [GKK10]). *Let $d \geq 1$ be any integer and $\rho = 12(d+4)\log n$ similar to Theorem 3.1 for $\epsilon = 1/2$ but d being $d+1$. Then with probability at least $1 - \mathcal{O}(\Lambda n^{-d})$, the values $\tilde{\boldsymbol{k}}$ returned by* `EstimateStrengths`(G, d) *satisfy $\tilde{k}_e \geq \frac{k_e}{8\rho}$ for all e.*

Proof. We aim to show that for all e, $l_e \geq \log_2(k_e/4\rho)$ to imply the lemma. The proof proceeds over all $\lambda \in [\Lambda]$ to show that with high probability, all edges of strength at least $k_e \geq 4\rho 2^\lambda$ are still part of $G^{(\lambda,\tau)}$, and hence $l_e \geq \lambda = \log_2(k_e/4\rho)$ for these edges.

To that extent, fix any λ and denote by C^e for any edge e the k_e-strong component that contains e. We proceed by induction on t to prove that with probability at least $(1 - n^{-d-1})^t$, all C^e remain internally connected in $G^{(\lambda,t)}$ for all edges e with strength at least $k_e \geq 4\rho 2^\lambda$, thus implying the statement for $G^{(\lambda,t)}$.

Let $t = 1$. We will show that the sampled graph in `Refine`$(G^{(\lambda,t-1)}, 2^{-\lambda})$ restricted to the strong components C^e is a $\frac{1}{2}$-cut-approximator of all C^e w.h.p., and hence w.h.p. all C^e remain internally connected after sampling. Indeed, for any e with $k_e \geq 4\rho 2^\lambda$ and any edge e' of C^e, $\tilde{p}_{e'} = c_{e'} 2^{-\lambda} \geq 4\rho \frac{c_{e'}}{k_e} \geq 4\rho \frac{c_{e'}}{k_{e'}}$ because every edge e' of C^e has strength at least k_e by construction of C^e. Let $C = \bigcup_e C^e$. By Corollary 3.2, $C[\tilde{\boldsymbol{p}}]$ is a $\frac{1}{2}$-cut-approximator of C with probability at least $1 - |V(C)|^{-d-1} \geq 1 - n^{-d-1}$, which implies that the connected components of C must remain connected after sampling, proving the base case.

Now assume as induction hypothesis that for some fixed $t \geq 1$, all the C^e are connected in $G^{(\lambda,t)}$ with probability at least $(1 - n^{-d-1})^t$. Hence none of the edges inside the C^e have been removed so far, implying that the edge strengths have not changed. Then by the same argument as above, with probability at least $1 - n^{-d-1}$, sampling does not internally disconnect any of the C^e, thus all C^e remain internally connected with probability at least $(1 - n^{-d-1})^{t+1}$, concluding the inductive step.

Hence with probability at least $(1 - n^{-d-1})^\tau$, the edges of strength at least $k_e \geq 4\rho 2^\lambda$ are still part of $G^{(\lambda,\tau)}$. In particular, for any edge e of strength $k_e \geq 4\rho 2^\lambda$, we have $l_e \geq \lambda$ and thus $\tilde{k}_e = 2^{l_e - 1} \geq \frac{1}{2} 2^\lambda \geq \frac{k_e}{8\rho}$ with high probability.

Applying Bernoulli's inequality $(1 - n^{-d-1})^\tau \geq 1 - \tau n^{-d-1}$ and noting that $\tau \leq n$ for sufficiently large n yields a success probability of at least $1 - n^{-d}$ for every λ. Now observe that Lemma 3.1 implies that the maximum strength of any edge is at most $\frac{1}{n-1} \sum_{e \in E} c_e$, thus our choice of Λ is sufficiently large that a union bound over all $\lambda \in [\Lambda]$ captures all edges. □

Lemma 3.4 (Adapted from [GKK10]). *For any integer $d \geq 1$, with probability at least $1 - \Lambda n^{-d}$, the values $\tilde{\mathbf{k}}$ returned by* `EstimateStrengths`(G, d) *satisfy $\tilde{k}_e \leq k_e$ for every $e \in E$.*

Proof. We aim to show that for every edge e, $l_e \leq \lceil \log_2 k_e \rceil$ with high probability to imply the lemma. The proof proceeds by showing that for all $\lambda \in [\Lambda]$, all 2^λ-weak edges are removed from $G^{(\lambda,\tau)}$ with probability at least $1 - n^{-d}$; a union bound over the failure probabilities for all $\lambda \in [\Lambda]$ will then complete the proof.

Fix any $\lambda \in [\Lambda]$ and let $k = 2^{\lambda-1}$. Benczúr and Karger [BK15] show that contracting edges of strength at least k does not change the strength of the k-weak edges, so for the sake of argument, we will consider the contracted graphs $H^{(\lambda,t)}$ resulting from contracting all k-strong components of $G^{(\lambda,t)}$ before sampling in `Refine`$(G^{(\lambda,t)}, 2^{-\lambda})$.

For any t, let $C^{(t)}$ denote the largest connected component of $H^{(\lambda,t)}$ and let $n^{(t)}$ be the number of vertices of $C^{(t)}$; the proof will show that as $t \to \tau$, $n^{(t)}$ must go to 1, implying that all k-weak edges will be removed from $G^{(\lambda,\tau)}$. Since all edges in $C^{(t)}$ are k-weak, the total capacity of all edges of $C^{(t)}$ is at most $k(n^{(t)} - 1)$ by Lemma 3.1 before sampling. Now let the random variable W denote the number of edges in $C^{(t)}$ after sampling. We have

$$\mathbb{E}[W] = \sum_{e \in E(C^{(t)})} c_e 2^{-\lambda} = \frac{1}{2k} \sum_{e \in E(C^{(t)})} c_e \leq \frac{1}{2k} k(n^{(t)} - 1) = \frac{1}{2}(n^{(t)} - 1)$$

i.e. after sampling and removing the corresponding edges also from $H^{(\lambda,t)}$, $C^{(t)}$ will have at most $\frac{1}{2}(n^{(t)} - 1)$ edges in expectation. A non-uniform Chernoff bound states that [CL06]

$$P[W \geq (1+\delta)\mathbb{E}[W]] \leq \exp\left(-\frac{\delta^2 \mathbb{E}[W]}{2 + \delta/3}\right)$$

which with $\delta = 1/2$ implies in particular that

$$P[W \geq \frac{3}{4}(n^{(t)} - 1)] \leq \exp\left(-\frac{n^{(t)} - 1}{17 + 1/3}\right)$$

i.e. with probability at least $1 - \exp(-\frac{1}{c} n^{(t)})$ where $c = 17 + 1/3$, $C^{(t)}$ is split into $n^{(t)}/4$ many connected components by the sampling in `Refine`$(G^{(\lambda,t)}, 2^{-\lambda})$. This implies that $n^{(t+1)}$ is at most $\frac{3}{4} n^{(t)}$ with probability at least $1 - \exp(-\frac{1}{c} n^{(t)})$.

To remedy the decreasing success probability as $n^{(t)}$ decays, we divide the algorithm into $\log_{4/3} n$ *phases*, where the i-th phase consists of r_i many iterations of refinement, and ensures that with high probability, $n^{(t)}$ has decreased by a factor of $3/4$ over its initial value at the start of the phase (which is at most $(3/4)^{i-1}$). The total number of iterations of the algorithm will then be given by $\sum_i r_i$. More concretely, the probability of failing to decrease $n^{(t)}$ by a factor of $3/4$ during r_i many repetitions is at most

$$\exp\left(-\frac{1}{c}(3/4)^{i-1}n\right)^{r_i} \overset{!}{\leq} \frac{1}{\log_{4/3} n} n^{-d} \iff r_i \geq \frac{c}{n}(4/3)^{i-1}\left(d\log n + \log\log_{4/3} n\right)$$

Hence the total number of iterations to reach $n^{(\tau)} = 1$ is at most

$$\tau = \sum_{i=1}^{\log_{4/3} n} r_i \leq \frac{c}{n}(d\log n + \log\log_{4/3} n) \cdot \frac{(4/3)^{\log_{4/3} n} - 1}{4/3 - 1} \leq 77 d \log n$$

with a success probability of at least

$$\left(1 - \frac{1}{\log_{4/3} n} n^{-d}\right)^{\log_{4/3} n} \geq 1 - n^{-d}$$

In other words, $G^{(\lambda,\tau)}$ contains no k-weak edges w.h.p., hence every k-weak edge e satisfies $l_e \leq \lambda - 1 \implies \tilde{k}_e \leq \frac{1}{2}k$. In particular, for those edges e of strength $\frac{1}{2}k \leq k_e < k$, we have $\tilde{k}_e \leq k_e$. Taking a union bound over the failure probabilities for all $\lambda \in [\Lambda]$ completes the proof for succeeding in all edges simultaneously. □

Combining both lemmas with Corollary 3.2 yields the following theorem:

Theorem 3.2. *There exists an algorithm* $\texttt{Sparsify}(G, \epsilon, d)$ *that, given any undirected graph* G *with polynomially bounded capacities, produces with probability at least* $1 - \mathcal{O}(n^{-d}\log n)$ *for any* $0 < \epsilon \leq 1$ *an* ϵ*-graph sparsifier* $\tilde{G}$ *of* G *containing at most* $\mathcal{O}(\epsilon^{-2} n \log^2 n)$ *many edges after performing at most* $\mathcal{O}(d \log^2 n \,\mathsf{sort}(m))$ *I/Os.*

Proof. Correctness follows from both lemmas with Corollary 3.2 and the fact that we assume capacities to be bounded by n^b for some constant b. For the number of I/Os, note that after computing $\tilde{\boldsymbol{k}}$, we can sample $\tilde{G}$ in only $\mathcal{O}(\mathsf{scan}(m))$ I/Os, hence we only need to consider the I/Os required to run $\texttt{EstimateStrengths}(G, d)$: The algorithm performs $\tau\Lambda$ many iterations, each of which consists of a call to a connected-components procedure as well as a constant number of sorts and scans to identify cross-component edges. Computing connected components (c.f. Lemma 2.21) requires at most $\mathcal{O}(\mathsf{sort}(m))$ I/Os with overwhelming probability $1 - \exp(-\Omega(m)) \gg 1 - \frac{1}{\tau\Lambda}n^{-d}$ and the theorem follows. □

Remark. Goel, Kapralov and Khanna [GKK10] develop their algorithm in the semi-streaming model (c.f. Section 1.2). This allows them run Benczúr and Karger's original algorithm after a preliminary sparsification, further reducing the number of edges by a logarithmic factor to only $\mathcal{O}(\epsilon^{-1} n \log n)$. This technique is not applicable in the (fully) external memory model. ◇

Chapter 4

Low Average Stretch Spanning Trees

Consider any undirected graph $G = (V, E, \boldsymbol{\ell})$ with edge lengths given by a vector $\boldsymbol{\ell}$. We are interested in finding trees on the vertices of G such that when removing all non-tree edges from G, the distances between vertices do not grow too much. Alternatively, one can understand the problem as trying to find a tree T such that the graph metric[1] induced by T dominates the graph metric induced by G, but as tightly as possible. A formulation involving the condition numbers of the graph Laplacians also exists due to Spielman and Woo [SW09].

Formally, denote for any tree T and $u, v \in V$ by $\mathsf{path}_T(u, v)$ the unique path $\langle u, \ldots, v \rangle$ between vertices u, v in the tree T. Let $\mathsf{path}_T(e)$ be the path between the endpoints of an edge $e \in E$ (c.f. Figure 4.1). Then the *stretch* of $e \in E$ is given by the ratio between the length ℓ_e of e, and the length between the endpoints of e in T, i.e.

Definition 4.1 (Edge Stretch)**.** For any $e \in E$, the *stretch* of e is given by

$$\mathsf{stretch}_T^{\ell}(e) = \frac{1}{\ell_e} \sum_{e_T \in \mathsf{path}_T(e)} \ell_{e_T}$$

◇

The problem of finding trees with low stretch was originally introduced by Alon et al. [Alo+95] when studying the 'k-server problem', but has received much attention since. Broadly speaking, variants of the problem can be divided into two major

[1] The graph metric induced by a graph is the metric on the vertices V induced by the shortest paths in G.

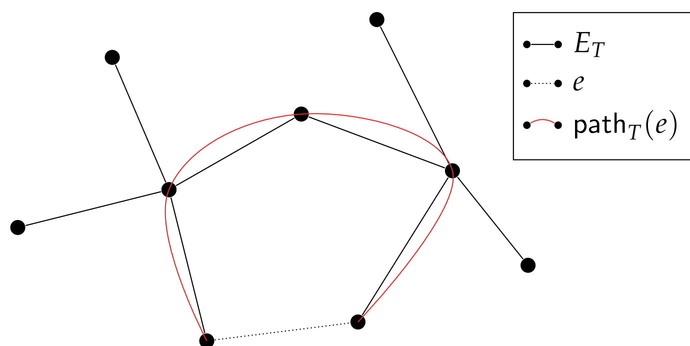

Figure 4.1: Paths in Trees

categories: (i) finding *spanning* trees on G, i.e. as subsets of the edges E, and (ii) relaxing this to allow addition of new edges.

Note that when restricted to spanning trees, there exist graphs for which the *maximum* edge stretch can be as large as $\Theta(n)$. This will be relevant for a related argument later in this thesis, so we devote some space here to making this precise:

Lemma 4.1. *For any n, there exists an n-vertex unit-weighted graph $G = (V, E)$ such that for any spanning tree T on the vertices of G, there exists an edge $e \in E$ with stretch $\mathsf{stretch}_T(e) \geq n - 1$.*

Proof. Consider the n-vertex cycle. For any spanning tree, exactly one edge is not part of the tree, and this edge clearly has stretch $n - 1$. □

The bound is tight in the following sense:

Lemma 4.2. *For any n-vertex graph $G = (V, E, \boldsymbol{\ell})$ with edge lengths $\boldsymbol{\ell}$, there exists a tree T such that no edge has stretch greater than $n - 1$.*

Proof. Consider a minimum spanning tree $T = (V, E_T)$ on G. Any edge that is part of the tree has stretch one. Any edge $e \in E \setminus E_T$ not part of the tree has stretch at most $n - 1$, because e closes a cycle with T, but because T is a minimum spanning tree, ℓ_e is at least as large as the length of any edge in this cycle. Since the cycle contains at most $n - 1$ edges apart from e, the stretch of e is at most $n - 1$. □

Alon et al. thus propose to find trees T with low *average* stretch:

Definition 4.2 (Average Stretch)**.** Given an undirected graph $G = (V, E, \boldsymbol{\ell})$ with edge lengths $\boldsymbol{\ell}$ and a tree $T = (V, E_T)$ on the vertices of G, the *average stretch* of T is defined as the average of edge stretches, i.e. as

$$\frac{1}{m} \sum_{e \in E} \mathsf{stretch}_T^{\ell}(e)$$

◇

Note that finding trees of low average stretch is a fundamentally harder, more global problem than e.g. computing minimum spanning trees. Consider for example the complete graph K_n. Any tree in this graph is a minimum spanning tree, but star graphs have much lower stretch in K_n than e.g. an n-vertex path.

Alon et al. prove the existence of graphs where any spanning tree has average stretch at least $\Omega(\log n)$ by asserting the existence of not-too-sparse n-vertex graphs with cycle length (girth) $\Omega(\log n)$, but this lower bound holds also in the general case [RR98; Gup01].

Low-average-stretch (henceforth just low-stretch) trees are an essential tool for approximating graph metrics on much simpler graphs. They find applications in a variety of problems, such as (among others, see also [FRT04; ABN08]) in network routing problems [Hu74], graph sparsification [ST14; Kol+10] and solving Laplacian linear systems [ST14], and, as relevant for this thesis, approximating the cut-flow structure of graphs using a distribution of trees [Mad11]. We note that the application of low-stretch spanning trees to solving Laplacian linear systems (linear systems of the form $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{y}$ where $\boldsymbol{L}$ is the Laplacian matrix of some graph) due to Spielman and Teng [ST14] is also used in the electrical flow based maximum-flow algorithm of Christiano et al. [Chr+11] and its later variations.

For some applications, the relaxed problem (allowing edge additions) suffices. A celebrated algorithm due to Fakcharoenphol, Rao and Talwar [FRT04] gives an optimal algorithm in this setting. For the purposes of this thesis however, the subgraph constraint will be vital. Alon et al. propose an algorithm achieving average stretch $\exp(\mathcal{O}(\sqrt{\log n \log\log n}))$,[2] which is also the basis for a parallel version due to Blelloch et al. [Ble+13]. Constructions with stretch closer to the lower bound have been found [Elk+05; ABN08], although they tend to be difficult to parallelise or implement cache-efficiently.

The remainder of this section is structured as follows: In Section 4.1, a variant of the algorithm from Alon et al. is given in a way that generalises the underlying *low-diameter decomposition*. This follows the work of Blelloch et al. [Ble+13], who use this algorithm in a parallel setting, but the actual low-diameter decomposition will be an improved version due to Miller, Peng and Xu [MPX13] presented in Section 4.2. Next, because the algorithm due to Alon et al. only achieves sublinear stretch for astronomically large input sizes on the order of $n \gg 10^{100}$, a more practical algorithm due to Becker et al. [Bec+19] is given that unfortunately relies on exact SSSP computations, which incur a substantial I/O overhead asymptotically.

4.1 The Algorithm of Alon et al.

Throughout this section, we use 'length' and 'weight' of the edges interchangeably. The section will also make use of the notion of a graph's radius:

Definition 4.3 (Notions of Graph Radius)**.** Define the *hop radius* x_0 $\mathsf{hoprad}_G(x_0)$ of some graph G with $x_0 \in V$ as the height of a BFS tree from x_0 in G. With no x_0 given, define the *hop radius* of G as the minimum hop radius from all $x_0 \in V$, i.e. $\mathsf{hoprad}(G) = \min_{x_0 \in V} \mathsf{hoprad}_G(x_0)$.

Likewise, the *weighted radius*[3] *from* x_0 $\mathsf{rad}_G(x_0)$ is the maximum shortest-path distance from x_0 to any $v \in V$, and the *weighted radius of* G with no x_0 given is $\mathsf{rad}(G) = \min_{x_0 \in V} \mathsf{rad}_G(x_0)$.

The (hop) diameter is twice the (hop) radius. ◇

The exhibition of the algorithm by Alon et al. follows that given in the work of Blelloch et al. [Ble+13], but generalises the algorithm by introducing the concept of a β-bucket-partition-oracle (see below), which helps clarify the main ideas of the approach. At a high level, the algorithm partitions the graph into low-radius clusters with few between-cluster edges. If no such partition exists, then the graph is sufficiently well-connected to build a low-average-stretch spanning tree. Else, the algorithm proceeds recursively in the clusters, and connects the clusters using the few between-cluster edges. In this way, only few edges have large stretch (namely those between clusters).

Definition 4.4 (β-bucket-partition-oracle)**.** An *β-bucket-partition-oracle*, for $1 \le \beta \le n$, is an algorithm that takes as input a radius parameter r and an unweighted, undirected graph $G = (V, E)$ with the edges partitioned into k buckets $E = E_1 \uplus \ldots \uplus E_k$, and outputs a partition of G into components $V = C_1 \uplus \ldots \uplus C_p$ such that

[2] Expressions of the form $\exp(\mathcal{O}(\sqrt{\log n \log\log n}))$ sit between poly-logarithmic and polynomial, i.e. for any $\epsilon > 0$ and integer d with sufficiently large n, $\log^d(n) \le \exp(\mathcal{O}(\sqrt{\log n \log\log n})) \le n^\epsilon$.

[3] This is sometimes also referred to as the *eccentricity* of x_0.

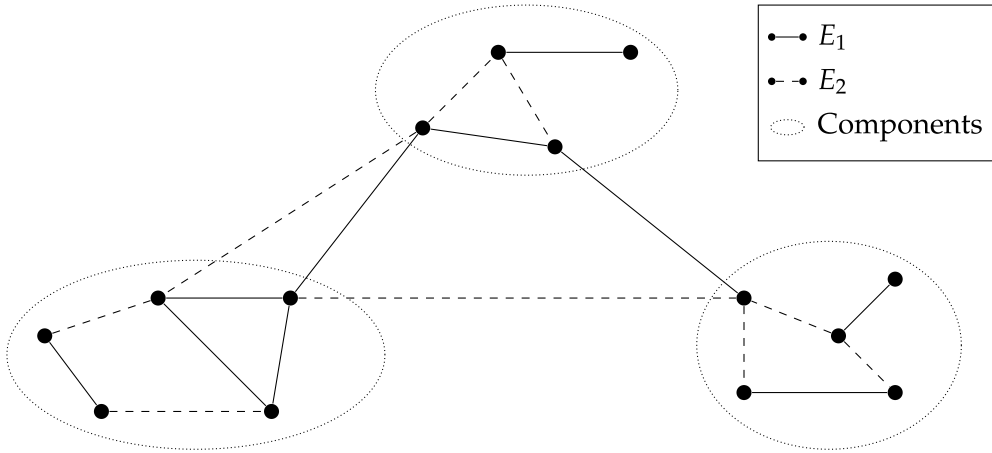

Figure 4.2: A bucket partition of a graph with two edge buckets.

1. The hop radius of each component C_i is at most r.
2. For every bucket E_i, the number of edges with endpoints in different components is at most $\frac{\beta}{r}k|E_i|$. ◇

Throughout this chapter, we assume $\beta \geq \log n$ when calculating optimal parameters for our algorithms; this matches the lower bound proved by Bartal [Bar98].

Blelloch et al. [Ble+13] provide an efficient PRAM construction of such an oracle with $\beta \leq \mathcal{O}(\log^3 n)$, which can be efficiently used in the EM model by employing the ball growing procedure from Subsection 2.6.4. An alternative construction by Miller, Peng and Xu [MPX13] achieves $\beta \leq \mathcal{O}(\log n)$, which we will use instead (see Section 4.2). We begin by showing how to construct low-stretch spanning trees from such low-radius decompositions:

Lemma 4.3 (Generalised from [Alo+95])**.** *Given a β-bucket-partition-oracle, there exists an algorithm* `AKPW-LSST` *that computes a spanning tree of the weighted input graph G with average stretch at most* $\exp(\mathcal{O}(\sqrt{\log n \log \beta}))$.

Algorithm 4.1 Alon et al. Low-Stretch Spanning Tree; [Alo+95; Ble+13]

1: **procedure** `AKPW-LSST`$(G, \boldsymbol{\ell})$
2: Let $\tau = \sqrt{\log n}/\sqrt{\log \beta}$ and $\rho = 4\beta\tau n^{3/\tau}$
3: Scale lengths such that $\min_e \ell_e = 1$. Let T be the initially empty tree.
4: Split $E = E_1^{(0)} \uplus \ldots$ such that $E_i^{(0)} = \{e \in E \mid \ell_e \in [\rho^{i-1}, \rho^i]\}$
5: **for** $t = 0, \ldots$ **do**
6: Let $(C_1, \ldots, C_p) = \texttt{Partition}(V^{(t)}, \rho/4, E_1^{(t)}, E_2^{t-1}, \ldots, E_t^{(0)})$
7: Add a BFS tree of each component to T
8: Contract all inner-component edges to yield the graph $(V^{(t+1)}, \biguplus_i E_i^{(t+1)})$
9: Remove self-loops (but maintain parallel edges)
10: If the graph has become empty, stop
11: **return** T

The algorithm is restated in Algorithm 4.1. To gain some intuitive insight, consider first the unit-weighted case. In each iteration, the inner-cluster edges are stretched by a

factor of at most $\frac{\rho}{2}$ because the diameter of the clusters is at most $\frac{\rho}{2}$. The remaining cross-component edges are passed on to the next iteration, where they are either included in a cluster, incurring an additional stretch factor of $\frac{\rho}{2}$ on top of the stretch ρ incurred by routing through the trees in the clusters of either endpoint, or again passed on. In general, in the t-th iteration, the inner-cluster edges have stretch at most $\rho^t/2$ in the original graph, and the number of edges in the t-th iteration for the unit-weighted case is $(\beta/\rho)^t m$. If $(\beta/\rho)^t$ decays sufficiently fast in comparison to the growth of ρ^t, then we can hope to achieve a spanning tree of low average stretch.

The weighted case merely extends this idea: Once a bucket of edges is added, its size decreases geometrically. Heavier buckets are added later because the stretch factor $\rho^t/2$ is decreased by the edge length ρ^i.

The formal proof of Lemma 4.3 follows in spirit that given by Alon et al. [Alo+95], but with substantial modifications to accommodate the more general concept of the bucket partition oracle. Note in particular that the meanings of β, ρ, γ, etc. in this section differ from those used by Alon et al. [Alo+95]. The proof begins by showing the 'sufficient decrease' of the edge buckets.

Lemma 4.4. *For any τ such that $\rho \geq 4\beta\tau n^{3/\tau}$ and iteration t, the i-th edge bucket for any $i \leq t$ satisfies $|E_i^{(t-i)}| \leq \left(\frac{4\beta\tau}{\rho}\right)^{(t-i)} |E_i^{(0)}|$.*

Proof. Write $\gamma = \frac{4\beta}{\rho}$. The number of between-component edges in the t-th iteration depends on the number of active partitions (at most t), which gives a naive bound of $|E_i^{(t-i)}| \leq \gamma^{t-i}(t-1)!|E_i^{(0)|}$. We will show that the number of active partitions is in fact always at most τ, and hence $|E_i^{(t-i)}| \leq (\gamma\tau)^{t-i}|E_i^{(0)|}$, proving the lemma.

For the first τ iterations, the statement holds immediately. If $(\gamma\tau)^\tau |E_i^{(0)}| < 1$, say because $(\gamma\tau)^\tau \leq n^{-3}$, then it also holds for all further iterations by induction. In principle, we are interested in finding the smallest τ that satisfies the inequality. It will however prove advantageous to leave τ variable and find instead find the smallest ρ such that $(\gamma\tau)^\tau \leq n^{-3}$. We have

$$(\gamma\tau)^\tau \leq n^{-3} \iff \tau \log(\gamma\tau) \leq -3\log n \iff \gamma\tau \leq n^{-3/\tau} \iff \rho \geq 4\beta\tau n^{3/\tau} \qquad \square$$

With the decrease of the edge buckets bounded, the next part is to bound the increase of the stretch incurred edges that remain between iterations.

Lemma 4.5 ([Alo+95])**.** *In any iteration t, if $\rho \geq 8$ then the weighted radius of any component C_i counted with edge lengths $\boldsymbol{\ell}$ in the original graph is at most ρ^{t+1}.*

Proof. The proof proceeds by induction on t. For $t = 1$, the statement holds immediately: The hop radius of every component in the first iteration is at most $\rho/4$, and every edge has length at most ρ, and $\rho^2/4 < \rho^2$. Assume then that the statement holds for all $t \leq k-1$ for some $k \geq 2$. Consider any path along the BFS tree computed for component C_i in iteration k. By construction, the path can have at most $\rho/4$ hops. Every vertex along the path may be a supervertex from a contraction in a previous iteration. By the induction hypothesis, expanding such vertices increases the radius of C_i by an addition of the diameter of at most $2\rho^k$. Moreover, any edge on the path has length at most ρ^k. Since there are at most $\rho/4$ edges and $\rho/4+1$ vertices along the path, C_i has weighted radius at most $\rho^{k+1}/4 + (\rho/4+1)2\rho^k \leq \rho^{k+1}$ for $2\rho^k \leq \rho^{k+1}/4 \iff \rho \geq 8$. $\square$

The ingredients are ready to be mixed:

Proof of Lemma 4.3. Assume $\rho \geq 4\beta\tau n^{3/\tau}$ and define again $\gamma = \frac{4\beta}{\rho}$. Begin by bounding the stretch added to T in any iteration t: By Lemma 4.5, the length of the tree path between the endpoints of any inner-component edges in iteration t is at most the diameter $2\rho^{t+1}$. By Lemma 4.4, the number of edges of length $\rho^{i-1} \leq \ell_e \leq \rho^i$ for $1 \leq i \leq t$ is at most a $(\gamma\tau)^{(t-i)}$-fraction of $|E_i^{(0)}|$. Hence the stretch added to the *total* stretch of T in iteration t is at most

$$\sum_{i=1}^{t} (\gamma\tau)^{t-i} \frac{2\rho^{t+1}}{\rho^{i-1}} |E_i^{(0)}| = \sum_{i=1}^{t} 2\rho^2 (4\beta\tau)^{t-i} |E_i^{(0)}|$$

Summing over all iterations, swapping summations and recalling that E_i vanishes after τ iterations gives

$$\sum_{e \in E} \mathsf{stretch}_T^{\ell}(e) \leq \sum_i \sum_{t=0}^{\tau-1} 2\rho^2 (4\beta\tau)^t |E_i^{(0)}| = 2m\rho^2 \frac{(4\beta\tau)^{\tau} - 1}{4\beta\tau - 1} \leq 4m\rho^2 (4\beta\tau)^{\tau-1}$$

Now use $\rho = 4\beta\tau n^{3/\tau}$ from Lemma 4.4 to arrive at a total stretch of at most $4n^{6/\tau}(4\beta\tau)^{\tau+1}$. Towards optimising this, write

$$\log\left(4n^{6/\tau}(4\beta\tau)^{\tau+1}\right) = \log 4 + \frac{6}{\tau}\log n + \tau \log(4\beta) + \log(4\beta\tau) + \tau \log \tau$$

and observe that a choice of $\tau := \frac{\sqrt{\log n}}{\sqrt{\log \beta}}$ with $\beta \geq \log n$ yields

$$\log\left(4n^{6/\tau}(4\beta\tau)^{\tau+1}\right) \leq \mathcal{O}(\sqrt{\log n \log \beta})$$

thereby bounding the average stretch of T to $\exp\left(\mathcal{O}(\sqrt{\log n \log \beta})\right)$. □

Remark. Computing the optimal τ that minimises the total stretch leads to[4]

$$\tau_{\mathsf{opt}} = \frac{1}{4e\beta} \exp\left(W(4e\beta(6\log n - 1))\right)$$

where W is the positive branch of Lambert's W function, defined as an inverse of $x \mapsto xe^x$. τ_{opt} has no representation using elementary functions; the next-best choice of τ is the one used above, balancing the dominating terms in the sum up to constant factors. ◇

The next section develops a bucket partitioning algorithm `Partition`$(G, \rho, E_1, \ldots, E_k)$ with $\beta \leq \mathcal{O}(\log n)$. As part of its implementation, `Partition` will also provide the BFS trees inside each component, thus these will not have to be computed. `Partition` succeeds with high probability after expending $\mathcal{O}(\log n \cdot (\rho\,\mathsf{scan}(m) + \mathsf{sort}(m)))$ I/Os. Combining with Lemma 4.3 yields the following theorem:

Theorem 4.1. *For graphs of polynomially-bounded edge length ratio* $U = \max_e \ell_e / \min_e \ell_e$, *`AKPW-LSST`$(G, \boldsymbol{\ell})$ can be implemented to return a spanning tree of average stretch at most* $\exp(\mathcal{O}(\sqrt{\log n \log\log n}))$ *after expending at most* $\mathcal{O}(\exp(\mathcal{O}(\sqrt{\log n \log\log n}))\,\mathsf{scan}(m))$ *I/Os.*

[4] Use that the average stretch is convex w.r.t. τ on the domain of interest, and after taking derivatives, observe that $x \log x = k \iff x = \exp(W(k))$ since $\exp(W(k))\log(\exp(W(k))) = k$ by definition of W.

Proof. The number of edge buckets in total is at most $\log_\rho U \leq \mathcal{O}(\log n)$. Since all buckets are emptied after being involved in at most τ iterations, the total number of iterations is at most $\mathcal{O}(\log n)$. Each iteration consists of a call to `Partition`, which also returns the BFS trees for each component, followed by edge contractions and the removal of self-loops. To perform the contraction, we can propagate each component's label along an Euler tour of every tree, building a list of vertex-to-component mappings $\{(u,x),\dots\}$. In a constant number of sorts and scans, we can then rename the endpoints of each edge to their component label, and filter out self-loops. The entire contraction process thus takes at most $\mathcal{O}(\mathsf{sort}(m))$ I/Os, which is dominated by the $\mathcal{O}(\rho\,\mathsf{scan}(m)\log^{2.5} n)$ from `Partition`. Noting that $\rho \leq \tilde{\mathcal{O}}(n^{3/\tau}) = \tilde{\mathcal{O}}(\exp(\mathcal{O}(\log n\sqrt{\log\log n}/\sqrt{\log n}))) \leq \exp(\mathcal{O}(\sqrt{\log n \log\log n}))$ completes the proof. □

Remark. The bound $\exp(c\cdot\sqrt{\log n\log\log n})$ scales very poorly with the hidden constant c. To beat the minimum spanning tree, i.e. $\exp(c\cdot\sqrt{\log n\log\log n}) \leq n-1$, for $c=1$ requires only $n \geq e$ as one can confirm with a calculator. But for $c \geq 7$, which roughly matches the hidden constant in the proof of Lemma 4.3 when assuming $\beta = \log n$ exactly, we already require roughly $n \geq 10^{120}$. Hence the algorithm is not practical for all realistic values of n. ◇

4.2 A Bucket-Partitioning Oracle

Analogous to the approach taken by Blelloch et al. [Ble+13], we first design a probabilistic bucket partitioning oracle for the single-bucket case, i.e. an algorithm that partitions $G=(V,E)$ into components such that for any $e \in E$, the probability that e goes between components is at most $\frac{\beta}{r}$ where $\beta \leq \mathcal{O}(\log n)$. We will then use this algorithm to construct a probabilistic multi-bucket oracle, and run it as often as necessary to produce a final 2β-bucket-partitioning-oracle. The algorithm presented here due to Miller, Peng and Xu [MPX13], who improve upon the ideas of Blelloch et al. [Ble+13] by greatly simplifying the algorithm and achieving a β of $\mathcal{O}(\log n)$ instead of $\mathcal{O}(\log^3 n)$.

4.2.1 The Algorithm `Split`

This section is devoted to the treatment of the single-bucket case. Formally, we will construct an algorithm `Split` that satisfies the bucket partitioning desiderata (c.f. Definition 4.4) with high probability:

Theorem 4.2 ([MPX13]). *Given an unweighted and undirected graph $G=(V,E)$ and a radius parameter r, `Split`(G,r) computes a partition of G into components $V = V_1 \uplus \dots \uplus V_k$ such that*

- *For any constant $d \geq 1$, with probability at least $1-n^{-d}$, every V_i has hop radius at most r.*
- *For every $e \in E$, the probability that the two endpoints of E lie in different components, i.e. that e is* cut, *is at most $\frac{(d+1)\log n}{r}$.*

`Split` requires at most $\mathcal{O}(r\,\mathsf{scan}(m)+\mathsf{sort}(m))$ I/Os.

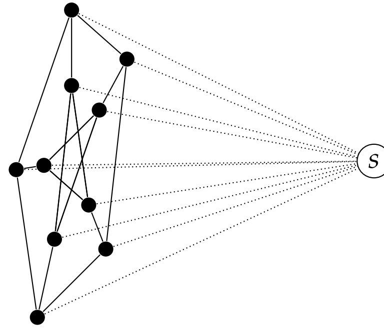

Figure 4.3: Graph Splitting with Artificial Source

At a high level, the algorithm works by attaching an artificial source s with edges of random length δ_v to every $v \in V$ (c.f. Figure 4.3). It then computes a shortest path tree from T, letting the subtrees of s define the components $V_1, \ldots, V_k$. The key idea is that the randomisation of δ_v makes it unlikely for an edge $e = \{u, v\}$ to be cut between components: Assume e is cut. This implies that u, v are in different subtrees from s, and thus the paths from s to u and v differ already at the first vertex $x_u \neq x_v$ after s. But because T is a shortest-path tree, the distances $\delta_{x_u} + d(x_u, u)$ and $\delta_{x_v} + d(x_v, v)$ must then be within an additive 1 of each other, as otherwise e would be part of the shortest-path tree. Now if we can select the δ_v in such a way that this happens with probability at most $\mathcal{O}(\frac{\log n}{r})$, while also ensuring that the components have radius at most r, then we have the desired decomposition.

Miller, Peng and Xu's key ingredient towards this is the exponential distribution:

Definition 4.5 (Exponential Distribution)**.** A random variable X is said to be *exponentially distributed with parameter* $\lambda > 0$, denoted $X \sim \text{Exp}(\lambda)$, if $P[X \leq x] = 1 - \exp(-\lambda x) =: F_{\text{Exp}}(x, \lambda)$ for $x \geq 0$, and 0 otherwise. ◇

An elementary consequence is that the distribution is *memoryless*, i.e. that $P[X \geq x \mid X \geq t] = P[X \geq x - t]$ for any $x, t \geq 0$.

With this notation, consider the pseudocode of Algorithm 4.2.

Algorithm 4.2 Graph Splitting; [MPX13]

1: **procedure** `Split`(G, r)
2: Let $d \geq 1$ be some parameter for the radius probability bound
3: Sample $\delta_v \overset{\text{i.i.d.}}{\sim} \text{Exp}\left(\frac{(d+1)\log n}{r}\right)$ for all $v \in V$
4: Add a vertex s to G with an edge to all $v \in V$ of length $\max_u \delta_u - \delta_v$
5: Compute a shortest-path tree T from s using edge length 1 for all $e \in E$
6: Let V_i be the vertices of the i-th subtree of T from s
7: **return** $V_1, \ldots, V_k$ along with their subtrees from T

The algorithm is closely related to the simultaneous ball growing procedure from Subsection 2.6.4, except that the cluster centres are selected randomly based on their randomised distance to the source s and the topology of the graph. In fact, we will show in Subsection 4.2.2 that T can be computed efficiently without a SSSP computation based on a modified ball growing procedure.

We begin by showing that the choice of δ_v indeed guarantees the low radius property:

Lemma 4.6. *For any constant $d \geq 1$, with probability at least $1 - n^{-d}$, every component V_i produced by* `Split`*(G, r) has hop radius at most r.*

Proof. Because the components are built using shortest paths from s, the hop radius of every component is at most half the maximum distance from any $v \in V$ to s. This in turn is at most $\max_v \delta_v$. To bound the probability that this exceeds r, recall that $\delta_v \sim \text{Exp}(\lambda)$ with $\lambda = \frac{(d+1)\log n}{r}$ and compute

$$P\left[\bigcup_{v \in V} \delta_v \geq r\right] \leq \sum_{v \in V} P\left[\delta_v \geq r\right] = n \exp(-\lambda r) = n \cdot n^{-(d+1)} = n^{-d}$$

□

Towards bounding the probability that an edge is cut between components, consider a collection of n i.i.d. exponentially distributed random variables X_i, and denote by $X^n_{(k)}$ the *k-th order statistic* of this collection, that is, $X^n_{(k)}$ is the random variable giving the k-th smallest value of the variables $X_1, \ldots, X_n$. The difference between adjacent order statistics is also exponentially distributed:

Lemma 4.7. *Let $X_1, \ldots, X_n \sim \text{Exp}(\lambda)$ be i.i.d. exponentially distributed random variables. Then $X^n_{(1)} \sim \text{Exp}(n\lambda)$ and for $1 \leq k \leq n-1$, $X^n_{(k+1)} - X_{(k)} \sim \text{Exp}((n-k)\lambda)$.*

Proof. For $X^n_{(1)}$, compute that for $x \geq 0$,

$$P[X^n_{(1)} \geq x] = P[\cap_{i=1}^n X_i \geq x] = (1 - F_{\text{Exp}}(x, \lambda))^n = \exp(-n\lambda x)$$

For the second claim, note that conditioning $X^n_{(k+1)}$ on $X^n_{(k)} = t$ gives exactly the same distribution as considering only the remaining $n-k$ random variables conditioned on being at least t, i.e. $P[X^n_{(k+1)} \geq x \mid X^n_{(k)} = t] = P[X^{n-k}_{(1)} \geq x \mid X^{n-k}_{(1)} \geq t]$. Hence compute for $x \geq 0$

$$\begin{aligned} P[X^n_{(k+1)} - X^n_{(k)} \geq x] &= \int_0^\infty P[X^n_{(k+1)} - t \geq x \mid X^n_{(k)} = t] P[X^n_{(k)} = t] dt \\ &= \int_0^\infty P[X^{n-k}_{(1)} \geq x + t \mid X^{n-k}_{(1)} \geq t] P[X^n_{(k)} = t] dt \\ &= \int_0^\infty P[X^{n-k}_{(1)} \geq x] P[X^n_{(k)} = t] dt = \exp(-(n-k)\lambda x) \end{aligned}$$

which proves the second claim. □

Recall that our aim is to show that the probability that any two adjacent vertices end up in different subtrees from s, i.e. have distance less than one from s, is small. The crucial lemma that will let us accomplish this is the following, which will help us bound the probability that the shortest and second-shortest paths to a vertex are close:

Lemma 4.8 ([MPX13]). *For arbitrary but fixed values $d_1 \leq \ldots \leq d_n$, let $X_i = d_i - \delta_i$ for all i, where $\delta_1, \ldots, \delta_n \overset{i.i.d.}{\sim} \text{Exp}(\lambda)$, and denote by $X^n_{(k)}$ again the k-th order statistic of the X_i. Then for any $c \geq 0$, $P[X^n_{(2)} - X^n_{(1)} \geq c] \geq \exp(-\lambda c)$.*

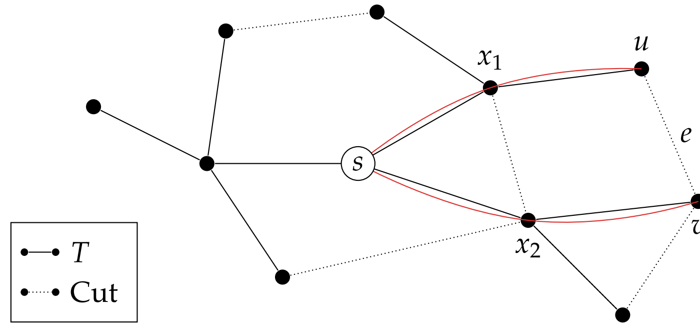

Figure 4.4: The shortest paths from s to u and v are via different neighbours of s, thus cutting the edge e.

Proof. Observe that shifting the d_i by a constant preserves the statement, and hence w.l.o.g. assume $d_1 = 0$. For all $S \subseteq \{1, \dots, n\}$, denote by A_S the event that $X_i \leq 0$ for all $i \in S$, and $X_j > 0$ for all $j \notin S$. Applying total probability, we have

$$P[X^n_{(2)} - X^n_{(1)} \leq c] = \sum_S P[X^n_{(2)} - X^n_{(1)} \leq c \mid A_S] P[A_S]$$

The case $S = \varnothing$ cannot occur because $X_1 = -\delta_1 \leq 0$ always. Consider next the case $|S| = 1$. Then $X_1 \leq 0$ and $X_i > 0$ for all $2 \leq i \leq n$, implying $X^n_{(1)} = X_1$ and thus

$$P[X^n_{(2)} - X^n_{(1)} \geq c \mid A_S] \geq P[-X_1 \geq c] = P[\delta_1 \geq c] = \exp(-\lambda c)$$

If $|S| \geq 2$, then $X^n_{(1)}, X^n_{(2)} \leq 0$, and hence these order statistics are defined only by the $|S|$ variables that are in S, allowing us to consider only these, conditioned on being non-positive by conditioning on A_S. Note that $P[-X_i \geq x \mid -X_i \geq 0] = P[\delta_i \geq x + d_i \mid \delta_i \geq d_i] = P[\delta_i \geq x]$ by the elementary properties of the exponential distribution. This implies that, conditioned on A_S, the *negations* of $X^n_{(1)}, X^n_{(2)}$ behave as the largest and second-largest order statistics of $|S|$ i.i.d. exponentially-distributed random variables. Write these as $Y^{|S|}_{(n)} \hat{=} \left(-X^n_{(1)} \mid A_S\right)$ and $Y^{|S|}_{(n-1)} \hat{=} \left(-X^n_{(2)} \mid A_S\right)$ respectively. Then by Lemma 4.7,

$$P[X^n_{(2)} - X^n_{(1)} \geq c \mid A_S] = P[-Y^{|S|}_{(n-1)} + Y^{|S|}_{(n)} \geq c] = \exp(-\lambda c)$$

Returning to the sum over all S, we have shown

$$P[X^n_{(2)} - X^n_{(1)} \leq c] = \sum_S P\left[X^n_{(2)} - X^n_{(1)} \leq c \mid A_S\right] P[A_S] \geq \exp(-\lambda c)$$

□

With this, we state the proof of Theorem 4.2 as given by Miller, Peng and Xu [MPX13], but incorporating a minor simplification adapted from Becker et al. [Bec+19].

Proof Theorem 4.2, correctness of `Split`. The first property holds by Lemma 4.6.

For the second property, let $e = \{u, v\}$ be any edge of G, with (w.l.o.g.) $d(s, u) \leq d(s, v) \leq d(s, u) + 1$. Since the components are defined via the subtrees of T, e is cut if and only if u, v are in different subtrees, i.e. if T contains paths $\langle s, x_1, \dots, v \rangle$ and $\langle s, x_2, \dots, u \rangle$ where $x_1 \neq x_2$ (see Figure 4.4). Note that $\langle s, x_2, \dots, u, v \rangle$ is also a path from s to v of length $d(s, u) + 1$. Hence because T is a shortest-path tree,

the probability that e is cut is at most the probability that there exist $x_1 \neq x_2$ with $\left|d(s, x_1) + d(x_1, v) - \left(d(s, x_2) - d(x_2, v)\right)\right| \leq 1$. This expression simplifies to

$$\begin{aligned} &\left|d(x_1, v) + \max_x \delta_x - \delta_{x_1} - \left(d(x_2, v) + \max_x \delta_x - \delta_{x_2}\right)\right| \leq 1 \\ \iff &\left|d(x_1, v) - \delta_{x_1} - (d(x_2, v) - \delta_{x_2})\right| \leq 1 \end{aligned}$$

Since x_1 and x_2 are defined through the shortest path from s to u resp. v, the probability of this event is exactly the probability that the smallest and second-smallest values of $\{d(x, v) - \delta_x\}_{x \in V}$ are within 1 of each other. Invoking Lemma 4.8 bounds this to at most $1 - \exp(-\lambda) \leq \lambda = \frac{(d+1)\log n}{r}$. □

With the correctness of Algorithm 4.2 proven, we now turn towards analysing its I/O complexity. As a first step, we ensure that the number of bits needed to store each δ_v is not too large.

Lemma 4.9. *With probability at least $1 - \mathcal{O}(n^{-d})$, the number of bits required for each δ_v is at most $(d+2)\log_2 n$. In particular, the number of records needed to store each δ_v is at most $\mathcal{O}(d)$.*

Proof. First, observe that we only require the ordering of the vertices provided by the δ_v, not their actual value. Hence it is sufficient to sample only as many bits as needed to make all values unique.

$(d+2)\log_2 n$ bits are not enough whenever two of the δ_v are within $2^{-(d+2\log_2 n)} = n^{-(d+2)}$ of each other. By Lemma 4.8 and a union bound over all $k \in [n-1]$, the probability of this occurring is at most

$$\sum_{k=1}^{n-1} 1 - \exp\left(-\frac{(n-k)}{rn^{d+3}}(d+1)\log n\right) \leq (n-1)\left(1 - \exp\left(-\frac{d+1}{n^{d+2}r}\log n\right)\right)$$

This should be at most $\mathcal{O}(n^{-d})$ to attain the high-probability bound. Compute

$$\begin{aligned} 1 - \exp\left(-\frac{d+1}{n^{d+2}r}\log n\right) &\leq \frac{1}{n-1}n^{-d} \\ \Longleftarrow \exp\left(-\frac{d+1}{n^{d+2}r}\log n\right) &\geq \exp\left(-n^{-d-1}\right) \\ \iff (d+1)\log n &\leq rn \end{aligned}$$

which holds for all sufficiently large n. Hence the union bound derived above is at most $\mathcal{O}(n^{-d})$ and the lemma follows. □

Sampling from the exponential distribution accurately when given access to a stream of random bits has been discussed at length in the literature [Dev86] and can be done efficiently, given that the number of records per sample is at most $\mathcal{O}(d)$ where d is constant.

We finish proving the I/O complexity of `Split` in the following subsection, where we show how to construct the shortest-path tree T using a modified ball-growing BFS

4.2.2 Delayed Breadth-First Search

The key observation for constructing T efficiently is that only the edges incidient to s have weights. Moreover, the entire procedure is effectively a ball-growing process with ball radius r. Hence by modifying the algorithm from Subsection 2.6.4, we can hope to achieve good performance for small values of r – recall from Section 4.1 that we choose $r = \rho/4$ where $\rho \leq \exp(\mathcal{O}(\sqrt{\log n \log\log n})) \leq n^{o(1)}$.

Algorithm 4.3 Delayed BFS for `Split`

1: **procedure** `DelayedBFS`(G, δ, s)
2: Let $\mathcal{A}$ be the lexicographically sorted arc list of G without $\{s\}$
3: Sort all $v \in V \setminus \{s\}$ by δ_v as a new list $\Delta = \{(\delta_v, v), \dots\}$
4: Sort and partition $\Delta = \Delta_0 \| \cdots \| \Delta_k$ by integer parts of δ_v
5: Overwrite $\Delta \leftarrow \{(\tilde{\delta}_v, v), \dots\}$ where $\tilde{\delta}_v$ is only the fractional part of δ_v
6: $L(0) \leftarrow \{(\tilde{\delta}_v, s, v) \mid (\tilde{\delta}_v, v) \in \Delta_0\}$
7: **for** $t = 1, \dots,$ until done **do**
8: Let $R \leftarrow \{(\tilde{\delta}_v, s, v) \mid (\tilde{\delta}_v, v) \in \Delta_t\}$ or $R \leftarrow \varnothing$ if $t > k$
9: **for all** $(\tilde{\delta}, \cdot u)$ in $L(t-1)$ and $(u, v) \in \mathcal{A}$ using tandem scan of both lists **do**
10: Append $(\tilde{\delta}, u, v)$ to R, remove (u, v) from $\mathcal{A}$
11: Sort R by target and $\tilde{\delta}$ lexicographically
12: Remove from R all requests to a target v with no incident $(v, \cdot)$ in $\mathcal{A}$
13: For duplicate requests to a target, keep only the request with smallest $\tilde{\delta}$
14: Set $L(t) \leftarrow R$
15: Remove the $\tilde{\delta}$ annotations from all $L(t)$ and **return** $\{L(t)\}_t$

Lemma 4.10. *`DelayedBFS` computes the shortest-path tree T from line 5 of Algorithm 4.2 in $\mathcal{O}(r\,\mathsf{scan}(m) + \mathsf{sort}(m))$ I/Os, where r is the radius parameter to `Split`(G, r).*

Proof. Consider Algorithm 4.3, a modified simultaneous ball growing procedure similar to `SimultaneousBallGrowing` from Subsection 2.6.4. Its correctness follows from the observation that the BFS level of any vertex v depends only on the integer part of δ_u, where u is the first vertex from s along the BFS path from s to v, while the arcs used for the BFS tree depend only on the tie-braking due to the fractional parts of the δ.

As shown in Lemma 4.9, δ takes at most $\mathcal{O}(n)$ records in memory, and hence can be scanned in $\mathcal{O}(\mathsf{scan}(n))$ I/Os. Throughout the execution of the algorithm, Δ is scanned exactly once, and all other operations are analogous to Algorithm 2.5. By Theorem 4.2, the radius of the largest cluster is at most r with high probability. Hence analogous to Lemma 2.15, `DelayedBFS` requires at most $\mathcal{O}(r\,\mathsf{scan}(m) + \mathsf{sort}(m))$ I/Os. □

`DelayedBFS` provides us with the last ingredient to prove Algorithm 4.2's I/O complexity.

Proof of Theorem 4.2, I/O complexity. By Lemma 4.9, sampling the δ_v and writing these values to memory takes at most $\mathcal{O}(dn\,\mathsf{scan}(n))$ I/Os. `DelayedBFS`(G, δ, s) implicitly constructs the supersource graph, hence we do not need to perform this operation explicitly. The call to `DelayedBFS` takes $\mathcal{O}(r\,\mathsf{scan}(m) + \mathsf{sort}(m))$ I/Os. The subtrees can then be identified using an Euler tour on T in $\mathcal{O}(\mathsf{sort}(m))$ I/Os. `DelayedBFS` dominates the I/O complexity when d is considered constant. □

4.2.3 Constructing the Bucket-Partitioning Oracle

This concludes the treatment of `Split`, which handles the single-bucket case. In this section, we show how to use `Split` to handle the multi-bucket case, following the original work of Blelloch et al. [Ble+13].

Theorem 4.3 (Following [Ble+13])**.** *There exists a randomised 2β-bucket-partitioning oracle $\texttt{Partition}(V, r, E_1, \ldots, E_k)$ for $\beta \leq (d+1)\log n$ that succeeds with probability at least $1 - n^{-d}$ and requires $\mathcal{O}(r\,\mathsf{scan}(m) + \mathsf{sort}(m))$ I/Os in expectation, or at most $\mathcal{O}(d \log n \cdot (r\,\mathsf{scan}(m) + \mathsf{sort}(m)))$ with probability at least $1 - \mathcal{O}(n^{-d})$.*

Algorithm 4.4 Bucket Partitioning Algorithm from [Ble+13]

```
1: procedure Partition(G, r, E_1, ..., E_k)
2:     repeat
3:         Let C_1, ... ← Split((G, ⊎_i E_i), r) together with their BFS trees
4:         For every i ∈ [k], compute the number x_i of edges from E_i that are cut
5:     until For all i ∈ [k], x_i ≤ (2(d+1) log n / r) k|E_i|
6:     return the components from the last call to Split
```

Proof. $\texttt{Partition}(V, r, E_1, \ldots, E_k)$ repeatedly invokes $\texttt{Split}(\bar{G}, r)$ from Subsection 4.2.1 with $\bar{G} = (V, \biguplus_i E_i)$ until the number of edges from E_i that are cut is at most $\frac{\beta}{r} k |E_i|$, for all E_i (c.f. Algorithm 4.4). Let X_i be the random variable giving the number of edges from E_i that are cut after a call to `Split`. Using the edge cut probability from Theorem 4.2, $\mathbb{E}[X_i] \leq \frac{\beta}{r}|E_i|$, where $\beta = (d+1)\log n$. Markov's inequality states that $P[X_i \geq 2k\mathbb{E}[X_i]] \leq \frac{1}{2k}$, hence with a union bound over all edge buckets, the probability that any edge bucket has too many edges cut in an iteration of `Partition` is at most $1/2$. Thus the number of iterations is upper-bounded by a geometric random variable with $p = 1/2$, and is at most 2 in expectation and at most $\mathcal{O}(d \log n)$ with probability at least $1 - n^{-d}$. The algorithm succeeds if upon termination, the components have hop radius at most r, which by Theorem 4.2 occurs with probability at least $1 - n^{-d}$.

Every iteration makes one call to `Split`, taking $\mathcal{O}(n + \mathsf{sort}(m))$ I/Os, and otherwise needs to compute the number of edges that are cut. This can be done in $\mathcal{O}(\mathsf{sort}(m))$ I/Os by annotating the endpoints of all edges by their components; an edge is cut if its endpoints have different component annotations. □

4.3 The Algorithm of Elkin et al.

The algorithm of Alon et al. [Alo+95] discussed in Section 4.1 yields a tree whose average stretch is both far away from the theoretical lower bound $\Omega(\log n)$ asymptotically, as well as impractical for any realistic values of n, since the average stretch is only improved over a naive minimum spanning tree for enormous n. Recall that the algorithm relies on carefully balancing the exponential increase of the radius of the contracted graphs with the exponential decrease of the number of remaining edges. The exponential radius increase is because we must assume that when taking a path through a contracted component, this path has length $\rho/2$.

In practice however, we might be able to build the tree in such a way that paths between multiple components are not increased by too much. This requires 'inverting'

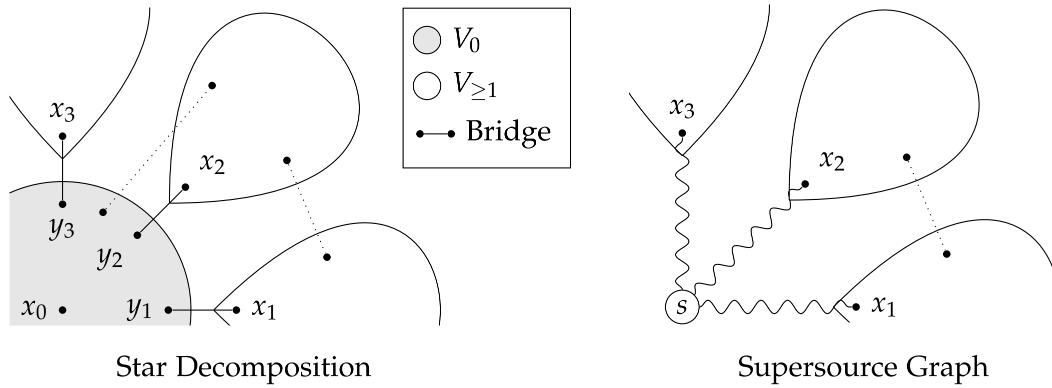

Figure 4.5: Star Decomposition and Supersource Graph

our approach: Instead of building the spanning tree from inside the components, oblivious to the remainder of the graph, we build the tree on some of the cross-component edges, and then recursively partition the components in such a way that the radius bound of the tree does not blow up exponentially.

This idea leads to the algorithm of Elkin et al. [Elk+05]. They grow large components with few between-component edges, structuring the decomposition such that there is a central component V_0 connected to all the other components (see Figure 4.5). The low-stretch tree is built on the *bridge edges* connecting V_0 to the other components, and all components are then decomposed recursively. In every step, the radius of T increases, because we reroute all paths leaving V_0 via the bridge edges. But because the endpoint of every bridge edge will serve as the centre for the next recursion, we implicitly keep track of the structure of T, allowing us to be much more careful to keep this radius increase small. The downside is that current methods for computing this decomposition efficiently still require an expensive single-source shortest path computation (c.f. Lemma 2.17) in every recursive call.

Elkin et al.'s algorithm is the (conceptual) basis for the current state-of-the-art algorithm due to Abraham, Bartal and Neiman [ABN08]. Hence despite the SSSP requirement implying that we cannot implement these algorithms efficiently, by taking some steps towards the algorithm of Elkin et al., we can illustrate almost the full extent of the current understanding of the problem in the literature, and highlight the chief difficulties that need to be overcome for a cache-efficient implementation.

Formally, we will recursively compute *star decompositions* of G:

Definition 4.6 ((β, γ)-Star Decomposition; [Elk+05])**.** Let $G = (V, E, \boldsymbol{\ell})$ be a weighted, undirected graph with edge lengths $\boldsymbol{\ell}$. A partition $V = V_0 \uplus \ldots \uplus V_k$ is called a *star decomposition* of G with centre x_0 (c.f. Figure 4.5) if

- The vertex-induced subgraphs $G[V_i]$ are connected
- For all $1 \leq i \leq n$ there exists a fixed $e = \{x_i, y_i\} \in E$ with $x_i \in V_i$ and $y_i \in V_0$. This e is called the *bridge* between V_0 and V_i.

Denote by $r_G(x)$ the maximum distance from x to any other vertex in G. Let $r_i = \mathsf{rad}_{G[V_i]}(x_i)$ and $r = \mathsf{rad}_G(x_0)$. The decomposition is a (β, γ)*-star decomposition* if, in addition to the above, $0 < \beta < 1/2$ and

1. $r_0 \leq (1-\beta)r$
2. $d(x_0, x_i) \geq \beta r$
3. $r_i + d(x_0, x_i) \leq (1+\gamma)r$ ◇

Properties 1 and 2 ensure that the component V_0 has radius not too small or large, while 3 ensures that the distance from x_0 to the periphery of G does not increase too much when rerouting paths over bridges. By recursively computing star decomposition of G and ensuring that the number of between-component edges is small, we will be able to build a low-stretch spanning tree: The bounded radius of $G[V_0]$ ensures that the recursion will terminate quickly, while property 3 ensures that the stretch incurred by using the bridge edges as spanning tree edges is small. Moreover, only few edges will need to be rerouted in any given step, since we keep the probability of cutting edges low. We will give the details later and focus first on obtaining a star decomposition of G.

Elkin et al. [Elk+05] compute V_0 as a ball around x_0 and then use a delicate cone-growing process to build the V_i around it. The procedure depends on slowly growing each cone in a sequential process, the existence of a cache-efficient implementation is therefore unlikely. Instead, we rely on Becker et al. [Bec+19], who give a distributed algorithm for computing star decompositions. They achieve substantial simplifications over Elkin et al. [Elk+05] by requiring that properties 1–3 hold only with high probability and that the number of between-component edges only be small *in expectation*. At the heart of the algorithm is a modified version of the graph splitting algorithm from Subsection 4.2.1.

4.3.1 Star Decomposition due to Becker et al.

Like Elkin et al., Becker et al. [Bec+19] first compute V_0 as a ball around x_0, but they do so only after randomising the radius of the ball to reduce the probability that an edge crosses the ball's boundary. They then remove V_0 from the graph and attach the artificial source s only to the bridge endpoints x_i (c.f. Figure 4.5), randomising the edge lengths using the exponential distribution. Using this, they show that (i) the probability that an edge crosses components is small, with an argument analogous to that in Subsection 4.2.1 but with weighted edges, and that (ii) property 3 is satisfied with high probability. Formally, by following Becker et al., we will prove the following:

Theorem 4.4 ([Bec+19]). *For any weighted, undirected graph $G = (V, E, \boldsymbol{\ell})$, the algorithm* `StarDecompose`$(G, x_0, \gamma)$ *outputs with probability at least $1 - n^{-d}$ (for any $d > 0$) a $(1/3, \gamma)$-star-decomposition of G. The decomposition has the additional property that for any $e \in E$, the probability that the endpoints of e lie in different components of the decomposition is at most $\mathcal{O}(\frac{(d+1)\log n}{\gamma r}\ell_e)$.* `StarDecompose` *makes two calls to a single-source shortest path computation and otherwise requires $\mathcal{O}(\mathsf{sort}(m))$ I/Os.*

Remark. It is possible to modify the algorithm to produce a (β, γ)-star-decomposition for any $0 < \beta < 1/2$ by sampling r_0 from $[\beta r, (1-\beta)r]$ in Algorithm 4.5. This will result in an algorithm where an edge is cut with probability at most

$$\frac{1}{(1-2\beta)r}\ell_e + \frac{(d+1)\log n}{\gamma r}\ell_e$$

◇

Algorithm 4.5 Star Decomposition; [Bec+19]

```
1: procedure StarDecompose(G, x_0, γ)
2:     Compute r ← rad_G(x_0) and sample r_0 ∈_uar [r/3, 2r/3]
3:     Compute V_0 ← Ball(x_0, r_0) and let X ← {x | {x, y} ∈ E ∧ x ∈ V_0 ∧ y ∉ V_0}
4:     Let λ = (d+1) log n / (γr) and sample δ_x ~ Exp(λ) for all x ∈ X
5:     Let H_s ← G[{s} ⊎ V \ V_0]
6:     Attach an edge of length d_G(x_0, x_i) + max_x δ_x − δ_{x_i} from s to all x_i ∈ X
7:     Compute T ← SSSP-Tree(H_s, s)
8:     Let V_i be the vertices of the i-th subtree of T from s
9:     return V_1, ..., V_k
```

Note that properties 1 and 2 both hold immediately by the choice of r_0. Hence to show that StarDecompose indeed returns a star decomposition, we only need to prove property 3, i.e. that rerouting paths leaving V_0 via bridge edges does not increase distances by much.

Lemma 4.11 ([Bec+19]). *The decomposition returned by Algorithm 4.5 satisfies property 3 of Definition 4.6 with probability at least* $1 - n^{-d}$.

Proof. Let $i \geq 1$ and $v \in V_i$ be arbitrary. By construction,

$$d_{H_s}(s, v) = d_G(x_0, x_i) + d_{H_s}(x_i, v) + \max_x \delta_x - \delta_{x_i} \geq d_G(x_0, x_i) + d_{H_s}(x_i, v)$$

To upper-bound $d_{H_s}(s, v)$, consider the shortest path from x_0 to v in G, and let x_v be the first vertex outside V_0 on this path. Because $\langle s, \ldots, x_v, \ldots, v \rangle$ is also a path in H_s of length $d_G(x_0, v) + \max_x \delta_x - \delta_{x_v}$, we have

$$d_{H_s}(s, v) \leq d_G(x_0, v) + \underbrace{\max_x \delta_x}_{\leq \gamma r} - \delta_{x_v} \leq (1 + \gamma)r$$

with probability at least $1 - n^{-d}$ by the calculation from the proof of Theorem 4.2.

Crucially, because V_i is chosen as a shortest-path subtree, we have $d_{H_s}(x_i, v) = d_{G[V_i]}(x_i, v)$. If we choose v at the periphery of $G[V_i]$ with $d_{G[V_i]}(x_i, v) = r_i$, then combining both bounds yields

$$d_G(x_0, x_i) + r_i \leq d_{H_s}(s, v) \leq (1 + \gamma)r$$

with probability at least $1 - n^{-d}$. □

We are also interested in keeping the probability of cutting an edge small. This probability can be bounded bounded analogously to Theorem 4.2, but with modifications to accommodate the weights on all edges.

Lemma 4.12 ([Bec+19]). *For any $e \in E$, the probability that e is cut, i.e. that the endpoints of e lie in different components after execution of Algorithm 4.5, is at most* $\mathcal{O}(\frac{(d+1)\log n}{\gamma r} \ell_e)$.

Proof. We consider the events that e is cut by the ball V_0 or lies between components V_i, V_j with $1 \leq i, j$ separately and then apply a union bound.

Let $e = \{u, v\} \in E$ be arbitrary and assume w.l.o.g. that $d_G(x_0, u) \leq d_G(x_0, v)$. e is cut by the ball V_0 if and only if $d_G(x_0, u) \leq r_0 < d_G(x_0, v)$. Using $d_G(x_0, v) \leq d_G(x_0, u) + \ell_e$ we obtain

$$P[e \text{ cut by } V_0] \leq P\left[r_0 \in [d_G(x_0, u), d_G(x_0, u) + \ell_e]\right] \leq \frac{\ell_e}{\frac{2r}{3} - \frac{r}{3}} = \frac{3}{r}\ell_e$$

because r_0 is chosen uniformly at random from $[\frac{r}{3}, \frac{2r}{3}]$.

If on the other hand $e = \{u, v\}$ is cut between components $V_i \neq V_j$ with $1 \leq i, j$, then by the same argument as in the proof of Theorem 4.2, there must exist two paths from s to v in H_s (one crossing u, and one not crossing u) whose lengths differ by at most ℓ_e. As in the proof of Theorem 4.2, invoking Lemma 4.8 with values $d_G(x_0, x_i) + d_G(x_i, v)$ bounds the probability of this occurring to at most $1 - \exp(-\lambda \ell_e) \leq \frac{(d+1)\log n}{\gamma r}\ell_e$.

Summing both probabilities in a union bound shows that the probability that e is cut is at most

$$\frac{3}{r}\ell_e + \frac{(d+1)\log n}{\gamma r}\ell_e \leq \mathcal{O}\left(\frac{(d+1)\log n}{\gamma r}\ell_e\right)$$

□

Lemma 4.13. *Algorithm 4.5 requires two shortest path tree computations, in addition to $\mathcal{O}(\mathsf{sort}(m))$ I/Os.*

Proof. The computation of r and V_0 can be implemented using one call to `SSSP-Tree`: From the returned tree, extract r, sample r_0, and then in a top-down tree computation assign to V_0 all vertices within distance r_0 from x_0. Every vertex then knows whether it is in V_0 or not, and hence in $\mathcal{O}(\mathsf{sort}(m))$ we can identify the set X. Sampling from the exponential distribution can be done efficiently as discussed in Lemma 4.9, since we again only rely on the ordering of the vertices provided by the δ_v. It is clear that the remainder of the algorithm can also be implemented in $\mathcal{O}(\mathsf{sort}(m))$ I/Os using top-down tree processing for generating the component assignments. □

4.3.2 From Star Decompositions to Low-Stretch Spanning Trees

The next step is to recursively star-decompose the graph, using the bridge edges as the edges of our low-stretch spanning tree. The simple recursive algorithm is given as pseudocode in Algorithm 4.6. Its correctness relies on the fact that every star decomposition uses the bridge endpoints as centre, and by construction takes care not to increase the radius too much from this centre. Moreover, because V_0 is chosen to be large, the recursion will terminate quickly, helping to mitigate the radius increase.

Algorithm 4.6 Low Average Stretch Spanning Tree from Star Decomposition; [Elk+05]

1: Fix $\gamma = \frac{1}{\log n}$ throughout
2: **procedure** `Star-LSST`(G, x_0)
3: If $|V| \leq 2$ then **return** G
4: Compute $(V_0, \dots, V_k, x_1, \dots, x_k, y_1, \dots, y_k) \leftarrow$ `StarDecompose`(G, x_0, γ)
5: Compute $T_i \leftarrow$ `Star-LSST`$(G[V_i], x_i)$ for $i = 0, \dots, k$
6: **return** T as the union of all edges $\{x_i, y_i\}$ and trees T_i

The first step in the analysis is to bound the radius increase of T from x_0 compared to G. Our proof for this replaces the recursive graph construction of Elkin et al. [Elk+05] with a simpler arithmetic argument.

Lemma 4.14. *If an invocation of* `Star-LSST` *has recursion depth* τ *and returns the tree* T*, then with probability at least* $1 - \tau n^{-d}$*,* $\mathsf{rad}_T(x_0) \le (1+\gamma)^\tau \mathsf{rad}_G(x_0)$*.*

Proof. The proof proceeds by induction on τ. For $\tau = 1$ (i.e. `Star-LSST` terminates without any further recursive calls), the statement holds trivially. Assume that the statement holds for some fixed $\tau \ge 1$ and consider an invocation of `Star-LSST` with recursion depth $\tau + 1$. Let the x_i be as computed in the topmost invocation of `Star-LSST`. Then by the induction hypothesis and property 3 of Definition 4.6, with probability at least $1 - n^{-d}$ it holds that

$$\mathsf{rad}_T(x_0) \le \max_i d(x_0, x_i) + \mathsf{rad}_{T_i}(x_i) \le (1+\gamma)r - r_i + (1+\gamma)^\tau r_i$$

To show that this is indeed bounded by $(1+\gamma)^{\tau+1} r$, use the facts that $r_i \le r$ and, for $k \ge 1$, $\binom{\tau}{k} = \binom{\tau+1}{k} - \binom{\tau}{k-1}$:

$$\begin{aligned}
(1+\gamma)r - r_i + (1+\gamma)^\tau r_i &= (1+\gamma)r - r_i + r_i \sum_{k=0}^{\tau} \binom{\tau}{k} \gamma^k \\
&\le (1+\gamma)r + r \sum_{k=1}^{\tau} \binom{\tau}{k} \gamma^k \\
&\le (1+\gamma)r + r \left(\sum_{k=1}^{\tau} \binom{\tau+1}{k} \gamma^k - \gamma \right) \\
&= r \sum_{k=0}^{\tau} \binom{\tau+1}{k} \gamma^k \le (1+\gamma)^{\tau+1} r
\end{aligned}$$

This concludes the inductive step, conditioned on the event that our bound for $d(x_0, x_i) + \mathsf{rad}_{T_i}(x_i)$ holds throughout. The probability for this is at least $(1 - n^{-d})^\tau \ge 1 - \tau n^{-d}$ (using Bernoulli's inequality). □

We now show that τ is at most logarithmic in n, hence the radius increases not by much – indeed, since $\gamma = \frac{1}{\log n}$, $(1+\gamma)^\tau \le \exp(\frac{\tau}{\log n}) \le \mathcal{O}(1)$ when $\tau \le \mathcal{O}(\log n)$, i.e. $\mathsf{rad}_T(x_0)$ will be at most a constant factor larger than $\mathsf{rad}_G(x_0)$.

Lemma 4.15 ([Bec+19])**.** *With probability at least* $1 - \tau n^{-d}$*,* `Star-LSST` *has recursion depth at most* $\tau \le \mathcal{O}(\log n + \log U)$*, where* $U = \max_e \ell_e / \min_e \ell_e$ *is the length ratio of the graph.*

Proof. By properties 2 and 3 from Definition 4.6, for $i \ge 1$

$$\left.\begin{aligned} r_i + d(x_0, x_i) &\le (1+\gamma)r \\ d(x_0, x_i) &\ge \beta r \end{aligned}\right\} \implies r_i \le (1+\gamma-\beta)r$$

Moreover, $r_0 \le (1-\beta)r < (1+\gamma-\beta)r$ by property 1. Thus at recursion depth t, the input graph has radius at most $(1+\gamma-\beta)^t r$. We may assume by rescaling that $\ell_e \ge 1$ for all e, and hence the recursion terminates when the radius decreases to strictly less than 1, i.e. when $t > -\frac{\log r}{\log(1+\gamma-\beta)}$. Let $U = \max_e \ell_e / \min_e \ell_e$ be the length ratio of the graph; then after rescaling $r \le nU$ and hence we arrive at a recursion depth of at most

$$\tau \le \left\lceil -\frac{\log n + \log U}{\log(1+\gamma-\beta)} \right\rceil \le \mathcal{O}(\log n + \log U)$$

This is predicated on all calls to `StarDecompose` satisfying Definition 4.6, which occurs with probability at least $(1 - n^{-d})^\tau \ge 1 - \tau n^{-d}$. □

Combining the small radius of T with the low edge cutting probability from the star decomposition almost yields the low-stretch spanning tree:

Lemma 4.16 ([Bec+19])**.** *For any edge $e \in E$, the expected stretch of e in the tree T with respect to the randomness of the algorithm is at most $\mathbb{E}[\mathsf{stretch}_T(e)] \leq \mathcal{O}((d+1)\log^3 n)$.*

Proof. Let T be the final tree returned by the algorithm. If e is an edge of T, then the distance between the endpoints of e in T is exactly ℓ_e. Else, e must be cut in some recursive call, and its depth depends only on this call and its descendant recursions. Hence it suffices to consider the case where e is cut in the first call of `Star-LSST` to later extend the argument to all recursive calls. If e is cut in the first call, then $\mathsf{path}_T(e)$ must cross x_0, and hence the distance in T between the endpoints of e is at most $2\,\mathsf{rad}_T(x_0)$, which Lemma 4.14 is bounded by $2 \cdot (1+\gamma)^\tau r$ with high probability.[5] If e is cut at a deeper recursion level, then the radius of that subtree is still at most $\mathsf{rad}_T(x_0)$, and hence the argument still applies.

To combine all cases formally, we rely on total expectation: Let $e = \{u, v\}$ and compute

$$\mathbb{E}[d_T(u,v)] = \mathbb{E}[d_T(u,v) \mid e \in E(T)]P[e \in E(T)] + \mathbb{E}[d_T \mid e \notin E(T)]P[e \notin E(T)]$$

The probability that e is cut in any given invocation is given by Lemma 4.12 as $\mathcal{O}(\frac{(d+1)\log n}{\gamma r}\ell_e)$, and by a union bound, the probability that it is cut in any one of the at most τ recursions that e is involved in is at most τ times that. Thus

$$\mathbb{E}[d_T(u,v)] \leq \ell_e + 2(1+\gamma)^\tau r \cdot \mathcal{O}\left(\tau \frac{(d+1)\log n}{\gamma r}\ell_e\right) \leq \mathcal{O}\left((d+1)\log^3 n \ell_e\right)$$

where we use that $\tau \leq \mathcal{O}(\log n), \gamma = \frac{1}{\log n}$ and thus $(1+\gamma)^\tau \leq \exp(\tau\gamma) \leq \mathcal{O}(1)$. Hence the expected stretch of e is at most

$$\mathbb{E}[\mathsf{stretch}_T(e)] \leq \mathcal{O}\left(\frac{(d+1)\log^3 n \ell_e}{\ell_e}\right) = \mathcal{O}\left((d+1)\log^3 n\right)$$

□

The lemma ensures that every edge has low stretch in expectation, but we must still find a single tree that achieves low average stretch over all edges. We show that we will produce such a tree after at most $\mathcal{O}((d+1)\log^3 n)$ calls to `Star-LSST`.

Theorem 4.5. *Let $G = (V, E, \boldsymbol{\ell})$ be some undirected graph with polynomially-bounded length ratio $\max_e \ell_e / \min_e \ell_e$. There exists an algorithm `LSST`(G, x_0) that outputs with high probability a spanning tree T of G such that T has average stretch $\mathcal{O}(\log^3 n)$. `LSST` expends at most $\mathcal{O}(\log^2 n\,\mathsf{SSSP}(n,m))$ I/Os, where $\mathsf{SSSP}(n,m) \leq \mathcal{O}(n + \mathsf{sort}(m)\log\frac{M}{DB})$ is the number of I/Os required to solve the single-source shortest path problem on an undirected graph.*

Proof. Applying linearity of expectation to Lemma 4.16 shows that in expectation, the average stretch of the tree T returned by `Star-LSST` is $\mathcal{O}((d+1)\log^3 n)$. By Markov's inequality, calling `Star-LSST` $\mathcal{O}(d \log n)$ times ensures that at least one of the

[5] We sweep the formal handling of the case where the bound does not hold under the rug, but it can be resolved with simple total probability argument if weights are polynomially bounded by choosing d sufficiently large. The unbounded case can be handled by a more involved construction that contracts all edges outside a specific length range between the recursions; see the original work of Elkin et al. [Elk+05] for more information.

produced trees has average stretch at most $\mathcal{O}((d+1)\log^3 n)$ with probability at least $1 - n^{-d}$. Note that we can compute the stretch of every edge by an algorithm similar to Algorithm 5.1 that will be shown in Section 5.1 using $\mathcal{O}(\log_2 m\, \mathsf{scan}(m))$ I/Os, which is less than the I/Os required for the recursive star decompositions.

In total, we perform $\mathcal{O}(d \log^2 n)$ calls to `StarDecompose`, with each call taking at most $\mathcal{O}(\mathsf{SSSP}(n,m))$ I/Os by Lemma 4.13, since the shortest-path computations dominate. Recall from Lemma 2.17 that $\mathsf{SSSP}(n,m) \le \mathcal{O}(n + \mathsf{sort}(m) \log \frac{M}{DB})$. □

4.4 Towards a Practical Cache-Efficient Algorithm

To the best of this thesis' author's knowledge, the only constructions of low-average-stretch *spanning* trees published in the literature are those of Alon et al. [Alo+95], which was discussed in Section 4.1, the star-decomposition based approach due to Elkin et al. [Elk+05] that was the previous section's subject, and the state-of-the-art method due to Abraham, Bartal and Neiman [ABN08] that conceptually builds on top of the star decomposition technique. Alon et al.'s algorithm admits an asymptotically cache-efficient implementation, but is not practical, whereas Elkin et al.'s algorithm is perhaps more practical, but even with the techniques of Becker et al. [Bec+19] still does not readily admit a cache-efficient implementation when building on current methods.

One might attempt to modify the algorithm to use only approximate shortest paths.[6] But recall that the low probability of cutting edges relies on small differences between the shortest and second-shortest paths to a vertex, and the fact that these shortest paths induce a metric: for any $u, v \in V$ we have the triangle inequality $d(s,u) \le d(s,v) + d(v,u)$. When using $(1+\epsilon)$-approximate shortest paths of length $\tilde{d}(s,\cdot)$, the triangle inequality weakens substantially to $\tilde{d}(s,u) \le (1+\epsilon)(\tilde{d}(s,v) + \ell_e)$, where $e = \{u,v\}$. Following the same calculations as Subsection 4.3.1 will show that the edge cutting probability using $(1+\epsilon)$-approximate SSSP is at most $\mathcal{O}(\frac{(d+1)\log n}{\gamma}(\frac{\ell_e}{r} + \epsilon))$, and thus the expected stretch is roughly $\mathcal{O}((d+1)\frac{\log^2 n}{\gamma}(1 + \frac{\epsilon r}{\ell_e}))$

what if we use an approx. that induces a metric?

We can use a construction of Elkin et al. [Elk+05] to ensure that the minimum edge lengths $\min_e \ell_e \ge \Omega(\frac{r}{n \log n})$ are large in any invocation to the star decomposition, but this still leaves an $\mathcal{O}(\epsilon n \log n)$-term in the expected stretch, requiring $\epsilon \le \frac{\mathsf{polylog}(n)}{n}$ to achieve polylogarithmic expected stretch.

This is still too small to achieve good performance; approximate shortest path algorithms tend to scale with ϵ^{-1} or even ϵ^{-2} and hence $\epsilon \ge \frac{1}{\mathsf{polylog}(n)}$ would be desirable. The next idea is to observe that we only require accurate shortest path computations in the neighbourhood of the edges that are cut, i.e. on the fringes of the components. We could for example grow the components with a crude approximation ratio, and then restart the computation for the component fringes to 'locally' refine the approximation. Becker, Emek and Lenzen [BEL20] devise a scheme that accomplishes this by carefully contracting the centre of the approximate SSSP tree and restarting the computation on the remaining vertices. The technique fails however for computing star decompositions, because we must eventually capture all vertices: Even after many refinement iterations, the fringes could still have $\Theta(n)$ vertices due to the structure of

[6] We mean $(1+\epsilon)$-approximate shortest paths in the sense that for any path $\langle s, \ldots, v\rangle$ in the tree, the length of the path is at most $(1+\epsilon)$ times the distance $d(s,v)$ from s to v along the truly shortest path.

the graph, and we would then still need to pay for a full SSSP computation.[7]

Becker, Emek and Lenzen do show how their technique can be used to compute other kinds of low-average-stretch trees; we will come back to this in Chapter 7.

Borradaile et al. [Bor+20] give a linear-time algorithm for unweighted graphs of bounded width.[8] Their algorithm essentially reduces to a minimum spanning tree computation on specially-crafted edge weights, and hence can be implemented cache-efficiently. It might be possible to interleave their algorithm with that of Section 4.1: Let $\varphi(G)$ denote the bandwidth of G, i.e. the minimum quantity $\max\{|f(u) - f(v)| \mid \{u,v\} \in E\}$ achievable over all bijective mappings $f : V \to \mathbb{N}$. One can show that $\varphi(G) \geq \Omega(\frac{n}{\mathsf{hoprad}(G)})$, hence graphs of low bandwidth have large radius. Intuitively, this means that the algorithm of Alon et al. performs worst on such graphs, because the radius of the contracted graph does not decrease as quickly. One might be able to devise a scheme that interleaves the algorithm of Borradaile et al. at appropriate stages to improve the stretch of the tree produced by Alon et al.'s algorithm.

Finally, in the context of this thesis, we will observe in Section 5.3 that we will not actually require a full low-average-stretch spanning tree on G, but rather a forest of low-average-stretch spanning trees. More concretely, we will be able to split G into a set of $n^{O(1)}$ disjoint connected components, and compute the spanning trees inside of these components. While this cannot accelerate the star-decomposition based algorithm (since we are still dealing with n vertices in total), it can improve the stretch produced by the algorithm of Alon et al., though only by effectively constant factors in the $\exp(\mathcal{O}(\sqrt{\log n \log\log n}))$ term.

[7] Becker, Emek and Lenzen confirm in private communication that their technique does not appear to be applicable for computing low-stretch spanning trees.

[8] In particular, for graphs of cutwidth or bandwidth at most b, they construct spanning trees of stretch at most $\mathcal{O}(b^2)$ in linear time with high probability. They also give a polynomial-time dynamic programming algorithm for graphs of bounded treewidth.

Chapter 5

Congestion Approximators

In this chapter, we connect low-stretch spanning trees to the minimum congestion flow problem, through a process due to Madry [Mad11] that we shall refer to as *Madry's decomposition*. The discussion in this thesis is based directly on his work, but tries to motivate the construction through a slower 'step-by-step' approach, while modifying the necessary parts to achieve a cache-efficient implementation (Madry designs his scheme in the standard RAM model). Recall from the introduction that we are aiming to devise an iterative algorithm for solving this problem, where every step $\tilde{\boldsymbol{f}}$ is selected by approximately minimising the potential function

$$\Phi(\tilde{\boldsymbol{f}}) = \|\boldsymbol{C}^{-1}\tilde{\boldsymbol{f}}\|_\infty + 2\alpha \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{B}\tilde{\boldsymbol{f}})$$

Here, $\boldsymbol{R}$ is an *α-congestion-approximator*, the focus of this chapter, which we now formally define as follows:

Definition 5.1 (α-congestion-approximator; [She13])**.** An *α-congestion-approximator* for an undirected, capacitated graph G is a linear operator $\boldsymbol{R} : B \to \mathbb{R}^k$ that for any valid demands $\boldsymbol{b}$ satisfies

$$\|\boldsymbol{R}\boldsymbol{b}\|_\infty \leq \mathsf{opt}_G(\boldsymbol{b}) \leq \alpha \|\boldsymbol{R}\boldsymbol{b}\|_\infty$$

where $B \subseteq \mathbb{R}^n$ is the linear subspace of valid demands. For technical reasons, we also require $k \leq \frac{1}{2}n^2$. ◇

For the purposes of solving optimisation problems involving $\boldsymbol{R}$, $\boldsymbol{R}$ can be thought of as a matrix, but a graph theoretic interpretation is more intuitive and indeed what this chapter will be working with. Under a graph-theoretic interpretation, $\boldsymbol{R}$ corresponds to a graph $G_R = (V, E_R, \mathbb{c})$ that dominates the original graph $G = (V, E, \boldsymbol{c})$ on all cuts, but conversely is dominated by the scaled-up graph $(V, E, \alpha\boldsymbol{c})$ on all cuts. In other words, for all $S \subseteq V$,

$$c_{S \overset{E}{\leftrightarrow} V \setminus S} \leq \mathbb{c}_{S \overset{E_R}{\leftrightarrow} V \setminus S} \leq \alpha c_{S \overset{E}{\leftrightarrow} V \setminus S}$$

In the following, the explicit annotation of the edge set is again omitted. Assume $\|\boldsymbol{R}\boldsymbol{b}\|_\infty = \mathsf{opt}_{G_R}(\boldsymbol{b})$ without making this correspondence precise at this point. The intention is that G_R is somehow a much simpler graph than G, making it easy to route demands in G_R. Because G_R dominates G on all cuts, the congestion incurred by an optimal routing in G_R under-approximates the congestion of an optimal routing in G: Let $S \subseteq V$ be a maximally congested of G_R in the sense of Theorem 2.1. Then

$$\|\boldsymbol{R}\boldsymbol{b}\|_\infty = \mathsf{opt}_{G_R}(\boldsymbol{b}) = \frac{|b_S|}{\mathbb{c}_{S \leftrightarrow V \setminus S}} \leq \frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} \leq \mathsf{opt}_G(\boldsymbol{b})$$

At the same time, this under-approximation cannot be too loose: Let S now be a maximally congested cut of G and compute

$$\alpha\|\boldsymbol{Rb}\|_\infty = \alpha\mathsf{opt}_{G_R}(\boldsymbol{b}) \geq \frac{\alpha|b_S|}{\mathbb{c}_{S\leftrightarrow V\setminus S}} \geq \frac{\alpha|b_S|}{\alpha c_{S\leftrightarrow V\setminus S}} = \mathsf{opt}_G(\boldsymbol{b})$$

thus we recover exactly the approximation bounds from Definition 5.1.

Consider with this motivation the following terminology:

Definition 5.2 (β-embedding)**.** An undirected, capacitated graph $\tilde{G} = (V, \tilde{E}, \tilde{\boldsymbol{c}})$ β-*embeds* into a graph $G = (V, E, \boldsymbol{c})$ if for every vertex-induced cut $S \subseteq V$,

$$\tilde{c}_{S \overset{\tilde{E}}{\leftrightarrow} V\setminus S} \leq \beta c_{S \overset{E}{\leftrightarrow} V\setminus S} \qquad \diamond$$

This definition is closely related to the notion of an ϵ-graph-sparsifier from Definition 3.1. Indeed, an ϵ-graph sparsifier $\tilde{G}$ of G is a graph that $(1+\epsilon)$-embeds into G, but such that G also $\frac{1}{1-\epsilon}$-embeds into $\tilde{G}$.

For the purposes of approximating congestions, we seek a graph G_R that 1-embeds into G, but conversely G should α-embed into G_R. The crucial difference to graph sparsifiers is that we additionally desire G_R to be so simple that solving the congestion minimisation problem in G_R becomes a trivial, ideally linear-time operation. It will be much easier to do so when we represent G_R as a convex combination $\{\lambda^{(i)}, G^{(i)}\}_i$ of even simpler graphs $G^{(i)} = (V, E^{(i)}, \boldsymbol{c}^{(i)})$. The graph G_R is recovered from the combination as $E_R = \bigcup_i E^{(i)}$ and $\mathbb{c}_e = \sum_i \lambda^{(i)} c_e^{(i)}$. Formally, define

Definition 5.3 (($\alpha, \mathbb{G}$)-decomposition; [Mad11])**.** Let $G = (V, E, \boldsymbol{c})$ be some undirected, capacitated graph and $\mathbb{G}$ be a set of graphs on G. An $(\alpha, \mathbb{G})$-*decomposition* of G is a convex combination $\{\lambda^{(i)}, G^{(i)}\}$ of graphs $G^{(i)} = (V, E^{(i)}, \boldsymbol{c}^{(i)}) \in \mathbb{G}$ satisfying

1. $\lambda^{(i)} > 0$ for all i and $\sum_i \lambda^{(i)} = 1$
2. G 1-embeds into every $G^{(i)}$
3. Conversely, every $G^{(i)}$ embeds into G such that the combination α-embeds into G on average, in the sense that for any cut $S \subseteq V$,

$$\sum_i \lambda^{(i)} c^{(i)}_{S \overset{E^{(i)}}{\leftrightarrow} V\setminus S} \leq \alpha c_{S \overset{E}{\leftrightarrow} V\setminus S} \qquad \diamond$$

Note that this implies

$$c_{S \overset{E}{\leftrightarrow} V\setminus S} \leq \sum_i \lambda^{(i)} c^{(i)}_{S \overset{E^{(i)}}{\leftrightarrow} V\setminus S} \leq \alpha c_{S \overset{E}{\leftrightarrow} V\setminus S}$$

for every $S \subseteq V$ because property 2 ensures that every $G^{(i)}$ is connected when G is connected (recall that this is a basic assumption throughout the thesis), ensuring that every $G^{(i)}$ has at least one edge crossing the cut $S \leftrightarrow V \setminus S$. In particular, the convex combination fulfils exactly the purpose of the imaginary graph G_R discussed above.

Lemma 5.1. *For any undirected, capacitated graph* $G = (V, E, \boldsymbol{c})$*, a convex combination* $\{(\lambda^{(i)}, G^{(i)})\}_i$ *of undirected, capacitated graphs* $G^{(i)} = (V, E^{(i)}, \mathbb{c}^{(i)}) \in \mathbb{G}$ *for some* $\mathbb{G}$ *is an*

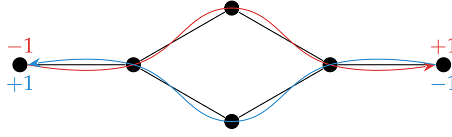

Figure 5.1: Non-Linearity of Optimal Routing

$(\alpha, \mathbb{G})$-decomposition of G if and only if G one-emebds into every $G^{(i)}$, and the graph G_R, defined through

$$G_R = (V, E_R, \mathbb{c}) \qquad E_R = \bigcup_i E^{(i)} \qquad \mathbb{c}_e = \sum_i \lambda^{(i)} \mathbb{c}_e^{(i)}$$

α-embeds into G, where we fix $\mathbb{c}_e^{(i)} = 0$ if $e \notin E^{(i)}$.

Proof. Properties 1 and 2 of Definition 5.3 are satisfied by the statement of the lemma. If $\{(\lambda^{(i)}, G^{(i)})\}_i$ is an $(\alpha, \mathbb{G})$-decomposition of G, then it follows directly from the construction of G_R that G_R α-emebds into G. For the other direction, assume G_R α-embeds into G. Then for any cut $S \subseteq V$,

$$\alpha c_{S \overset{E}{\leftrightarrow} V \setminus S} \geq \mathbb{c}_{S \overset{E_R}{\leftrightarrow} V \setminus S} = \sum_i \lambda^{(i)} \mathbb{c}^{(i)}_{S \overset{E^{(i)}}{\leftrightarrow} V \setminus S}$$

and the lemma follows immediately. □

A minor caveat in this graph-based construction of congestion approximators is that require $\boldsymbol{R}$ to be linear for the sake of optimising the potential function $\Phi(\tilde{\boldsymbol{f}})$, but optimal routings on G_R are not necessarily linear, in the sense that if $\boldsymbol{f}^{(1)}$ and $\boldsymbol{f}^{(2)}$ are optimal routings of $\boldsymbol{b}^{(1)}$ and $\boldsymbol{b}^{(2)}$, then $\boldsymbol{f}^{(1)} + \boldsymbol{f}^{(2)}$ is not necessarily an optimal routing of $\boldsymbol{b}^{(1)} + \boldsymbol{b}^{(2)}$. Consider for example the unit-weighted graph and demands shown in Figure 5.1. Adding both demands together yields the zero demands, but the two flows add to a cycle flow, which has non-zero congestion. This is further discussed in Section 7.2; for now it suffices to observe that this problem does not occur when routing is only performed on trees, which are acyclic. This yields the following lemma, the proof of which is deferred to the end of Section 5.1.

Lemma 5.2 ([She13])**.** *If there exists an $(\alpha, \tilde{\mathbb{T}}_V)$-decomposition of some graph $G = (V, E, \boldsymbol{c})$, where $\tilde{\mathbb{T}}_V$ is the set of trees on V, then there exists an α-congestion approximator of G.*

A useful technique for proving graph embeddings will be in the form of *embedding flows* resulting from a special multicommodity flow problem, which we now define.

Definition 5.4 (Minimum-Congestion Concurrent Flow Problem)**.** Given some undirected, capacitated graph $G = (V, E, \boldsymbol{c})$ along with demands $b^{\{u,v\}} \geq 0$ for all unordered pairs of vertices $\{u, v\} \in \{\{u, v\} \mid u, v \in V\}$, the *minimum-congestion concurrent flow problem* seeks to find flows $\boldsymbol{f}^{\{u,v\}}$ such that

1. Every $\boldsymbol{f}^{\{u,v\}}$ routes exactly $b^{\{u,v\}}$ units of flow between u and v (in some direction), and satisfies flow conservation everywhere else.

2. The *maximum concurrent congestion*

$$\max_{e \in E} \frac{1}{c_e} \sum_{\{u,v\}} |f_e^{\{u,v\}}|$$

is minimised.

The *value* of the problem is the maximum congestion incurred by an optimal assignment of flows. If demands are only provided for a subset of all vertex-pairs, assume the remaining demands are zero. ◇

The concurrent flow problem relates to graph embeddings via the following lemma:

Lemma 5.3. *For any two undirected, capacitated graphs $\bar{G} = (V, \bar{E}, \bar{\boldsymbol{c}})$ and $G = (V, E, \boldsymbol{c})$ on the same set of vertices, consider the minimum-congestion concurrent flow problem in G of routing demands $\bar{c}_{\bar{e}}$ between the endpoints of every $\bar{e} \in \bar{E}$. $\bar{G}$ β-embeds into G if and only if the value of this problem is at most β.*

Proof. Consider an arbitrary optimal solution $\{\boldsymbol{f}^{\{u,v\}}\}_{\{u,v\}}$ to the concurrent flow problem, and let $S \subseteq V$ be an arbitrary cut. The flow $\boldsymbol{f}$ in G given by $f_e = \sum_{\{u,v\}} |f_e^{\{u,v\}}|$ routes exactly $\bar{c}_{S \overset{\bar{E}}{\leftrightarrow} V \setminus S}$ units of flow across the cut S. At the same time, each edge of G has congestion at most β, so the flow cannot route more than $\beta c_{S \overset{\bar{E}}{\leftrightarrow} V \setminus S}$ units of flow across the cut. Hence $\bar{c}_{S \overset{\bar{E}}{\leftrightarrow} V \setminus S} \leq \beta c_{S \overset{\bar{E}}{\leftrightarrow} V \setminus S}$. Since S was chosen arbitrarily, this shows that $\bar{G}$ β-embeds into G.

For the only-if direction, assume $\bar{G}$ β-embeds into G. Then for any cut $S \subseteq V$, the demand across the cut is exactly $\bar{c}_{S \overset{\bar{E}}{\leftrightarrow} V \setminus S}$. We now invoke a multi-commodity analogue of Lemma 2.3 [LR99] to conclude that there exists some cut S such that an optimal solution to the concurrent flow problem has congestion exactly $\frac{\bar{c}_{S \overset{\bar{E}}{\leftrightarrow} V \setminus S}}{c_{S \overset{\bar{E}}{\leftrightarrow} V \setminus S}} \leq \beta$. □

Definition 5.5 (Embedding Flow; adapted from [Mad11])**.** Consider some graphs $\bar{G} = (V, \bar{E}, \bar{\boldsymbol{c}})$ and $G = (V, E, \boldsymbol{c})$. We say that a flow $\boldsymbol{f}$ is a *β-embedding flow* of $\bar{G}$ into G if there exists some (not necessarily optimal) solution $\{\boldsymbol{f}^{\{u,v\}}\}_{\{u,v\}}$ to the concurrent flow problem of Lemma 5.3 with maximum congestion at most β, such that for all $e \in E$, $|f_e| = \sum_{\{u,v\}} |f_e^{\{u,v\}}|$. By Lemma 5.3, there exists a β-embedding flow of $\bar{G}$ into G if and only if $\bar{G}$ β-embeds into G.

The *embedding congestion* of $\bar{G}$ into G is the maximum congestion of an optimal embedding flow. ◇

In particular, this means that in order to prove that $\bar{G}$ β-embeds into some graph G, it suffices to find a β-embedding flow of $\bar{G}$ into G. This can be an easier problem to reason about. If for example G is a tree, then the embedding flow of an edge $\bar{e}$ from $\bar{G}$ into G can only be routed along $\mathsf{path}_T(\bar{e})$, and the overall embedding flow is the sum of these flows. The embedding-flow formulation also allows computing capacities for G to achieve a desired embedding factor for $\bar{G}$ into G, because the embedding flow of $\bar{G}$ into G corresponds to the capacities in G required for a one-embedding of $\bar{G}$ into G. This will become more clear in Section 5.1, where we give an algorithm for embedding any graph G into a tree T on the same vertices by computing the corresponding embedding flow, and then use this flow to define capacities $\mathbb{c}$ on T to achieve a 1-embedding of G into T.

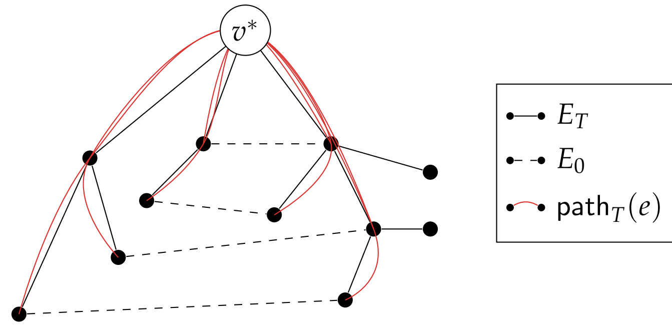

Figure 5.2: Embedding G_0 into a Tree

5.1 Embedding into Trees

Trees, being the simplest connected graphs, play a crucial role in finding good $(\alpha, \mathbb{G})$-decompositions. In a tree $T = (V, E_T)$, every edge $e_T \in E_T$ *induces* a cut in the sense that removing e_T splits the vertices V into two connected components. Recall that we assume an arbitrary but fixed orientation on the edges E. We can thus uniquely define the cut set $S \subseteq V$ induced by $e_T \in E_T$ as the component which e_T points towards, i.e.

Definition 5.6 (Tree-Edge Induced Cut)**.** For any undirected graph $G = (V, E)$ with an arbitrary orientation $\vec{E}$ of the edges E, and a spanning tree $T = (V, E_T)$ of G, the cut $\mathcal{X}_T(e_T) \subseteq V$ *induced* by a tree edge $e_T \in E_T$ is defined by the connected component of T after removal of e_T such that e_T is oriented towards the component. ◇

For any valid demands $\boldsymbol{b}$, there exists a unique flow that routes $\boldsymbol{b}$, hence routing demands is easy. Moreover, the embedding flow $\boldsymbol{f}$ of a graph $G = (V, E, \boldsymbol{c})$ into T is unique: For every tree edge $e_T \in E_T$, the flow on e_T must be exactly $|f_{e_T}| = c_{\mathcal{X}_T(e_T) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_T(e_T)}$.

This suggests a simple recursive algorithm for computing $\boldsymbol{f}$ due to Madry [Mad11], who adapted it from a related algorithm in an earlier version of Spielman and Teng [ST14]. To illustrate the approach, root T at some vertex v^* and assume that $v^* \in \mathsf{path}_T(e)$ for all edges $e \in E$ (c.f. Figure 5.2). Then we can collect the required flow across the tree edges in a bottom-up fashion, because any tree edge $\{u, v\}$, where we assume u is a descendant of v, must route all of the flow that arrives at u from its descendants (since by assumption all these flows must get routed up via v to the root v^*), in addition to the flow required to embed any edges of G that are incident to v. In general, G will also contain edges whose path in T does not cross the root v^*, but these will be entirely contained in some subtree of v^*, and can thus be routed in the recursively smaller trees. More pointedly, every recursion of the algorithm partitions the edges E into

$$E_0 = \{e \in E \mid v^* \in \mathsf{path}_T(e)\} \qquad E_i = \{e \in E \mid \mathsf{path}_T(e) \subseteq T_i\}$$

where $T_1, \ldots, T_k$ are the subtrees of v^*. Then $G_0 = (V, E_0, \boldsymbol{c})$ can be embedded into T as described above, and the $G_i = (V, E_i, \boldsymbol{c})$ are embedded recursively into the subtrees T_i. Selecting appropriate roots v^* to guarantee fast termination yields the final algorithm, given in pseudocode form in Algorithm 5.1.

Algorithm 5.1 Tree Embedding; [Mad11]

1: **procedure** `EmbedTree`(G, T)
2: If $E(T) = \varnothing$, return the empty flow
3: Identify a splitter vertex $v^* \leftarrow$ `FindSplitter`(G, T)
4: Root T at v^*
5: Propagate subtree labels T_i to all descendants of v^*
6: Build partition $E = E_0 \uplus \ldots \uplus E_k$
7: ▷ *Now begin processing* G_0 ◁
8: Compute for all $v \in V$ the value $d_v = c_{\{v\} \overset{E_0}{\leftrightarrow} V \setminus \{v\}}$
9: $\forall v \in V$, let $\sigma_v \leftarrow \sum_{u \in V(T_v)} d_u$ where $V(T_v)$ are the descendants of v, including v
10: For every edge $\{u, v\} \in E(T)$ where u is a descendant of v, let $|f_e^{(0)}| \leftarrow \sigma_u$
11: ▷ *Recurse on subtrees* ◁
12: For all subtrees T_i of v^*, compute $\boldsymbol{f}^{(i)} \leftarrow$ `EmbedTree`(G_i, T_i)
13: **return** $\boldsymbol{f} = \sum_{i=0}^{k} \boldsymbol{f}^{(i)}$
14: **procedure** `FindSplitter`(G, T)
15: Construct an Euler tour on T, initialise $w \leftarrow 0$
16: **for all** arcs (u, v) along the tour, starting with v being some leaf **do**
17: Label the arc with the current sum of weights w
18: If this is the first arc into v, increment w by $1 + \deg_G(v)$
19: ▷ *The size of some subtree is now given by the arc label of the incoming arc from the tree, minus the arc label of the outgoing arc into the tree* ◁
20: Using the Euler tour, compute for each vertex the size of all its subtrees
21: Scan all vertices to identify some splitter v^*, and **return** v^*

Lemma 5.4 (Tree Embedding; [Mad11]). *Let $T = (V, E_T)$ be a tree and $G = (V, E, \boldsymbol{c})$ be some graph. There exists an algorithm `EmbedTree`(G, T) that computes the unique embedding flow $\boldsymbol{f}$ of G into T in $\mathcal{O}(\log_2 m \cdot \mathsf{scan}(m))$ I/Os.*

Proof. We begin by proving correctness of the algorithm `EmbedTree` given in Algorithm 5.1. Define for any tree T' on V the *size* of T' as $\mathsf{size}_G(T') = \sum_{v \in V(T')}(1 + \deg_G(v))$. A *splitter* $v \in v^*$ is a vertex such that all subtrees $T_1, \ldots, T_k$ of v^* (without v^*) have size at most one-half the size of T, i.e. for all $i \in [k]$, $\mathsf{size}_G(T_i) \leq \frac{1}{2}\mathsf{size}_G(T)$. Lemma 5.5 below proves that v^* always exists and can be found in $\mathcal{O}(\mathsf{sort}(m))$ I/Os.

Now consider the partition $E = E_0 \uplus \ldots \uplus E_k$ described above, and let G_i be the induced graphs on the edges E_i. We claim that `EmbedTree` correctly computes an embedding of G_0 into T. Consider any tree edge $e_T \in E_T$. The required embedding flow of G_0 into T on e_T is given by (c.f. Figure 5.2)

$$|f_{e_T}^{(0)}| = c_{\mathcal{X}_T(e_T) \overset{E_0}{\leftrightarrow} V \setminus \mathcal{X}_T(e_T)} = \sum_{\substack{e \in E_0 \\ e_T \in \mathsf{path}_T(e)}} c_e$$

By construction of E_0, $\mathsf{path}_T(e)$ contains the tree edge $e_T = \{u, v\}$, where we let u be the descendant of v, if and only if an endpoint of e is a descendant of u, or u itself. Let T_u be the tree rooted at u and containg exactly all descendants of u, including u. Then the sum collapses to

$$|f_{e_T}^{(0)}| = \sum_{\substack{e \in E_0 \\ e \cap V(T_u) \neq \varnothing}} c_e = \sum_{w \in V(T_u)} c_{\{w\} \overset{E_0}{\leftrightarrow} V \setminus \{w\}}$$

which is exactly the value σ_u computed by the algorithm `EmbedTree`, and the value assigned to $|f^{(0)}_{e_T}|$. Hence `EmbedTree` correctly computes the embedding of G_0 into T.

With a simple inductive argument, one can then show that after the recursive calls of `EmbedTree`, the flow f returned by the algorithm is indeed the embedding flow of G into T.

For the number of I/Os required, observe that the Euler tour only needs to be constructed once (requiring $\mathcal{O}(\mathsf{sort}(m))$ I/Os), and can then be maintained in $\mathcal{O}(\mathsf{scan}(m))$ I/Os when splitting the tree. Given the Euler tour (with the first incoming and last outgoing arc of each vertex labelled), all operations except partitioning the edges E are easily implemented in $\mathcal{O}(\mathsf{scan}(m))$ I/Os. To accelerate the partitioning of the edges, we maintain E as an edge list sorted by incidence to the Euler tour (with each edge appearing only once in the list, for its endpoint that is earliest along the tour), and label each edge by the signed number of hops in the tour between its endpoints. Then when building the partition, we scan the edge list together with the subtree sizes produced by `FindSplitter`(G, T). Whenever the number of hops to the other endpoint of an edge is larger than the remaining number of hops in the subtree, we move the edge to E_0, else we keep it in E_i. Note that the sorted order of the edge list thereby remains intact for all E_i. This implies also that we can compute the values d_v and σ_v in $\mathcal{O}(\mathsf{scan}(m))$ using another scan of the Euler tour.

Hence after an initial $\mathcal{O}(\mathsf{sort}(m))$ I/Os to construct the tour and labelled edge list, each level of the recursion requires only $\mathcal{O}(\mathsf{scan}(m))$ I/Os. By the choice of v^*, the recursion has depth at most $\log_2(n+m)$, leading to an total of $\mathcal{O}(\log_2 m \cdot \mathsf{scan}(m))$ I/Os. □

Lemma 5.5 (Tree Splitters). *Consider any tree $T = (V, E_T)$ with arbitrary weights $\boldsymbol{w}$ on the vertices, and denote for any subtree T' by $\mathsf{size}(T') = \sum_{v \in V(T')} w_v$ the weight of the subtree T'. Then the tree T contains a splitter vertex $v^* \in V$ such that for all subtrees T_i induced by the removal of v^* from T, $\mathsf{size}(T_i) \leq \frac{1}{2}\mathsf{size}(T)$. Moreover, v^* can be found in $\mathcal{O}(\mathsf{sort}(m))$ I/Os.*

Proof. We first prove existence of a splitter for any tree T with arbitrary weights $\boldsymbol{w}$. Assume towards a contradiction that T contains no splitter, and consider the tree walk defined by the following function $\omega : V \to V$: Start the walk at any leaf. To take a step $\omega(u)$ from some vertex u, walk to the neighbouring vertex $\omega(u) = v$ that roots a subtree of maximum weight among all neighbours of u, breaking ties according to some arbitrary, predefined order. We claim that this process eventually enters a two-cycle (see Figure 5.3).

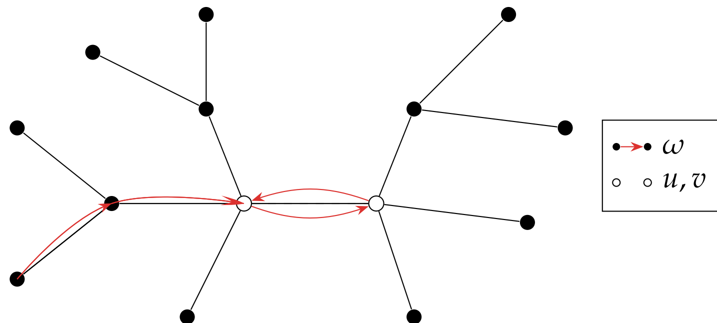

Figure 5.3: Tree walk entering a two-cycle.

Indeed, note that $\omega : V \to V$ is a function (i.e. $\omega(u)$ always takes the same value, irrespective of the history of the walk), and since V is finite, iterating ω must enter a cycle $v_1, \ldots, v_k, v_1$ of length $k \geq 2$, since $\omega(u) \neq u$ for all u. Moreover, ω being a function implies that in one revolution along the cycle, every vertex must be visited exactly once, and hence $v_1, \ldots, v_k, v_1$ is also a cycle in T. But T is acyclic, thus it must hold that $k = 2$.

Let then u, v be the vertices defining the cycle, let T_v be the subtree of u rooted at v, and likewise let T_u be the subtree of v rooted at u. Note that this ensures that T_u and T_v are disjoint, but when connected with $\{u, v\}$ make up the entire tree T. At the same time, since by the walking process, T_u and T_v are heaviest subtrees of v and u respectively, and since T by assumption contains no splitter, it must hold that $\mathsf{size}(T_u) > \frac{1}{2}\mathsf{size}(T)$ and $\mathsf{size}(T_v) > \frac{1}{2}\mathsf{size}(T)$. This is a contradiction, because it must also hold that $\mathsf{size}(T_u) + \mathsf{size}(T_v) = \mathsf{size}(T)$. Hence T must contain some splitter vertex v^*.

To find v^*, proceed as in `FindSplitter` from Algorithm 5.1, using the weights $\boldsymbol{w}$ instead of $1 + \deg_G(v)$. It is not hard to see that the algorithm correctly computes all subtree sizes by appropriately traversing the labelled arcs, and thus also correctly identifies a splitter v^*. Constructing the Euler tour takes $\mathcal{O}(\mathsf{sort}(m))$ I/Os, which dominates the cost of `EmbedTree`. □

Note that we can also use the flow $\boldsymbol{f} \leftarrow$ `EmbedTree`(G, T) to assign capacities $\mathbb{c}_e = f_e$ to the tree edges such that G one-embeds into T. This will be the main application of `EmbedTree` for the purpose of building congestion approximators; the example below illustrates the approach.

Sherman [She13] proves that a maximum spanning tree provides an m-congestion-approximator. We follow this, but incorporate the results developed above to pull everything together in an example.

Lemma 5.6 (Maximum Spanning Tree Congestion; [She13]). *Let $G = (V, E, \boldsymbol{c})$ be some undirected, capacitated graph, and let $T = (V, E_T)$ be a maximum spanning tree in G, with capacities $\mathbb{c}$ given by $\mathbb{c} \leftarrow$ `EmbedTree`(G, T).*

Then the matrix $\boldsymbol{R} \in \mathbb{R}^{(n-1) \times n}$ with one row for each tree edge $e_T \in E_T$ given by

$$(\boldsymbol{R})_{e_T, v} = \begin{cases} \frac{1}{\mathbb{c}_{e_T}} & \text{if } v \in \mathcal{X}_T(e_T) \\ 0 & \text{otherwise} \end{cases}$$

is an m-congestion-approximator.

Proof. We prove that a maximum spanning tree T with capacities $\mathbb{c} \leftarrow$ `EmbedTree`(G, T) m-embeds into G and then rely on the proof of Lemma 5.2 given below.

First, note that by construction of $\mathbb{c}$, G one-embeds into T. At the same time, for any tree edge $e_T \in E_T$, $\mathbb{c}_{e_T} = \sum_{e \in (\mathcal{X}_T(e_T) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_T(e_T))} c_e \leq m c_{e_T}$ because T is a maximum spanning tree. In particular, the identity embedding flow of T into G has congestion at most m. Thus T m-embeds into G by Lemma 5.3, the lemma then follows from Lemma 5.2, proved below. □

Remark. One can tighten the bound to $m - n + 2$ by noting that the graph must be connected, hence for every tree edge e_T and induced cut $\mathcal{X}_T(e_T) \leftrightarrow V \setminus \mathcal{X}_T(e_T)$, at least $n - 2$ edges must reside entirely within $\mathcal{X}_T(e_T)$ or $V \setminus \mathcal{X}_T(e_T)$, hence at most $m - n + 2$ edges can cross the cut. ◇

Let us finally prove Lemma 5.2 from the previous section to show how we will construct congestion approximation operators $\boldsymbol{R}$ from arbitrary $(\alpha, \tilde{\mathbb{T}})$-decompositions.

Proof of Lemma 5.2; [She13]. We show how to construct an α-congestion-approximator $\boldsymbol{R}$ for an arbitrary $(\alpha, \tilde{\mathbb{T}}_V)$-decomposition $\{(\lambda^{(i)}, T^{(i)})\}_i$ of some graph $G = (V, E, \boldsymbol{c})$, where we recall that $\tilde{\mathbb{T}}_V$ is the set of trees on V. Write $T^{(i)} = (V, E_T^{(i)})$ and let $\mathbb{c}^{(i)}$ be the capacity vector of $T^{(i)}$. Now define $\boldsymbol{R}$ to be the matrix containing one row for each edge $e \in E_T^{(i)}$ of each tree $T^{(i)}$ given as in Lemma 5.6 by

$$(\boldsymbol{R})_{e,v} = \begin{cases} \frac{1}{\mathbb{c}_e^{(i)}} & \text{if } v \in \mathcal{X}_{T^{(i)}}(e) \\ 0 & \text{otherwise} \end{cases}$$

We claim that $\boldsymbol{R}$ is an α-congestion-approximator. Indeed, consider any tree $T^{(i)}$ and any edge $e \in E_T^{(i)}$ of this tree, with the corresponding induced cut $\mathcal{X}_{T^{(i)}}(e)$. Then because G one-embeds into every $T^{(i)}$,

$$|(\boldsymbol{R}\boldsymbol{b})_e| = \frac{b_{\mathcal{X}_{T^{(i)}}(e)}}{\mathbb{c}^{(i)}_{\mathcal{X}_{T^{(i)}}(e) \overset{E_T^{(i)}}{\leftrightarrow} V \setminus \mathcal{X}_{T^{(i)}}(e)}} \leq \frac{b_{\mathcal{X}_{T^{(i)}}(e)}}{c_{\mathcal{X}_{T^{(i)}}(e) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_{T^{(i)}}(e)}} \leq \mathsf{opt}_G(\boldsymbol{b})$$

where we use Lemma 2.2 in the last step. In particular, $\|\boldsymbol{R}\boldsymbol{b}\|_\infty \leq \mathsf{opt}_G(\boldsymbol{b})$.

To show $\alpha\|\boldsymbol{R}\boldsymbol{b}\|_\infty \geq \mathsf{opt}_G(\boldsymbol{b})$, consider the graph $G_R = (V, E_R, \mathbb{c})$ of Lemma 5.1. We route $\boldsymbol{b}$ in each tree $T^{(i)}$ as $\boldsymbol{f}^{(i)}$ and then use the flow $\boldsymbol{f} = \sum_i \lambda^{(i)} \boldsymbol{f}^{(i)}$ to route $\boldsymbol{b}$ in G_R. This incurs a congestion on any edge $e \in E_R$ of

$$\mathsf{cong}_f^{\mathbb{c}}(e) = \left| \frac{\sum_i \lambda^{(i)} f_e^{(i)}}{\sum_i \lambda^{(i)} \mathbb{c}_e^{(i)}} \right|$$

If we interpret the $\lambda^{(i)}$ as a probability distribution on the $T^{(i)}$, then this is equivalent to stating that the congestion on $e \in E_R$ is at most the expected flow on e, divided by the expected capacity of e. As in Lemma 2.5, there must exist a joint outcome of both random variables, i.e. a single tree $T^{(i)}$, that attains

$$\mathsf{cong}_f^{\mathbb{c}}(e) \leq \frac{f_e^{(i)}}{\mathbb{c}_e^{(i)}} = \mathsf{cong}_{f^{(i)}}^{\mathbb{c}^{(i)}}(e) \leq \|\boldsymbol{R}\boldsymbol{b}\|_\infty$$

since $\boldsymbol{R}$ accounts for the congestion on all edges of all trees. By Lemma 5.1, G_R α-embeds into G, and hence if S is a maximally-congested cut of G in the sense of Theorem 2.1, then

$$\alpha\|\boldsymbol{R}\boldsymbol{b}\|_\infty \geq \frac{\alpha b_S}{\mathbb{c}_{S \overset{E_R}{\leftrightarrow} V \setminus S}} \geq \frac{\alpha b_S}{\alpha c_{S \overset{E}{\leftrightarrow} V \setminus S}} = \mathsf{opt}_G(\boldsymbol{b})$$

□

5.2 Using Multiple Trees

Can we do better than an m-congestion-approximator when using trees? For a single tree, it unfortunately turns out that the answer is 'not much':

Lemma 5.7 (Lower Bound for Tree Embedding). *For any n and $m \geq \Omega(n)$, there exists a unit-weighted n-vertex and m-edge graph $G(n,m)$ such that for any spanning tree T of $G(n,m)$ with capacities $\mathbb{c} \leftarrow \texttt{EmbedTree}(G(n,m),T)$, the identity embedding flow of T back into $G(n,m)$ has maximum congestion at least $\Omega(\sqrt{m})$.*

Proof. Consider first the complete graph K_q on q vertices, and let T_q be a tree in K_q. Any leaf of T_q is incident to $q-1$ edges of K_q, but only one edge of T_q, hence this edge incurs identity embedding congestion $q-1 \geq \Omega(q)$.

Now let $G(n,m)$ be some graph containing K_q as a subgraph, where we fix q to be as large as possible while ensuring $\binom{q}{2} + (n-q) \leq m$ such that $G(n,m)$ has no more than m edges. One may compute $q = \Theta(\sqrt{m-n}) = \Theta(\sqrt{m})$. Any spanning tree of $G(n,m)$ must also contain a spanning tree of K_q, and thereby must contain an edge of identity embedding congestion at least $\Omega(\sqrt{m})$. □

The lemma implies that when constructing spanning trees T such that G one-embeds into T, using construction of $\boldsymbol{R}$ given above cannot be sufficient to yield a tight congestion approximator.

This finally motivates the use of a convex combination in Definition 5.3 containing more than a single spanning tree to approximate congestions in arbitrary graphs. This section is dedicated to presenting a direct decomposition into spanning trees $\mathbb{T}_G$, which is the first step of Madry's decomposition for approximating congestion of arbitrary graphs. It will then become clear why even this does not suffice, with the further steps of the decomposition then presented in Section 5.3 and Section 5.4.

Similar to how low-stretch spanning trees in some graphs always contain an edge of high stretch, but admit low *average* stretch, we can hope to achieve low *average* embedding congestion for our congestion-approximating trees, and then combine multiple trees together in a convex combination so that the embedding congestion of the entire combination approaches the average congestion of its members. Note that this is exactly the goal of the flow packing procedure from Section 2.4: If the trees $T^{(i)}$ are *spanning* trees of G, then the embedding flow of G into the $T^{(i)}$ is also a flow in G, and using flow packing on these flows corresponds to building a convex combination of trees $T^{(i)}$ such that the weighted combination has overall low embedding congestion.

In somewhat greater detail, the idea is to use the $(\mathbb{F}, G)$-system *induced* by the *identity embedding flows* of the spanning trees $\mathbb{T}_G$ into G. An *identity embedding flow* of a subgraph $H = (V, E_H, \mathbb{c})$ into $G = (V, E, \boldsymbol{c}), E_H \subseteq E$ is simply the embedding flow that routes the flow required for each $e \in E_H$ on the edge itself, i.e. $|f_e| = \mathbb{c}_e$. Using the flow packing from Theorem 2.2 for these identity embedding flows produces exactly a decomposition of G into trees:

Lemma 5.8. *Given a β-oracle for the $(\mathbb{F}, G)$-system induced by the identity embedding flows of the spanning trees $\mathbb{T}_G$ of G into G, running the flow packing algorithm from Theorem 2.2 and collecting the trees produced by the oracle as $\{(\lambda^{(i)}, T^{(i)})\}$ yields a $(\beta \cdot (1+3\delta), \mathbb{T}_G)$-decomposition of G.*

Proof. Construct the graph $G_R = (V, E_R, \mathbb{c})$ of the convex combination of trees as in Lemma 5.1, and let $\boldsymbol{f}^{(i)}$ be the identity embedding flow of $T^{(i)}$ into G, assuming all common edges of the $T^{(i)}$ are oriented equally. Then $\boldsymbol{f} = \sum_i \lambda^{(i)} \boldsymbol{f}^{(i)}$ is the identity embedding flow of G_R into G, and for any edge $e \in E$, since all flows $\boldsymbol{f}^{(i)}$ have the

same sign on e,

$$\mathrm{cong}_f(e) = \frac{1}{c_e}\left|\sum_i \lambda^{(i)} f_e^{(i)}\right| = \sum_i \lambda^{(i)} \mathrm{cong}_{f^{(i)}}(e) \leq \beta \cdot (1+3\delta)$$

by Theorem 2.2. Hence by Lemma 5.3, G_R $(\beta \cdot (1+3\delta))$-embeds into G, which with Lemma 5.1 implies that the convex combination of trees is a $(\beta \cdot (1+3\delta), \mathbb{T}_G)$-decomposition of G. □

To solve the (relaxed) system with the approach from Theorem 2.2, we need to define an oracle for querying this identity-embedding $(\mathbb{F}, G)$-system with edge weights $\boldsymbol{w}$, which is where the connection to low-stretch spanning trees will be made. Formally, we prove the following lemma:

Lemma 5.9 ([Mad11]). *Given (i) an undirected and capacitated graph $G = (V, E, \boldsymbol{c})$, (ii) an edge weight vector $\boldsymbol{w}$, and (iii) an algorithm that, for any edge lengths, computes a spanning tree of average stretch at most β on a multigraph with at most n^2 edges, there exists an algorithm that finds a spanning tree $T \in \mathbb{T}_G$ such that G 1-embeds into T and the identity embedding flow $\boldsymbol{f}$ of T into G satisfies $\sum_{e\in E} w_e \mathrm{cong}_f(e) \leq 2\beta\|\boldsymbol{w}\|_1$.*

Proof. Let T be some for now arbitrary spanning tree in G with capacities $\mathbb{c} \leftarrow$ `EmbedTree`(G, T) such that G 1-embeds into T. Note that for any tree edge $e_T \in E_T$

$$\mathbb{c}_{e_T} = c_{\mathcal{X}_T(e_T) \overset{E}{\leftrightarrow} V\setminus\mathcal{X}_T(e_T)} = \sum_{\substack{e\in E \\ e_T \in \mathrm{path}_T(e)}} c_e$$

Hence the identity embedding flow $\boldsymbol{f}$ of T into G produces *weighted* congestion

$$\sum_{e\in E} w_e \mathrm{cong}_f(e) = \sum_{e_T\in E_T} w_{e_T} \mathrm{cong}_f(e_T) = \sum_{e_T\in E_T} w_{e_T} \frac{\mathbb{c}_{e_T}}{c_{e_T}} = \sum_{e_T\in E_T} \frac{w_{e_T}}{c_{e_T}} \sum_{\substack{e\in E \\ e_T\in\mathrm{path}_T(e)}} c_e =$$

$$= \sum_{e\in E} c_e \sum_{e_T\in\mathrm{path}_T(e)} \frac{w_{e_T}}{c_{e_T}} = \sum_{e\in E} w_e \mathrm{stretch}_T^{\ell}(e) \overset{!}{\leq} 2\beta \sum_{e\in E} w_e$$

where we set $\ell_e = \frac{w_e}{c_e}$ and we recall the definition of stretch from Definition 4.1. Thus the problem of finding an oracle for a flow packing of the $\mathbb{T}_G$ identity embedding flows is equivalent to finding a tree of low total stretch w.r.t. edge lengths $\ell_e = \frac{w_e}{c_e}$, but additionally weighted by edge weights w_e. Madry [Mad11] follows Alon et al. [Alo+95] in reducing this weighted problem to the usual formulation of the low-stretch tree problem by duplicating some edges to yield a multigraph.

Let $\bar{G} = (V, \bar{E})$ be the multigraph containing $d_e = 1 + \lfloor \frac{m w_e}{\|\boldsymbol{w}\|_1} \rfloor$ copies of every edge e. The total number of edges increases only by a constant factor:

$$|\bar{E}| = \sum_{e\in E} 1 + \left\lfloor \frac{w_e}{\|\boldsymbol{w}\|_1} \right\rfloor \leq m + \frac{m}{\|\boldsymbol{w}\|_1} \sum_{e\in E} w_e = 2m \leq n^2$$

where the last inequality follows from the fact that G is a simple graph. Now let T be some spanning tree of $\bar{G}$ with total stretch $\beta|\bar{E}|$ w.r.t. edge lengths $\boldsymbol{\ell}$. Hence

$$\sum_{\bar{e}\in\bar{E}} \mathrm{stretch}_T^{\ell}(\bar{e}) = \sum_{e\in E} d_e \mathrm{stretch}_T^{\ell}(e) \leq \beta|\bar{E}|$$

Lower-bounding d_e yields

$$d_e \geq \frac{mw_e}{\|\boldsymbol{w}\|_1} = \frac{2mw_e}{2\|\boldsymbol{w}\|_1} \geq w_e \frac{|\bar{E}|}{2\|\boldsymbol{w}\|_1} \implies \sum_{e \in E} w_e \mathsf{stretch}_T^{\ell}(e) \leq 2\beta \|\boldsymbol{w}\|_1$$

and hence a spanning tree T of $\bar{G}$ with average stretch β on edge lengths $\boldsymbol{\ell}$ satisfies $\sum_{e \in E} w_e \mathsf{cong}_f(e) \leq 2\beta \|\boldsymbol{w}\|_1$, where $\boldsymbol{f}$ is the identity embedding flow of T into G. □

Remark. Recall that both of our low-stretch tree algorithms require the edge length ratio $U^{\ell} := \max_e \ell_e / \min_e \ell_e$ to be polynomially bounded. If we refer back to the proof of Lemma 2.6, we observe that w_e can be as large as $\exp(\mathcal{O}(\beta))$, while the smallest edge weights might forever remain at 1. Since $\beta \gg \Omega(\log n)$ for both algorithms, the length ratio might not be polynomially bounded. We resolve this in Lemma 5.12, somewhat paradoxically by *increasing* the lengths by some additive factor proportional to $\|\boldsymbol{w}\|_1$.◇

We refer to the process of building a convex combination of trees using the oracle defined above as *tree packing*. Recall from Theorem 2.2 that the number of packing iterations, and hence the number of calls to the low-stretch tree algorithm, depends on the *tightness* of the oracle defined in Definition 2.4 to be the number of edges with congestion at least one-half the maximum congestion. Even if all tree edges satisfies this criterion, all $m - n + 1$ non-tree edges of G do not (since their congestion is zero), hence the tightness κ for any tree packing is bounded by $n - 1$. Recall that the number of iterations of flow packing, and hence the number of calls to the low-stretch tree algorithm, scales roughly with $\frac{m}{\kappa n}$. This only becomes sublinear after sparsification of G (see Chapter 3).

Is sparsification enough to yield a sublinear number of calls to the LSST algorithm? Unfortunately, the answer is no:

Lemma 5.10 (Tree Packing Oracle Tightness)**.** *For any $m \geq \Omega(n)$, there exist m-edge graphs on which the packing oracle defined in Lemma 5.9 has tightness at most $\mathcal{O}(\sqrt{m}\beta)$.*

Proof. Assume the oracle has tightness k. Use $\boldsymbol{w} = \mathbf{1}$ in Lemma 5.9 to compute

$$\frac{1}{2}k \cdot \max_e \mathsf{cong}_f(e) \leq \sum_e \mathsf{cong}_f(e) \leq 2\beta m \implies \beta \geq \frac{k}{4m} \cdot \max_e \mathsf{cong}_f(e)$$

Lemma 5.7 shows that there exist graphs on n vertices and m edges such that $\max_e \mathsf{cong}_f(e) \geq \Omega(\sqrt{m})$.Plugging this into the equation above yields

$$\beta \geq \frac{k}{4m}\Omega(m - n) \geq \Omega(k) \implies k \leq \mathcal{O}(\beta)$$

□

In the absence of possibly profound structural restrictions on the sparsified graphs that we perform the flow packing on, we would need at least (c.f. Theorem 2.2)

$$\frac{4\beta(1+3\delta)m \log m}{k\delta^2} \geq \frac{4(1+3\delta)\sqrt{m}\log m}{\delta^2} \geq \Omega(\sqrt{m}\log m)$$

calls to the flow packing oracle, and hence also to the low-stretch spanning tree algorithm. For $m \leq n\,\mathsf{polylog}(n)$, this is still on the order of $\sqrt{n}$. With each call to the LSST taking at least $\Omega(\mathsf{scan}(n))$ I/Os, we could only hope to achieve an algorithm of $\Omega(\mathsf{scan}(n^{1.5}))$ I/Os – far from being nearly linear.

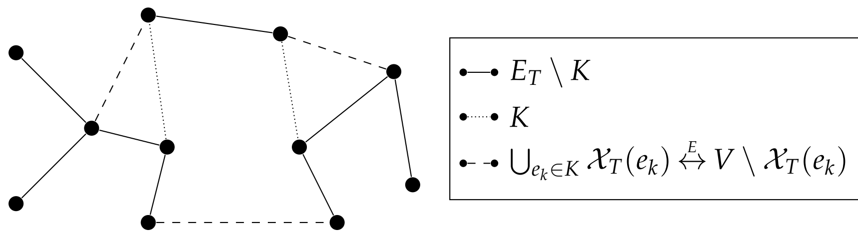

Figure 5.4: j-sliced Tree

5.3 Slicing Trees

To remedy this, Madry proposes to 'slice' the trees into smaller subtrees at the heavily-congested edges by removing these edges e_T from the tree T and instead replacing them with all cross-cut edges $\mathcal{X}_T(e_T) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_T(e_T)$. This ostensibly sacrifices the easy routability of trees, but we will recover a (recursive) decomposition into trees later in Section 5.4. We define a *sliced tree* as follows:

Definition 5.7 (j-sliced Tree; [Mad11])**.** Let $G = (V, E)$ be some graph and let $T = (V, E_T)$ be a spanning tree on G. Let $K \subseteq E_T$ with $|K| \leq j$ be some subset of the tree edges. The *j-sliced tree* $H_G(T, K) = (V, E_H)$ is the graph on the vertices V connected by the edges (c.f. Figure 5.4)

$$E_H = (E_T \setminus K) \cup \left(\bigcup_{e_k \in K} \left(\mathcal{X}_T(e_T) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_T(e_T) \right) \right)$$

i.e. $H_G(T, K)$ contains parts $E_T \setminus K$ of the tree, together with all edges that cross the cut induced by a tree edge from K.

The set of j-sliced trees in G is denoted by $\mathbb{H}_G[j]$. ◇

Slicing trees along heavily congested edges gives us the necessary freedom to achieve low tightness. To illustrate the approach, consider the subset of tree edges with congestion at least one-half the maximum congestion $\kappa := \{e \in E \mid \mathsf{cong}_f(e) \geq \frac{1}{2} \max_{e'} \mathsf{cong}_f(e')\}$. We want to grow this set by adding some edges from T to the slicing set K to produce the graph $H_G(T, K)$. The only tool we are given for this is to reduce the term for the maximum congestion by moving the maximally congested edges to K, at the expense of making the resulting graph more difficult to work with (if we move all edges to K, we are left with the original graph G, which is not a useful decomposition). The trick is that if κ is small, e.g. contains only a single edge, then by moving only these edges to K, the maximum congestion decreases by a factor of at least one-half, whereas if κ is large, then we already have large tightness and are done.

Iterating on this, we can partition the edges into buckets with congestion at least 2^{-i} times the maximum congestion, and we want to find a bucket containing many edges (the number of edges defines the tightness of the oracle), but such that the number of edges in all higher-congested buckets is small (these are the edges moved to K). What remains to be done is to analyse this formally, and to balance the size of K with the tightness of the oracle.

Lemma 5.11 ([Mad11])**.** *Let β be as in Lemma 5.9. There exists a $(2\beta + 1)$-oracle with tightness $1 \leq k \leq n - 1$ for the flow system induced by the identity embedding flows of $\mathbb{H}_G\left[(k-1)\lfloor \log_2 mU \rfloor\right]$, where $U = \max_e c_e / \min_e c_e$ is the capacity ratio of the graph G.*

Proof. Let $T = (V, E_T)$ be the tree as produced by Lemma 5.9. For the tree edges $E_T \setminus K$, use the capacities $\mathbb{c}$ as produced by `EmbedTree`(G, T), and for all other edges of $H_G(T, K)$, use the original capacities $\boldsymbol{c}$. Now begin by proving that for any choice of K, $H_G(T, K)$ satisfies $\sum_{e \in E} w_e \mathsf{cong}_f(e) \leq (2\beta + 1)\|\boldsymbol{w}\|_1$, where $\boldsymbol{f}$ is again the identity embedding flow of $H_G(T, K)$ into G. This part is a simple calculation:

$$\sum_{e \in E} w_e \mathsf{cong}_f(e) = \sum_{e \in E_T \setminus K} w_e \mathsf{cong}_f(e) + \sum_{e \in E \setminus E_T} w_e \underbrace{\mathsf{cong}_f(e)}_{\leq 1} \leq 2\beta\|\boldsymbol{w}\|_1 + \|\boldsymbol{w}\|_1 = (2\beta + 1)\|\boldsymbol{w}\|_1$$

Proceed by constructing the set K as outlined above. Observe that $\max_e \mathsf{cong}_f(e) \leq mU$, which would be the congestion incurred at the least-capacity edge if all other edges are of maximum capacity and routed over this edge. Moreover, for any tree edge $e_T \in E_T$, $\mathsf{cong}_f(e_T) \geq 1$ because the edge capacities in T are always at least as large as those of G. This implies that the tree edges can be fully split into buckets $E_i = \{e_T \in E_T \mid 2^{-(i+1)}mU < \mathsf{cong}_f(e_T) \leq 2^{-i}mU\}$ for $i = 0, \ldots, \lfloor \log_2 mU \rfloor$. Let E_j be the bucket with the smallest j that satisfies $|E_j| \geq k$, or let $j = \lfloor \log mU \rfloor$ if no such bucket exists.

By moving the first $j - 1$ buckets to K, $\max_e \mathsf{cong}_f(e)$ reduces to at most $2^{-j}mU$ (where $\boldsymbol{f}$ is now the embedding flow of this $H_G(T, K)$ to G) because all edges of $H_G(T, K)$ that are not in $E_T \setminus K$ now have congestion exactly 1. Thus after this, $|E_j|$ is exactly the tightness of the oracle. The number of edges in K is $|K| = \sum_{i=0}^{j-1} |E_i| \leq j(k-1) \leq (k-1)\lfloor \log_2 mU \rfloor$ because by construction of j, none of these buckets contains more than $k - 1$ edges, and $j \leq \lfloor \log_2 mU \rfloor$.

If $|E_j| \geq k$, then the tightness holds immediately by the above construction. If however no such bucket was found, then $j = \lfloor \log_2 mU \rfloor$ and hence the maximum congestion after constructing $H_G(T, K)$ is 2, thus the oracle has tightness $n - 1 \geq k$ regardless. □

The dependency on the capacity ratio U is not hugely important, because the external memory model assumes that the edge capacities have constant encoding size and thus $U \in \mathcal{O}(1)$. However, we must still resolve the conundrum remarked earlier: We also require the length ratio $U^\ell := \max_e \ell_e / \min_e \ell_e$ to be polynomially bounded for the low-stretch spanning tree computations. It turns out that both problems can be resolved with a modification of the edge lengths due to Sherman [She13]. First, note that U in the previous lemma was only used to bound the maximum congestion on any edge. Shifting the edge lengths will let us ensure that the congestion is always at most $\tilde{\mathcal{O}}(m)$, while at the same time increasing the minimum edge length $\min_e \ell_e$ such that U^ℓ becomes polynomially bounded.

To see this, observe that the maximum congestion of T from Lemma 5.9 is in fact $2\beta\|\boldsymbol{w}\|_1/b$ if $w_e \geq b$ for all e, because then $b \max_e \mathsf{cong}_f(e) \leq \sum_{e \in E} w_e \mathsf{cong}_f(e) \leq 2\beta\|\boldsymbol{w}\|_1$. Shifting weights such that b becomes proportional to $\|\boldsymbol{w}\|_1$ thus decreases the maximum edge congestion drastically, while also increasing $\min_e \ell_e$ enough to guarantee $U^\ell \leq \mathsf{poly}(n)$. The following lemma makes this precise:

Lemma 5.12 ([She13; Mad11])**.** *Let β be as in Lemma 5.9. There exists a $(4\beta + 1)$-oracle with tightness $1 \leq k \leq n - 1$ for the flow system induced by the identity embedding flows of $\mathbb{H}_G\left[(k-1)\lfloor \log_2(4\beta m) \rfloor\right]$. Moreover, if the capacities of G are polynomially bounded, then the length ration $U^\ell = \max_e \ell_e / \min_e \ell_e$, where $\boldsymbol{\ell}$ are the lengths of the graph passed to the low-stretch tree algorithm, is also polynomially bounded.*

Proof. Modify the calculation from Lemma 5.9 to use the length function $\ell'_e = \frac{w_e + a}{c_e}$ (for some a fixed below) in the construction of the low stretch spanning tree:

$$\sum_{e \in E} w_e \mathsf{cong}_f(e) \le \sum_{e \in E} (w_e + a)\,\mathsf{cong}_f(e) \le \cdots \le 2\beta\left(\|\boldsymbol{w}\|_1 + am\right)$$

and hence

$$\max_{e \in E} \mathsf{cong}_f(e) \le \sum_{e \in E} \mathsf{cong}_f(e) \le 2\beta\left(\frac{\|\boldsymbol{w}\|_1}{a} + m\right)$$

Choose $a = \frac{\|\boldsymbol{w}\|_1}{m}$ to arrive at $\sum_{e \in E} w_e \mathsf{cong}_f(e) \le 4\beta\|\boldsymbol{w}\|_1$ and $\max_e \mathsf{cong}_f(e) \le 4\beta m$. In other words, the modified length function is a 4β-oracle for $\mathbb{H}_G[0]$, and the resulting tree has maximum congestion at most $4\beta m$. Using this bound instead of mU in Lemma 5.11 results in a $(4\beta + 1)$-oracle with tightness k for the embedding flows of $\mathbb{H}_G\left[(k-1)\lfloor \log_2(4\beta m) \rfloor\right]$.

Finally, observe that $\min_e \ell_e \ge \min_e \frac{\|\boldsymbol{w}\|_1}{mc_e} \ge \frac{\|\boldsymbol{w}\|_1}{\mathsf{poly}(n)}$ and $\max_e \ell_e \le \max_e w_e + a \le (1 + 1/m)\|\boldsymbol{w}\|_1$. Thus $U^\ell \le (1 + 1/m)\,\mathsf{poly}(n) \le \mathsf{poly}(n)$. □

The final flow-packing oracle algorithm is given in pseudocode form in Algorithm 5.2.

Algorithm 5.2 A Flow-Packing Oracle for the $(\alpha, \mathbb{H}_G)$-decomposition

1: **procedure** `SlicedTreeOracle`$^k(G, \boldsymbol{w})$
2: Let $\ell'_e = \frac{w_e}{c_e} + \frac{\|\boldsymbol{w}\|_1}{mc_e}$ for every $e \in E$
3: Let $\bar{G} = (V, \bar{E})$ be the multigraph s.t. $\bar{E}$ has $1 + \lfloor \frac{mw_e}{\|\boldsymbol{w}\|_1} \rfloor$ copies of all edges $e \in E$
4: Compute a spanning tree T of $\bar{G}$ of average stretch at most β w.r.t. $\boldsymbol{\ell}'$
5: Compute the embedding capacities $\mathbb{c}$ using `EmbedTree`(G, T)
6: Sort the edges e of T by their embedding congestion $\mathbb{c}_e / c_e$ in descending order
7: Split this list into buckets $E_i = \{e \in E \mid 4\beta m \cdot 2^{-(i+1)} < \mathbb{c}_e / c_e < 4\beta m \cdot 2^{-i}\}$
8: Find the smallest j such that $|E_i| \ge k$, or let $j = \lfloor \log_2 4\beta m \rfloor$ otherwise
9: Let $K = \bigcup_{i=0}^{j-1} E_i$ and construct $H_G(T, K)$
10: **return** the identity embedding flow of $H_G(T, K)$ into G
11: **procedure** `SlicedTreeDecompose`$^t(G)$
12: Let $k = \lfloor 2^8 \beta t^{-1} m \log m \rfloor$
13: Run `RelaxedFlowPacking`$(G, \delta = 1/2)$ with the oracle `SlicedTreeOracle`^{k}
14: **return** the convex combination $\{\lambda^{(i)}, H_G^{(i)}(T, K)\}$

Combining the oracle from Algorithm 5.2 with Theorem 2.2 yields the following summarising theorem:

Theorem 5.1 ([She13; Mad11]). *Assume there exists an algorithm that computes, for any multigraph of at most n^2 edges, a spanning tree of average stretch at most β. Then there exists an algorithm `SlicedTreeDecompose`^{t} that for any undirected, capacitated graph $G = (V, E, \boldsymbol{c})$ and $\Omega\left(\frac{\beta m \log m}{n}\right) \le t \le \mathcal{O}(\beta m \log m)$, computes an $\left(\mathcal{O}(\beta), \mathbb{H}_G\left[\mathcal{O}(\frac{\beta m \log^2 m}{t})\right]\right)$-decomposition of G containing at most t many sliced trees.*

`SlicedTreeDecompose`^{t} makes $\mathcal{O}(t)$ calls to the low-stretch tree oracle, and otherwise requires $\mathcal{O}(t \log m\, \mathsf{scan}(m))$ I/Os.

Proof. We prove the theorem with explicit constants. By Theorem 2.2, the number of trees in the decomposition for an oracle of tightness k is

$$t = \frac{4(4\beta+1)(1+3\delta)}{k\delta^2} m \log m \qquad 0 < \delta \leq 1/2$$

Choose e.g. $\delta = 1/2$, then $t \leq 2^8 \beta k^{-1} m \log m \implies k \leq 2^8 \beta t^{-1} m \log m$. Since we require $1 \leq k \leq n-1$, this also constrains

$$\frac{2^8}{n-1} \beta m \log m \leq t \leq 2^8 \beta m \log m$$

Now invoke Algorithm 2.2 with $\texttt{SlicedTreeOracle}^k$ for $k = \lfloor 2^8 \beta t^{-1} m \log m \rfloor$, adding every tree constructed by the oracle to the decomposition along with the coefficient $\lambda^{(i)}$ used by the flow packing procedure. In every iteration, the flow packing procedure adds a new sliced tree to the decomposition, hence the decomposition will contain at most t many sliced trees. By Lemma 5.12, the number of edges on which these trees are sliced is at most

$$(k-1) \cdot \lfloor \log_2(4\beta m) \rfloor \leq 2^8 \beta t^{-1} m \log m \log_2(4\beta m) \leq \mathcal{O}\left(\frac{\beta m \log^2 m}{t}\right)$$

Due to Theorem 2.2, the cut approximation ratio of the final decomposition is at most

$$(4\beta+1)(1+3\delta) \leq 14\beta + 7/2 \leq \mathcal{O}(\beta)$$

The cost of $\texttt{SlicedTreeOracle}^k$ is dominated by the call to `EmbedTree` (which takes $\mathcal{O}(\log_2 m \, \mathsf{scan}(m))$ I/Os), in addition to the construction of a low stretch spanning tree. Hence finding the entire decomposition requires at most t calls to the low-stretch spanning tree algorithm, in addition to the $\mathcal{O}(t \log m \, \mathsf{scan}(m))$ I/Os required on top of this by the calls to $\texttt{SlicedTreeOracle}^k$. □

As the statement of the theorem indicates, we will be interested in keeping the number of sliced trees produced by the decomposition small. Another gadget available to us towards this goal is the following 'sparsification' result, used implicitly by Sherman and formalised here:

Lemma 5.13. *If $\{\lambda^{(i)}, G^{(i)}\}_{i\in[t]}$ is an $(\alpha, \mathbb{G})$-decomposition of G, then for any $\delta \in (0,1)$, removing the δt graphs of smallest coefficient $\lambda^{(i)}$ and scaling up the remaining $\lambda^{(i)}$ to sum to one yields a $(\frac{\alpha}{1-\delta}, \mathbb{G})$-decomposition of G.*

Proof. Write $\sum_{i=1}^{\delta t} \lambda^{(i)}$ for the sum over the coefficients of the δt smallest $\lambda^{(i)}$. Assume towards a contradiction that $\sum_{i=1}^{\delta t} \lambda^{(i)} > \delta$. On average, the coefficients in the sum have value $\frac{1}{\delta t} \sum_{i=1}^{\delta t} \lambda^{(i)} > 1/t$, hence there must exist some $\lambda^{(i)}$ among the δt smallest coefficient of value strictly greater than $1/t$. But then the remaining $(1-\delta)t$ coefficients must sum to strictly more than $(1-\delta)t\frac{1}{t} = 1-\delta$, which together with the first δt smallest coefficients is more than one, arriving at a contradiction.

Thus the δt smallest coefficients can only sum to at most δ, and the remaining coefficients must sum to at least $1-\delta$. Hence we scale the $\lambda^{(i)}$ of the remaining graphs by at most $\frac{1}{1-\delta}$ and the lemma follows. □

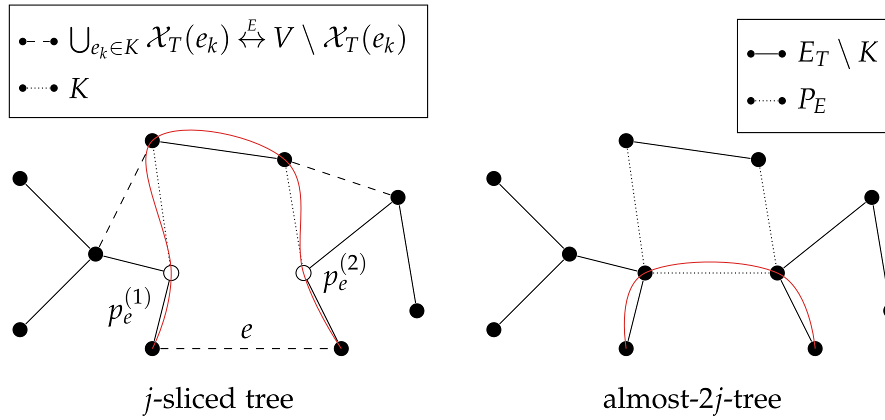

Figure 5.5: Construction of an almost-2j-tree (right) from a j-sliced tree (left), with $\mathsf{path}_T(e)$ and the corresponding portal vertices $p_e^{(1)}, p_e^{(2)}$ indicated for a cut edge on the left, and the corresponding embedding flow of e from $H_G(T, K)$ into $I_G(T, K)$ on the right.

5.4 From Sliced Trees to j-Trees

j-sliced trees have a well-defined structure, but it is still not evident how to efficiently compute optimal routings in such sliced trees. Even then, directly routing in j-sliced trees would still run into the non-linearity of routing discussed in the preamble to Lemma 5.2, and so we seek to recover a decomposition into trees $\tilde{\mathbb{T}}_V$ on the vertices V of G.

The mechanism that accomplishes this is an algorithm to one-embed the j-sliced trees into *j-trees*, which are graphs consisting of j many disjoint trees connected through an arbitrary j-vertex graph (recall Figure 1.2). This will be a two-step process: First, every j-sliced tree is one-embedded into an *almost-2j-tree*, which is the union of a single tree and an arbitrary 2j-vertex graph. As a second step, these almost-2j-trees are then embedded into $\mathcal{O}(j)$-trees.

Lemma 5.14 ([Mad11]). *There exists an algorithm* `AlmostTreeify` *takes a j-sliced tree $H_G(T, K)$ and finds an almost-2j-tree $I_G(T, K)$ that is three-embeddable into $H_G(T, K)$, and such that $H_G(T, K)$ is one-embeddable into $I_G(T, K)$.* `AlmostTreeify` *requires $\mathcal{O}(\log m\, \mathsf{scan}(m))$ I/Os.*

Proof. Write $G = (V, E, \mathbf{c})$, $T = (V, E_T)$, and let $\mathbb{c}$ be the capacities of $H_G(T, K)$. Recall that $\mathcal{X}_T(e_k) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_T(e_k)$ denotes the set of edges from E that cross the cut $\mathcal{X}_T(e_k)$ induced by an edge $e_k \in K \subseteq E_T$. For any $e \in \mathcal{X}_T(e_T) \leftrightarrow V \setminus \mathcal{X}_T(e_T)$ and any $e_k \in K$, let $p_e^{(1)}, p_2^{(2)} \in V$ be the first and last vertices on the path $\mathsf{path}_T(e)$ in T between the endpoints of e that are incident to an edge in K. Refer to these vertices *portals* between the slices crossed by the cut edge $e \in \mathcal{X}_T(e_k) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_T(e_T)$, and call $\{p_e^{(1)}, p_e^{(2)}\} = e_p$ the *portal edge* of e. Let P_E be the set of portal edges, i.e.

$$P_E = \left\{ \left\{ p_e^{(1)}, p_e^{(2)} \right\} \middle| \; e \in \left(\mathcal{X}_T(e_k) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_T(e_k) \right) \text{ for some } e_k \in K \right\}$$

Likewise, denote by P_V the set of all portal vertices, i.e. the set of endpoints of the edges in P_E. Note that every portal vertex in P_V is incident to an edge in K, hence there can be at most $2|K| \leq 2j$ vertices in P_V.

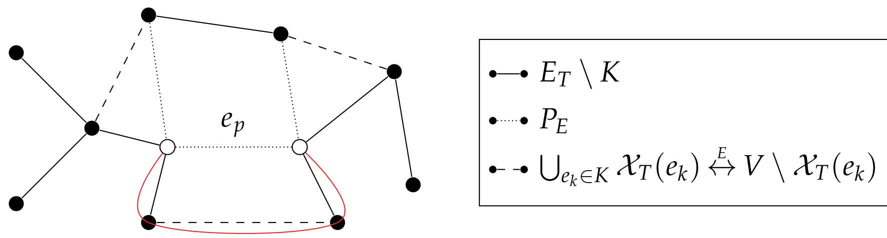

Figure 5.6: Embedding flow of a portal edge back into $H_G(T, K)$.

Now define $I_G(T, K)$ as the graph consisting of the edges in $E_T \setminus K$ and all portal edges P_E, i.e. (c.f. Figure 5.5)

$$I_G(T, K) = \Big(V, (E_T \setminus K) \cup P_E\Big)$$

Observe that by construction, there is a path between the endpoints of every cross-cut edge $e \in \mathcal{X}_T(e_k) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_T(e_k)$, and every tree slice of the sliced tree is also contained in $I_G(T, K)$, hence the $I_G(T, K)$ is connected. As such, it contains a spanning tree, and together with the subgraph (P_V, P_E), it forms an almost-$2j$-tree.

To compute the set of portal vertices P_V and portal edges P_E, we use a recursive tree splitting algorithm similar to `EmbedTree` from Lemma 5.4. When routing a cross-cut edge e through the splitter vertex v^*, we use the Euler tour to identify the two portal vertices of e, i.e. the first ancestors of either endpoint that are endpoints of an edge in K. The entire construction requires $\mathcal{O}(\log m \,\mathsf{scan}(m))$ I/Os as argued in Lemma 5.4.

Finally, to guarantee mutual embeddability of $I_G(T, K)$ into $H_G(T, K)$ and vice-versa, we need to define appropriate capacities for the edges in the almost-$2j$-tree. To that extent, denote the set of edges e for which e_p is a portal edge as $e \in \Pi(e_p)$, i.e. $\Pi(e_p)$ is the set of cut edges which are routed through the portal edge e_p (we can track this set during the construction of the portal edges without any additional overhea). We use the symbol $\mathbb{i}$ to denote the capacities that we define for $I_G(T, K)$. The idea is to use that by construction of $H_G(T, K)$, the original graph G one-embeds into T, hence we can route the embedding flow of the cut edges $(\mathcal{X}_T(e_k) \overset{E}{\leftrightarrow} V \setminus \mathcal{X}_T(e_k))$ through the three edges $E_T \setminus K$ in $I_G(T, K)$ up to and then through their portal edge. Define to that extent the capacities $\mathbb{i}$ for $I_G(T, K)$ as follows:

$$\mathbb{i}_e = \begin{cases} 2\mathbb{c}_e & \text{if } e \in E_T \setminus K \\ \sum_{e' \in \Pi(e)} \mathbb{c}_{e'} & \text{otherwise, i.e. } e \in P_E \end{cases}$$

With this choice, $H_G(T, K)$ one-embeds into the almost-$2j$-tree $I_G(T, K)$: For the tree edges $e_T \in E_T \setminus K$, we use the identity embedding, using up $\mathbb{c}_e$ units of capacity and leaving the remaining $\mathbb{c}_e$ for the cut edges. For these, we route the flow along the tree edges in E_T up to and then through the corresponding portal edge (c.f. Figure 5.5) – the choice tree and portal edge capacities ensures one-embeddability.

An embedding of $I_G(T, K)$ into $H_G(T, K)$ can be constructed as follows: For the tree edges $e_T \in E_T \setminus K$, an identity embedding flow has congestion two. For the portal edges $e_p \in P_E$, use that the capacity $\mathbb{i}_{e_p}$ of the portal edge in $I_G(T, K)$ is given by the sum of capacities of cut edges $e' = \{u, v\} \in \Pi(e_p)$ with $\mathsf{path}_T(e') = \langle u, \ldots, p_e^{(1)}, \ldots, p_e^{(2)}, \ldots, v \rangle$. Hence $\langle p_e^{(1)}, \ldots, u, v, \ldots, p_e^{(2)} \rangle$ (c.f. Figure 5.6) is a path in $H_G(T, K)$ between the endpoints of the portal edge e_p, and we can route $\mathbb{c}_{e'}$ units of flow along this path with

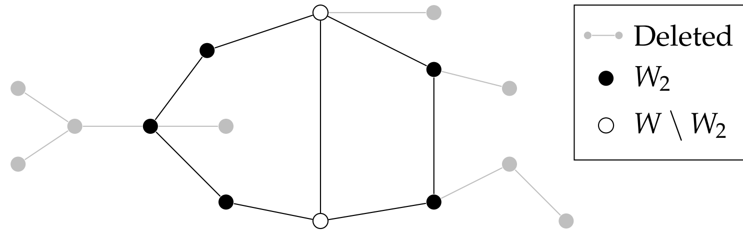

Figure 5.7: A graph after recursively deleting all degree-one vertices.

congestion 1 on the non-tree edges. Doing this for all such edges $e' \in \Pi(e_p)$ routes all of $\mathbb{i}_{e_p}$. At the same time, the congestion on the tree edges $e_T \in E_T \setminus K$ increases by at most an additional 1, because their capacities $\mathbb{c} \leftarrow \texttt{EmbedTree}(G, T)$ are chosen to accommodate the routing of all cut edges e'. This results in an overall embedding factor of three. □

The construction is summarised in Algorithm 5.3.

Algorithm 5.3 From j-sliced Trees to Almost-$2j$-Trees

1: **procedure** $\texttt{AlmostTreeify}(H_G(T, K))$
2: Build the star-tree $\tilde{T}$
3: Using recursive tree splitting, compute the set of portal edges P_E
4: Compute $E_I \leftarrow (E_T \setminus K) \cup P_E$ and capacities $\mathbb{i}$
5: **return** $(V, E_I, \mathbb{i})$

We proceed to the second step of transforming $I_G(T, K)$ into an $\mathcal{O}(j)$-tree. The leaves of $I_G(T, K)$, if any, can be reused as leaves in $J_G(T, K)$, so we can recursively clip off the leaves from $I_G(T, K)$ until obtaining some subgraph G_W on $W \subseteq V$ that contains only vertices of degree at least two in G_W. Consider from these in particular the vertices $W_2 \subseteq W$ of degree exactly two. W_2 gives rise to a set of disconnected paths or cycles, and breaking every cycle at some edge results in a forest on W_2. By reattaching the clipped-off vertices and suitably connecting this forest using an arbitrary graph on at most $6j - 2$ vertices, we will obtain $J_G(T, K)$.

Lemma 5.15 ([Mad11]). *There exists an algorithm* $\texttt{Treeify}$ *that, given an almost-$2j$-tree $I_G(T, K) = (V, E_I, \mathbb{i})$, finds a $(6j - 2)$-tree $J_G(T, K) = (V, E_J, \mathbb{j})$ that is three-embeddable into $I_G(T, K)$, and such that $I_G(T, K)$ is one-embeddable into $J_G(T, K)$.*

Proof. Let $G_W = (W, E_W)$ be the graph obtained by repeatedly removing from $I_G(T, K)$ all vertices of degree one, until only vertices of degree at least two in G_W remain. We distinguish two cases:

Case (i): There exists at least one vertex of degree strictly greater than two. Let $W_2 \subseteq W$ be the set of vertices of degree exactly two. The *core* of $J_G(T, K)$, which is the arbitrary $(6j - 2)$-vertex graph, will be built on $W \setminus W_2$ (c.f. Figure 5.7). To that extent, decompose G_W into a set of edge-disjoint paths $\{\mathcal{P}^{(i)}\}_i$ such that the (not necessarily distinct) endpoints of every path are in $W \setminus W_2$, and the inner vertices of every path are from W_2 only. Write $\mathcal{P}^{(i)} = \langle v_1^{(i)}, \ldots, v_{k_i}^{(i)} \rangle$. To ensure that $J_G(T, K)$ is acyclic except on $W \setminus W_2$, we remove from every path an edge of minimum capacity $e_{\min}^{(i)} = \arg\min_{e \in P^{(i)}} \mathbb{i}_e$, and instead add an edge between the endpoints of $\mathcal{P}^{(i)}$ (if the

endpoints are distinct and the edge does not already exist) to retain connectivity. The resulting graph may be described as follows:

$$J_G(T,K) = \left(V, E_I[\![V \setminus W]\!] \cup \left(\bigcup_i \left(\mathcal{P}^{(i)} \setminus \{e^{(i)}_{\min}\} \right) \cup \left\{ \{v_1^{(i)}, v_{k_i}^{(i)}\} \right\} \right) \right)$$

where $E_I[\![V \setminus W]\!]$ are the edges of E_I where at least one endpoint lies in $V \setminus W$.

Let us now prove that $J_G(T,K)$ is indeed a $(6j-2)$-tree: First, $J_G(T,K)$ acyclic except on $W \setminus W_2$, because any cycle must include only vertices from W, but the only edges with both endpoints in W are those added in the above construction. Assume for now that the cycle includes at least one vertex from $W \setminus W_2$. If this cycle includes even one vertex from W_2, then the corresponding two edges must have been part of one of the paths $\mathcal{P}^{(i)}$ whose endpoints are also both part of the cycle, hence by construction the cycle must have been split. Hence the cycle can include only vertices from $W \setminus W_2$, or only vertices from W_2. But the latter case is not possible, since the graph is connected and contains at least one vertex of degree strictly greater than two.

Second, we show that the number of vertices $|W \setminus W_2|$ in the core is at most $6j-2$: Let W_T be the set of vertices from G_W that are incident only to the edges of the spanning tree $E_T \setminus P_E$. Because $I_G(T,K)$ is an almost-$2j$-tree, we have $|W_T| \geq |W| - 2j$. Moreover, the number of edges in T restricted to G_W is at most $2|W|-2$, hence

$$2|W_T \cap W_2| + 3|W_T \setminus W_2| \leq \sum_{v \in W_T} \deg_{G_W}(v) \leq 2|W| - 2 \leq 2|W_T| + 4j - 2$$

and thus with $|W_T| = |W_T \cap W_2| + |W_T \setminus W_2|$, we have

$$|W_T \setminus W_2| \leq 4j - 2$$

In particular, the core of $J_G(T,K)$ consists of the vertices $W \setminus W_2$, for which we have

$$|W \setminus W_2| = |W_T \setminus W_2| + |W \setminus (W_T \cup W_2)| \leq 4j - 2 + 2j \leq 6j - 2$$

Hence $J_G(T,K)$ is indeed a $(6j-2)$-tree.

Finally, we must define appropriate capacities $\mathbb{j}$ for this $(6j-2)$-tree. Denote by $\pi(\{v_1^{(i)}, v_{k_i}^{(i)}\}) = \{j \mid \mathcal{P}^{(j)} = \langle v_1^{(i)}, \ldots, v_{k_i}^{(i)} \rangle\}$ the set of indices of the paths that start and end at $v_1^{(i)}$ and $v_{k_i}^{(i)}$. Then for any $e \in E_J$, let

$$\mathbb{j}_e = \begin{cases} \mathbb{i}_e & \text{if } e \in E_I[\![V \setminus W]\!] \\ \mathbb{i}_e + \mathbb{i}_{e^{(i)}_{\min}} & \text{if } e \in \mathcal{P}^{(i)} \text{ for some } i \\ \sum_{i \in \pi(e)} \mathbb{i}_{e^{(i)}_{\min}} & \text{otherwise} \end{cases}$$

To show that $I_G(T,K)$ one-embeds into $J_G(T,K)$, we construct the corresponding embedding flow explicitly. For any edge $e \in E_I[\![V \setminus W]\!]$, we use the identity embedding, incurring embedding congestion 1. Likewise, we use the identity embedding for all edges of the paths $\mathcal{P}^{(i)}$, except the edge $e^{(i)}_{\min} = \{u, v\}$, which we embed along the path $\langle u, \ldots, v_1^{(i)}, v_{k_i}^{(i)}, \ldots, v \rangle$ (c.f. Figure 5.8). Note that by construction of $\mathbb{j}_e$, this also incurs embedding congestion one, as none of the edges along the routing of $e^{(i)}_{\min}$ are from $E_I[\![V \setminus W]\!]$.

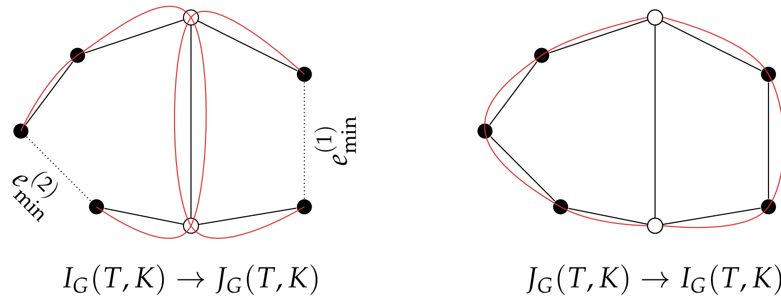

Figure 5.8: Embedding $I_G(T,K)$ into $J_G(T,K)$ and vice-versa.

For the three-embedding of $J_G(T,K)$ into $I_G(T,K)$, use the identity embedding for all edges except the $\{v_1^{(i)}, v_{k_i}^{(i)}\}$. By choice of the $e_{\min}^{(i)}$, this does not overflow the capacities by more than a factor of two. For the remaining edges e, route for every $i \in \pi(e)$ exactly $e_{\min}^{(i)}$ units of flow along the path $\mathcal{P}^{(i)}$, increasing the congestion on these edges by at most one. In total, the congestion on all edges of $I_G(T,K)$ is not more than three.

Case (ii): All vertices in G_W have degree exactly two. Because G_W is connected, this implies that G_W is just one cycle, and removing any edge $e_{\min}$ of minimum capacity $\mathbb{i}_{e_{\min}}$ and setting $\mathbb{j}_e = \mathbb{i}_e + \mathbb{i}_{e_{\min}}$ for all other edges of the cycle as above will suffice. Note that the final graph is a proper tree. □

Algorithm 5.4 From Almost-$2j$-Trees to $(6j-2)$-Trees

```
1: procedure Treeify(I_G(T,K))
2:     Label each vertex by whether or not it is incident to a portal edge
3:     Perform leaf elimination on T up to the first labelled vertex in each subtree
4:     Call the remaining graph G_W, construct W_2 ← {v ∈ W | deg_{G_W}(v) = 2}
5:     Compute Euler tours on the subgraph induced by W_2
6:     Identify edges ℰ_2 ⊆ P_E incident to exactly one vertex of W_2
7:     Build {𝒫^(i)}_i as described below
8:     Compute the edges e_min^(i)
9:     Compute 𝕛 and return J_G(T,K)
```

Lemma 5.16. *Treeify can be implemented in* $\mathcal{O}(\mathsf{sort}(m))$ *I/Os.*

Proof. Observe that recursive deletion of the degree-one vertices deletes exactly all vertices of $I_G(T,K)$ that are not involved in any cycle. The only cycles of $I_G(T,K)$ are those formed by the portal edges P_E together with the tree T. We claim that leaf elimination on T, stopping once a vertex is incident to an edge $P_E \setminus E_T$, correctly performs this recursive deletion in $\mathcal{O}(\mathsf{sort}(m))$ I/Os. Indeed, it is not hard to see that this leaf elimination cannot delete vertices that would have been part of a cycle. At the same time, the resulting graph has no degree-one vertices left; if it did, then this vertex must be incident *only* to an edge $P_E \setminus E_T$, but this violates the assumption that T is a spanning tree.

Next, we can compute in $\mathcal{O}(\mathsf{sort}(m))$ I/Os whether the remaining graph G_W has any vertices of degree strictly greater than two. If not, we can easily identify the edge

$e_{\min}$ and build the tree $J_G(T, K)$ in $\mathcal{O}(\mathsf{scan}(m))$ I/Os, so consider only the case where W_2 is non-empty. To find the path decomposition $\{\mathcal{P}^{(i)}\}_i$, extract in $\mathcal{O}(\mathsf{sort}(m))$ I/Os the vertex-induced subgraph on W_2. Since every vertex has at degree at most two, we can proceed as in Lemma 2.8 to build an Euler tour around every path in $\mathcal{O}(\mathsf{sort}(m))$ I/Os. Then in $\mathcal{O}(\mathsf{scan}(m))$ I/Os, extract the leafs of every path, i.e. the vertices that we still need to connect to $W \setminus W_2$. In $\mathcal{O}(\mathsf{sort}(m))$ I/Os, identify the edges incident to exactly one vertex of W_2, and sort these edges by that vertex. A tandem scan of the sorted edges together the vertices extracted from the Euler tours yields the paths $\{\mathcal{P}^{(i)}\}_i$.

This leaves us with a delimited list of edges in path order. In $\mathcal{O}(\mathsf{scan}(m))$ I/Os we can now identify the edges $e^{(i)}_{\min}$ for all paths, and in another scan produce the new capacities $\mathbb{j}_e$ for all edges, including the newly added edges $\{v_1^{(1)}, v_{k_i}^{(2)}\}$. A final $\mathcal{O}(\mathsf{sort}(m))$ I/Os merges all newly-added parallel edges and sums their capacities. □

Note that the total number of edges does not increase during the entire treeification process:

Lemma 5.17. *Given an m-edge j-sliced tree $H_G(T, K)$, the treeification of this tree through `AlmostTreeify` followed by `Treeify` results in a $(6j - 2)$-tree with at most m edges.*

Proof. Consider first `AlmostTreeify`. For every edge added by the construction, i.e. every portal edge, at least one cross-cut edge from $H_G(T, K)$ is removed, hence the total number of edges of the resulting almost-$2j$-tree cannot be more than m.

Likewise, `Treeify` removes at least one edge $e^{(i)}_{\min}$ for every edge it adds. Hence the total number of edges is still at most m. □

5.5 Decomposing Recursively

This concludes the discussion of how to decompose G into a convex combination of $\mathcal{O}(j)$-trees. We now just need to apply this decomposition recursively to the core of ever $\mathcal{O}(j)$-tree in such a way that the total number of trees does not explode. Recall that Lemma 5.13 allows us to drop a substantial fraction of the trees without increasing the overall approximation ratio too much. Following Sherman [She13], the algorithm will thus proceed as follows: For some parameter η, we will compute a decomposition into $\frac{n}{\eta}$-trees. We will then keep only η trees, so that in total we are left with at most n trees in total when the recursion is done. The procedure is given as pseudocode in Algorithm 5.5.

In the following, we substitute Sherman's inductive proof with a number of lemmas that will better permit us to study the decomposition's I/O complexity.

Lemma 5.18. *`ComputeTrees`(G, η) has recursion depth $\log n / \log \eta$, and at the end of the i-th level of recursion, there are at most η^i graphs in total, each having at most n/η^i vertices and $\mathcal{O}\left(\frac{n}{\eta^{i-1}} \log^2 \frac{n}{\eta^{i-1}}\right)$ edges.*

Proof. Consider the first level of recursion. After sparsification by Theorem 3.2, $\tilde{G}$ has at most $\tilde{m} \leq \mathcal{O}(n \log^2 n)$ edges. Hence by Theorem 5.1, there exists some constant c_1 such that the sliced trees returned by $\texttt{SlicedTreeDecompose}^t(\tilde{G})$ are sliced in at most

$$c_1 \cdot \frac{\beta \tilde{m} \log \tilde{m}}{t} = \frac{n}{6\eta}$$

Algorithm 5.5 Recursive j-Tree Decomposition; [She13]

1: **procedure** `ComputeTrees`(G, η)
2: $\tilde{G} = (V, \tilde{E}, \tilde{\boldsymbol{c}}) \leftarrow$ `Sparsify`$(G, \epsilon = 1/2)$
3: Scale up the capacities of $\tilde{G}$ by 2
4: If $\tilde{G}$ has at most $n-1$ edges, **return** $\tilde{G}$
5: Let $n \leftarrow |V|$, $t \leftarrow 24 \cdot 2^8 \cdot \eta\beta|\tilde{E}|\log|\tilde{E}|$ ▷ *c_1 is 2^8 from Theorem 5.1*
6: $\{\lambda^{(i)}, H_G^{(i)}(T,K)\}_i \leftarrow$ `SlicedTreeDecompose`$^t(\tilde{G})$
7: Keep only the η trees of largest $\lambda^{(i)}$, throw away the rest
8: Rescale the $\lambda^{(i)}$ to sum to 1
9: $I_G^{(i)}(T,K) \leftarrow$ `AlmostTreeify`$(H_G^{(i)}(T,K))$ for all i
10: $J_G^{(i)}(T,K) \leftarrow$ `Treeify`$(I_G^{(i)}(T,K))$ for all i
11: **for all** $i \in [\eta]$ **do**
12: Let $C^{(i)}$ be the core of $J_G^{(i)}(T,K)$
13: If $C^{(i)}$ is non-empty, compute $\{\lambda^{(j)}, T^{(j)}\}_j \leftarrow$ `ComputeTrees`$(C^{(i)}, \eta)$
14: Let $F^{(i)}$ be the forest of $J_G^{(i)}(T,K)$, i.e. with all core edges removed
15: Append $\{\lambda^{(i)}, F^{(i)}, \{\lambda^{(j)}, T^{(j)}\}\}$ to the output distribution of trees
16: **return** the final distribution of trees

edges. After treeifying all sliced trees, we are left with $(\frac{n}{\eta} - 2)$-trees (c.f. Lemma 5.15), of which only the η 'most significant' trees are kept. Thus by induction, the number of graphs after the i-th level of recursion is at most η^i, each of which has a core consisting of up to n/η^i vertices. The number of edges in the cores is at most $\mathcal{O}\left(\frac{n}{\eta^{i-1}}\log^2\frac{n}{\eta^{i-1}}\right)$ due to sparsification at the previous level and the fact that treeification does not increase the number of edges (Lemma 5.17). Finally, note that after $n \leq \eta^i \iff i \geq \log n/\log\eta$ recursions, the tree decomposition is complete. □

Lemma 5.19. *With high probability,* `ComputeTrees`(G,η) *returns an $(\alpha, \tilde{\mathbb{T}}_V)$-decomposition of G, where $\alpha \leq \mathcal{O}(\beta^{2\log n/\log\eta} n \log^{3\log n/\log\eta} n)$-decomposition of G.*

Proof. Note that all involved algorithms succeed with probability at least $1 - n^{-d}$ for any d. The number of calls to these algorithms is polynomially bounded, hence one can choose d such that `ComputeTrees` succeeds with high probability.

Now consider the first level of the recursion. `Sparsify`$(G, \epsilon = 1/2)$ returns a graph $\tilde{G}$ with capacities $\tilde{\boldsymbol{c}}$ such that for any cut S, $\frac{1}{2}c_{S \overset{E}{\leftrightarrow} V\setminus S} \leq \tilde{c}_{S \overset{\tilde{E}}{\leftrightarrow} V\setminus S} \leq \frac{3}{2}c_{S \overset{E}{\leftrightarrow} V\setminus S}$. Hence after scaling up the capacities of $\tilde{G}$ by a factor of two, G one-embeds into $\tilde{G}$ while $\tilde{G}$ three-embeds into G. Hence G also one-embeds into every $J_G^{(i)}(T,K)$, and thus the sliced tree decomposition of $\tilde{G}$ is also a sliced tree decomposition of G in the sense of Definition 5.3.

We then throw away a $(1 - \eta/t)$-fraction of the sliced trees, increasing the embedding factor of the decomposition from $\mathcal{O}(\beta)$ to $\mathcal{O}(\beta t/\eta) \leq \mathcal{O}(\beta^2 \log^3 n)$ by Lemma 5.13. The treeification increases this by a further constant factor. Hence by induction over the recursion depth $\tau = \log n/\log\eta$, the final decomposition is an $(\alpha, \tilde{\mathbb{T}}_V)$-decomposition of G with

$$\alpha \leq \prod_{i=0}^{\tau-1} O(1)\beta^2 \log^3(n\eta^{-i}) \leq n^{\mathcal{O}(1/\log\eta)}\beta^{2\tau}\log^{3\tau} n$$

where the $n^{\mathcal{O}(1/\log \eta)}$ is removed for sufficiently large n by merging it into the 'lost' $\prod_{i=0}^{\tau} \frac{\log n\eta^{-i}}{\log n}$ from the naive bound of $\log n\eta^{-i}$ used above. □

Lemma 5.20. *`ComputeTrees`(G, η) requires*

$$\mathcal{O}\left(\log^2 n \,\mathsf{sort}(m) + \frac{\eta}{\log \eta}\beta \log^4 n \cdot \left(\mathsf{LSST}(n, \mathcal{O}(n \log^2 n)) + \log^3 n \,\mathsf{scan}(n)\right)\right)$$

I/Os, where $\mathsf{LSST}(n, m)$ is the number of I/Os required to compute a tree of average stretch at most β on a graph of n vertices and m edges.

Proof. For the initial recursion, we expend $\mathcal{O}(\log^2 n \,\mathsf{sort}(m))$ I/Os when sparsifying the graph by Theorem 3.2. By Lemma 5.18, in all later levels of recursion, every n-vertex graph has at most $\mathcal{O}(\eta n \log^2(\eta n))$ edges. Let therefore $T(n)$ denote the number of I/Os performed by `ComputeTrees`(G, η) on a graph of n vertices and at most $\mathcal{O}(\eta n \log^2(\eta n))$ edges.

During such a call, we sparsify G in $\mathcal{O}(\log^2 n \,\mathsf{sort}(n \log^2 n)) \leq \mathcal{O}(\log^4 n \,\mathsf{scan}(n))$ I/Os to have only $\tilde{m} \leq \mathcal{O}(n \log^2 n)$ edges. We then compute a decomposition into $t \leq \mathcal{O}(n\beta \log^3 n)$ sliced trees. This decomposition takes

$$\mathcal{O}(t \cdot (\mathsf{LSST}(n, \tilde{m}) + \log \tilde{m} \,\mathsf{scan}(\tilde{m}))) \leq \mathcal{O}(\eta\beta \log^3 n \cdot (\mathsf{LSST}(n, \tilde{m}) + \log^3 n \,\mathsf{scan}(n)))$$

I/Os. The η 'most significant' trees are extracted in $\mathcal{O}(\mathsf{sort}(t))$ I/Os and treeified individually in $\mathcal{O}(\log \tilde{m} \,\mathsf{scan}(\tilde{m}))$ I/Os each.

Keeping only the dominating terms, we obtain the following recurrence:

$$T(n) \leq \eta T(n/\eta) + \mathcal{O}\left(\eta\beta \log^3 n \left(\mathsf{LSST}(n, \mathcal{O}(n \log^2 n)) + \log^3 n \,\mathsf{scan}(n)\right)\right)$$

which is bounded by

$$T(n) \leq \mathcal{O}\left(\frac{\eta}{\log \eta}\beta \log^4 n \left(\mathsf{LSST}(n, \mathcal{O}(n \log^2 n)) + \log^3 n \,\mathsf{scan}(n)\right)\right)$$

Adding the I/Os required for the initial sparsification completes the proof. □

Remark. For $\eta \leq n^{1/M}$, using the I/O master theorem for recursive algorithms [Dem+18] gives a tighter bound, replacing a $\tau = \log n / \log \eta$ factor by M in the bound of $T(n)$.⋄

Corollary 5.21. *For any constant k, an $\exp \mathcal{O}(k\sqrt{\log n \log \log n})$-congestion-approximator $\boldsymbol{R}$ can be computed with high probability in $\mathcal{O}(\log^2 n \,\mathsf{sort}(m) + \mathsf{scan}(n^{1+1/k}))$ I/Os.*

Proof. Use $\eta = n^{1/2k}$ in Lemma 5.19 and Lemma 5.20 together with the low–average-stretch spanning tree algorithm from Theorem 4.1. Then observe that for constant k, $\exp(\mathcal{O}(\sqrt{\log n \log \log n})) \leq n^{1/2k}$ for sufficiently large n. Hence by hiding all lower-terms in a $n^{1/2k}$-factor, we obtain a total of $\mathcal{O}(\log^2 n \,\mathsf{sort}(m) + n^{1/2k} \,\mathsf{scan}(n^{1+1/2k}))$ I/Os.

Note that by Lemma 5.2, the tree decomposition of G immediately yields a congestion approximator $\boldsymbol{R}$, although we must still ensure that the number of rows is at most $\frac{1}{2}n^2$ to satisfy Definition 5.1. Due to the recursive construction, many trees share the same outer edges (with the same capacities), and we can at least conceptually collapse

all these into a single row. The total number of rows is then the sum of the number of non-core, i.e. forest, edges, which is at most

$$\sum_{i=1}^{\tau} \eta^{i-1} \left(\frac{n}{\eta^{i-1}} - \frac{n}{\eta^i} \right) = \tau(1 - 1/\eta)n = k(n - n^{1-1/k}) \leq kn$$

This is indeed less than $\frac{1}{2}n^2$ for sufficiently large n (for smaller n, we can use Lemma 5.13 with $\delta = 1/2$). □

We will however avoid constructing $\boldsymbol{R}$ explicitly as a matrix: even after collapsing shared rows, matrix-vector products would still require $\mathcal{O}(n^2)$ arithmetic operations. We can improve on this by exploiting the tree structure that defines $\boldsymbol{R}$.

Recall that $\boldsymbol{R}$ as defined in the proof of Lemma 5.2 has $n-1$ rows for each tree, one row per tree edge, and such that $(\boldsymbol{Rb})_e$ is just the (signed) congestion on the edge e when routing $\boldsymbol{b}$ in the tree to which e belongs. Hence to compute $\boldsymbol{Rb}$, we can route $\boldsymbol{b}$ in all trees, and do so by exploiting the recursive structure of the trees. The demand flowing across every edge is the sum of the demands in the cutset $\mathcal{X}_T(e) \subseteq V$, which can easily be computed in $\mathcal{O}(\mathsf{sort}(n))$ I/Os using the leaf elimination technique from Lemma 2.11. We will perform this leaf elimination on the outer forest part of every level of the decomposition, and then recurse on the core. This is made more precise in Lemma 5.23.

We will also have to compute products of the form $\boldsymbol{R}^T\boldsymbol{x}$. Note that $\boldsymbol{R}^T$ has one row per vertex and one column for every tree-edge of every tree. Hence $\boldsymbol{x}$ is a vector of values x_e for each tree edge e, and $(\boldsymbol{R}^T\boldsymbol{x})_v$ is the sum over $\frac{x_e}{\mathbb{c}_e}$ over all edges e where $v \in \mathcal{X}_T(e)$. If we orient the edges e towards the leaves, then we can again collect all these terms using leaf elimination – we just need to ensure that this orientation is consistent with the leaf elimination used for computing $\boldsymbol{Rb}$. Recall that since $\boldsymbol{b}$ are valid demands, for any $S \subseteq V$, $b_S = -b_{S\setminus V}$. Hence flipping the orientation of an edge will correspond to changing the sign in the computation of $(\boldsymbol{Rb})_e$.

The idea of using leaf elimination is due to Sherman [She13], but the presentation here is more explicit about keeping `ComputeR` and `ComputeR`^{T} consistent. The (high-level) pseudocode for both procedures is given in Algorithm 5.6.

We first prove correctness and performance of `ComputeR`^{T}, since this algorithm requires us to define the sets $\mathcal{X}_T(e)$ by giving an orientation of the tree edges.

Lemma 5.22. *Given the decomposition* `ComputeTrees`(G, η) *and values* $\boldsymbol{x}$ *on the edges of the decomposition,* `ComputeR`^{T} *computes* $\boldsymbol{R}^T\boldsymbol{x}$ *in* $\mathcal{O}(\frac{\log n}{\log \eta}\eta\, \mathsf{sort}(n))$ *I/Os.*

Proof. We first show how to use leaf elimination on the tree T to compute the vertex values $\boldsymbol{\psi}$. Begin by modifying the Euler tour construction to maintain the values $\frac{x_e}{\mathbb{c}_e}$ on the arcs. The modified construction still requires $\mathcal{O}(\mathsf{sort}(n))$ I/Os.

Now perform leaf elimination on T, starting the Euler tour traversal at some leaf. When traversing the last outgoing arc (v, w) with value $\frac{x_e}{\mathbb{c}_e}$ from some vertex v, output $\nu_v = f(v, \{\nu_1, \ldots, \nu_k\}) = \frac{x_e}{\mathbb{c}_e} + \sum_{i=1}^{k} \nu_i$ (note that this function is efficiently tree-foldable). This corresponds to the sum of the values $\frac{x_{e'}}{\mathbb{c}_{e'}}$ for all edges e' in the 'lower' subtree of v, plus the value on the outgoing arc (v, w). When done with the tour, subtract again the value of this outgoing arc for all vertices.

One can see now that the resulting vertex values correspond exactly to the desired $(\boldsymbol{R}^T\boldsymbol{x})_v$ when orienting the edges away from the leaves during leaf elimination.

Algorithm 5.6 Computing $\boldsymbol{Rb}$ and $\boldsymbol{R}^T\boldsymbol{x}$ efficiently; [She13]

```
1: procedure ComputeR({λ^(i), F^(i), {λ^(j), T^(j)}_j}_i, b)
2:     for all λ^(i), F^(i){λ^(j), T^(j)}_j in the decomposition do
3:         Let b' be b restricted to the core
4:         Compute μ' ← ComputeR({λ^(j), T^(j)}_j, b')
5:         Build a tree T by connecting the forest F^(i) via a new vertex s
6:         Route b on f on T using leaf elimination
7:         Let μ be μ' concatenated with the congestions incurred on F^(i)
8:     return the concatenation of all μ
9: procedure ComputeR^T({λ^(i), F^(i), {λ^(j), T^(j)}_j}_i, x)
10:    for all λ^(i), F^(i){λ^(j), T^(j)}_j in the decomposition do
11:        Let x' be x restricted to the core
12:        Compute ψ' ← ComputeR^T({λ^(j), T^(j)}_j, x')
13:        Build a tree T by connecting the forest F^(i) via a new vertex s
14:        Using leaf elimination, compute vertex potentials ψ
15:        Concatenate ψ' to ψ
16:    return the concatenation of all ψ
```

Let $T(n)$ be the number of I/Os used during $\texttt{ComputeR}^T$ when all trees in the decomposition have at most n vertices. The decomposition contains at most η forests, hence we obtain the recurrence $T(n) = \eta T(n/\eta) + \mathcal{O}(\eta\,\mathsf{sort}(n))$, where $\mathsf{sort}(n)$ is from the cost of constructing an Euler tour on the at most $n - n/\eta \leq n$ forest edges. This is bounded by $\mathcal{O}(\tau\eta\,\mathsf{sort}(m))$ for $\tau = \log n / \log \eta$; the lemma follows. □

Remark. For $\eta \leq n^{1/M}$, the I/O master theorem [Dem+18] again gives a tighter bound of $\mathcal{O}(M\eta\,\mathsf{sort}(n))$. ◇

The orientation of the edges in the decomposition is now defined depending on the leaf elimination traversal order. We now show how to compute $\boldsymbol{Rb}$ using the same leaf elimination traversals:

Lemma 5.23. *Given valid demands* $\boldsymbol{b}$ *and the decomposition* $\texttt{ComputeTrees}(G, \eta)$*,* $\texttt{ComputeR}$ *computes* $\boldsymbol{Rb}$ *in* $\mathcal{O}(\frac{\log n}{\log \eta}\eta\,\mathsf{sort}(n))$ *I/Os.*

Proof. To route $\boldsymbol{b}$ in some tree T, perform leaf elimination on T to sum the values b_v into σ_v at each $v \in V$. When traversing the last outgoing arc (u, v) from some vertex u, let its congestion $(\boldsymbol{Rb})_e$ be $-\sigma_v/c_e$, where $e = \{u, v\}$. Note that $b_{\mathcal{X}_T(e)} = -b_{V\setminus\mathcal{X}_T(e)} = -\sigma_v$, hence this leaf elimination correctly computes $(\boldsymbol{Rb})_e$. The number of I/Os follows the same reasoning as for Lemma 5.22. □

5.6 Expander Graphs

For special cases, we can design much simpler congestion approximators than Madry's rather involved decomposition. In some cases however, such as planar graphs or grid graphs, even simpler methods than Sherman's algorithm suffice for computing exact maximum flows.[1] A non-trivial result is that we can efficiently compute approximate

[1] These graphs can be traversed cache-efficiently [TZ02], and hence standard algorithms for these cases are immediately efficient.

maximum flows on graphs of large *conductance* (defined below) due to an observation of Sherman [She13].

Definition 5.8 (Conductance)**.** Let $G = (V, E, \boldsymbol{c})$ be some undirected, capacitated graph. For any $S \subseteq V$, the *volume* of S is the sum of weighted degrees in S, i.e. $\mathsf{vol}_G(S) = \sum_{v \in S} c_{\{v\} \leftrightarrow V \setminus \{v\}}$. The *conductance* $\varphi(S)$ *of* S is the ratio

$$\varphi(S) = \frac{c_{S \leftrightarrow V \setminus S}}{\min\{\mathsf{vol}_G(S), \mathsf{vol}_G(V \setminus S)\}}$$

The *conductance of* G is the minimum conductance of any cut $S \subseteq V$. ◇

Lemma 5.24 (Congestion Approximation in Expanders; [She13])**.** *Let $G = (V, E, \boldsymbol{c})$ be an undirected, capacitated graph of conductance ϕ. Then the diagonal matrix $\boldsymbol{R} \in \mathbb{R}^{\times n}$ with $(\boldsymbol{R})_{v,v} = c^{-1}_{\{v\} \leftrightarrow V \setminus \{v\}}$ is a ϕ^{-1}-congestion-approximator.*

Proof. By Lemma 2.2,

$$|(\boldsymbol{Rb})_v| = \frac{|b_v|}{c_{\{v\} \leftrightarrow V \setminus \{v\}}} \leq \mathsf{opt}_G(\boldsymbol{b})$$

At the same time, any cut $S \subseteq V$ has demand at most

$$|b_S| \leq \sum_{v \in S} |b_v| = \sum_{v \in V} |(\boldsymbol{Rb})_v| c_{\{v\} \leftrightarrow V \setminus \{v\}} \leq \sum_{v \in V} \|\boldsymbol{Rb}\|_\infty c_{\{v\} \leftrightarrow V \setminus \{v\}} = \|\boldsymbol{Rb}\|_\infty \mathsf{vol}_G(S)$$

and thus if w.l.o.g. $\mathsf{vol}_G(S) \leq \mathsf{vol}_G(V \setminus S)$ then

$$\frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} = \varphi(S)^{-1} \frac{|b_S|}{\mathsf{vol}_G(S)} \leq \varphi(S)^{-1} \|\boldsymbol{Rb}\|_\infty \leq \phi^{-1} \|\boldsymbol{Rb}\|_\infty$$

which holds in particular for the maximally congested cut of Theorem 2.1. □

Expander graphs are essentially families of graphs where the second-largest eigenvalue λ_2 of the graph Laplacian is large. Cheeger's inequality relates this to the conductance of these graphs:

$$\frac{1}{2}\lambda_2 \leq \varphi(G) \leq \sqrt{2\lambda_2}$$

Note that computing $c_{\{v\} \leftrightarrow V \setminus \{v\}}$ can be easily done in $\mathcal{O}(\mathsf{sort}(m))$ I/Os on the arc list representation of the graph. Hence for expander graphs, we immediately have a $\mathcal{O}(\lambda_2^{-1})$-congestion-approximator that requires only $\mathcal{O}(\mathsf{sort}(m))$ I/Os to construct, and $\mathcal{O}(\mathsf{scan}(n))$ I/Os to evaluate $\boldsymbol{Rb}$ and $\boldsymbol{R}^T\boldsymbol{x}$.

Chapter 6

Sherman's Approximate Maximum Flow Algorithm

Recall from the introduction that the aim throughout the thesis is to develop an algorithm for solving the maximum flow problem by iteratively taking almost-optimal steps $\boldsymbol{f}^{(i)}$. Each step is selected by approximately minimising a potential function that involves the congestion approximator $\boldsymbol{R}$, namely

$$\boldsymbol{f}^{(i)} \approx \arg\min_{\boldsymbol{f}} \|\boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty + 2\alpha\|\boldsymbol{R}(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f})\|_\infty$$

Thus, we will develop two algorithms: An algorithm `Route` that adds together all the steps while ensuring that the final solution satisfies the constraints $\boldsymbol{B}\boldsymbol{f} = \boldsymbol{b}$ exactly, and an algorithm `AlmostRoute` that minimises the potential function to find the steps $\boldsymbol{f}^{(i)}$. When we are done, we will also want to return a cut $S \subseteq V$ that certifies the optimality of our solution by Lemma 2.2: Since $\frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} \leq \mathsf{opt}_G(\boldsymbol{b})$, if the final solution $\boldsymbol{f}$ has congestion at most $\|\boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty \leq (1+\epsilon)\frac{|b_S|}{c_{S \leftrightarrow V \setminus S}}$, then certainly $\|\boldsymbol{C}^{-1}\boldsymbol{f}\| \leq (1+\epsilon)\mathsf{opt}_G(\boldsymbol{b})$ also. To efficiently construct the certificate S, we will rely on the dual problem from Section 2.3.

In somewhat greater detail, we will in Section 6.2 develop `AlmostRoute` to satisfy the following guarantees:

Theorem 6.1 ([She13]). *There exists an algorithm* `AlmostRoute`$(G, \boldsymbol{b}, \epsilon)$ *that, given* $0 < \epsilon \leq 1/2$ *and an undirected graph* G *with an* α*-congestion-approximator* $\boldsymbol{R}$ *for it, returns a flow* $\boldsymbol{f}$ *and vertex-induced cut* $S \subseteq V$ *such that*

$$\left\|\boldsymbol{C}^{-1}\boldsymbol{f}\right\|_\infty + 2\alpha\left\|\boldsymbol{R}(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f})\right\|_\infty \leq (1+\epsilon)\frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} \leq (1+\epsilon)\mathsf{opt}_G(\boldsymbol{b})$$

The algorithm makes at most $\mathcal{O}(\alpha^2\epsilon^{-3}\log n \log \alpha)$ *iterations, each of which requires a multiplication by* $\boldsymbol{R}$ *and* $\boldsymbol{R}^T$ *in addition to* $\mathcal{O}(\mathsf{scan}(m))$ *I/Os.*

Note that `AlmostRoute` does not route all the demand $\boldsymbol{b}$ exactly. We will therefore construct the algorithm `Route` that invokes `AlmostRoute` in such a way that the residual demands left by `AlmostRoute` quickly become small and trivial to route in G.

Formally, `Route` satisfies the following:

Theorem 6.2 ([She13]). *There exists an algorithm* `Route`$(G, \boldsymbol{b}, \epsilon)$ *that, given* $0 < \epsilon \leq 1/2$ *and an undirected graph* G *with an* α*-congestion approximator for it, returns a flow* $\boldsymbol{f}$ *satisfying* $\boldsymbol{Bf} = \boldsymbol{b}$ *and vertex-induced cut* $S \subseteq V$ *such that*

$$\mathsf{opt}_G(\boldsymbol{b}) \leq \left\| \boldsymbol{C}^{-1}\boldsymbol{f} \right\|_\infty \leq (1+\epsilon)\frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} \leq (1+\epsilon)\mathsf{opt}_G(\boldsymbol{b})$$

The algorithm makes at most $\mathcal{O}(\log m)$ *calls to* `AlmostRoute` *and additionally requires* $\mathcal{O}(\log m\, \mathsf{scan}(m))$ *I/Os with overwhelming probability* $1 - \exp(-\Omega(m))$.

Note in particular that for the s-t flow problem where $b_s = -1, b_t = 1$, and $b_v = 0$ for all other vertices, S is a $(1+\epsilon)$-minimum s-t cut.

We begin with discussing the workings of `Route`.

6.1 The Algorithm `Route`

The initial call to `AlmostRoute` provides us with a flow $\boldsymbol{f}^{(0)}$ and cut S such that $\boldsymbol{f}^{(0)}$ already has low congestion, but does not satisfy all demands exactly. But because the potential function overemphasises the cost of the residual demands by factor of two, the initial call to `AlmostRoute` already pays for the congestion of routing the remaining demands: Since $\boldsymbol{f}^{(0)}$ is within a factor of $(1+\epsilon)$ from optimal, we have

$$\left\| \boldsymbol{C}^{-1}\boldsymbol{f}^{(0)} \right\|_\infty \leq (1+\epsilon)\mathsf{opt}_G(\boldsymbol{b}) - 2\alpha \left\| \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{Bf}^{(0)}) \right\|_\infty \leq (1+\epsilon)\mathsf{opt}_G(\boldsymbol{b}) - 2\mathsf{opt}_G(\boldsymbol{b} - \boldsymbol{Bf}^{(0)})$$

and hence even if we route the remaining demands $\boldsymbol{b} - \boldsymbol{Bf}$ via $\tilde{\boldsymbol{f}}$ only within a factor of 2 from optimal, we still satisfy

$$\left\| \boldsymbol{C}^{-1}(\boldsymbol{f} + \tilde{\boldsymbol{f}}) \right\|_\infty \leq \|\boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty + 2\mathsf{opt}_G(\boldsymbol{b} - \boldsymbol{Bf}) \leq (1+\epsilon)\mathsf{opt}_G(\boldsymbol{b})$$

At a high level, `Route` will therefore have the following structure: (i) Construct the flow $\boldsymbol{f}^{(0)}$ and the corresponding certificate cut S, (ii) iteratively route the residual demands almost-optimally, whith the congestion of the resulting flows $\boldsymbol{f}^{(i)}$ already paid for by the initial flow. Once these residual demands are very small, we can (iii) route them within a factor m of optimal along a maximum spanning tree (recall Lemma 5.6) to satisfy the demands exactly, while ensuring that all previous iterations have already paid for the congestion incurred by this final flow.

To make this work, after routing $\boldsymbol{f}^{(0)}$ within $(1+\epsilon)$ of optimal, we route all further $\boldsymbol{f}^{(i)}$ within $(1 + 1/2)$ of optimal, rather than the factor 2 that the previous iteration has paid for. We thereby accumulate 'slack' towards allowing the final flow to be routed within a factor of m from optimal.

The algorithm is shown in Algorithm 6.1. To show that the iterations converge quickly towards an optimal solution, we will make use of the following lemma:

Lemma 6.1 ([She13]). *For* $1 \leq i \leq \tau$, $\left\| \boldsymbol{Rb}^{(i+1)} \right\|_\infty \leq 1/2 \left\| \boldsymbol{Rb}^{(i)} \right\|_\infty$.

Proof. Let $\boldsymbol{f}$ be an optimal routing of $\boldsymbol{b}^{(i+1)}$, and let $\boldsymbol{f}^{(i)}$ be the routing of $\boldsymbol{b}^{(i)}$ returned by `AlmostRoute` in the i-th iteration. Note that $\boldsymbol{f} + \boldsymbol{f}^{(i)}$ is a routing of $\boldsymbol{b}^{(i)}$, since $\boldsymbol{B}(\boldsymbol{f} + \boldsymbol{f}^{(i)}) = \boldsymbol{b}^{(i+1)} + \boldsymbol{Bf}^{(i)} = \boldsymbol{b}^{(i)}$. Hence

$$\mathsf{opt}_G(\boldsymbol{b}^{(i)}) \leq \left\| \boldsymbol{C}^{-1}(\boldsymbol{f}^{(i)} + \boldsymbol{f}) \right\|_\infty \leq \left\| \boldsymbol{C}^{-1}\boldsymbol{f}^{(i)} \right\|_\infty + \left\| \boldsymbol{C}^{-1}\boldsymbol{f} \right\|_\infty$$

Algorithm 6.1 The Algorithm `Route`; [She13]

1: **procedure** `Route`$(G, \boldsymbol{b}, \epsilon)$
2: Let $\boldsymbol{f}^{(0)}, S \leftarrow$ `AlmostRoute`$(G, \boldsymbol{b}, \epsilon)$, let $\boldsymbol{b}^{(1)} \leftarrow \boldsymbol{b} - \boldsymbol{B}\boldsymbol{f}^{(0)}$
3: Define $\tau = \lceil \log_2(2m) \rceil$
4: **for** $t = 1, \dots, \tau$ **do**
5: Let $\boldsymbol{f}^{(t)} \leftarrow$ `AlmostRoute`$(G, \boldsymbol{b}^{(t)}, 1/2)$, let $\boldsymbol{b}^{(t+1)} \leftarrow \boldsymbol{b}^{(t)} - \boldsymbol{B}\boldsymbol{f}^{(t)}$
6: Compute a maximum spanning tree T of G with capacities $\boldsymbol{c}$ as edge weights
7: Compute the flow $\boldsymbol{f}^{(\tau+1)}$ that routes $\boldsymbol{b}^{(\tau+1)}$ along T
8: **return** $\sum_{i=0}^{\tau+1} \boldsymbol{f}^{(i)}$ and S

The optimal routing $\boldsymbol{f}$ satisfies

$$\left\| \boldsymbol{C}^{-1}\boldsymbol{f} \right\|_\infty = \mathsf{opt}_G(\boldsymbol{b}^{(i+1)}) \le \alpha \left\| \boldsymbol{R}\boldsymbol{b}^{(i+1)} \right\|_\infty$$

Recall also that by Theorem 6.1,

$$\left\| \boldsymbol{C}^{-1}\boldsymbol{f}^{(i)} \right\|_\infty \le \frac{3}{2}\mathsf{opt}_G(\boldsymbol{b}^{(i)}) - 2\alpha \left\| \boldsymbol{R}\boldsymbol{b}^{(i+1)} \right\|_\infty$$

Combining both yields

$$\mathsf{opt}_G(\boldsymbol{b}^{(i)}) \le \frac{3}{2}\mathsf{opt}_G(\boldsymbol{b}^{(i)}) - \alpha \left\| \boldsymbol{R}\boldsymbol{b}^{(i+1)} \right\|_\infty$$

and hence

$$\alpha \left\| \boldsymbol{R}\boldsymbol{b}^{(i+1)} \right\|_\infty \le \frac{1}{2}\mathsf{opt}_G(\boldsymbol{b}^{(i)}) \le \frac{\alpha}{2} \left\| \boldsymbol{R}\boldsymbol{b}^{(i)} \right\|_\infty$$

Divide by α and the lemma follows. □

We are now ready to state the proof of Theorem 6.2.

Proof of Theorem 6.2. Denote by $\sigma^{(t)} = 2\alpha \|\boldsymbol{R}(\boldsymbol{b}^{(t)} - \boldsymbol{B}\boldsymbol{f}^{(t)})\|_\infty$ the 'slack' left by the t-th iteration. Because by Lemma 5.6, routing along a maximum spanning tree incurs congestion at most m times optimal, the flow returned by `Route` satisfies

$$\begin{aligned}\left\| \boldsymbol{C}^{-1} \sum_{i=0}^{\tau+1} \boldsymbol{f}^{(i)} \right\|_\infty &\le \sum_{i=0}^{\tau+1} \left\| \boldsymbol{C}^{-1}\boldsymbol{f}^{(i)} \right\|_\infty \\ &\le \underbrace{(1+\epsilon)\frac{|b_S|}{c_{S \leftrightarrow V \setminus S}}}_{\|\boldsymbol{C}^{-1}\boldsymbol{f}^{(0)}\|_\infty} + \underbrace{m\mathsf{opt}_G(\boldsymbol{b}^{(\tau+1)})}_{\|\boldsymbol{C}^{-1}\boldsymbol{f}^{(\tau+1)}\|_\infty} + 3/2 \sum_{i=1}^{\tau} \mathsf{opt}_G(\boldsymbol{b}^{(i)}) - \sum_{i=0}^{\tau} \sigma^{(i)}\end{aligned}$$

where we rely on correctness of `AlmostRoute` from Theorem 6.1, i.e. that for $1 \le i \le \tau$, $\|\boldsymbol{C}^{-1}\boldsymbol{f}^{(i)}\|_\infty \le (1 + 1/2)\mathsf{opt}_G(\boldsymbol{b}^{(i)})$. Now by definition of an α-congestion-approximator, $\sigma^{(i)} \ge 2\mathsf{opt}_G\left(\boldsymbol{b}^{(i)} - \boldsymbol{B}\boldsymbol{f}^{(i)}\right) = 2\mathsf{opt}_G\left(\boldsymbol{b}^{(i+1)}\right)$ and hence

$$\left\| \boldsymbol{C}^{-1} \sum_{i=0}^{\tau+1} \boldsymbol{f}^{(i)} \right\|_\infty \le (1+\epsilon)\frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} + m\mathsf{opt}_G(\boldsymbol{b}^{(\tau+1)}) - 1/4 \sum_{i=0}^{\tau-1} \sigma^{(i)}$$

At the same time, by induction with Lemma 6.1,

$$m\mathsf{opt}_G(\boldsymbol{b}^{(\tau+1)}) \le m\alpha \left\| \boldsymbol{R}\boldsymbol{b}^{(\tau+1)} \right\| \le m\alpha 2^{-\tau} \left\| \boldsymbol{R}\boldsymbol{b}^{(1)} \right\| \le m 2^{-\tau-1}\sigma^{(0)}$$

and hence choosing $\tau = \lceil \log_2(2m) \rceil$ guarantees

$$\left\| C^{-1} \sum_{i=0}^{\tau+1} f^{(i)} \right\|_\infty \leq (1+\epsilon)\frac{|b_S|}{c_{S\leftrightarrow V\setminus S}} + \frac{1}{4}\sigma^{(0)} - \frac{1}{4}\sum_{i=0}^{\tau-1}\sigma^{(i)} \leq (1+\epsilon)\frac{|b_S|}{c_{S\leftrightarrow V\setminus S}}$$

which proves both correctness and the claimed number of iterations.

A maximum spanning tree can be computed in $\mathcal{O}(\mathsf{sort}(m))$ I/Os with overwhelming probability (c.f. Lemma 2.21). Routing demands on trees can be implemented using leaf elimination to sum the demands at every vertex in $\mathcal{O}(\mathsf{sort}(m))$ I/Os. The flow along the tree edges can then be computed in a constant number of sorts and scans similar to Algorithm 5.1. Apart from the calls to `AlmostRoute`, an iteration of `Route` requires $\mathcal{O}(\mathsf{scan}(m))$ I/Os to sum the flows. With $\tau \leq \mathcal{O}(\log m)$, this adds up to a total of $\mathcal{O}(\log m\, \mathsf{scan}(m))$ I/Os besides the calls to `AlmostRoute`. □

Remark. The analysis suffers from the difficulty of proving a lower bound for $\sum_{i=0}^{\tau-1}\sigma^{(i)}$ and uses only the first term in the sum. When implementing the algorithm in practice, it makes sense to maintain the current sum $1/4\sum_{i=0}^{t}\sigma^{(i)}$ at the end of the t-th iteration, and stop early when it exceeds $m\alpha \left\| Rb^{(t+1)} \right\| \geq m\mathsf{opt}_G(b^{(t+1)})$. ◇

6.2 The Algorithm `AlmostRoute`

This section develops the solver `AlmostRoute` to find the steps $f^{(i)}$ with

$$f^{(i)} \approx \arg\min_f \|C^{-1}f\|_\infty + 2\alpha\|R(b - Bf)\|_\infty$$

Here we finally get to make use of the congestion approximator R and tie together all the methods developed in this thesis. While `AlmostRoute` will use gradient descent, escaping the combinatorial formulation of the maximum flow problem, we will still rely on the combinatorial graph representation of R to compute Rb and $R^T v$ – matrix-vector multiplication would be too expensive for an almost linear I/O algorithm.

After (almost) minimising the potential, we will then again jump back to the convex programming formulation to generate vertex potentials ψ for the dual problem (the maximally congested cut problem) from the gradient, and use these together with `CongestedCut` from Section 2.3 for generating the certificate cut $S \subseteq V$.

Let us focus now on minimising the potential function for finding $f^{(i)}$. Since the supremum norm is not differentiable, Sherman follows standard practice of replacing it by the smooth 'smax' function, which closely approximates it. We have

$$\mathrm{smax}(x) = \log\left(\sum_i e^{x_i} + e^{-x_i}\right) \qquad \nabla\,\mathrm{smax}(x) = \frac{1}{\sum_i e^{x_i} + e^{-x_i}}\begin{pmatrix} e^{x_1} - e^{-x_1} \\ \vdots \end{pmatrix}$$

and obtain the following potential function, where the scalars have been moved inside of smax to slightly simplify the analysis later.

$$\Phi(f) = \mathrm{smax}\left(C^{-1}f\right) + \mathrm{smax}\left(2\alpha R(b - Bf)\right)$$

Towards performing gradient descent, compute also by linearity of R

$$\nabla\Phi(f) = C^{-1}\left(\nabla\,\mathrm{smax}\left(C^{-1}f\right)\right) - 2\alpha B^T R^T\left(\nabla\,\mathrm{smax}\left(2\alpha R(b - Bf)\right)\right)$$

The following lemma will help us bound the number of gradient descent steps:

Lemma 6.2 (Fact 2.3 without proof in [She13]). *For all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$, the* smax *function satisfies*

1. $\|\boldsymbol{x}\|_\infty \leq \frac{1}{\lambda}\operatorname{smax}(\lambda \boldsymbol{x}) \leq \|\boldsymbol{x}\|_\infty + \frac{1}{\lambda}\log(2d)$ *for all* $\lambda > 0$
2. $\|\nabla \operatorname{smax}(\boldsymbol{x})\|_1 \leq 1$
3. $(\nabla \operatorname{smax}(\boldsymbol{x}))^T \boldsymbol{x} \geq \operatorname{smax}(\boldsymbol{x}) - \log(2d)$
4. $\|\nabla \operatorname{smax}(\boldsymbol{x}) - \nabla \operatorname{smax}(\boldsymbol{y})\|_1 \leq \|\boldsymbol{x} - \boldsymbol{y}\|_\infty$

We defer the proof of this to Subsection 6.2.1.

The first statement implies that when replacing $\|\cdot\|_\infty$ with an smax, we may incur an error of up to an additive $\log(2d)$. To circumvent the additive error and minimise to within $(1+\epsilon)$, we can scale scale $\boldsymbol{f}, \boldsymbol{b}$ by some appropriate factor λ. Indeed, say we have minimised the smax potential $\Phi(\lambda \boldsymbol{f})$ to within a multiplicative $(1+\delta)$ of its optimal value $\tilde{v}$ (in the smoothed smax potential) for the scaled demands $\lambda \boldsymbol{b}$, i.e.

$$\operatorname{smax}\left(\lambda \boldsymbol{C}^{-1}\boldsymbol{f}\right) + \operatorname{smax}\left(2\alpha\lambda \boldsymbol{R}\left(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f}\right)\right) \leq (1+\delta)\tilde{v}$$

$\tilde{v}$ is bounded by the second inequality of property 1 when noting that $\boldsymbol{C}^{-1} \in \mathbb{R}^{m\times m}$ for $m \leq \frac{1}{2}n^2$ and $\boldsymbol{R} \in \mathbb{R}^{n\times k}$ for $k \leq \frac{1}{2}n^2$: Let $\boldsymbol{f}_{\mathsf{opt}}$ be an optimal routing of $\boldsymbol{b}$. Then

$$\frac{1}{\lambda}\tilde{v} \leq \underbrace{\|\boldsymbol{C}^{-1}\boldsymbol{f}_{\mathsf{opt}}\|_\infty}_{=\mathsf{opt}_G(\boldsymbol{b})} + \frac{1}{\lambda}\log(2m) + \underbrace{2\alpha\|\boldsymbol{R}(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f}_{\mathsf{opt}})\|_\infty}_{=0} + \frac{1}{\lambda}\log(n^2)$$

and hence property 1 implies

$$\|\boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty + 2\alpha\|\boldsymbol{R}(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f})\|_\infty \leq \frac{1+\delta}{\lambda}\tilde{v} \leq (1+\delta)\left(\mathsf{opt}_G(\boldsymbol{b}) + \frac{4}{\lambda}\log n\right)$$

By definition of a congestion approximator, $\|\boldsymbol{R}\boldsymbol{b}\|_\infty \leq \mathsf{opt}_G(\boldsymbol{b}) \leq \alpha\|\boldsymbol{R}\boldsymbol{b}\|_\infty$. Hence if $\lambda \geq \frac{8\log n}{\epsilon\mathsf{opt}_G(\boldsymbol{b})} \geq \frac{8\log n}{\epsilon\alpha\|\boldsymbol{R}\boldsymbol{b}\|_\infty}$, then

$$\frac{1+\delta}{\lambda}\tilde{v} \leq (1+\delta)\left(\mathsf{opt}_G(\boldsymbol{b}) + \frac{1}{2}\epsilon\mathsf{opt}_G(\boldsymbol{b})\right) \leq (1+\epsilon)\mathsf{opt}_G(\boldsymbol{b})$$

for $\delta \leq \epsilon/4$. While we could aggressively set λ to $\frac{8\log n}{\epsilon\|\boldsymbol{R}\boldsymbol{b}\|_\infty}$ (because $\tilde{v} \geq \|\boldsymbol{R}\boldsymbol{b}\|_\infty$), this could increase the number of iterations until we achieve a good-enough solutions, and hence we begin the search for the 'correct' scaling λ at the lower bound $\frac{8\log n}{\epsilon\alpha\|\boldsymbol{R}\boldsymbol{b}\|_\infty}$ and interleave scaling of λ with gradient descent steps in such a way that performance is improved by almost a factor of α.[1]

A final consideration is the selection of an appropriate termination condition and the generation of a certificate in the form of a cut S in the sense of Theorem 2.1 without an expensive graph traversal, for which we will rely on the dual problem described in Section 2.3. In the proof of `AlmostRoute`, we will never make use of the above scaling intuition explicitly, but use it implicitly from the way in which scaling is interleaved with the descent steps to ensure $\Phi(\boldsymbol{f}) \geq \Omega(\epsilon^{-1}\log n)$, allowing the proof to work entirely in the formulation of the maximally-congested cut.

[1]Sherman claims that more careful scaling can additionally remove a factor of ϵ^{-1}, but the specifics of how this should be done are not given.

Algorithm 6.2 The Algorithm `AlmostRoute`; [She13]

1: **procedure** `AlmostRoute`$(G, \boldsymbol{b}, \epsilon)$
2: Initialise $\boldsymbol{f} \leftarrow \boldsymbol{0}$, let $\lambda \leftarrow \frac{8}{\epsilon\alpha\|\boldsymbol{Rb}\|_\infty} \log n$, scale $\boldsymbol{b} \leftarrow \lambda \boldsymbol{b}$
3: **repeat**
4: **while** $\lambda < \frac{16 \log n}{\epsilon\|\boldsymbol{Rb}\|_\infty}$ and $\Phi(\boldsymbol{f}) \leq 16\epsilon^{-1} \log n$ **do**
5: Scale $\lambda \leftarrow \frac{17}{16}\lambda$, $\boldsymbol{f} \leftarrow \frac{17}{16}\boldsymbol{f}, \boldsymbol{b} \leftarrow \frac{17}{16}\boldsymbol{b}$
6: Compute $\delta \leftarrow \|\boldsymbol{C}\nabla\Phi(\boldsymbol{f})\|_1$
7: For all $e \in E$, set $f_e \leftarrow f_e - \frac{\delta}{1+4\alpha^2}\text{sign}(\nabla\Phi(\boldsymbol{f})_e)c_e$
8: **until** $\delta \leq \epsilon/4$
9: Let $\boldsymbol{\psi} = \boldsymbol{R}^T \nabla \operatorname{smax}(2\alpha \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{Bf}))$ be vertex potentials
10: Let $\tilde{\boldsymbol{\psi}} = \|\boldsymbol{CB}^T\boldsymbol{\psi}\|_1^{-1}\boldsymbol{\psi}$ and compute $S \leftarrow$ `CongestedCut`$(G, \boldsymbol{b}, \tilde{\boldsymbol{\psi}})$
11: **return** $\lambda^{-1}\boldsymbol{f}$ and S

The resulting algorithm is stated in Algorithm 6.2.[2] There is some additional complexity in the way that the steps are chosen to accelerate the descent, it will become clear why this is so in the analysis with Lemma 6.3.

Lemma 6.3 ([She13]). *Every gradient step (line 7) except the last reduces the potential value by at least* $\Omega(\epsilon^2\alpha^{-2})$.

Proof. Let $\boldsymbol{h}$ be the step taken in line 7, i.e. $h_e = -\frac{\delta}{1+4\alpha^2}\text{sign}(\partial_e\Phi(\boldsymbol{f}))c_e$. We are interested in proving an upper bound on $\Phi(\boldsymbol{f} + \boldsymbol{h})$. To do so, the proof will follow the proof of the usual descent lemmas based on the fundamental theorem of calculus and Hölder's inequality, but then combine the result with the properties from Lemma 6.2 to achieve a tighter bound. The fundamental theorem of calculus provides

$$\Phi(\boldsymbol{f} + \boldsymbol{h}) = \Phi(\boldsymbol{f}) + \int_{\theta=0}^{1} \nabla\Phi(\boldsymbol{f}_\theta)^T \boldsymbol{h} d\theta \qquad \text{where } \boldsymbol{f}_\theta = \boldsymbol{f} + \theta\boldsymbol{h}$$

which we rewrite into

$$\Phi(\boldsymbol{f}) + \nabla\Phi(\boldsymbol{f})^T\boldsymbol{h} + \int_{\theta=0}^{1} \langle \nabla\Phi(\boldsymbol{f}_\theta) - \nabla\Phi(\boldsymbol{f}), \boldsymbol{h}\rangle d\theta$$

and proceed to upper-bound $\langle \nabla\Phi(\boldsymbol{f}_\theta) - \nabla\Phi(\boldsymbol{f}), \boldsymbol{h}\rangle$. Define for the sake of brevity the functions $d(\boldsymbol{x}, \boldsymbol{y}) = \nabla \operatorname{smax}(\boldsymbol{x}) - \nabla \operatorname{smax}(\boldsymbol{y})$ and $r(\boldsymbol{f}) = 2\alpha\boldsymbol{R}(\boldsymbol{b} - \boldsymbol{Bf})$. Then using $\langle \boldsymbol{x}, \boldsymbol{y}\rangle \leq \|\boldsymbol{x}\|_1\|\boldsymbol{y}\|_\infty$ from Hölder's inequality, compute

$$\begin{aligned}
\langle \nabla\Phi(\boldsymbol{f}_\theta) - \nabla\Phi(\boldsymbol{f}), \boldsymbol{h}\rangle &= \left\langle d(\boldsymbol{C}^{-1}\boldsymbol{f}_\theta, \boldsymbol{C}^{-1}\boldsymbol{f}), \boldsymbol{C}^{-1}\boldsymbol{h}\right\rangle + 2\alpha\left\langle d\big(r(\boldsymbol{f}), r(\boldsymbol{f}_\theta)\big), \boldsymbol{RBh}\right\rangle \\
&\leq \left\|d(\boldsymbol{C}^{-1}\boldsymbol{f}_\theta, \boldsymbol{C}^{-1}\boldsymbol{f})\right\|_1 \left\|\boldsymbol{C}^{-1}\boldsymbol{h}\right\|_\infty + 2\alpha \left\|d\big(r(\boldsymbol{f}), r(\boldsymbol{f}_\theta)\big)\right\|_1 \|\boldsymbol{RBh}\|_\infty
\end{aligned}$$

Now using property 4 from Lemma 6.2, bound this and simplify as

$$\begin{aligned}
\langle \nabla\Phi(\boldsymbol{f}_\theta) - \nabla\Phi(\boldsymbol{f}), \boldsymbol{h}\rangle &\leq \left\|\boldsymbol{C}^{-1}(\boldsymbol{f}_\theta - \boldsymbol{f})\right\|_\infty \left\|\boldsymbol{C}^{-1}\boldsymbol{h}\right\|_\infty + 4\alpha^2 \|\boldsymbol{RB}(\boldsymbol{f} - \boldsymbol{f}_\theta)\|_\infty \|\boldsymbol{RBh}\|_\infty \\
&= \theta \left\|\boldsymbol{C}^{-1}\boldsymbol{h}\right\|_\infty^2 + 4\alpha^2\theta \|\boldsymbol{RBh}\|_\infty^2 \\
&\leq \theta \left\|\boldsymbol{C}^{-1}\boldsymbol{h}\right\|_\infty^2 + 4\alpha^2\theta \left\|\boldsymbol{C}^{-1}\boldsymbol{h}\right\|_\infty^2 = \theta\frac{\delta^2}{1+4\alpha^2}
\end{aligned}$$

[2]Some minor adjustments to Sherman's algorithm have been made: The scaling condition ensures that the algorithm eventually stops scaling, and it is specified how the cut S is computed.

where the last line follows from the observation that $(\boldsymbol{C}^{-1}\boldsymbol{h})_e = \frac{\delta}{1+4\alpha^2}$ and $\|\boldsymbol{RBh}\|_\infty \leq \|\boldsymbol{C}^{-1}\boldsymbol{h}\|_\infty$. This is because $\boldsymbol{h}$ is a routing of the demands $\boldsymbol{Bh}$ and thus $\|\boldsymbol{RBh}\|_\infty \leq \mathsf{opt}_G(\boldsymbol{Bh}) \leq \|\boldsymbol{C}^{-1}\boldsymbol{h}\|_\infty$. Substituting this bound into the integral derived earlier and evaluating yields

$$\Phi(\boldsymbol{f}+\boldsymbol{h}) \leq \Phi(\boldsymbol{f}) + \nabla\Phi(\boldsymbol{f})^T\boldsymbol{h} + \frac{1}{2}\frac{\delta^2}{1+4\alpha^2}$$

Finally, $\nabla\Phi(\boldsymbol{f})^T\boldsymbol{h} = -\frac{\delta^2}{1+4\alpha^2}$ because

$$\begin{aligned}
\nabla\Phi(\boldsymbol{f})^T\boldsymbol{h} &= \sum_e -(\partial_e\Phi(\boldsymbol{f}))\mathrm{sign}(\partial_e\Phi(\boldsymbol{f}))c_e\frac{\delta}{1+4\alpha^2} \\
&= \sum_e -|\partial_e\Phi(\boldsymbol{f})|c_e\frac{\delta}{1+4\alpha^2} = -\|\boldsymbol{C}\nabla\Phi(\boldsymbol{f})\|_1\frac{\delta}{1+4\alpha^2} \\
&= -\frac{\delta^2}{1+4\alpha^2}
\end{aligned}$$

and thus our final bound for $\Phi(\boldsymbol{f}+\boldsymbol{h})$ is

$$\Phi(\boldsymbol{f}+\boldsymbol{h}) \leq \Phi(\boldsymbol{f}) - \frac{\delta^2}{2+8\alpha^2}$$

Noting that $\delta > \epsilon/4$ whenever a step is taken (except for the last step) completes the proof. □

Lemma 6.4 ([She13]). *Algorithm 6.2 performs at most $\mathcal{O}(\epsilon^{-3}\alpha^2\log n\log\alpha)$ gradient steps.*

Proof. After the initial scaling, $\Phi(\lambda\boldsymbol{f})$ has a value of at most $16\epsilon^{-1}\log n + 4\log n$ (by Lemma 6.2, property 1, and using that $\boldsymbol{R}$ has at most $\frac{1}{2}n^2$ rows), and whenever the scaling loop finishes, $\Phi(\lambda\boldsymbol{f}) \leq 17\epsilon^{-1}\log n$. In either case, by Lemma 6.3, after at most $\mathcal{O}(\frac{\epsilon^{-1}\log n}{\epsilon^2\alpha^{-2}}) = \mathcal{O}(\epsilon^{-3}\alpha^2\log n)$ gradient steps, the value of $\Phi(\lambda\boldsymbol{f})$ decreases below the scaling threshold again, or the algorithm terminates. Since λ is bounded from above such that the algorithm scales at most $\mathcal{O}(\log\alpha)$ times, there are at most $\mathcal{O}(\epsilon^{-3}\alpha^2\log n\log\alpha)$ gradient steps until λ stops scaling, at which point a further $\mathcal{O}(\epsilon^{-3}\alpha^2\log n)$ gradient steps would reduce the value of $\Phi(\lambda\boldsymbol{f})$ below a value of zero. This is a contradiction with the positivity of $\Phi(\lambda\boldsymbol{f})$, thus the algorithm must reach $\delta \leq \epsilon/4$ before then and terminate; the lemma follows. □

Lemma 6.5 ([She13]). *When Algorithm 6.2 terminates, it outputs a flow $\boldsymbol{f}$ and vertex-induced cut $S \subseteq V$ such that*

$$\|\boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty + 2\alpha\|\boldsymbol{R}(\boldsymbol{b}-\boldsymbol{Bf})\|_\infty \leq (1+\epsilon)\frac{|b_S|}{c_{S\leftrightarrow V\setminus S}} \leq (1+\epsilon)\mathsf{opt}_G(\boldsymbol{b})$$

Proof. Let $\boldsymbol{f}, \boldsymbol{b}$ be the flow as return by the algorithm, i.e. with any scaling undone, and let λ be its final value. The proof proceeds as follows: We use that $\tilde{\boldsymbol{\psi}}$ is a feasible solution of the maximally congested cut dual program for $\lambda\boldsymbol{b}$, and show that $\lambda\boldsymbol{b}^T\tilde{\boldsymbol{\psi}} \geq \frac{1}{1+\epsilon}\Phi(\lambda\boldsymbol{f})$. Since we are able to construct a cut S from $\tilde{\boldsymbol{\psi}}$ and the congestion of S is at most $\mathsf{opt}_G(\lambda\boldsymbol{b})$ by Lemma 2.2 (the max-flow min-cut theorem), this shows $\Phi(\lambda\boldsymbol{f}) \leq (1+\epsilon)\lambda\boldsymbol{b}^T\tilde{\boldsymbol{\psi}} \leq (1+\epsilon)\lambda\frac{|b_S|}{c_{S\leftrightarrow V\setminus S}} \leq (1+\epsilon)\mathsf{opt}_G(\lambda\boldsymbol{b})$. Then dividing by λ and using Lemma 6.2 will complete the proof.

First we establish that upon termination, we can assume $\Phi(\lambda \boldsymbol{f}) \geq 16e^{-1} \log n$ without loss of generality, despite this condition not necessarily being satisfied, either because λ reaches is upper bound or because the last step can decrease the potential below the scaling threshold: To remedy these issues, observe that by Lemma 6.3, the last step can only improve the solution, and hence we can prove correctness after undoing the last step without loss of generality. Moreover, by property 1 of Lemma 6.2, $\lambda^{-1}\Phi(\lambda \boldsymbol{f}) \geq \mathsf{opt}_G(\boldsymbol{b}) \geq \|\boldsymbol{R}\boldsymbol{b}\|_\infty$, and hence if λ ever reaches its upper bound, then it must hold $\Phi(\lambda \boldsymbol{f}) \geq 16\epsilon^{-1} \log n$ regardless.

With this out of the way, obtain a lower bound on $\lambda \boldsymbol{b}^T \tilde{\boldsymbol{\psi}}$ by computing

$$\begin{aligned}
\delta\Phi(\lambda \boldsymbol{f}) &\geq \|\boldsymbol{C}\nabla\Phi(\lambda \boldsymbol{f})\|_1 \|\lambda \boldsymbol{C}^{-1}\boldsymbol{f}\|_\infty \\
&\geq \nabla\Phi(\lambda \boldsymbol{f})^T \lambda \boldsymbol{f} && \text{(Hölder's inequality)} \\
&= \langle \nabla \operatorname{smax}(\lambda \boldsymbol{C}^{-1}\boldsymbol{f}), \lambda \boldsymbol{C}^{-1}\boldsymbol{f}\rangle - 2\alpha\langle \boldsymbol{R}^T \nabla \operatorname{smax}(2\alpha\lambda \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f})), \lambda \boldsymbol{B}\boldsymbol{f}\rangle \\
&= \langle \nabla \operatorname{smax}(\lambda \boldsymbol{C}^{-1}\boldsymbol{f}), \lambda \boldsymbol{C}^{-1}\boldsymbol{f}\rangle - 2\alpha\langle \boldsymbol{\psi}, \lambda \boldsymbol{B}\boldsymbol{f}\rangle && (\dagger)
\end{aligned}$$

At the same time, using Lemma 6.2, property 3,

$$\begin{aligned}
&\langle \nabla \operatorname{smax}(\lambda \boldsymbol{C}^{-1}\boldsymbol{f}), \lambda \boldsymbol{C}^{-1}\boldsymbol{f}\rangle + 2\alpha\langle \boldsymbol{\psi}, \lambda(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f})\rangle && (\ddagger) \\
&\quad = \langle \nabla \operatorname{smax}(\lambda \boldsymbol{C}^{-1}\boldsymbol{f}), \lambda \boldsymbol{C}^{-1}\boldsymbol{f}\rangle + \langle \nabla \operatorname{smax}(2\alpha\lambda \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f})), 2\alpha\lambda \boldsymbol{R}(\boldsymbol{b} - \boldsymbol{B}\boldsymbol{f})\rangle \\
&\quad\quad \geq \Phi(\lambda \boldsymbol{f}) - 4\log n \geq \Phi(\lambda \boldsymbol{f})(1 - \epsilon/4)
\end{aligned}$$

where we again use that $\boldsymbol{R}$ has at most $\frac{1}{2}n^2$ rows and that $\Phi(\lambda \boldsymbol{f}) \geq 16\epsilon^{-1} \log n$. Subtracting ($\dagger$) from ($\ddagger$) leaves only

$$2\alpha\lambda \boldsymbol{\psi}^T \boldsymbol{b} \geq \Phi(\lambda \boldsymbol{f})(1 - \epsilon/4 - \delta)$$

Now bound $\|\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1$ as follows:

$$\begin{aligned}
2\alpha\|\boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1 &= \|\nabla \operatorname{smax}(\lambda \boldsymbol{C}^{-1}\boldsymbol{f}) - \nabla \operatorname{smax}(\lambda \boldsymbol{C}^{-1}\boldsymbol{f}) + 2\alpha \boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1 \\
&\leq \|\nabla \operatorname{smax}(\lambda \boldsymbol{C}^{-1}\boldsymbol{f})\|_1 + \|\nabla \operatorname{smax}(\lambda \boldsymbol{C}^{-1}\boldsymbol{f}) - 2\alpha \boldsymbol{C}\boldsymbol{B}^T\boldsymbol{\psi}\|_1 \\
&\leq 1 + \delta
\end{aligned}$$

where we used Lemma 6.2, property 2, for the first of the two norms, and the definition of δ for the second. Hence with $\delta \leq \epsilon/4$ and $\epsilon \leq 1/2$,

$$\lambda \boldsymbol{b}^T \tilde{\boldsymbol{\psi}} \geq \Phi(\lambda \boldsymbol{f}) \frac{1 - \epsilon/4 - \delta}{1 + \delta} \geq \Phi(\lambda \boldsymbol{f}) \frac{1}{1 + \epsilon}$$

Constructing the cut S as in Algorithm 2.1 such that

$$\Phi(\lambda \boldsymbol{f}) \leq (1 + \epsilon)\lambda \boldsymbol{b}^T \tilde{\boldsymbol{\psi}} \leq (1 + \epsilon)\lambda \frac{|b_S|}{c_{S \leftrightarrow V \setminus S}} \leq (1 + \epsilon)\mathsf{opt}_G(\lambda \boldsymbol{b})$$

and dividing by λ together with property 1 from Lemma 6.2 completes the proof. □

Remark. Let us reflect again on the scaling operations: The fact that $\Phi(\lambda \boldsymbol{f}) \geq \Omega(\epsilon^{-1} \log n)$ is used to implicitly bound the additive error introduced by the use of smax – if $\Phi(\lambda \boldsymbol{f})$ were to be smaller, then the $4 \log n$ term would begin to dominate.

The scaling factor $\frac{17}{16}$ on the other hand is somewhat arbitrary. Can we do better by being more careful about how we scale? Unfortunately, the answer is no, at least not

with the construction of `AlmostRoute` as given here. Consider for example scaling by $(1+\eta)$ for some η fixed throughout the execution. The number of scaling iterations would be at most $\log \alpha / \log(1+\eta) \leq \mathcal{O}(\eta^{-1} \log \alpha)$ for $0 < \eta \leq \frac{1}{2}$, and the number of gradient steps until the scaling threshold is reached again is at most $\mathcal{O}(\eta \epsilon^{-3} \alpha^2 \log n)$. Note that the value of η cancels out, hence we modifying η will not lead to an improvement in asymptotic runtime.

Nevertheless, Sherman claims that more careful scaling can decrease the runtime to $\mathcal{O}(\epsilon^{-2} \cdots)$, but does not make clear how this would be done. By the above argument, it could be possible to find some optimal way of varying η between iterations, perhaps in combination with other modifications to the algorithm, in the hope of improving performance. ◇

6.2.1 Proof of Lemma 6.2

The first three properties of smax claimed in Lemma 6.2 are elementary:

Proof of Lemma 6.2, 1 – 3.

1. Let $|x_i| = \|\boldsymbol{x}\|_\infty$ and use $e^x \geq 0$:

$$|x_i| = \log\left(e^{|x_i|}\right) \leq \operatorname{smax}(\boldsymbol{x}) \leq \log\left(d \cdot 2e^{|x_i|}\right) \leq |x_i| + \log(2d)$$

   This shows the case for $\lambda = 1$; it is trivial to then apply scaling by λ.

2. The statement follows directly from the triangle inequality $|e^{x_i} - e^{-x_i}| \leq |e^{x_i}| + |e^{-x_i}|$.

3. By convexity, $\operatorname{smax}(\boldsymbol{x} - \boldsymbol{x}) \geq \operatorname{smax}(\boldsymbol{x}) - \nabla \operatorname{smax}(\boldsymbol{x})^T \boldsymbol{x}$. With $\operatorname{smax}(0) = \log(2d)$ this yields $\nabla \operatorname{smax}(\boldsymbol{x})^T \boldsymbol{x} \geq \operatorname{smax}(\boldsymbol{x}) - \log(2d)$ as desired. □

The fourth property, namely that $\|\nabla \operatorname{smax}(\boldsymbol{x}) - \nabla \operatorname{smax}(\boldsymbol{y})\|_1 \leq \|\boldsymbol{x} - \boldsymbol{y}\|_\infty$ for all $\boldsymbol{x}, \boldsymbol{y}$, requires some more elaborate insight from convex optimisation ([Bec17], chapter 5). In general, this property is referred to as L-smoothness:

Definition 6.1 (L-smoothness; [Bec17])**.** A function $f : \mathbb{R}^n \to \mathbb{R}$ is *L-smooth* (in $D \subseteq \mathbb{R}^n$), for $L \geq 0$, if it is differentiable (in D) and satisfies

$$\|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\|_* \leq L\|\boldsymbol{x} - \boldsymbol{y}\|$$

for all $\boldsymbol{x}, \boldsymbol{y}$ (in D), where $\|\cdot\|_*$ is the dual norm of $\|\cdot\|$.[3] ◇

Note that the 1-norm is the Hölder-congruent norm to the supremum norm and thus its dual. The key observations for the purpose of proving 1-smoothness of smax with respect to the supremum norm can be summarised in the following lemma, which takes some shortcuts through the more general theory developed in Beck [Bec17] (in fact, this condition is also necessary):

Lemma 6.6 (Sufficient Condition for L-Smoothness; [Bec17])**.** *If f is convex and $\langle \boldsymbol{d}, \nabla^2 f(\boldsymbol{x}) \cdot \boldsymbol{d}\rangle \leq L\|\boldsymbol{d}\|^2$ for all $\boldsymbol{x}, \boldsymbol{d} \in \mathbb{R}^n$, then f is L-smooth on $\mathbb{R}^n$.*

We defer the proof for later and focus first on applying the lemma to smax.

[3] That is, $\|\boldsymbol{x}\|_* = \max_{\boldsymbol{v}}\{\langle \boldsymbol{x}, \boldsymbol{v}\rangle \mid \|\boldsymbol{v}\| = 1\}$.

Proof of Lemma 6.2, 4. We prove a somewhat stronger result, namely 1-smoothness of the unsymmetric LogSumExp function given by $S(\boldsymbol{x}) = \log\left(\sum_i e^{x_i}\right)$. Instantiating $S(\cdot)$ with $[\boldsymbol{x}, -\boldsymbol{x}]$ recovers the symmetric function, so this comes without loss of generality. One may compute that the Hessian of S is given by

$$\nabla^2 S(\boldsymbol{x}) = \operatorname{diag}(\sigma(\boldsymbol{x})) - \sigma(\boldsymbol{x})\sigma(\boldsymbol{x})^T \qquad\qquad \sigma(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Then compute

$$\begin{aligned}\langle \boldsymbol{d}, \nabla^2 S(\boldsymbol{x}) \cdot \boldsymbol{d} \rangle &= \boldsymbol{d}^T \operatorname{diag}(\sigma(\boldsymbol{x}))\boldsymbol{d} - (\sigma(\boldsymbol{x})^T \boldsymbol{d})^2 \\ &\leq \boldsymbol{d}^T \operatorname{diag}(\sigma(\boldsymbol{x}))\boldsymbol{d} \\ &\leq \|\sigma(\boldsymbol{x})\|_\infty \|\boldsymbol{d}\|_\infty^2 \leq \|\boldsymbol{d}\|_\infty^2\end{aligned}$$

which by Lemma 6.6 proves 1-smoothness of S and thereby also smax. □

Finally, let us prove Lemma 6.6:

Proof of Lemma 6.6; based on [Bec17]. By Taylor's theorem, because f is differentiable there exists for any $\boldsymbol{x}, \boldsymbol{d} \in \mathbb{R}^n$ a $\boldsymbol{\xi} \in \mathbb{R}^n$ such that

$$f(\boldsymbol{x}+\boldsymbol{d}) = f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{d} + \frac{1}{2}\langle \boldsymbol{d}, \nabla^2 f(\boldsymbol{\xi}) \cdot \boldsymbol{d} \rangle \leq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{d} + \frac{L}{2}\|\boldsymbol{d}\|^2$$

where we make use of the assumption that $\langle \boldsymbol{d}, \nabla^2 f(\boldsymbol{\xi}) \cdot \boldsymbol{d} \rangle \leq L\|\boldsymbol{d}\|^2$. We continue by proving that any f satisfying this inequality is L-smooth. For notational convenience, define the Bregman distance[4]

$$D_f(\boldsymbol{x}, \boldsymbol{d}) = f(\boldsymbol{x}+\boldsymbol{d}) - f(\boldsymbol{x}) - \nabla f(\boldsymbol{x})^T \boldsymbol{d}$$

From the inequality, it must hold that $D_f(\boldsymbol{x}, \boldsymbol{d}) \leq \frac{L}{2}\|\boldsymbol{d}\|^2$ for all $\boldsymbol{x}, \boldsymbol{d}$. Because f is convex, we also have $f(\boldsymbol{x}+\boldsymbol{d}) \geq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{d}$ and hence $D_f(\boldsymbol{x}, \boldsymbol{d}) \geq 0$, for all $\boldsymbol{d}$. In other words, we have the property that for any $\boldsymbol{x}, \boldsymbol{d}$, $0 \leq D_f(\boldsymbol{x}, \boldsymbol{d}) \leq \frac{L}{2}\|\boldsymbol{d}\|^2$.

To prove L-smoothness, we need to obtain terms of the form $\nabla f(\boldsymbol{x})^T \boldsymbol{v}$ and $\nabla f(\boldsymbol{y})^T \boldsymbol{v}$ for $\boldsymbol{v}$ independent of $\boldsymbol{x}, \boldsymbol{y}$ so that we can obtain statements about the dual norm of $\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})$; however all expressions so far have $\boldsymbol{v}$ dependant on $\boldsymbol{x}$ or $\boldsymbol{d}$, which we can fix to e.g. $\boldsymbol{d} = \boldsymbol{y} - \boldsymbol{x}$. To get around this, we additionally shift $\boldsymbol{d}$ by some $\boldsymbol{\delta}$ under our control. Formally, fix any $\boldsymbol{x}, \boldsymbol{d}$ and write for some $\boldsymbol{\delta} \in \mathbb{R}^n$

$$D_f(\boldsymbol{x}, \boldsymbol{d}+\boldsymbol{\delta}) = f(\boldsymbol{x}+\boldsymbol{d}+\boldsymbol{\delta}) - f(\boldsymbol{x}) - \langle \nabla f(\boldsymbol{x}), \boldsymbol{d}+\boldsymbol{\delta} \rangle$$

Bounding $f((\boldsymbol{x}+\boldsymbol{d})+\boldsymbol{\delta})$ through the inequality at the beginning of this proof yields

$$\begin{aligned}D_f(\boldsymbol{x}, \boldsymbol{d}+\boldsymbol{\delta}) &\leq f(\boldsymbol{x}+\boldsymbol{d}) - f(\boldsymbol{x}) - \langle \nabla f(\boldsymbol{x}), \boldsymbol{d}+\boldsymbol{\delta} \rangle + \langle \nabla f(\boldsymbol{x}+\boldsymbol{d}), \boldsymbol{\delta} \rangle + \frac{L}{2}\|\boldsymbol{\delta}\|^2 \\ &= D_f(\boldsymbol{x}, \boldsymbol{d}) + \langle \nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x}), \boldsymbol{\delta} \rangle + \frac{L}{2}\|\boldsymbol{\delta}\|^2\end{aligned}$$

[4] The Bregman distance is defined in the context of strictly convex functions, so this is a slight abuse of terminology in the interest of providing some context.

With the very specific choice of $\delta = -\frac{1}{L}\|\nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x})\|_* \boldsymbol{v}$ where $\boldsymbol{v}$ is chosen such that $\|\boldsymbol{v}\| = 1$ and $\langle \nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x}), \boldsymbol{v}\rangle = \|\nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x})\|_*$ (i.e., $\boldsymbol{v}$ is the vector defines the dual norm of $\nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x})$), we obtain

$$\begin{aligned} 0 &\le D_f(\boldsymbol{x}, \boldsymbol{d}+\delta) \\ &\le D_f(\boldsymbol{x}, \boldsymbol{d}) + \frac{1}{L}\|\nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x})\|_* \langle \nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x}), \boldsymbol{v}\rangle \\ &\qquad + \frac{1}{2L}\|\nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x})\|_*^2 \\ &= D_f(\boldsymbol{x}, \boldsymbol{d}) - \frac{1}{2L}\|\nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x})\|_*^2 \end{aligned}$$

Hence after rearranging terms and applying the upper bound on $D_f(\boldsymbol{x}, \boldsymbol{d})$,

$$\frac{1}{2L}\|\nabla f(\boldsymbol{x}+\boldsymbol{d}) - \nabla f(\boldsymbol{x})\|_*^2 \le D_f(\boldsymbol{x}, \boldsymbol{d}) \le \frac{L}{2}\|\boldsymbol{d}\|^2$$

Using $\boldsymbol{d} = \boldsymbol{y} - \boldsymbol{x}$, multiplying by $2L$ and taking the square root completes the proof.□

6.3 Discussion

`AlmostRoute` scales with ϵ^{-3}, which is significantly worse than the ϵ^{-2} or even e^{-1} that convex optimisation solvers typically take. The upside is that the bound does not depend on any initial guess. Sidford and Tian [ST18] design a coordinate-descent based solver that terminates after time

$$\mathcal{O}\left(m^{1+o(1)} + \frac{\sqrt{n}\|\boldsymbol{C}^{-1}\boldsymbol{f}_{\mathsf{opt}}\|_2}{\epsilon}\,\mathsf{polylog}(n)\right) \le \mathcal{O}\left(m^{1+o(1)} + \frac{\sqrt{nm}}{\epsilon}\,\mathsf{polylog}(n)\right)$$

in the standard RAM model, where $\boldsymbol{f}_{\mathsf{opt}}$ is a flow that routes $\boldsymbol{b}$ optimally and we assume a polylogarithmic congestion approximator. When $\|\boldsymbol{C}^{-1}\boldsymbol{f}_{\mathsf{opt}}\|_2 \le \frac{m^{1+o(1)}}{\sqrt{n}DB}$, the coordinate-descent based solver vastly improves upon `AlmostRoute`. In general however, coordinate descent relies on random access to $\boldsymbol{f}$, hence the algorithm will not always be cache-efficient a priori. More involved analysis of Sidford and Tian's solver might however show that it can be implemented cache-efficiently with some modification.

In later work, Sherman [She16] notes that one can employ the multiplicative weights update method to obtain a solver that scales with ϵ^{-2}. His construction abstracts away the underlying details, but if we interpret his claim that the iterations are 'simple' as requiring only $\mathcal{O}(\mathsf{scan}(m))$ I/Os apart from applying $\boldsymbol{R}$ and $\boldsymbol{R}^T$, then the result should carry over immediately to the EM model.

Chapter 7

Conclusion and Future Work

Combining `Route` with the congestion approximator built around the low-stretch spanning tree from Theorem 4.1 finally yields an asymptotically cache-efficient algorithm for solving the maximum flow problem:

Theorem 7.1. *For any constant $k \geq 1$ and any $\epsilon \leq 1/2$, the maximum flow problem can approximated to within a factor of $(1-\epsilon)$ from optimal on undirected graphs with polynomially bounded capacities in $\mathcal{O}(\mathsf{scan}(m^{1+1/k})\epsilon^{-3})$ I/Os. The algorithm returns a cut $S \subseteq V$ that certifies the approximate optimality.*

Proof. Use the congestion approximator from Corollary 5.21, and note that for constant k and large enough m, $\mathcal{O}(\log^2 n\, \mathsf{sort}(m)) \leq \mathcal{O}(\mathsf{scan}(m^{1+1/k}))$. Combine this with `ComputeR` and `ComputeR`^{T} for $\eta = n^{1/2k}$ in Theorem 6.1 and Theorem 6.2, and use that the lower-order terms are dominated by $m^{1/2k}$ asymptotically. □

For expander graphs, Lemma 5.24 even yields

Theorem 7.2. *The maximum flow problem can approximated to within $(1-\epsilon)$ on undirected graphs in $\mathcal{O}(\lambda_2^{-2} \log \lambda_2^{-1} \log^2 n \epsilon^{-3}\, \mathsf{scan}(m))$, where λ_2 is the second-largest eigenvalue of the graphs's Laplacian.*

Unfortunately, as already discussed for the low-stretch spanning tree in Theorem 4.1, the term $\exp(\mathcal{O}(\sqrt{\log n \log\log n}))$ hidden in the bound for general graphs scales poorly with its constant factors. The algorithm will only yield a speedup over more naive methods when n is astronomically large.

7.1 Towards a Practical Algorithm

Is the entire framework discussed in this thesis doomed to be impractical? Assume we had some algorithm that computes a spanning tree of optimal average stretch $\mathcal{O}(\log n)$ in only $\mathcal{O}(\mathsf{sort}(m))$ I/Os, and that we could also sparsify our graphs to only $\mathcal{O}(n \log n)$ edges. Running `Route` would then take 'only' $\tilde{\mathcal{O}}(\epsilon^{-3} \log^{8k+2} n\, \mathsf{scan}(m^{1+1/k}))$ I/Os for any constant k, while we could construct $\boldsymbol{R}$ in $\tilde{\mathcal{O}}(\log^7 n\, \mathsf{scan}(n^{1+1/k}))$ I/Os after initial sparsification of G (we let $\tilde{\mathcal{O}}$ hide lower-order terms). The exponents in the polylogarithmic terms are still substantial, but likely subject to future improvement: (i) `AlmostRoute` scales with α^2. Even if this cannot be reduced to α, this still implies that any improvements to the term α will count double towards improving the overall

performance. Moreover, (ii) throwing away a substantial portion of the sliced trees while increasing the embedding factor α by $\mathcal{O}(\beta \operatorname{polylog}(n))$ in each iteration is a somewhat primitive way of working around the large number of trees generated by the flow packing. An improved procedure for generating the convex combination of trees might be able to do away with this additional increase of α, thus improving the convergence of `AlmostRoute` while additionally decreasing the number of queries to the LSST algorithm.

While using flow packing for generating the convex combination is an elegant reduction, it comes at the cost of requiring us to find embedding flows of the trees into G to update the flow packing weights $\boldsymbol{w}$. This was the reason for using low-stretch *spanning* trees as part of the flow packing oracle. Indeed, Lemma 5.9, where we relate the low-stretch trees to flow packing, explicitly requires an *identity* embedding flow, because it requires the embedding congestions to update the weights of the flow packing procedure. The identity embedding flow only exists for spanning trees, but finding low-stretch spanning trees is much harder than finding arbitrary low-stretch trees.

This should seem unnecessarily restrictive: the congestion approximator $\boldsymbol{R}$ only has to approximate the *value* of the cuts in G, but not their structure. Hence it does not make sense for us to restrict ourselves to *spanning* trees in the vital part of the construction; there must exist some better method for packing a convex combination of cut-approximating trees. As a first step towards this, it might be possible to conceive a way of quickly building a cut-preserving *complete* graph on V, such that any spanning tree on the complete graph is actually an arbitrary tree on V, and analyse the flow packing in this setting.

We could also consider the possibilities of using *projected trees* [BEL20]. These are trees on a superset of V, where each vertex of V maps to a set of vertices of the tree, and every edge of the tree likewise maps back to an edge in G. Thus we can easily obtain embedding flows of these trees into G by using the mapping of the edges, which allows us to use the flow packing routine to generate a convex combination of these trees. But now have to deal with there being multiple tree vertices for each $v \in V$ across which we have to distribute the demands b_v when approximating congestions. Since we must ultimately build a *linear* congestion approximator $\boldsymbol{R}$, we cannot do this depending on the demands $\boldsymbol{b}$, but must devise some allocation scheme that applies equally well to any demands $\boldsymbol{b}$. This is challenging because the definition of stretch in this case is that for any $\{u, v\} \in E$, the exist some representatives of u, v in the projected tree between which the path is short, but in order to have some constant scheme of allocating the demands, we would require the paths between *all* representatives of u, v to be short.

Nevertheless, the projected tree is sufficiently close to our desired tree (since it is a low-stretch tree that is efficiently constructable and easily embeds into G) that there might be some method of making this construction work. One would presumably need to modify the flow packing to account for the allocation scheme that distributes $\boldsymbol{b}$ over the vertices of the tree.

Ultimately, we can hope that future insight will be able to do away with or drastically improve the flow packing procedure, substituting it for something more directly applicable that does not require us to have embedding flows of our trees into G. This hypothetical scheme might then also produce fewer graphs in the decomposition, which will most likely also lead to more practical exponents in the polylogarithmic terms.

As a last resort, we can replace all of Madry's decomposition (Chapter 5) with a

different congestion approximation scheme. Räcke, Shah and Täubig [RST14] show how to compute a *single* tree in almost-linear time $\mathcal{O}(m^{1+o(1)})$ that $\mathcal{O}(\log^4 n)$-embeds into G.[1] The algorithm relies internally on computing approximate maximum flows. In a surprising result, Peng [Pen16] resolves this 'chicken-and-egg situation' using a circular reduction on recursively *larger* graphs by interleaving with *ultra-sparsifiers* [ST14; Kol+10] at the right moment. Multiple constructions of such ultra-sparsifiers have been found, but these again rely on low-stretch spanning trees. Blelloch et al. [Ble+13] show however that sparse low-stretch subgraphs suffice for their parallel construction of an ultra-sparsifier, hence this problem should be solvable in a cache-efficient manner. Their construction is based on a modification of the low-stretch spanning tree algorithm discussed in Section 4.1, improving the average stretch from $\exp(\mathcal{O}(\sqrt{\log n \log\log n}))$ to $\mathsf{polylog}(n)$ for their low-stretch subgraphs. Unfortunately, the exponents in the polylogarithm are still considerable. Whether their approach would make Peng's circular reduction a practical and cache-efficient algorithm would remain to be seen.

Finally, we note that related maximum flow algorithms, namely those of Kelner et al. [Kel+14] and Kyng et al. [Kyn+19], suffer from the same difficulties that we encountered in this thesis: In the case of Kelner et al., we require an *oblivious routing scheme* on G, which is essentially a congestion approximator that additionally preserves the structure of G, i.e. constructing an oblivious routing scheme is at least as hard as constructing $\boldsymbol{R}$. Kyng et al. observe that a *non-linear* congestion approximator will inherently lead to better performance of their version of `AlmostRoute`, and show how to construct and use what they call an *adaptive preconditioner*, which can be thought of as an non-linear congestion approximator. Their construction again revolves around a low-stretch spanning tree to build a modified ultra-sparsifier with additional guarantees.

All of this points to the following conclusion: Either an improved algorithm for low-stretch spanning trees is found, presumably making all related algorithms immediately cache-efficient, or the decomposition is improved to not require *spanning* trees, making the maximum flow problem cache-efficient in practice but leaving related problems open. The only known algorithms for constructing low-stretch spanning trees are those of Alon et al. [Alo+95], Elkin et al. [Elk+05], and Abraham, Bartal and Neiman [ABN08], of which only the first has a cache-efficient counterpart. Entirely new methods might be needed to make progress in the external memory (and parallel or distributed) setting.

7.2 Decomposing into Non-Trees

If computing low-stretch spanning trees is hard, can we decompose into graphs that are not trees? We answer this question partially in the negative by showing that the combinatorial formulation of a congestion approximator $\boldsymbol{R}$ as a combination of simple graphs on which we perform optimal routing exists only for trees. This is fundamentally because we require $\boldsymbol{R}$ to be linear. Consider again Figure 7.1, as previously given in Chapter 5. For a linear operator $\boldsymbol{R}$, $\boldsymbol{Rb} - \boldsymbol{Rb}$ is zero, but if we add the routing of $-\boldsymbol{b}$ in a cyclic graph to that of $\boldsymbol{b}$, we might obtain a non-zero flow. Can we maybe select the flows according to some clever scheme that avoids this issue? It turns out that this is impossible for cyclic graphs.

[1] This ostensibly contradicts our lower bound from Lemma 5.7. Recall however that the bound discusses identity embedding flows; the scheme of Räcke, Shah and Täubig does not rely on identity embeddings.

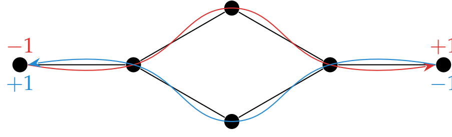

Figure 7.1: Non-Linearity of Optimal Routing

In more precise terms, if $\boldsymbol{Rb}$ is a flow resulting from routing $\boldsymbol{b}$ optimally on some combination of graphs, then these graphs must be trees, or $\boldsymbol{R}$ cannot be linear.

Lemma 7.1. *Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be some function that maps valid demands $\boldsymbol{b}$ to optimal flows $\boldsymbol{f}$ that route $\boldsymbol{b}$ with minimal congestion in some cyclic graph. Then f cannot be a linear function.*

Proof. Consider the n-vertex cycle, and let $\langle u, v, w \rangle$ be any path in this cycle. Let $\boldsymbol{b}^{(1)}$ be the demands given by $b_u = -1, b_v = 1$, and zero everywhere else. There is a unique optimal routing of $\boldsymbol{b}^{(1)}$ in the cycle of congestion one-half. Now repeat the same process for $\boldsymbol{b}^{(2)}$ being $b_v = -1$, $b_w = 1$, and zero everywhere else.

If we consider the added demands $\boldsymbol{b} = \boldsymbol{b}^{(1)} + \boldsymbol{b}^{(2)}$, then there still exists an optimal routing of congestion one-half, but the flow $\boldsymbol{f} = f(\boldsymbol{b}^{(1)}) + f(\boldsymbol{b}^{(2)})$ has congestion one: The flow on the path $\langle u, v, w \rangle$ is zero, whereas it is one on the remaining edges of the cycle. Thus optimal routing in cyclic graphs cannot be linear. □

This implies that we can only construct congestion approximators that have a graph-theoretic interpretation as routing flows optimally in some simpler graph, when these simpler graphs are trees.

Less pessimistically, if we want to use combinatorial congestion approximators that are not composed of trees, then we cannot use optimal routing on these approximators, but must at least interleave with other schemes such that routing only occurs on the acyclic parts of the decomposition. Alternatively, we could consider modifying the `AlmostRoute` solver to handle non-linear congestion approximators, following Kyng et al. [Kyn+19]. However, this closely marries the congestion approximator to the solver, whereas we would ideally keep both algorithms independent to provide a more general framework.

7.3 Handling Directed Graphs

Linear congestion approximators can only exist on undirected graphs, since we require that $\boldsymbol{R}(-\boldsymbol{b}) = -\boldsymbol{Rb}$. Can we nevertheless use our algorithm for approximate undirected max-flow to compute approximate maximum flows on directed graphs? Madry [Mad11] gives a reduction from directed to undirected maximum flow. Unfortunately, if we modify his construction to handle the approximate case, we see that we would in general need a prohibitively tight approximation ratio ϵ to obtain good results:

Lemma 7.2 (Directed to Undirected Max-Flow; [Mad11])**.** *Given an algorithm `MaxFlow` that computes a $(1-\epsilon)$-maximum flow on an undirected, capacitated graph G in $T(n, m, U)$ I/Os, where $U = \max c_e / \min c_e$, there exists an algorithm `DirectedMaxFlow` that computes flow on any* directed*, capacitated $\vec{G}$ in $T(\vec{n}, \mathcal{O}(\vec{m}), \vec{n} \cdot \vec{U}) + \mathsf{sort}(\vec{m})$ I/Os, such that the flow*

has value $\vec{v}$ at least

$$\vec{v} \geq (1-\epsilon)\mathsf{opt}_{\vec{G}}(\vec{G}) - \frac{\epsilon}{2}\sum_{e\in\vec{E}}\vec{c}_e$$

See Madry for a proof in the exact case; the modification to the EM model and approximate maximum flows is straightforward.

It is surprising that such a reduction exists at all, and this motivates the speculation that improved reductions may eventually be designed, even though current methods do not indicate how this might be accomplished.

References

[ABN08] Ittai Abraham, Yair Bartal and Ofer Neiman. 'Nearly Tight Low Stretch Spanning Trees'. In: *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, Oct. 2008. DOI: 10.1109/focs.2008.62. URL: https://doi.org/10.1109%2Ffocs.2008.62.

[ABT04] Lars Arge, Gerth Stølting Brodal and Laura Toma. 'On external-memory MST, SSSP and multi-way planar graph separation'. In: *Journal of Algorithms* 53.2 (Nov. 2004), pp. 186–206. DOI: 10.1016/j.jalgor.2004.04.001. URL: https://doi.org/10.1016/j.jalgor.2004.04.001.

[ABW02] Abello, Buchsbaum and Westbrook. 'A Functional Approach to External Graph Algorithms'. In: *Algorithmica* 32.3 (Mar. 2002), pp. 437–458. DOI: 10.1007/s00453-001-0088-5. URL: https://doi.org/10.1007/s00453-001-0088-5.

[ACZ12] Deepak Ajwani, Adan Cosgaya-Lozano and Norbert Zeh. 'A Topological Sorting Algorithm for Large Graphs'. In: *ACM J. Exp. Algorithmics* 17 (Sept. 2012). ISSN: 1084-6654. DOI: 10.1145/2133803.2330083. URL: https://doi.org/10.1145/2133803.2330083.

[AHK12] Sanjeev Arora, Elad Hazan and Satyen Kale. 'The Multiplicative Weights Update Method: a Meta-Algorithm and Applications'. In: *Theory Comput.* 8.1 (2012), pp. 121–164. DOI: 10.4086/toc.2012.v008a006. URL: https://doi.org/10.4086/toc.2012.v008a006.

[Alo+95] Noga Alon et al. 'A Graph-Theoretic Game and Its Application to the k-Server Problem'. In: *SIAM J. Comput.* 24.1 (1995), pp. 78–100. DOI: 10.1137/S0097539792224474. URL: https://doi.org/10.1137/S0097539792224474.

[Arg+08] Lars Arge et al. 'Fundamental parallel algorithms for private-cache chip multiprocessors'. In: *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*. Ed. by Friedhelm Meyer auf der Heide and Nir Shavit. ACM, 2008, pp. 197–206. DOI: 10.1145/1378533.1378573. URL: https://doi.org/10.1145/1378533.1378573.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. 'The Input/Output Complexity of Sorting and Related Problems'. In: *Commun. ACM* 31.9 (1988), pp. 1116–1127. DOI: 10.1145/48529.48535. URL: https://doi.org/10.1145/48529.48535.

[Bar98] Yair Bartal. 'On Approximating Arbitrary Metrices by Tree Metrics'. In: *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*. Ed. by Jeffrey Scott Vitter. ACM, 1998, pp. 161–168. DOI: 10.1145/276698.276725. URL: https://doi.org/10.1145/276698.276725.

[Bec+19] Ruben Becker et al. 'Distributed Algorithms for Low Stretch Spanning Trees'. In: *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*. Ed. by Jukka Suomela. Vol. 146. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 4:1–4:14. DOI: 10.4230/LIPIcs.DISC.2019.4. URL: https://doi.org/10.4230/LIPIcs.DISC.2019.4.

[Bec17] Amir Beck. *First-Order Methods in Optimization*. Philadelphia: Society for Industrial & Applied Mathematics (SIAM) and Mathematical Optimization Society (MOS), 2017. ISBN: 9781611974997.

[BEL20] Ruben Becker, Yuval Emek and Christoph Lenzen. 'Low Diameter Graph Decompositions by Approximate Distance Computation'. In: *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*. Ed. by Thomas Vidick. Vol. 151. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 50:1–50:29. DOI: 10.4230/LIPIcs.ITCS.2020.50. URL: https://doi.org/10.4230/LIPIcs.ITCS.2020.50.

[Bes+20] Maciej Besta et al. 'Log(Graph): A Near-Optimal High-Performance Graph Representation'. In: *CoRR* abs/2010.15879 (2020). arXiv: 2010.15879. URL: https://arxiv.org/abs/2010.15879.

[BK15] András A. Benczúr and David R. Karger. 'Randomized Approximation Schemes for Cuts and Flows in Capacitated Graphs'. In: *SIAM Journal on Computing* 44.2 (Jan. 2015), pp. 290–319. DOI: 10.1137/070705970. URL: https://doi.org/10.1137%2F070705970.

[Ble+13] Guy E. Blelloch et al. 'Nearly-Linear Work Parallel SDD Solvers, Low-Diameter Decomposition, and Low-Stretch Subgraphs'. In: *Theory of Computing Systems* 55.3 (Mar. 2013), pp. 521–554. DOI: 10.1007/s00224-013-9444-5. URL: https://doi.org/10.1007%2Fs00224-013-9444-5.

[Bor+20] Glencora Borradaile et al. 'Low-Stretch Spanning Trees of Graphs with Bounded Width'. In: *17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22-24, 2020, Tórshavn, Faroe Islands*. Ed. by Susanne Albers. Vol. 162. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 15:1–15:19. DOI: 10.4230/LIPIcs.SWAT.2020.15. URL: https://doi.org/10.4230/LIPIcs.SWAT.2020.15.

[Bra+21] Jan van den Brand et al. 'Minimum cost flows, MDPs, and ℓ_1-regression in nearly linear time for dense instances'. In: *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*. Ed. by Samir Khuller and Virginia Vassilevska Williams. ACM, 2021, pp. 859–869. DOI: 10.1145/3406325.3451108. URL: https://doi.org/10.1145/3406325.3451108.

[Bro+04] Gerth Stølting Brodal et al. 'Cache-Oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths'. In: *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*. Ed. by Torben Hagerup and Jyrki Katajainen. Vol. 3111. Lecture Notes in Computer Science. Springer, 2004, pp. 480–492. DOI: 10.1007/978-3-540-27810-8_41. URL: https://doi.org/10.1007/978-3-540-27810-8%5C_41.

[BS05] David A. Bader and Vipin Sachdeva. 'A Cache-Aware Parallel Implementation of the Push-Relabel Network Flow Algorithm and Experimental Evaluation of the Gap Relabeling Heuristic'. In: *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems, September 12-14, 2005 Imperial Palace Hotel, Las Vegas, Nevada, USA*. Ed. by Michael J. Oudshoorn and Sanguthevar Rajasekaran. ISCA, 2005, pp. 41–48.

[BSS14] Joshua D. Batson, Daniel A. Spielman and Nikhil Srivastava. 'Twice-Ramanujan Sparsifiers'. In: *SIAM Rev.* 56.2 (2014), pp. 315–334. DOI: 10.1137/130949117. URL: https://doi.org/10.1137/130949117.

[Chr+11] Paul Christiano et al. 'Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs'. In: *Proceedings of the 43rd annual ACM symposium on Theory of computing - STOC '11*. ACM Press, 2011. DOI: 10.1145/1993636.1993674. URL: https://doi.org/10.1145%2F1993636.1993674.

[CL06] Fan Chung and Linyuan Lu. 'Concentration Inequalities and Martingale Inequalities: A Survey'. In: 3.1 (Jan. 2006), pp. 79–127. DOI: 10.1080/15427951.2006.10129115. URL: https://doi.org/10.1080/15427951.2006.10129115.

[CM89] Joseph Cheriyan and S. N. Maheshwari. 'Analysis of Preflow Push Algorithms for Maximum Network Flow'. In: *SIAM J. Comput.* 18.6 (1989), pp. 1057–1086. DOI: 10.1137/0218072. URL: https://doi.org/10.1137/0218072.

[DB08] Andrew Delong and Yuri Boykov. 'A Scalable graph-cut algorithm for N-D grids'. In: *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008), 24-26 June 2008, Anchorage, Alaska, USA*. IEEE Computer Society, 2008. DOI: 10.1109/CVPR.2008.4587464. URL: https://doi.org/10.1109/CVPR.2008.4587464.

[DBS21] Laxman Dhulipala, Guy E. Blelloch and Julian Shun. 'Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable'. In: *ACM Trans. Parallel Comput.* 8.1 (2021), 4:1–4:70. DOI: 10.1145/3434393. URL: https://doi.org/10.1145/3434393.

[Dem+18] Erik D. Demaine et al. 'Fine-grained I/O Complexity via Reductions: New Lower Bounds, Faster Algorithms, and a Time Hierarchy'. In: *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*. Ed. by Anna R. Karlin. Vol. 94. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 34:1–34:23. DOI: 10.4230/LIPIcs.ITCS.2018.34. URL: https://doi.org/10.4230/LIPIcs.ITCS.2018.34.

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer, 1986. ISBN: 978-1-4613-8645-2. DOI: 10.1007/978-1-4613-8643-8. URL: https://doi.org/10.1007/978-1-4613-8643-8.

[Din06] Yefim Dinitz. 'Dinitz' Algorithm: The Original Version and Even's Version'. In: *Theoretical Computer Science, Essays in Memory of Shimon Even*. Ed. by Oded Goldreich, Arnold L. Rosenberg and Alan L. Selman. Vol. 3895. Lecture Notes in Computer Science. Springer, 2006, pp. 218–240. DOI: 10.1007/11685654_10. URL: https://doi.org/10.1007/11685654%5C_10.

[Elk+05] Michael Elkin et al. 'Lower-stretch spanning trees'. In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing - STOC '05*. ACM Press, 2005. DOI: 10.1145/1060590.1060665. URL: https://doi.org/10.1145%2F1060590.1060665.

[Fei+04] Joan Feigenbaum et al. 'On Graph Problems in a Semi-streaming Model'. In: *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*. Ed. by Josep Díaz et al. Vol. 3142. Lecture Notes in Computer Science. Springer, 2004, pp. 531–543. DOI: 10.1007/978-3-540-27836-8_46. URL: https://doi.org/10.1007/978-3-540-27836-8%5C_46.

[Fri+99] Matteo Frigo et al. 'Cache-Oblivious Algorithms'. In: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. IEEE Computer Society, 1999, pp. 285–298. DOI: 10.1109/SFFCS.1999.814600. URL: https://doi.org/10.1109/SFFCS.1999.814600.

[FRT04] Jittat Fakcharoenphol, Satish Rao and Kunal Talwar. 'A tight bound on approximating arbitrary metrics by tree metrics'. In: *J. Comput. Syst. Sci.* 69.3 (2004), pp. 485–497. DOI: 10.1016/j.jcss.2004.04.011. URL: https://doi.org/10.1016/j.jcss.2004.04.011.

[Gha+18] Mohsen Ghaffari et al. 'Near-Optimal Distributed Maximum Flow'. In: *SIAM J. Comput.* 47.6 (2018), pp. 2078–2117. DOI: 10.1137/17M113277X. URL: https://doi.org/10.1137/17M113277X.

[GHR95] Raymond Greenlaw, H James Hoover and Walter L Ruzzo. *Limits to parallel computation*. en. New York, NY: Oxford University Press, Apr. 1995.

[GKK10] Ashish Goel, Michael Kapralov and Sanjeev Khanna. 'Graph Sparsification via Refinement Sampling'. In: *CoRR* abs/1004.4915 (2010). arXiv: 1004.4915. URL: http://arxiv.org/abs/1004.4915.

[GLP21] Yu Gao, Yang P. Liu and Richard Peng. 'Fully Dynamic Electrical Flows: Sparse Maxflow Faster Than Goldberg-Rao'. In: *CoRR* abs/2101.07233 (2021). arXiv: 2101.07233. URL: https://arxiv.org/abs/2101.07233.

[GSS82] Leslie M. Goldschlager, Ralph A. Shaw and John Staples. 'The Maximum Flow Problem is Log Space Complete for P'. In: *Theor. Comput. Sci.* 21 (1982), pp. 105–111. DOI: 10.1016/0304-3975(82)90092-5. URL: https://doi.org/10.1016/0304-3975(82)90092-5.

[GT04] F. Glineur and T. Terlaky. 'Conic Formulation for ℓ_p-Norm Optimization'. In: *Journal of Optimization Theory and Applications* 122.2 (Aug. 2004), pp. 285–307. DOI: 10.1023/b:jota.0000042522.65261.51. URL: https://doi.org/10.1023%2Fb%3Ajota.0000042522.65261.51.

[GT86] A V Goldberg and R E Tarjan. 'A new approach to the maximum flow problem'. In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing - STOC '86*. ACM Press, 1986. DOI: 10.1145/12130.12144. URL: https://doi.org/10.1145%2F12130.12144.

[Gup01] Anupam Gupta. 'Steiner points in tree metrics don't (really) help'. In: *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*. Ed. by S. Rao Kosaraju. ACM/SIAM, 2001, pp. 220–227. URL: http://dl.acm.org/citation.cfm?id=365411.365448.

[HH10] Zhengyu He and Bo Hong. 'Dynamically tuned push-relabel algorithm for the maximum flow problem on CPU-GPU-Hybrid platforms'. In: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. IEEE, 2010, pp. 1–10. DOI: 10.1109/IPDPS.2010.5470401. URL: https://doi.org/10.1109/IPDPS.2010.5470401.

[HRR98] Monika Rauch Henzinger, Prabhakar Raghavan and Sridhar Rajagopalan. 'Computing on data streams'. In: *External Memory Algorithms, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, May 20-22, 1998*. Ed. by James M. Abello and Jeffrey Scott Vitter. Vol. 50. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1998, pp. 107–118. DOI: 10.1090/dimacs/050/05. URL: https://doi.org/10.1090/dimacs/050/05.

[Hu74] T. C. Hu. 'Optimum Communication Spanning Trees'. In: *SIAM J. Comput.* 3.3 (1974), pp. 188–195. DOI: 10.1137/0203015. URL: https://doi.org/10.1137/0203015.

[Kel+14] Jonathan A. Kelner et al. 'An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations'. In: *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*. Ed. by Chandra Chekuri. SIAM, 2014, pp. 217–226. DOI: 10.1137/1.9781611973402.16. URL: https://doi.org/10.1137/1.9781611973402.16.

[KKT95] David R. Karger, Philip N. Klein and Robert Endre Tarjan. 'A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees'. In: *J. ACM* 42.2 (1995), pp. 321–328. DOI: 10.1145/201019.201022. URL: https://doi.org/10.1145/201019.201022.

[KLS20] Tarun Kathuria, Yang P. Liu and Aaron Sidford. 'Unit Capacity Maxflow in Almost $\mathcal{O}(m^{4/3})$ Time'. In: *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. IEEE, 2020, pp. 119–130. DOI: 10.1109/FOCS46700.2020.00020. URL: https://doi.org/10.1109/FOCS46700.2020.00020.

[Kol+10] Alexandra Kolla et al. 'Subgraph sparsification and nearly optimal ultrasparsifiers'. In: *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*. Ed. by Leonard J. Schulman. ACM, 2010, pp. 57–66. DOI: 10.1145/1806689.1806699. URL: https://doi.org/10.1145/1806689.1806699.

[KS96] Vijay Kumar and Eric J. Schwabe. 'Improved algorithms and data structures for solving graph problems in external memory'. In: *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, SPDP 1996, New Orleans, Louisiana, USA, October 23-26, 1996*. IEEE Computer Society, 1996, pp. 169–176. DOI: 10.1109/SPDP.1996.570330. URL: https://doi.org/10.1109/SPDP.1996.570330.

[KX16] Ioannis Koutis and Shen Chen Xu. 'Simple Parallel and Distributed Algorithms for Spectral Graph Sparsification'. In: *ACM Trans. Parallel Comput.* 3.2 (2016), 14:1–14:14. DOI: 10.1145/2948062. URL: https://doi.org/10.1145/2948062.

[Kyn+19] Rasmus Kyng et al. 'Flows in almost linear time via adaptive preconditioning'. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*. Ed. by Moses Charikar and Edith Cohen. ACM, 2019, pp. 902–913. DOI: 10.1145/3313276.3316410. URL: https://doi.org/10.1145/3313276.3316410.

[LR99] Frank Thomson Leighton and Satish Rao. 'Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms'. In: *J. ACM* 46.6 (1999), pp. 787–832. DOI: 10.1145/331524.331526. URL: https://doi.org/10.1145/331524.331526.

[LRS13] Yin Tat Lee, Satish Rao and Nikhil Srivastava. 'A new approach to computing maximum flows using electrical flows'. In: *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*. Ed. by Dan Boneh, Tim Roughgarden and Joan Feigenbaum. ACM, 2013, pp. 755–764. DOI: 10.1145/2488608.2488704. URL: https://doi.org/10.1145/2488608.2488704.

[Mad11] Aleksander Madry. 'From Graphs to Matrices, and Back: New Techniques for Graph Algorithms'. AAI0823802. PhD thesis. USA, 2011.

[Mad13] Aleksander Madry. 'Navigating Central Path with Electrical Flows: From Flows to Matchings, and Back'. In: *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*. IEEE Computer Society, 2013, pp. 253–262. DOI: 10.1109/FOCS.2013.35. URL: https://doi.org/10.1109/FOCS.2013.35.

[MM02] Kurt Mehlhorn and Ulrich Meyer. 'External-Memory Breadth-First Search with Sublinear I/O'. In: *Algorithms — ESA 2002*. Springer Berlin Heidelberg, 2002, pp. 723–735. DOI: 10.1007/3-540-45749-6_63. URL: https://doi.org/10.1007/3-540-45749-6_63.

[MPX13] Gary L. Miller, Richard Peng and Shen Chen Xu. 'Parallel graph decompositions using random shifts'. In: *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, June 2013. DOI: 10.1145/2486159.2486180. URL: https://doi.org/10.1145%2F2486159.2486180.

[MR99] Kameshwar Munagala and Abhiram Ranade. 'I/O-Complexity of Graph Algorithms'. In: *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '99. Baltimore, Maryland, USA: Society for Industrial and Applied Mathematics, 1999, pp. 687–694. ISBN: 0898714346.

[MZ03] Anil Maheshwari and Norbert Zeh. 'A Survey of Techniques for Designing I/O-Efficient Algorithms'. In: *Algorithms for Memory Hierarchies*. Springer Berlin Heidelberg, 2003, pp. 36–61. DOI: 10.1007/3-540-36574-5_3. URL: https://doi.org/10.1007%2F3-540-36574-5_3.

[MZ06] Ulrich Meyer and Norbert Zeh. 'I/O-Efficient Undirected Shortest Paths with Unbounded Edge Lengths'. In: *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*. Ed. by Yossi Azar and Thomas Erlebach. Vol. 4168. Lecture Notes in Computer Science. Springer, 2006, pp. 540–551. DOI: 10.1007/11841036_49. URL: https://doi.org/10.1007/11841036%5C_49.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. 'A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph'. In: *Algorithmica* 7.1-6 (June 1992), pp. 583–596. DOI: 10.1007/bf01758778. URL: https://doi.org/10.1007%2Fbf01758778.

[OG21] James B. Orlin and Xiao-Yue Gong. 'A fast maximum flow algorithm'. In: *Networks* 77.2 (2021), pp. 287–321. DOI: 10.1002/net.22001. URL: https://doi.org/10.1002/net.22001.

[Pen16] Richard Peng. 'Approximate Undirected Maximum Flows in $O(m\text{polylog}(n))$ Time'. In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*. Ed. by Robert Krauthgamer. SIAM, 2016, pp. 1862–1867. DOI: 10.1137/1.9781611974331.ch130. URL: https://doi.org/10.1137/1.9781611974331.ch130.

[RR98] Yuri Rabinovich and Ran Raz. 'Lower Bounds on the Distortion of Embedding Finite Metric Spaces in Graphs'. In: *Discret. Comput. Geom.* 19.1 (1998), pp. 79–94. DOI: 10.1007/PL00009336. URL: https://doi.org/10.1007/PL00009336.

[RST14] Harald Räcke, Chintan Shah and Hanjo Täubig. 'Computing Cut-Based Hierarchical Decompositions in Almost Linear Time'. In: *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*. Ed. by Chandra Chekuri. SIAM, 2014, pp. 227–238. DOI: 10.1137/1.9781611973402.17. URL: https://doi.org/10.1137/1.9781611973402.17.

[San00] Peter Sanders. 'Fast Priority Queues for Cached Memory'. In: *ACM J. Exp. Algorithmics* 5 (2000), p. 7. DOI: 10.1145/351827.384249. URL: https://doi.org/10.1145/351827.384249.

[She13] Jonah Sherman. 'Nearly Maximum Flows in Nearly Linear Time'. In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE, Oct. 2013. DOI: 10.1109/focs.2013.36. URL: https://doi.org/10.1109%2Ffocs.2013.36.

[She16] Jonah Sherman. 'Generalized Preconditioning and Network Flow Problems'. In: *CoRR* abs/1606.07425 (2016). arXiv: 1606.07425. URL: http://arxiv.org/abs/1606.07425.

[SS11] Daniel A. Spielman and Nikhil Srivastava. 'Graph Sparsification by Effective Resistances'. In: *SIAM J. Comput.* 40.6 (2011), pp. 1913–1926. DOI: 10.1137/080734029. URL: https://doi.org/10.1137/080734029.

[ST14] Daniel A. Spielman and Shang-Hua Teng. 'Nearly Linear Time Algorithms for Preconditioning and Solving Symmetric, Diagonally Dominant Linear Systems'. In: *SIAM Journal on Matrix Analysis and Applications* 35.3 (Jan. 2014), pp. 835–885. DOI: 10.1137/090771430. URL: https://doi.org/10.1137%2F090771430.

[ST18] Aaron Sidford and Kevin Tian. 'Coordinate Methods for Accelerating ℓ^∞ Regression and Faster Approximate Maximum Flow'. In: *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*. Ed. by Mikkel Thorup. IEEE Computer Society, 2018, pp. 922–933. DOI: 10.1109/FOCS.2018.00091. URL: https://doi.org/10.1109/FOCS.2018.00091.

[ST81] Daniel Dominic Sleator and Robert Endre Tarjan. 'A Data Structure for Dynamic Trees'. In: *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*. ACM, 1981, pp. 114–122. DOI: 10.1145/800076.802464. URL: https://doi.org/10.1145/800076.802464.

[SW09] Daniel A. Spielman and Jaeoh Woo. 'A Note on Preconditioning by Low-Stretch Spanning Trees'. In: *CoRR* abs/0903.2816 (2009). arXiv: 0903.2816. URL: http://arxiv.org/abs/0903.2816.

[Tar84] Robert Endre Tarjan. 'A simple version of Karzanov's blocking flow algorithm'. In: *Operations Research Letters* 2.6 (Mar. 1984), pp. 265–268. DOI: 10.1016/0167-6377(84)90076-2. URL: https://doi.org/10.1016/0167-6377(84)90076-2.

[TZ02] Laura Toma and Norbert Zeh. 'I/O-Efficient Algorithms for Sparse Graphs'. In: Jan. 2002, pp. 85–109. ISBN: 978-3-540-00883-5. DOI: 10.1007/3-540-36574-5_5. URL: https://doi.org/10.1007/3-540-36574-5_5.