

CS598  
Topics in Graph Algorithms

Pingbang Hu

September 10, 2024

## Abstract

This is an advanced graduate-level graph algorithm course taught by [Chandra Chekuri](#) at University of Illinois Urbana-Champaign.



This course is taken in Fall 2024, and the date on the cover page is the last updated time.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Minimum Spanning Tree . . . . .	2
1.2	Tree Packing . . . . .	10
1.3	Min-Cuts . . . . .	12
1.4	Metric Embedding . . . . .	21

# Chapter 1

## Introduction

### Lecture 1: Overview

Throughout the course, we consider a graph  $G = (V, E)$  such that  $n := |V|$  and  $m := |E|$ . Let's see some examples about the recent breakthroughs. 27 Aug. 11:00

**Example (Shortest paths with negative length).** The classical algorithm runs in  $O(mn)$ . In 2022, [BNW] came up with an algorithm  $O(m \log^3 n C)$ , where  $C$  is the largest absolute value of the integer length.

cite

This is not a strongly polynomial time algorithm. In 2024 [Fineman] come up with  $\tilde{O}(mn^{8/9})$ , and soon after 2024 [HJQ] improve this to  $\tilde{O}(mn^{4/5})$ .

cite

cite

**Example ( $s$ - $t$  max-flow).** The tradition running time is  $O(mn \log m/n)$ , and it's later improved to be  $O(m\sqrt{n} \log n C)$ . Recently, [Chen et-al] improve to  $O(m^{1+o(1)})$ , which is almost-linear.<sup>a</sup>

cite

<sup>a</sup>This can be also applied to min-cost flow and quadratic-cost flow.

### 1.1 Minimum Spanning Tree

Finding the minimum cost **spanning tree** (MST) in a connected graph is a basic algorithmic problem that has been long-studied. We introduce the problem formally.

**Definition 1.1.1 (Spanning tree).** A *spanning tree*  $T$  of a connected graph  $G = (V, E)$  is an induced subgraph of  $G$  which spans  $G$ , i.e.,  $V(T) = V$  and  $E(T) \subseteq E$ .

Then, the problem can be formalized as follows.

**Problem 1.1.1 (Minimum spanning tree).** Given a connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ , find the min-cost **spanning tree**.

**Remark.** The edge costs need not be positive, but we can make them positive by adding a large number without affecting correctness.

Standard algorithm that are covered in most undergraduate courses are **Kruskal's algorithm**, **Jarnik-Prim's (JP) algorithm**,<sup>1</sup> and (sometimes) **Borůvka's algorithm**. There are many algorithms for **MST** and their correctness relies on two simple rules (structural properties). The first one is about **cuts**:

**Lemma 1.1.1 (Cut rule).** If  $e$  is a minimum cost edge in a cut  $\delta(S)$  for some  $S \subseteq V$ , then  $e$  is in some **MST**. In particular, if  $e$  is the unique minimum cost edge in the cut, then  $e$  is in every **MST**.

<sup>1</sup>This is typically attributed usually to Prim but first described by Jarnik

**Definition 1.1.2 (Light).** An edge  $e$  is *light* or *safe* if there exists a cut  $\delta(S)$  such that  $e$  is the cheapest cost edge crossing the cut. We also say that  $e$  is *light* w.r.t. a set of edges  $F \subseteq E$  if  $e$  is light in  $(V, F)$ .

Another one is about [cycles](#):

**Lemma 1.1.2 (Cycle rule).** If  $e$  is the highest cost edge in a cycle  $C$ , then there exists an [MST](#) that does not contain  $e$ . In particular, if  $e$  is the unique highest cost edge in  $C$ , then  $e$  cannot be in any [MST](#).

**Definition 1.1.3 (Heavy).** An edge  $e$  is *heavy* or *unsafe* if there exists a cycle  $C$  such that  $e$  is the highest cost edge in  $C$ . We also say that  $e$  is *heavy* w.r.t. a set of edges  $F \subseteq E$  if  $e$  is heavy in  $(V, F)$ .

**Corollary 1.1.1.** Suppose the edge costs are unique and  $G$  is connected. Then the [MST](#) is unique and consists of the set of all [light](#) edges.

**Remark.** Without loss of generality, we can assume that the cost are unique by, e.g., perturbation or consistent tie-breaking rule.

### 1.1.1 Standard Algorithms

Let's review the basic algorithms, the data structures they use, and the run-times that they yield.

#### Kruskal's Algorithm

Intuitively speaking, [Kruskal's algorithm](#) sorts the edges in increasing cost order and greedily inserts edges in this order while maintaining a maximal forest  $F$  at each step. When considering the  $i^{\text{th}}$  edge  $e_i$ , the algorithm needs to decide if  $F + e_i$  is a forest or whether adding  $e$  creates a cycle.

---

**Algorithm 1.1:** Kruskal's algorithm

---

**Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$

**Result:** A [MST](#)  $T = (V, F)$

```

1 Sort the edges such that  $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$  //  $O(m \log n)$ 
2  $F \leftarrow \emptyset$  // Initialize the tree
3 for  $i = 1, \dots, m$  do
4   if  $e_i + F$  has no cycle then
5      $F \leftarrow F + e_i$ 
6 return  $(V, F)$ 
```

---

**Theorem 1.1.1.** [Kruskal's algorithm](#) takes  $O(m \log n)$ .

**Proof.** Sorting takes  $O(m \log n)$  time. The standard solution for [line 4](#) is to use a [union-find](#) data structure. Union-find data structure with path compression yields a total run time, after sorting, of  $O(m\alpha(m, n))$  where  $\alpha(m, n)$  is [inverse Ackerman function](#) which is extremely slowly growing. Thus, the bottleneck is sorting, and the run-time is  $O(m \log n)$ . ■

#### Jarnik-Prim's Algorithm

[Jarnik-Prim's algorithm](#) grows a tree starting at some arbitrary root vertex  $r$  while maintaining a tree  $T$  rooted at  $r$ . In each iteration it adds the cheapest edge leaving  $T$  until  $T$  becomes [spanning](#). Thus, the [Jarnik-Prim's algorithm](#) takes  $n - 1$  iterations.

**Algorithm 1.2:** Jarnik-Prim's algorithm**Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ **Result:** A **MST**  $T = (V, F)$ 

```

1  $r \leftarrow \text{uniform}(V)$  // Sample a root
2  $V' \leftarrow \{r\}, F \leftarrow \emptyset$  // Initialize the tree
3 while  $V' \neq V$  do
4    $e \leftarrow \arg \min_{e=(u,v) \in \delta(V'), u \in V'} c(e)$ 
5    $F \leftarrow F + e, V' \leftarrow V' + v$  // Update the tree
6 return  $(V, F)$ 

```

**Theorem 1.1.2.** Jarnik-Prim's algorithm takes  $O(m + n \log n)$ .

**Proof.** To find the cheapest edge leaving  $T$  (line 4), one typically uses a priority queue where we maintain vertices not yet in the tree with a key for  $v$  equal to the cost of the cheapest edge from  $v$  to the current tree. When a new vertex  $v$  is added to  $T$  the algorithm scans the edges in  $\delta(v)$  to update the keys of neighbors of  $v$ . Thus, one sees that there are a total of  $O(m)$  decrease-key operations,  $O(n)$  extract-min operations, and initially we set up an empty queue. Standard priority queues implement decrease-key and extract-min in  $O(\log n)$  time each, so the total time is  $O(m \log n)$ . However, Fibonacci heaps and related data structures show that one can implement decrease-key in amortized  $O(1)$  time which reduces the total run time to  $O(m + n \log n)$ . ■

**Remark.** The Jarnik-Prim's algorithm runs in linear-time for moderately dense graphs!

**Borůvka's Algorithm**

Borůvka's algorithm seems to be the first **MST** algorithm, which has very nice properties and essentially uses no data structures. The algorithm works in phases. We describe it recursively to simplify the description, while refer to Algorithm 1.3 for the real implementation. In the first phase the algorithm finds, for each vertex  $v$  the cheapest edge in  $\delta(v)$ . By the cut rule this edge is in every **MST**.

**Note.** An edge  $e = uv$  may be the cheapest edge for both  $u$  and  $v$ .

The algorithm collects all these edges, say  $F$ , and adds them to the tree. It then shrinks the connected components induced by  $F$  and recurses on the resulting graph  $H = (V', E')$ . It's easy to see that Borůvka's algorithm can be parallelized, unlike the other two algorithms.

**Algorithm 1.3:** Borůvka's algorithm**Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ **Result:** A **MST**  $T = (V, F)$ 

```

1  $F = \emptyset$  // Initialize the tree
2  $\mathcal{S} \leftarrow \{S_v = \{v\}\}$  // Collection of all sets
3 while  $|\mathcal{S}| > 1$  do
4   for  $S \in \mathcal{S}$  do
5      $e_S = (u, v) \leftarrow \arg \min_{e \in \delta(S)} c(e)$ 
6      $\mathcal{S} \leftarrow \mathcal{S} - \{S_u, S_v\} + S_u \cup S_v$  // Merge (i.e., shrink)
7      $F \leftarrow F + e_S$  // Update the tree
8 return  $(V, F)$ 

```

**Notation.** In line 6,  $S_u$  and  $S_v$  both refer to  $S := S_u \cup S_v$  later in the algorithm.

**Theorem 1.1.3.** Borůvka's algorithm takes  $O(m \log n)$ .

**Proof.** The first phase needs  $O(m)$  from a linear scan of the adjacency lists, and also computing  $H$  (i.e., shrinking) can be done in  $O(m)$  time. The main observation is that  $|V'| \leq |V|/2$  since each vertex  $v$  is in a connected component of size at least 2 as we add an edge leaving  $v$  to  $F$ . Thus, the algorithm terminates in  $O(\log n)$  phases for a total of  $O(m \log n)$  time. ■

### 1.1.2 Faster Algorithms

A natural question is whether there is a linear-time, i.e.,  $O(m)$ , **MST** algorithm. The following is the history of fast **MST** algorithms:

- Very early on, Yao, in 1975, obtained an algorithm that ran in  $O(m \log \log n)$  [Yao75], which leverages the idea developed in 1974 for the linear-time Selection algorithm.
- In 1987, Fredman and Tarjan [FT87] developed the Fibonacci heaps and give an **MST** algorithm which runs in  $O(m \log^* n)$ .<sup>2</sup> This was further improved to  $O(m \log \log n)$  [Gab+86].
- Karger, Klein, and Tarjan [KKT95] obtained a linear time randomized algorithm that will be the main topic of this lecture.
- Chazelle's algorithm [Cha00] that runs in  $O(m \alpha(m, n))$  is the fastest known deterministic algorithm.

**Note.** Pettie and Ramachandran gave an optimal deterministic algorithm in the comparison model without known what its actual running time is [PR02]!

Perhaps an easier question is the following.

**Problem 1.1.2 (MST verification).** Given a graph  $G$  and a tree  $T$ , decide  $T$  is an **MST** of  $G$  or not.

One can always use an **MST** algorithm to solve the **verification** problem, but not necessarily the other way around. Interestingly, there is indeed a linear-time **MST verification** algorithm based on several non-trivial ideas and data structures and was first developed in the RAM model by Dixon, Rauch, and Tarjan [DRT92] with insights from Komlós [Kom85]. Simplification is done by King [Kin97].

**Note (RAM model).** The RAM model allows bit-wise operation on  $O(\log n)$  bit words in  $O(1)$  time.

**Theorem 1.1.4 (MST verification).** There is a linear-time **MST verification** algorithm in the RAM model. In fact, the algorithm is based on a more general result that we will need: Given a graph  $G = (V, E)$  with edge costs and a **spanning tree**  $T = (V, F)$ , there is an  $O(m)$ -time algorithm that outputs all the **F-heavy** edge of  $G$ .

**Proof.** The original complicated algorithm has been simplified over the years. See lecture notes of Gupta and Assadi for accessible explanation, also the MST surveys [Eis97; Mar08]. ■

### Fredman-Tarjan's Algorithm

Here we briefly describe Fredman and Tarjan's algorithm [FT87; Mar08] via Fibonacci heaps, which is reasonably simple to describe and analyze modulo a few implementation details that we will gloss over for the sake of brevity. First, we develop a simple  $O(m \log \log n)$  time algorithm by combining **Borůvka's algorithm** and **Jarník-Prim's algorithm**.

**As previously seen.** **Jarník-Prim's algorithm** takes  $O(m + n \log n)$  time via Fibonacci heaps where the bottleneck is when  $m = o(n \log n)$ . On the other hand, **Borůvka's algorithm** starts with a graph on  $n$  nodes and after  $i^{\text{th}}$  phases, reduces the number of nodes to  $n/2^i$ ; each phase takes  $O(m)$  times.

<sup>2</sup>Formally, it runs in  $O(m \beta(m, n))$ , where  $\beta(m, n)$  is the minimum value of  $i$  such that  $\log^{(i)} n \leq m/n$ , where  $\log^{(i)} n$  is the logarithmic function iterated  $i$  times. Since  $m \leq n^2$ ,  $\beta(m, n) \leq \log^* n$ .

**Intuition.** Suppose we run [Borůvka's algorithm](#) for  $k$  phases and then run [Jarnik-Prim's algorithm](#) once the number of nodes is reduced. We can see that the total run time is  $O(mk)$  for the  $k$  phases of [Borůvka's algorithm](#), and  $O(m + n/2^k \log n/2^k)$  for the [Jarnik-Prim's algorithm](#) on the reduced graph. Thus, if we choose  $k = \log \log n$ , we obtain a total run-time of  $O(m \log \log n)$ .

Tarjan and Fredman obtained a more sophisticated scheme based on the [Jarnik-Prim's algorithm](#), but the basic idea is to reduce the number of vertices. The algorithm runs again in phases. We describe the first phase here.

**Intuition (First phase).** Start growing the tree. If the heap gets too big, we stop.

Consider an integer parameter  $t$  such that  $1 < t \leq n$ . Pick an arbitrary root  $r_1$  and grow a tree  $T_1$  via [Jarnik-Prim's algorithm](#) with a Fibonacci heap. We stop the tree growth when the heap size exceeds  $t$  for the first time or if we run out of vertices. All the vertices in the tree are marked as visited. Now pick an arbitrary, unmarked vertex as root  $r_2 \in V - T$  and grow a tree  $T_2$ , and we stop growing  $T_2$  if it touches  $T_1$ , in which case it merges with it, or if the heap size exceeds  $t$  or if we run out of vertices. The algorithm proceeds in this fashion by picking new roots and growing them until all nodes are marked.

**Note.** While growing  $T_2$ , the heap may contain previously marked vertices. It is only when the algorithm finds one of the marked vertices as the cheapest neighbor of the current tree that we merge the trees and stop.

It's easy to see that the [first phase of Fredman-Tarjan algorithm](#) correctly adds a set of [MST](#) edges  $F$ . After this, we simply shrink these trees and recurse on the smaller graph.

---

**Algorithm 1.4:** Fredman-Tarjan's algorithm

---

**Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$

**Result:** A [MST](#)  $T = (V, F)$

---

```

1   $V' \leftarrow V, F \leftarrow \emptyset$                                 // Initialize the tree
2  while  $|V'| > 1$  do
3       $T \leftarrow \text{Grow}(G)$                                     // First phase
4       $F \leftarrow F \cup E(T)$                                   // Update the tree
5      Shrink  $G$  w.r.t.  $T$ , update  $V$  and  $E$                     // Second phase
6  return  $(V', F)$ 
7
8  Grow( $G$ ):
9       $V' \leftarrow \emptyset, F \leftarrow \emptyset, T \leftarrow (V', F)$  // Initialize the forest
10     while  $V' \neq V$  do
11          $r \leftarrow \text{uniform}(V - V')$                         // Pick an unmarked vertex
12          $T' \leftarrow (\{r\}, \emptyset)$                         // Initialize a tree
13         while  $|N(T')| < t$  or  $V(T') \cap V' \neq \emptyset$  do
14             Run one more step of Jarnik-Prim( $r, T'$ )        // Starting at  $r$ , maintaining  $T'$ 
15              $V' \leftarrow V' \cup V(T')$                         // Mark
16              $F \leftarrow F \cup E(T')$                         // Update the forest by merging the tree
17     return  $(V, F)$                                            // Return a forest of  $G$ 

```

---

**Note.** This can be seen as a parameterized version of [Borůvka's algorithm](#).

The difficult part is to determine its runtime. We have the following.

**Theorem 1.1.5.** [Fredman-Tarjan's algorithm](#) takes  $O(m\beta(m, n))$ .

**Proof.** Firstly, the total time to scan edges and insert vertices into heaps and do **decrease-key** is  $O(m)$  since an edge is only visited twice, once from each end point. Since each heap is not allowed to grow to more than size  $t$ , the total time for all the **extract-min** operations take  $O(n \log t)$ . With the fact that the initialization of each data structure is easy as it starts as an empty one, hence, the



first phase takes  $O(m + n \log t)$ . We claim that it also reduces the number of vertices to  $2m/t$ .

**Claim.** The number of connected components induced by  $F$  is  $\leq 2m/t$  after the first phase.

**Proof.** Let  $C_1, \dots, C_h$  be the connected components of  $F$ . If for every  $C_i$ ,  $\sum_{v \in C_i} \deg(v) \geq t$ ,

$$2m = \sum_{v \in V} \deg(v) = \sum_{i=1}^h \sum_{v \in C_i} \deg(v) \geq ht \Rightarrow h \leq \frac{2m}{t}.$$

To see why the assumption holds, consider the growth of a tree  $T'$  in line 14:

- If we stop  $T'$  because heap size  $|N(T')|$  exceeds  $t$ , then each of the vertex in the heap is a witness to a unique edge incident to  $T'$ , hence the property holds.
- If  $T'$  merged with a previous tree, then the property holds because the previous tree already had the property and adding vertices can only increase the total degree of the component.

The only reason the property may not hold is if line 17 terminates a tree because all vertices are already included in it, but then that phase finishes the algorithm.  $\otimes$

The question reduces to choosing  $t$ .

**Intuition.** We want linear time in the first phase, i.e.,  $n \log t$  to be no more than  $O(m)$ , leading to  $t = 2^{2m/n}$ . If we do this in every iteration, then this leads to  $O(m)$  time per iteration.

We now bound the number of iteration. Consider  $t_1 := 2^{2m/n}$  and  $t_i := 2^{2m/n_i}$ ,<sup>a</sup> where  $n_i$  and  $m_i$  are the number of vertices and edges at the beginning of the  $i^{\text{th}}$  iteration, with  $m_1 = m$  and  $n_1 = n$ . From the previous claim,  $n_{i+1} \leq 2m_i/t_i$ , which gives

$$t_{i+1} = 2^{2m/n_{i+1}} \geq 2^{\frac{2m}{2m_i/t_i}} \geq 2^{t_i}.$$

Thus,  $t_i$  is a power of twos with  $t_1 = 2^{2m/n}$ , and the Fredman-Tarjan's algorithm stops if  $t_i \geq n$  since it will grow a single tree and finish. Thus, the algorithm needs at most  $\beta(m, n)$  iterations, giving the total time  $O(m\beta(m, n))$ .  $\blacksquare$

<sup>a</sup>Technically, we need to choose  $t_i := 2^{\lceil 2m/n_i \rceil}$ , but we will be a bit sloppy and ignore the ceilings here.

## Lecture 2: MST and Tree Packing

### Linear-Time Randomized Algorithm

29 Aug. 11:00

Using randomization, it's possible to derive a linear-time algorithm for MST.

**Theorem 1.1.6 ([KKT95]).** Karger-Klein-Tarjan's algorithm takes  $O(m)$  time that computes the MST with probability at least  $1 - 1/\text{poly}(m)$ .

Karger-Klein-Tarjan's algorithm relies on the so-called sampling lemma, which we first discussed.

**Lemma 1.1.3 (Sampling lemma).** Given a graph  $G = (V, E)$ , and let  $E' \subseteq E$  be obtained by sampling each edge  $e$  with probability  $p \in (0, 1)$ . Let  $F$  be a minimum spanning forest<sup>a</sup> in  $G' = (V, E')$ . Then the expected number of  $F$ -light edge in  $G$  is less than  $(n - 1)/p$ .

<sup>a</sup>As  $G'$  can be disconnected.

**Proof.** The proof is based on the *principle of deferred decisions* in randomized analysis. Let  $A$  be the set of  $F$ -light edges. Note that both  $A$  and  $F$  are random sets that are generated by the process of sampling  $E'$ . To analyze  $\mathbb{E}[|A|]$ , we consider Kruskal's algorithm to obtain  $F$  from  $E'$ , where we generate  $E'$  on the fly:

**Algorithm 1.5:** Sampling Process**Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ , probability  $p \in (0, 1)$ **Result:** A minimum spanning forest  $F$  and the set of *F-light* edges  $A$ 

```

1 Sort the edges such that  $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$ 
2  $A \leftarrow \emptyset, F \leftarrow \emptyset, E' \leftarrow \emptyset$ 
3 for  $i = 1, \dots, m$  do
4    $r \leftarrow \text{Ber}(p)$  // Toss a biased coin
5   if  $r = 1$  then
6      $E' \leftarrow E' + e_i$ 
7     if  $F + e_i$  is a forest then
8        $F \leftarrow F + e_i$ 
9        $A \leftarrow A + e_i$ 
10  else if  $e_i$  is F-light then
11     $A \leftarrow A + e_i$ 
12 return  $F, A$ 

```

The following is exactly the same as the above, but easier to analyze:

**Algorithm 1.6:** Sampling Process with Tweaks**Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ , probability  $p \in (0, 1)$ **Result:** A minimum spanning forest  $F$  and the set of *F-light* edges  $A$ 

```

1 Sort the edges such that  $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$ 
2  $A \leftarrow \emptyset, F \leftarrow \emptyset$ 
3 for  $i = 1, \dots, m$  do
4   if  $e_i$  is F-light then // Sorting implies  $F + e_i$  is a forest  $\Leftrightarrow e_i$  is F-light
5      $A \leftarrow A + e_i$ 
6      $r \leftarrow \text{Ber}(p)$  // Toss a biased coin
7     if  $r = 1$  then
8        $F \leftarrow F + e_i$ 
9 return  $F, A$ 

```

The second algorithm makes the following observation clear.

**Intuition.** An edge  $e_i$  is added to  $A$  implies that it is added to  $F$  with probability  $p$ .Hence,  $p\mathbb{E}[|A|] = \mathbb{E}[|F|] \leq n - 1$ , hence  $\mathbb{E}[|A|] \leq (n - 1)/p$ . ■

With the [sampling lemma](#), we know that when  $p = 1/2$ , the number of *F-light* edges from  $E$  is at most  $2n$ . Hence, we can eliminate most of the edges from  $E \setminus E'$  from consideration given the fact that we can efficiently compute the *F-heavy* edges via the [MST verification theorem](#). It's worth noting that to work with the [sampling lemma](#) via the natural recursion that it implies means that we need to work with potentially disconnected graph. That is, we will need to consider disconnected graph. Hence, we make the following generalization.

**Definition 1.1.4 (Spanning forest).** A *spanning forest*  $T$  of a graph  $G = (V, E)$  (potentially disconnected) is an induced subgraph of  $G$  which spans  $G$ , i.e.,  $V(T) = V$  and  $E(T) \subseteq E$ .

**Problem 1.1.3 (Minimum spanning forest).** Given a graph  $G = (V, E)$  (potentially disconnected) with edge capacity  $c: E \rightarrow \mathbb{R}_+$ , find the min-cost [spanning forest](#).

**Note.** [MST](#) and [MSF](#) are closely related and one is reducible to the other in linear time, and the [cut](#) and [cycle rules](#) can be generalized to [MSF](#) easily.

Now, consider the following natural recursive divide and conquer algorithm for computing [MSF](#).

**Algorithm 1.7:** Natural Recursive Algorithm from [Sampling Lemma](#)


---

**Data:** A graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$   
**Result:** A [MSF](#)  $T = (V, F)$

```

1 if  $|V| < n_0$  then                                     //  $n_0$  is some constant
2   return Standard-MST( $G, c$ )                             // Use a standard deterministic algorithm
3
4 Sample each edge i.i.d. from  $\text{Ber}(1/2)$  to obtain  $E_1 \subseteq E$ 
5  $(V, F_1) \leftarrow \text{Karger-Klein-Tarjan}((V, E_1))$        // Recursively compute MSF
6  $E_2 \leftarrow \text{Light-Edge}(G, F_1)$                      // Compute all F1-light edges with Theorem 1.1.4
7  $(V, F_2) \leftarrow \text{Karger-Klein-Tarjan}((V, E_2))$      // Recursively compute MSF
8 return  $(V, F_2)$ 

```

---

The correctness of [Algorithm 1.7](#) is clear from the [cut](#) and [cycle rules](#). The issue is the running time:

**Claim.** [Algorithm 1.7](#) is not efficient enough.

**Proof.** The expected number of edges in  $G_1 := (V, E_1)$  is  $m/2$ , and the expected number of edges in  $G_2 := (V, E_2)$ , via the [sampling lemma](#), is at most  $2n$ . We see that the algorithm does  $O(m+n)$  work outside the two recursive calls ([line 5](#), [line 7](#)). Let  $T(m, n)$  be the expected running time of the algorithm on an  $m$ -edge  $n$ -node graph. Informally, we see the following recurrence:

$$T(m, n) \leq c(m+n) + T(m/2, n) + T(2n, n).$$

If we take the problem size to be  $n+m$ , then [Algorithm 1.7](#) generates two sub-problems of expected size  $m/2 + n$  and  $2n + n$ , with the total size being  $4n + m/2$ . If  $m > 10n$ , say, then the total problem size is shrinking by a constant factor, and we obtain a linear-time algorithm. However, this is generally not the case. \*

The problem becomes reducing the graph size, which is the trick of [Karger-Klein-Tarjan's algorithm](#): we run [Borůvka's algorithm](#) for a few iterations as a preprocessing step, reducing the number of vertices:

**Algorithm 1.8:** Karger-Klein-Tarjan's Algorithm [[KKT95](#)]

---

**Data:** A connected graph  $G = (V, E)$ <sup>a</sup> with edge capacity  $c: E \rightarrow \mathbb{R}_+$   
**Result:** A [MSF](#)  $T = (V, F)$

```

1 if  $|V| < n_0$  then                                     //  $n_0$  is some large constant
2   return Standard-MST( $G, c$ )                             // Use a standard deterministic algorithm
3
4  $G' = (V', E'), T' = (V', F') \leftarrow \text{Borůvka}(G, c, 2)$  // Run two iterations with  $|V'| \leq |V|/4$ .
5
6 Sample each edge in  $G'$  i.i.d. from  $\text{Ber}(1/2)$  to obtain  $E_1 \subseteq E'$ 
7  $(V', F_1) \leftarrow \text{Karger-Klein-Tarjan}((V', E_1))$      // Recursively compute MSF
8  $E_2 \leftarrow \text{Light-Edge}(G_1, F_1)$                      // Compute F2-light edges with Theorem 1.1.4
9  $(V', F_2) \leftarrow \text{Karger-Klein-Tarjan}((V', E_2))$      // Recursively compute MSF
10 return  $(V, F' \cup F_2)$ 

```

---

<sup>a</sup>Assume no connected component of  $G$  is small.

Now, we provide the proof sketch of [Theorem 1.1.6](#), which can be made precise with expectation.

**Proof Sketch of Theorem 1.1.6.** The correctness is easy to see as before. As for the running time, we see that [Borůvka's algorithm](#) takes  $O(m)$  time for each phase, so the total time for the preprocessing ([line 4](#)) is  $O(m)$ . Then, the recurrence for  $T(m, n)$  is

$$T(m, n) \leq c(m+n) + T(m/2, n/4) + T(2n/4 + n/4),$$

i.e., the resulting sub-problem is of size  $n/4 + m/2 + n/4 + n/2 = n + m/2$ , which is good enough assuming  $m \geq n - 1$ .<sup>a</sup> By a simple inductive proof, we can show that  $T(m, n) = O(n + m)$ . ■

<sup>a</sup>Since we eliminate small components including singletons.

**Remark.** A more refined analysis of the [sampling lemma](#) can be used to show that the running time is linear with high probability as well.

Many properties of forests and spanning trees can be understood in the more general context of *matroids*. In many cases this perspective is insightful and also useful. The [sampling lemma](#) applies in this more general context and has various applications [Kar95; Kar98]. Obtaining a deterministic  $O(m)$  time algorithm is a major open problem. Obtaining a simpler linear-time [MST verification](#) algorithm, even randomized, is also a very interesting open problem.

## 1.2 Tree Packing

We turn to another interesting problem, [tree packing](#).

**Problem 1.2.1 (Tree packing).** Given a multigraph  $G = (V, E)$ , find all the edge-disjoint [spanning trees](#) in  $G$ . In particular, find the maximum number,  $\tau(G)$ , of edge-disjoint [spanning trees](#) of  $G$ .

### 1.2.1 Bound on the Tree Packing Number

There is a beautiful theorem that provides a min-max formula for this. We first introduce some notation.

**Notation.** Let  $\mathcal{P}$  be the collection of partitions of  $V$ , and  $E_P$  is the edge between connected components induced by a partition  $P \in \mathcal{P}$ , i.e.,  $e \in E_P$  if its endpoints are in different parts of  $P$ .

It's easy to see that any [spanning tree](#) must contain at least  $|P| - 1$  edges from  $E_P$ . Thus, if  $G$  has  $k$  edge-disjoint [spanning trees](#), then

$$k \leq \frac{|E_P|}{|P| - 1}.$$

More generally, we have the following.

**Theorem 1.2.1.** The maximum number of edge-disjoint [spanning trees](#) in a graph  $G$  is given by

$$\tau(G) = \left\lfloor \min_{P \in \mathcal{P}} \frac{|E_P|}{|P| - 1} \right\rfloor.$$

**Remark.** [Theorem 1.2.1](#) is a special case of a theorem on matroid base packing where it is perhaps more natural to see [Sch+03].

A weaker version of the theorem is regarding fractional packing. In fractional packing, we allow one to use a fraction amount of a tree. The total amount to which an edge can be used is at most 1 (or  $c(e)$  in the capacitated case). Clearly, an integer packing is also a fractional packing. The advantage of fractional packings is that one can write a linear program for it, and they often have some nice properties. Let  $\tau_{\text{frac}}(G)$  be the fraction [tree packing](#) number. Clearly, we have  $\tau_{\text{frac}}(G) \geq \tau(G)$ .

**Corollary 1.2.1.** Given a graph  $G$ , we have

$$\tau_{\text{frac}}(G) = \min_{P \in \mathcal{P}} \frac{|E_P|}{|P| - 1}.$$

A second important corollary that is frequently used is about the min-cut. We see that while the min-cut size  $\lambda(G)$  of  $G$  is upper-bounding  $\tau(G)$ , i.e.,  $\tau(G) \leq \lambda(G)$ , this is not tight at all.

**Corollary 1.2.2.** Let  $G$  be a capacitated graph and let  $\lambda(G)$  be the global min-cut size. Then

$$\tau_{\text{frac}}(G) \geq \frac{\lambda(G)}{2} \frac{n}{n-1}.$$

**Proof.** Let  $P^*$  be the optimum partition that induces  $\tau_{\text{frac}}(G)$ . Then,  $\tau(G) = |E_{P^*}|/(|P^*| - 1)$ . Since for every connected component induced by  $P^*$ , at least  $\lambda(G)$  edges are going out, hence

$$\tau_{\text{frac}}(G) = \frac{|E_{P^*}|}{|P^*| - 1} \geq \frac{\lambda(G)/2 \cdot |P^*|}{|P^*| - 1} \geq \frac{\lambda(G)}{2} \frac{n}{n-1},$$

where we use the fact that  $|P^*| \leq n$  and  $i/(i-1)$  is decreasing.  $\blacksquare$

We first see a tight example.

**Example (Cycle).** Consider the  $n$ -node cycle  $C_n$ . Clearly,  $\tau(C_n) = 1$ , and  $\tau_{\text{frac}}(C_n) \leq n/(n-1)$  since each tree has  $n-1$  edges and there are  $n$  edges in the graph. Indeed, we have  $\tau_{\text{frac}}(C_n) = n/(n-1)$ . Finally, we see that  $\lambda(G) = 2$ .

**Proof.** Consider the  $n$  trees in  $C_n$  (corresponding to deleting each of the  $n$  edge) and assigning a fraction value of  $1/(n-1)$  for each of them, with the corresponding tight partition consists of the  $n$  singleton vertices.  $\circledast$

**Note.** Theorem 1.2.1 and its corollaries naturally extend to the capacitated case. For integer packing, we can assume  $c_e$  is an integer for each edge  $e$ , and the formula is changed to

$$\tau(G) = \left\lfloor \min_{P \in \mathcal{P}} \frac{c(E_P)}{|P| - 1} \right\rfloor.$$

Typically, one uses the connection between [tree packing](#) and min-cut to argue about the existence of many disjoint [trees](#), since the global minimum cut is easier to understand than  $\tau(G)$ . However, we will see that one can use [tree packing](#) to compute  $\lambda(G)$  exactly which may seem surprising at first due to the approximate relationship [Corollary 1.2.2](#).

## 1.2.2 Proof of Theorem 1.2.1

First, we prove the fractional version of [Theorem 1.2.1](#) (i.e., [Corollary 1.2.1](#)) via LP duality.

**Proof of Corollary 1.2.1 [CQ17].** Consider  $\mathcal{T}_G := \{T \mid T \text{ is a spanning tree of } G\}$ . Then, consider the following primal and the dual linear program:

$$\begin{array}{ll} \max & \sum_{T \in \mathcal{T}_G} y_T \\ & \sum_{T \ni e} y_T \leq c(e) \quad \forall e \in E; \\ \text{(P)} & y_T \geq 0 \quad \forall T \in \mathcal{T}_G; \end{array} \quad \begin{array}{ll} \min & \sum_{e \in E} c(e)x_e \\ & \sum_{e \in T} x_e \geq 1 \quad \forall T \in \mathcal{T}_G; \\ \text{(D)} & x_e \geq 0 \quad \forall e \in E. \end{array}$$

Let  $y^*$  and  $x^*$  be the optimal solution to the primal and the dual. Then from the strong duality,

$$\sum_{T \in \mathcal{T}_G} y_T^* = \tau_{\text{frac}}(G) = \sum_{e \in E} c(e)x_e^*.$$

We see that if there exists  $e$  such that  $x_e^* = 0$ , then we can just contract all these edges, so without loss of generality,  $x_e^* > 0$  for all  $e \in E$ .

**Intuition.** If  $x_e^* = 0$ , we can effectively increase  $c(e)$  to  $\infty$  without affecting the value of the dual solution, i.e.,  $e$  is not a bottleneck in the primal [tree packing](#), hence safe to contract.

**Claim.** If  $x_e^* > 0$  for all  $e \in E$ , then  $\tau_{\text{frac}}(G)$  is achieved via the singleton partition  $P$ . In particular,

$$\tau_{\text{frac}}(G) = \frac{\sum_{e \in E} c(e)}{n-1}.$$

**Proof.** From complementary slackness, we know that  $\sum_{T \ni e} y_T^* = c(e)$  for all  $e \in E$ . Hence,

$$(n-1) \sum_{T \in \mathcal{T}_G} y_T^* = \sum_{T \in \mathcal{T}_G} \sum_{e \in T} y_T^* = \sum_{e \in E} \sum_{T \ni e} y_T^* = \sum_{e \in E} c(e),$$

implying that  $\sum_{T \in \mathcal{T}_G} y_T^* = \sum_{e \in E} c(e)/(n-1)$ . ⊗

The above claim gives us the desired conclusion via induction: this is true if  $x_e^* > 0$  for all  $e \in E$ ; otherwise, we contract edges with  $x_e^* = 0$  and reduce to this case. ■

**Remark.** In the above proof, the dual can be interpreted as a relaxation for the min-cut problem. In fact, if  $x_e \in \{0, 1\}$ , then this is exact.

### 1.2.3 Finding an Optimum Tree Packing and Approximating Tree Packing

If the linear program in the [proof](#) of [Corollary 1.2.1](#) can be solved efficiently to get  $\tau_{\text{frac}}(G)$ , then it will also yield an algorithm for the value of the integer packing  $\tau(G)$  since it's just the floor of which. The problem is that while the primal has an exponentially many variables, the dual has an exponentially many constraints. We recall the following fact.

**As previously seen.** The Ellipsoid method needs a *separation oracle*. For example, applying it to the dual, we need to answer the following question efficiently:

- Given  $x \in \mathbb{R}^E$ , is it the case that  $\sum_{e \in T} x_e \geq 1$  for all  $T \in \mathcal{T}_G$ ?
- If not, find a tree  $T$  such that  $\sum_{e \in T} x_e < 1$ .

We see that this corresponds to solving [MST](#), hence, the dual admits an efficient solution via the Ellipsoid method. One can convert an exact algorithm for the dual to an exact algorithm for the primal.

**Remark.** There are combinatorial algorithms for solving [tree packing](#) (both integer version and fraction versions) in strongly polynomial time [[Sch+03](#)].

On the other hand, we're also interested in whether we can find a faster algorithm for [tree packing](#) if one allows approximation. With an adaption of the *multiplicative weights update* (MWU) method and data structures for [MST](#) maintenance, there is a near-linear time algorithm:

**Theorem 1.2.2** ([[CQ17](#)]). There is a deterministic algorithm to compute a  $(1 - \epsilon)$ -approximate fractional [tree packing](#) in  $O(m \log^3 n / \epsilon^2)$ .

## Lecture 3: Global Min-Cut with Tree Packing

### 1.3 Min-Cuts

3 Sep. 11:00

Consider the following famous problems about min-cuts.

**Problem 1.3.1 (*s-t* min-cut).** Given a graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ , the *s-t min-cut* problem aims to find  $\min_{S \subseteq V: s \in S, t \in V \setminus S} c(\delta(S))$ .

**Problem 1.3.2 (Global min-cut).** Given a graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ , the *global min-cut* problem aims to find  $\min_{\emptyset \neq S \subsetneq V} c(\delta(S))$ .

In what follows, we will simply use [min-cut](#) to refer to [Problem 1.3.2](#) problem. A naive way to solve it is to first fix one end  $s \in V$ , and compute the *s-t min-cut* for all  $t \in V - s$ . Fairly recent work shows how one can do it with only poly-log max-flow computations.

Over the years, several very different algorithmic approaches have been developed for these problems. One of the surprising ones is based on MA-orderings [NI92], which is a combinatorial  $O(mn + n^2 \log n)$  time algorithm that does not rely on flow at all.<sup>3</sup> Another approach is to combine several flow computations together via the push-relabel method [HO94], which also works for directed graphs. Karger developed elegant and powerful random contraction based algorithms for [global min-cuts](#) [Kar95], leading to many results. Two notable consequences are the following.

**Theorem 1.3.1** ([KS96]). There is a randomized algorithm that runs in  $O(n^2 \log n)$  time and outputs the [min-cut](#) with high probability.<sup>a</sup>

<sup>a</sup>This is a Monte-Carlo algorithm, so we cannot guarantee that the min-cut found is the correct one.

The following is a consequence of Karger’s contraction algorithm [Kar95].

**Theorem 1.3.2** (Approximate min-cut [Kar00]). The number of [α-approximate min-cuts](#) in a graph is at most  $O(n^{2\alpha})$ .

Karger then developed another approach via [tree packing](#) to obtain a randomized near-linear time algorithm for [min-cut](#). He also was able to refine the bound on [approximate min-cuts](#) via this approach.

**Theorem 1.3.3** ([Kar00]). There is a randomized algorithm that runs in time  $O(m \log^3 n)$  and outputs the [min-cut](#) with high probability.

While the random contraction based algorithm is taught quite frequently due to its elegance and simplicity, the [tree packing](#) approach is more technical. More recently, the [tree packing](#) approach has led to several new results, which we now discuss.

### 1.3.1 Tree Packing-Based Algorithm for Min-Cut

Recall that [Corollary 1.2.2](#), which gives

$$\frac{\lambda(G)}{2} \frac{n}{n-1} \leq \tau_{\text{frac}}(G) \leq \lambda(G).$$

Intuitively speaking, even if we can compute  $\tau_{\text{frac}}(G)$  exactly, the above only gives a 2-approximation to  $\lambda(G)$ . However, this already leads a crucial observation as follows.

**Intuition.** On average, each tree can’t cross the [min-cut](#) more than twice.

To formalize the above intuition, consider the following definition.

**Definition 1.3.1** (Respecting). Let  $T = (V, E_T)$  be a [spanning tree](#) and  $(S, V \setminus S)$  be a cut. The for an integer  $h \geq 1$ , we say  $T$  is *h-respecting* w.r.t.  $S$  if  $|E_T \cap \delta(S)| \leq h$ .

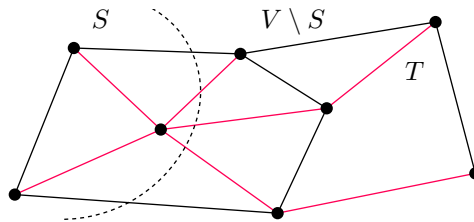


Figure 1.1: The [spanning tree](#)  $T$  is shown in red edges.  $T$  is [3-respecting](#) the cut  $(S, V \setminus S)$ .

We can now formalize the intuition in [Lemma 1.3.1](#).

<sup>3</sup>This approach generalizes to symmetric submodular functions.



**Lemma 1.3.1.** Suppose  $\{y_T\}_{T \in \mathcal{T}_G}$  is a  $(1 - \epsilon)$ -approximate **tree packing** of  $G$ , and  $\delta(S)$  is a **min-cut** of  $G$ . Let  $\ell_T := |E_T \cap \delta(S)|$  be the number of edges of  $T$  that cross the cut  $S$ . Furthermore, let  $p_T = y_T / \sum_{T \in \mathcal{T}_G} y_T$  and  $q := \sum_{T: \ell(T) \leq 2} p_T$ . Then,

$$q \geq \frac{1}{2} \left( 3 - \frac{2}{1 - \epsilon} \left( 1 - \frac{1}{n} \right) \right).$$

In particular, if  $\epsilon = 0$ , then  $1 \geq 1/2 + 1/n$ , and if  $\epsilon < 1/5$ , then  $q > 1/4$ .

**Proof.** From the assumption  $\sum_{T \in \mathcal{T}_G} y_T \geq (1 - \epsilon) \tau_{\text{frac}}(G)$ . With **Corollary 1.2.2**, i.e.,  $\tau_{\text{frac}}(G) \geq n\lambda(G)/2(n - 1)$ , we have

$$\sum_{T \in \mathcal{T}_G} y_T \geq (1 - \epsilon) \frac{n}{n - 1} \frac{\lambda(G)}{2}.$$

Now, let  $S \subseteq V$  be a **min-cut**, we have  $1 = \sum_{T \in \mathcal{T}_G} p(T) = \sum_{T: \ell(T) \leq 2} p_T + \sum_{T: \ell(T) \geq 3} p_T$ . We observe that

- each tree  $T$  with  $\ell(T) \geq 3$  uses up at least 3 edges from  $\delta(S)$ ; while
- each tree  $T$  with  $\ell(T) \leq 2$  uses up at least 1 edge from  $\delta(S)$ .

Since the total capacity of  $\delta(S)$  is  $\lambda(G)$ , and the **tree packing** solution is valid, we have

$$\sum_{T: \ell(T) \leq 2} y_T + 3 \sum_{T: \ell(T) \geq 3} y_T \leq \lambda(G) \Rightarrow q + 3(1 - q) \leq \frac{\lambda(G)}{\sum_{T \in \mathcal{T}_G} y_T} \leq \frac{2}{1 - \epsilon} \left( 1 - \frac{1}{n} \right),$$

where the last inequality follows from the very first inequality we have derived.  $\blacksquare$

**Remark.** **Lemma 1.3.1** states that if the **tree packing** is sufficiently good, then a constant fraction of the trees in the **packing** will cross the **min-cut** at most twice.

Now, we're ready to see Karger's algorithm for **min-cut** [Kar00]. However, the original algorithm was more involved since at that time, there was no near-linear time approximation algorithm for **tree packing**, so he used a form of sparsification and then applied an approximation **tree packing** algorithm on the sparsified graph which is quite a feat. In our case, recall that following.

**As previously seen.** **Theorem 1.2.2** states that we can compute a  $(1 - \epsilon)$ -approximate **tree packing** of  $G$ , given by  $\{y_T\}_{T \in \mathcal{T}_G}$ , in  $O(m \log^3 n / \epsilon^2)$  time.

By black-boxing this near-linear time **tree packing** algorithm, consider the following.

---

**Algorithm 1.9:** **Tree Packing-Based Min-Cut** Algorithm [Kar00; CQ17]

---

**Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ ,  $\epsilon_0 \in < 1/5$

**Result:** A cut  $S$

- 1  $\{y_T\}_{T \in \mathcal{T}_G} \leftarrow \text{Approximate-Tree-Packing}(G, c, \epsilon_0)$  //  $O(m \log^3 n)$
  - 2 Sample a tree  $T$  with probability  $p_T = y_T / \sum_{T \in \mathcal{T}_G} y_T$
  - 3 Find the cheapest cut  $(S, V \setminus S)$  in  $G$  such that  $T$  is **2-respecting** w.r.t.  $S$
  - 4 **return**  $S$
- 

Firstly, we see that **Algorithm 1.9** admits the following.

**Lemma 1.3.2.** **Algorithm 1.9** outputs the **min-cut** of  $G$  with probability at least  $1/4$ .

**Proof.** It's immediate from **Lemma 1.3.1**.  $\blacksquare$

To boost the success probability, we can simply repeat the last two steps (line 2, line 3)  $\Theta(\log n)$  times, which results in a success probability to at least  $1 - 1/n^c$  for any constant  $c$ . To analyze the running time, a key ingredient is line 3. Karger showed that one can implement line 3 via a clever dynamic programming coupled with link-cut tree data structure:



**Theorem 1.3.4** ([Kar00]). Given a graph  $G = (V, E)$  and a **spanning tree**  $T = (V, E_T)$ . There is a deterministic algorithm that computes a minimum cut  $(S, V \setminus S)$  such that  $T$  is **2-respecting** w.r.t.  $S$  in  $O(m \log^2 n)$  time.

We can now prove **Theorem 1.3.3**.

**Proof of Theorem 1.3.3.** Since **line 1** takes  $O(m \log^3 n)$  for  $\epsilon_0$  being a constant, and observe that once the approximated **tree packing**  $\{y_T\}_{T \in \mathcal{T}}$  is computed, we can reuse them and apply the repetition for **line 2** and **line 3** to boost the probability of success. With  $\Theta(\log n)$  repetitions, we obtain an  $O(m \log^3 n)$  time algorithm as desired with the running time guaranteed by **Theorem 1.3.4**. ■

### 1.3.2 Bounding the Number of Approximate Min-Cuts

As hinted in **Theorem 1.3.2**, we're now interested in how many distinct **min-cuts** can an undirected graph have. The following theorem was shown a long time ago:

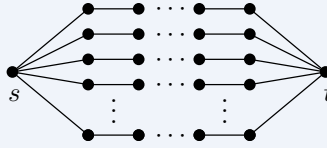
**Theorem 1.3.5** ([DKL76]). The number of distinct **min-cuts** in an undirected graph is at most  $\binom{n}{2}$ .

**Example** (Cycle). The worst case example is an  $n$ -cycle  $C_n$ .

**Remark.** All the **min-cuts** of a graph can be represented in a nice and compact data structure called the cactus (cactus representation), which was also shown in [DKL76].

In contrast, for  **$s$ - $t$  min-cuts**, it can be exponentially many in  $n$ .

**Example.** Consider the following multi-highway-like graph, which has exponentially many  **$s$ - $t$  min-cuts** since if we choose one of the road section in each line of the road, it'll be a  **$s$ - $t$  min-cut**.



Hence, we're interested in the number of  **$\alpha$ -approximation min-cut**:

**Definition 1.3.2** (Approximate min-cut). For  $\alpha \geq 1$  an  $\alpha$ -approximate min-cut is a cut  $(S, V \setminus S)$  such that  $c(\delta(S)) \leq \alpha \lambda(G)$ .

Recall **Theorem 1.3.2**, where Karger used **tree packing** to prove that the number of  **$\alpha$ -approximation min-cuts** is at most  $O_\alpha(n^{\lfloor 2\alpha \rfloor})$ . Before we prove **Theorem 1.3.2**, we recall some basic facts from linear programming.

**As previously seen.** A solution  $x^*$  to a linear program which has  $n$  non-trivial constraints means that the support size of  $x$  is at most  $n$ , i.e.,  $x_i > 0$  for at most  $n$  many  $i$ 's.

We're now ready to prove **Theorem 1.3.2**, which is based on [CQX20].

**Proof of Theorem 1.3.2.** Consider an optimum fraction **tree packing** solution  $\{(T, y_T^*)\}_{T \in \mathcal{T}_G}$ . In the **proof** of **Corollary 1.2.1**, where we define the fractional **tree packing** linear program, we know that there are only  $m$  non-trivial constraints, hence there are only  $m$  many  $T$ 's such that  $y_T^* > 0$ .

Consider an  **$\alpha$ -approximate min-cut**  $S \subseteq V$ , and let  $h = \lceil 2\alpha \rceil$ . Now, let  $q_{h,\alpha}$  be the fraction of **tree packing** that  $h$ -respects  $S \subseteq V$ , i.e.,

$$q_{h,\alpha} := \sum_{T: \ell(T) \leq h} p_T.$$

Using a similar analysis as the one in [Lemma 1.3.1](#), we can argue that

$$q_{h,\alpha} \geq \frac{1}{h}(1 - (2\alpha - \lfloor 2\alpha \rfloor)) \left(1 - \frac{1}{n}\right).$$

The main intuition is the following:

**Intuition.** Say at least one tree in the [packing](#) *h-respects* the cut (which is the case). Then, the total number of  *$\alpha$ -approximate min-cuts* is at most  $m \cdot n^h \leq m \cdot n^{\lfloor 2\alpha \rfloor}$ .

But we can do better by noticing that  $q_{h,\alpha} > 0$  is a fixed constant for any fixed  $\alpha$ . Suppose  $N$  is the number of  *$\alpha$ -approximate min-cuts*. For any fixed  *$\alpha$ -approximate min-cut*,  $q_{h,\alpha}$  fraction of the *tree packing* is *h-respecting* w.r.t. the cut. Consider the following question:

**Problem.** Fix a single tree  $T$ , how many distinct cuts are there such that  $T$  *h-respects* w.r.t.?

**Answer.** We can remove at most  $h$  edges from  $T$  to create at most  $h + 1$  components and combine these components into two sides of a cut, hence, each tree  $T$  correspond to at most  $2^{h+1} \binom{n-1}{h} \leq 2^{h+1} n^h$  cuts.  $\otimes$

Thus, the number of  *$\alpha$ -approximate min-cuts* is at most  $2^{h+1} n^h / q_{h,\alpha}$ .  $\blacksquare$

## Lecture 4: Steiner Min-Cut with Isolating Cuts

### 1.3.3 Steiner Min-Cut

5 Sep. 11:00

Consider the following problem that generalizes the *s-t min-cut* and *global min-cut*.

**Problem 1.3.3 (Steiner min-cut).** Given a graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$  and a set  $T \subseteq V$  of terminals, the *Steiner min-cut* problem aims to find the min-cut  $(S, V \setminus S)$  which separates some pair of terminals, i.e.,  $S \cap T \neq \emptyset$  and  $(V \setminus S) \cap T \neq \emptyset$ .

**Remark.** *Steiner min-cut* generalizes both *s-t min-cut* and *global min-cut*.

**Proof.** *s-t min-cut* corresponds to  $T = \{s, t\}$ , while *global min-cut* corresponds to  $T = V$ .  $\otimes$

A simple algorithm for the *Steiner min-cut* is the same as the *global min-cut* by solving *s-t min-cut*: for  $T = \{t_1, \dots, t_k\}$ , fix a terminal, say  $t_1$ , then compute  *$t_1$ - $t_i$  min-cut* for all  $i \geq 2$ . This requires  $|T| - 1$  max-flow computations. In fact, this is the best known algorithm even for the *global min-cut* till [NI92].

Quite recently, a simple yet striking approach that computes the *Steiner min-cut* with high probability using only  $O(\log^3 n)$  *s-t cut* computations is developed [LP20], which is based on *isolating cut*.

### Submodular Function

The main interest here, i.e., solving *isolating cut*, will be essentially based on properties of symmetric *submodular functions*. Although we can prove various properties by appealing to only graph theoretic facts, it's useful to see the proofs via *submodularity*. Here, we give some background, and specifically, for the cut function of graphs.

**Definition.** Given a finite ground set  $V$ , consider a real-valued set function  $f: 2^V \rightarrow \mathbb{R}$ .

**Definition 1.3.3 (Modular).** The function  $f$  is *modular* if for all  $A, B \subseteq V$ ,

$$f(A) + f(B) = f(A \cap B) + f(A \cup B).$$

**Definition 1.3.4 (Submodular).** The function  $f$  is *submodular* if for all  $A, B \subseteq V$ ,

$$f(A \cap B) + f(A \cup B) \leq f(A) + f(B).$$

**Definition 1.3.5 (Supermodular).** The function  $f$  is *supermodular* if for all  $A, B \subseteq V$ ,

$$f(A \cap B) + f(A \cup B) \geq f(A) + f(B).$$

**Definition 1.3.6 (Posi-modular).** The function  $f$  is *posi-modular* if for all  $A, B \subseteq V$ ,

$$f(A - B) + f(B - A) \geq f(A) + f(B).$$

We note that perhaps a more common definition of *submodularity* is *diminishing marginal utility*, i.e., if  $f(A + v) - f(A) \geq f(B + v) - f(B)$  for all  $A \subseteq B$ . Here, we see some examples.

**Example (Modular function as weight function).**  $f$  is *modular* if and only if there exists some  $w: V \rightarrow \mathbb{R}$  such that  $f(A) = \sum_{v \in A} w(v) + c$  for some shift  $c$ .

**Example.** If  $f$  and  $g$  are *submodular*, then so is  $\alpha f + \beta g$  for some  $\alpha, \beta \geq 0$ .

One of the reason that *submodularity* is important for graphs is because of the following.

**Example (Cut).** Given a graph  $G = (V, E)$ , the cut size function  $|\delta_G(\cdot)|: 2^V \rightarrow \mathbb{R}_+$  is *submodular*.

**Proof.** We simply note that for any  $A, B \subseteq V$ ,

$$|\delta_G(A)| + |\delta_G(B)| = |\delta_G(A \cap B)| + |\delta_G(A \cup B)| + 2|E(A \setminus B, B \setminus A)| \geq |\delta_G(A \cap B)| + |\delta_G(A \cup B)|,$$

where  $E(X, Y)$  is the set of edges crossing  $X$  and  $Y$  for some  $X, Y \subseteq V$ . ⊗

The above argument extends naturally to non-negative capacitated graph. Moreover, this is also true for directed graph.

**Example.** Let  $G = (V, E)$  be a directed graph.  $|\delta^+(\cdot)|$ , and hence by symmetry  $|\delta^-(\cdot)|$  are *submodular*.

We're also interested in the following property.

**Definition 1.3.7 (Symmetric).** A set function is *symmetric* if  $f(A) = f(V \setminus A)$  for all  $A \subseteq V$ .

Clearly,  $|\delta_G(\cdot)|$  is *symmetric*. However, for directed graph, this is not necessarily the case. Finally, we see that *symmetric submodular* function satisfies another important property.

**Example.** A *symmetric submodular* function is automatically *posi-modular*.

Now, we discuss uncrossing, a common and powerful technique that is frequently used in working with *submodular functions*. We illustrate this in the context of *min-cuts*.

**Lemma 1.3.3.** Let  $G = (V, E)$  be a graph and  $(A, V \setminus A)$ ,  $(B, V \setminus B)$  be two *s-t min-cuts*. Then  $(A \cap B, V \setminus (A \cap B))$  and  $(A \cup B, V \setminus (A \cup B))$  are also *s-t min-cuts*.

**Proof.** From *submodularity*, we have  $|\delta(A)| + |\delta(B)| \geq |\delta(A \cap B)| + |\delta(A \cup B)|$ . However, as both  $A \cup B$  and  $A \cap B$  are themselves *s-t cuts*, all terms need to be equal. ■

**Corollary 1.3.1.** For any graph  $G = (V, E)$ , there is a unique (inclusion-wise) minimal *s-t min-cut*.<sup>a</sup>

<sup>a</sup>While maybe not that useful, from the same logic, there is a unique maximal *s-t min-cut*.

**Proof.** If there are two  $s$ - $t$  min-cuts  $A, B$  that are both minimal and distinct, then  $A \setminus B \neq \emptyset$  and  $B \setminus A \neq \emptyset$  since otherwise one will be contained in the other, contradicting the minimality. From Lemma 1.3.3,  $A \cap B$  is also a  $s$ - $t$  min-cut and  $A \cap B$  is a strict subset of  $A$  and  $B$ , again contradicting minimality of  $A$  and  $B$ . ■

The above proof applies to directed graph as well since we only used submodularity.

**Remark (Graphic matroid).** A second aspect of submodularity in graphs comes via *matroids*. We will not discuss it here but the rank function of a matroid is a special class of submodular functions; and in a formal sense, matroid rank functions are building blocks for all submodular functions. Given an undirected graph  $G = (V, E)$  there is a fundamental matroid associated with the edge set of  $G$  called the *graphic matroid*.<sup>a</sup> Several properties of trees and forests can be better understood in the context of the graphic matroid including the *Tutte-Nash-Williams theorem*.

<sup>a</sup>There are other matroids that are also defined from graphs including the dual graphic matroid for instance.

## Isolating Cuts via Poly-log Max-flow Computations

We can now formally introduce the isolating cut problem.

**Problem 1.3.4 (Isolating cut).** Given a graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$  and a set  $T \subseteq V$  of terminals. The  $t_i$ -isolating cut problem aims to find a cut  $(S_i, V \setminus S_i)$  such that  $t_i \in S_i$  and  $t_j \notin S_i$  (i.e.,  $t_j \in V \setminus S_i$ ) for all  $j \neq i$ .

**Intuition.** In other words, the cut isolates  $t_i$  from the rest of the terminals.

The minimum capacity  $t_i$ -isolating cut can be found by a single max-flow computation: by shrinking the terminals in  $T - t_i$  into a single vertex  $s$  and computing the  $s$ - $t_i$  min-cut. Thus, naively, computing all isolating cuts require  $k$  max-flow computations. The upshot is that this can be done in only  $O(\log k)$  max-flow. Before we describe the algorithm, we first note that from submodularity, we also have a similar structural result, just like Corollary 1.3.1.

**Lemma 1.3.4.** There is a unique minimal  $t_i$ -isolating min-cut  $(S_i^*, V \setminus S_i^*)$  such that if  $(S_i, V \setminus S_i)$  is any  $t_i$ -isolating min-cut, then  $S_i^* \subseteq S_i$ .

We now describe the algorithm for computing the isolating cuts. Basically, we consider  $h$  bi-partitions  $(A_1, B_1), \dots, (A_h, B_h)$  with  $h = \lceil \log k \rceil$ , and compute a cut separating each bi-partition. Then, we take the intersection among the resulting cut sets, which will be isolating cuts as we will see. Finally, with the structural property Lemma 1.3.4, we can then find the minimum isolating cuts from them.

**Algorithm 1.10: Isolating Cut [LP20]** (also developed independently in [AKT21])

---

**Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ , terminal  $T = \{t_i\}_{i=1}^k$   
**Result:** A set of isolating cuts  $\{(S_i^*, V \setminus S_i^*)\}_{i=1}^k$  isolating  $t_i$ 's

---

```

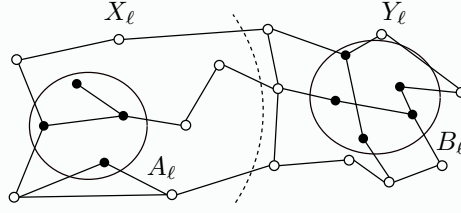
1  $h \leftarrow \lceil \log k \rceil$ 
2 for  $\ell = 1, \dots, h$  do                                     //  $h$  bi-partitions  $(A_\ell, B_\ell)$  of  $T$ 
3    $A_\ell \leftarrow \{t_i \mid \text{binary representation of } i \text{ has } 1 \text{ in } \ell^{\text{th}} \text{ bit}\}$ 
4    $B_\ell \leftarrow T \setminus A_\ell$ 
5    $(X_\ell, Y_\ell) \leftarrow s\text{-}t\text{-min-cut}(G, A_\ell, B_\ell)^a$            //  $Y_\ell = V \setminus X_\ell$ 
6 for  $i = 1, \dots, k$  do
7    $S_i \leftarrow \bigcap_{\ell: t_i \in A_\ell} X_\ell \cap \bigcap_{\ell: t_i \in B_\ell} Y_\ell$ 
8    $H_i = (V_i, E_i) \leftarrow \text{shrinking } V \setminus S_i \text{ to a single vertex } s_i$ 
9    $(S_i^*, V \setminus S_i^*) \leftarrow s\text{-}t\text{-min-cut}(H_i, t_i, s_i)$ 
10 return  $\{(S_i^*, V \setminus S_i^*)\}_{i=1}^k$ 

```

---

<sup>a</sup>This can be done via shrinking  $A_\ell$  and  $B_\ell$  into two separate nodes, and compute the  $s$ - $t$  min-cut.

**Intuition.** The following illustrates [line 5](#), where terminals are black vertices.  $(A_\ell, B_\ell)$  is a bi-partition of  $T$ , while  $(X_\ell, Y_\ell)$  is a [min-cut](#) that separates  $(A_\ell, B_\ell)$ .



Additionally, [line 7](#) is created by considering the intersections of all  $X_\ell$  (or  $Y_\ell$ ) that includes  $t_i$ .

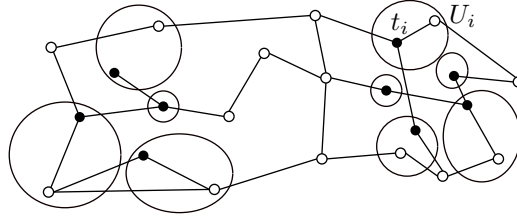
To see the correctness of the [Algorithm 1.10](#), we want to say that [s<sub>i</sub>-t<sub>i</sub> min-cut](#) is exactly the minimum cost [t<sub>i</sub>-isolating cut](#) in  $G$ . This is due to [Lemma 1.3.5](#).

**Lemma 1.3.5.** For each  $i$ ,  $(S_i, V \setminus S_i)$  is a [t<sub>i</sub>-isolating cut](#). Furthermore,  $S_i$ 's are pairwise disjoint.

**Proof.** Firstly,  $t_i \in A_\ell \subseteq X_\ell$  or  $t_i \in B_\ell \subseteq Y_\ell$ , implying  $t_i \in S_i$ . Consider  $t_j$  with  $j \neq i$ . As  $i \neq j$ , there is some index  $\ell$  in the binary representation of  $i$  and  $j$  differ in the bit position. Suppose  $i$  has 1 in the  $\ell^{\text{th}}$  position and  $j$  has 0, then  $t_i \in A_\ell$  and  $t_j \in B_\ell$ , implying  $t_i \in X_\ell$  and  $t_j \notin X_\ell$  as  $t_j \in B_\ell \subseteq Y_\ell$  and  $Y_\ell \cap X_\ell = \emptyset$ . This means  $t_j \notin S_i$ .

We now prove that  $S_i \cap S_j = \emptyset$  for all  $i \neq j$ . Firstly, there exists some  $\ell$  such that  $t_i \in A_\ell$  and  $t_j \in B_\ell$  (or  $t_i \in B_\ell$  and  $t_j \in A_\ell$ ). Suppose  $v \in X_\ell$ , then  $v$  can't be in  $S_j \subseteq Y_\ell$  and if  $v \in Y_\ell$ , then  $v$  can't be in  $S_i \subseteq X_\ell$ , hence  $v$  can't be in both  $S_i$  and  $S_j$ . ■

[Lemma 1.3.5](#) gives the following picture, where each  $t_i$  lives in exactly one  $S_i$ .



Hence, for each  $i$ , we have a [t<sub>i</sub>-isolating cut](#)  $(S_i, V \setminus S_i)$ . Now, [Lemma 1.3.4](#) states that there is a [t<sub>i</sub>-isolating min-cut](#)  $(S_i^*, V \setminus S_i^*)$  where  $S_i^*$  is a subset of any [t<sub>i</sub>-isolating min-cut](#), it doesn't say it will be a subset of  $S_i$  in particular, as  $(S_i, V \setminus S_i)$  is only a [t<sub>i</sub>-isolating cut](#). However, we do not lose anything:

**Lemma 1.3.6.** The minimal [t<sub>i</sub>-isolating min-cut](#)  $(S_i^*, V \setminus S_i^*)$  is in  $(S_i, V \setminus S_i)$ , i.e.,  $S_i^* \subseteq S_i$ .

**Proof.** It suffices to prove that if  $t_i \in A_\ell$  then  $S_i^* \subseteq X_\ell$ . Assume not, then  $S_i^* \cap (V \setminus X_\ell) \neq \emptyset$ . But since  $S_i^* \cap X_\ell$  is a [t<sub>i</sub>-isolating cut](#), while  $S_i^*$  is the minimal [t<sub>i</sub>-isolating min-cut](#),  $|\delta(S_i^* \cap X_\ell)| > |\delta(S_i^*)|$ .

Moreover, it's trivial to see that  $S_i^* \cup X_\ell$  is a  $A_\ell$ - $B_\ell$  cut (not necessarily minimum, just a cut), hence we also have  $|\delta(S_i^* \cup X_\ell)| \geq |\delta(X_\ell)|$ . From [submodularity](#) of  $|\delta(\cdot)|$ , we have

$$|\delta(S_i^*)| + |\delta(X_\ell)| \geq |\delta(S_i^* \cap X_\ell)| + |\delta(S_i^* \cup X_\ell)|,$$

which is a contradiction. ■

With all the lemmas, it's now easy to see that [Algorithm 1.10](#) is at least correct. Firstly, from [Lemma 1.3.6](#), we know that there the optimal [t<sub>i</sub>-isolating min-cut](#)  $S_i^* \subseteq S_i$  (here,  $S_i^*$  is not necessary the one found by [Algorithm 1.10](#): indeed, we're trying to argue this). As  $S_i$ 's are disjoint, each terminal  $t_i$  lives in exactly one  $S_i$ , hence computing [s<sub>i</sub>-t<sub>i</sub> min-cut](#) will indeed recover  $S_i^*$ .

**Intuition.** When we contract  $V \setminus S_i$ , we do not lose the optimal [isolating cut](#)  $S_i^*$ .

**Theorem 1.3.6 ([LP20]).** Algorithm 1.10 is a deterministic algorithm that given  $G = (V, E)$  and a terminal set  $T \subseteq V$  with  $|T| = k$ , computes all the **isolating cuts** using  $O(\log k)$  max-flow computations on graphs with  $|V|$  vertices and  $|E|$  edges each.

**Proof.** We analyze the runtime. It's easy to see that line 2 requires  $O(\log k)$  max-flow computations on  $G$ . It's also easy to show that computing  $S_i$ 's in line 7 can be done in  $O((m+n)\log k)$  time given  $(X_\ell, Y_\ell)$  for  $\ell \in [h]$ . However, line 8 and line 9 seem to require  $k$  max-flow computations.

**Claim.** In total, line 9 only requires  $O(\log k)$  max-flow computations.

**Proof.** Let us understand the size of  $H_i$ . It has  $n_i + 1$  vertices where  $n_i = |S_i|$ , and it has  $m_i$  edges where  $m_i = |E(S_i)| + |\delta(S_i)|$ . Thus, the running time of max-flow on  $H_i$  is  $T(n_i + 1, m_i)$  where  $T(a, b)$  is the running time of max-flow on graph with  $a$  nodes and  $b$  edges. We observe that  $\sum_i (n_i + 1) \leq 2n$  since  $S_i$ 's are disjoint, while  $\sum_i m_i \leq 2m$ : consider any edge  $uv \in E$ . If  $uv \in E(S_i)$  for some  $i$ , then it does not contribute to any other  $H_j$ . If  $uv \in \delta(S_i)$  for some  $i$ , then it can be in  $\delta(S_j)$  for only one more index  $j \neq i$ .

Thus, the total time to compute all  $k$  max-flows is  $\sum_i T(n_i, m_i) \leq T(2n, 2m)$  under reasonable assumption, specifically,  $T(a, b)$  is super-additive.<sup>a</sup> ⊗

<sup>a</sup>Formally, we first create a single  $H$  that includes each  $H_i$  as a copy in it, and we can run a single max-flow on  $H$  to recover all the max-flow values in each  $H_i$ .  $H$  will have  $O(n)$  vertices and  $O(m)$  edges.

With the correctness of Algorithm 1.10, the theorem is proved. ■

We see that this could have been discovered many years ago in terms of its simplicity. Algorithm 1.10 has been very influential in the last few years for a number of problems.

**Note.** Another perspective of the bi-partitions is that they are a way to *derandomize* a natural randomized algorithm that picks some  $O(\log k)$  bi-partitions of  $T$  at random and computes the cuts between them. With high probability, every  $t_i, t_j$  with  $i \neq j$  will be separated in at least one of the random bi-partitions.

**Remark.** The core idea of **isolating cuts** relies only on **submodularity** and symmetry, thus, this applies in much more generality and to several other problems. This is explicitly discussed in [CQ21], though the ideas are implicit in [LP20].

## Randomized Algorithm for Steiner Min-Cut via Isolating Cuts

**Isolating cut** naturally lead to a simple randomized algorithm for **Steiner min-cut**. The basic idea is quite simple. Consider an optimum **Steiner min-cut**  $(S, V \setminus S)$  and let  $T_1 := S \cap T$  and  $T_2 := (V \setminus S) \cap T$ , with  $k_1 = |T_1|$  and  $k_2 = |T_2|$ . We may assume that  $1 \leq k_1 \leq k_2$ .

**Note.**  $(S, V \setminus S)$  is a  $t_i$ - $t_j$  **min-cut** for any  $i \neq j$  since otherwise, it induces a lower-cost cut.

The basic intuition is the following.

**Intuition.** If we can sample exactly one terminal in one side of the **Steiner min-cut**, then we can simply use the **isolating cut** to recover the **Steiner min-cut**.

Say we know  $k_1$ . We can sample each terminal in  $T$  independently with probability  $1/k_1$  to obtain  $T' \subseteq T$  such that with constant probability,  $|T' \cap T_1| = 1$  and  $|T' \cap T_2| \geq 1$  (recall  $k_1 \leq k_2$ ). Suppose  $T'$  satisfies these properties and let  $T' \cap T_1 = \{t_i\}$ . Then,  $(S, V \setminus S)$  is a minimum cost  $t_i$ -**isolating cut** w.r.t.  $T'$ . Hence, by computing  $t_i$ -**isolating cuts** for all  $t_i \in T'$  and choosing the cheapest one identifies the **Steiner min-cut** for  $T$ .

The problem is that we don't know  $k_1$ , and trying all possible values for  $k_1$  (from 1 to  $k/2$ ) will be too expensive. The idea is that the above sampling procedure is robust: say if we sample with probability, say,  $1/2k_1$ , everything still happens with constant probability. Hence, we only need to try  $k_i = 2^i$ , i.e.,  $O(\log k)$  different sampling probabilities.

**Algorithm 1.11: Steiner Min-Cut****Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ , terminal  $T = \{t_i\}_{i=1}^k$ **Result:** A possible **Steiner min-cut**  $(U^*, V \setminus U^*)$ 

```

1  $U^* \leftarrow \emptyset$  // Initialize Steiner min-cut
2 for  $i = 0, \dots, \lceil \log k \rceil$  do
3    $T' \leftarrow \text{Sample}(T, 1/2^i)$  // Sample each terminal in  $T$  with probability  $1/2^i$ 
4    $\{(S_i^*, V \setminus S_i^*)\}_{i=1}^{|T'|} \leftarrow \text{Isolating-Cut}(G, c, T')$ 
5    $U^* \leftarrow \text{Min-Cost}(\{(S_i^*, V \setminus S_i^*)\}_{i=1}^{|T'|} \cup \{(U^*, V \setminus U^*)\})$  // Update minimum cost cut
6 return  $(U^*, V \setminus U^*)$ 

```

We now formally prove the robustness we have mentioned.

**Lemma 1.3.7.** Algorithm 1.11 finds the **Steiner min-cut** for  $T$  with a constant probability.

**Proof.** We see that for  $k_1 = 1$ , Algorithm 1.11 is correct (deterministically) since  $i$  can only be 0 and  $T' = T$ . Hence, let  $k_1 > 1$ . Consider the case that  $1/2^{i+1} < 1/k_1 \leq 1/2^i$ , where  $i$  will be tried at some point during  $i = 0$  to  $\lceil \log k \rceil$  since  $1 \leq k_1 \leq k/2$ . Let  $\ell = 2^i$ , i.e.,  $\ell \leq k_1 \leq 2\ell$ .

Now, let  $\mathcal{E}_1$  be the event that  $|T_1 \cap T'| = 1$ , i.e., exactly one terminal from  $T_1$  is chosen. Then

$$\Pr(\mathcal{E}_1) = k_1 \cdot \frac{1}{\ell} \cdot \left(1 - \frac{1}{\ell}\right)^{k_1-1} \geq \left(1 - \frac{1}{\ell}\right)^{2\ell} \geq \frac{1}{e^2}.$$

On the other hand, let  $\mathcal{E}_2$  be the event that  $T_2 \cap T' \neq \emptyset$ . We see that

$$\Pr(\mathcal{E}_2) \geq 1 - \left(1 - \frac{1}{\ell}\right)^{k_2} \geq \left(1 - \frac{1}{\ell}\right)^{\ell} \geq 1 - \frac{1}{e}.$$

Since  $T_1$  and  $T_2$  are disjoint,  $\mathcal{E}_1$  and  $\mathcal{E}_2$  are independent, we have

$$\Pr(\mathcal{E}_1 \cap \mathcal{E}_2) \geq \left(1 - \frac{1}{e}\right) \cdot \frac{1}{e^2},$$

which is a constant. ■

**Theorem 1.3.7.** There is a randomized algorithm that given  $G = (V, E)$  and terminal set  $T \subseteq V$  with  $|T| = k$ , outputs the **Steiner min-cut** with high probability using in  $O(\log^2 k \log n)$  max-flow computations.<sup>a</sup>

<sup>a</sup>Again, potentially on graphs with  $|V|$  vertices and  $|E|$  edges each from Theorem 1.3.6.

**Proof.** From Lemma 1.3.7, Algorithm 1.11 succeeds with a constant probability. We further boost the overall success probability by rerunning Algorithm 1.11  $\Theta(\log n)$  times. With Theorem 1.3.6, this requires  $O(\log^2 k \log n)$  max-flow computations. ■

**Remark (Deterministic algorithm).** Li and Panigraphy [LP20] also developed deterministic **min-cut** and **Steiner min-cut** algorithms using additional ideas based on expander decomposition.

## Lecture 5: Metric Embedding and Multi-Cut Problem

### 1.4 Metric Embedding

10 Sep. 11:00

Consider a directed graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ . Consider the following linear program relaxation of the **s-t min-cut** where  $s, t$  are two distinct vertices, where the integer version is



with constraints  $x_e \in \{0, 1\}$ :

$$\begin{aligned}
 \min \quad & \sum_{e \in E} c(e)x_e & \max \quad & \sum_{P \in \mathcal{P}_{s,t}} y_P \\
 \sum_{e \in P} x_e \geq 1 & \quad P \in \mathcal{P}_{s,t} & \sum_{P \ni e} y_P \leq c(e) & \quad \forall e \in E \\
 \text{(P)} \quad x_e \geq 0 & \quad \forall e \in E; & \text{(D)} \quad y_P \geq 0 & \quad \forall P \in \mathcal{P}_{s,t},
 \end{aligned} \tag{1.1}$$

where  $\mathcal{P}_{s,t}$  is the set of all  $s$ - $t$  paths.

**Remark.** An  $s$ - $t$  cut is often also defined as  $\delta^+(S)$  for some  $S \subseteq V$  where  $s \in S$  and  $t \in V \setminus S$ . Suppose  $E'$  is a  $s$ - $t$  cut and  $S$  is the set of nodes reachable from  $s$  in  $G - E'$ . Then,  $\delta(S) \subseteq E'$  and  $\delta(S)$  is an  $s$ - $t$  cut. Hence, it suffices to focus on such limited type of cuts.<sup>a</sup>

<sup>a</sup>In some more general settings, it is useful to keep these notions separate.

### 1.4.1 Max-Flow Min-Cut Theorem

It is well-known that  $s$ - $t$  min-cut can be computed efficiently via  $s$ - $t$  max-flow, which also establishes the **max-flow min-cut theorem**. This is a fundamental theorem in combinatorial optimization with many direct and indirect applications. The proof of the **max-flow min-cut** theorem is typically established via the augmenting path algorithm for computing a max-flow, here, we give another proof that is based on the above linear program.

**Theorem 1.4.1 (Max-flow min-cut).** Let  $G = (V, E)$  be a directed graph with rational edge capacities  $c: E \rightarrow \mathbb{Q}_+$  and let  $s, t \in V$  be two distinct vertices. The  $s$ - $t$  max-flow value in  $G$  is equal to the  $s$ - $t$  min-cut value, and both can be computed in strongly polynomial time. Furthermore, if  $c$  is integer valued, then there exists an integer-valued max-flow as well.

**Proof.** We first observe that the linear program can be viewed as assigning lengths to the edges such that the shortest path between  $s$  and  $t$  according to the lengths is at least 1, which is a fractional relaxation of the cut. Now, we show that it's possible to round the fractional solution of the primal to the exact  $s$ - $t$  min-cut without any loss.

---

#### Algorithm 1.12: $\theta$ -Rounding Algorithm

---

**Data:** A directed graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ ,  $s, t \in V$

**Result:** A  $s$ - $t$  min-cut  $F$

---

```

1  $\{x_e\}_{e \in E} \leftarrow \text{LP-Solve}(\text{Min-Cut-LP}(G, c, s, t))$  // Solve the primal
2  $\theta \leftarrow \text{Uniform}((0, 1))$ 
3 for  $v \in V$  do
4    $d_x(s, v) \leftarrow \text{Shortest-Path-Dist}(s, v, G, x)$ 
5  $F \leftarrow \delta^+(B_x(s, \theta))$  //  $B_x(s, \theta) = \{v \in V \mid d_x(s, v) \leq \theta\}$ 
6 return  $F$ 

```

---

We see that **Algorithm 1.12** will definitely output a valid  $s$ - $t$  cut since  $d_x(s, t) \geq 1$  by feasibility of the linear program solution  $x$  and hence  $t \notin B_x(s, \theta)$  for any  $\theta < 1$ .

**Claim.** For any  $e \in E$ ,  $\Pr(e \text{ is cut by Algorithm 1.12}) \leq x_e$ .

**Proof.** The edge  $e = (u, v)$  is cut if and only if  $d_x(s, u) \leq \theta < d_x(s, v)$ . Hence, the edge is not cut if  $d_x(s, v) \leq d_x(s, u)$ . If  $d_x(s, v) > d_x(s, u)$ , we have  $d_x(s, v) - d_x(s, u) \leq x_{(u,v)}$ . Since  $\theta$  is chosen uniformly at random from  $(0, 1)$ , the probability that  $\theta$  lies in the interval  $[d_x(s, u), d_x(s, v)]$  is at most  $x_{(u,v)}$ . ⊗

With this claim, from linearity of expectation, we see that  $\mathbb{E}[c(\delta^+(B_x(s, \theta)))] \leq \sum_{e \in E} c(e)x_e$ . As  $B_x(s, \theta)$  will always be a valid  $s$ - $t$  cut, this implies that there is an integral cut whose cost is at most that of the linear program relaxation, implying that the linear program relaxation yields an optimum solution.



Finally, observe that the dual is the *path version* of the  $s$ - $t$  max-flow. Hence, from strong duality, the optimal value of  $s$ - $t$  min-cut is the same as the  $s$ - $t$  max-flow. Moreover, we note that the primal is strongly polynomial-time solvable if we have a separation oracle. In this case, given  $\{x_e\}_{e \in E}$ , we need to answer either this is a feasible solution, or outputs some path  $p$  such that  $\sum_{e \in p} x_e < 1$ , which is exactly the shortest  $s$ - $t$  path algorithm and can be solved efficiently. ■

**Intuition (Line embedding).** The rounding can be thought as putting every vertex on a line from  $s$  to  $t$ , sorting by their distances given by  $x_e$ 's. Then, observe that on that line, any two vertices  $u, v \in V$  has distance at most  $x_{(u,v)}$ , and picking  $\theta$  corresponds to picking a threshold on the line.

The above intuition not only helps the analysis, but also gives a way to derandomize Algorithm 1.12. Basically, we can try all possible  $\theta$ 's, and if we adapt this line embedding viewpoint, the only interesting  $\theta$ 's are given by the  $n$  values  $d_x(s, v)$ .

### 1.4.2 Multi-cut and Approximation via Randomized Decomposition

Now, consider a more general question than  $s$ - $t$  cut called **multi-min-cut**.

**Problem 1.4.1 (Multi-min-cut).** Given a graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$  and  $k$  pairs of vertices  $\{(s_i, t_i)\}_{i=1}^k$ , the *multi-min-cut problem* aims to find a minimum capacity cut that separates all pairs.

**Remark.** **Multi-min-cut** is NP-hard even on trees. In general, it is a NP-complete problem.

Since this is a hard problem, we ask for an approximation algorithm. Firstly, an  $O(k)$ -approximation algorithm is trivial since we can simply output the union of all  $s_i$ - $t_i$  min-cuts. The goal is to design an  $O(\log k)$ -approximation algorithm, which also proves a multi-commodity flow-cut gap.

**Note.** It turns out that  $O(\log k)$  is tight in general graphs. For planar graphs, one can get an  $O(1)$ -approximation and flow-cut gap. These results are only for undirected graphs since the situation is more complicated in directed graphs, and we will discuss that later.

Again, we can write the following linear program relaxation for the **multi-min-cut problem**:

$$\begin{array}{ll}
 \min \sum_{e \in E} c(e)x_e & \max \sum_{i=1}^k \sum_{P \in \mathcal{P}_{s_i, t_i}} y_P \\
 \sum_{e \in P} x_e \geq 1 & P \in \mathcal{P}_{s_i, t_i}, i \in [k] \\
 \text{(P)} \quad x_e \geq 0 & \forall e \in E; \quad \text{(D)} \quad y_P \geq 0 \quad \forall P \in \mathcal{P}_{s_i, t_i}, i \in [k]. \\
 & \sum_{i=1}^k \sum_{P \ni e} y_P \leq c(e) \quad \forall e \in E \quad (1.2)
 \end{array}$$

The primal assigns distance labels  $x_e$  to edges so that, on each path  $P$  between  $s_i$  and  $t_i$ , the distance labels of these edges on  $P$  sum up to at least one, just like the  $s$ - $t$  min-cut.

**Remark.** Although the primal again has exponentially many constraints, from ellipsoid method, this is still efficiently solvable.

**Proof.** The separation oracle is as follows. Given distance labels  $x_e$ , we set the length of each edge to  $x_e$  and for each pair  $(s_i, t_i)$ , we compute the length of the shortest path between  $s_i$  and  $t_i$  and check whether it is at least one. This only takes  $k$  many times compared to the previous separation oracle for the  $s$ - $t$  min-cut linear program relaxation, hence it's still polynomial time. ⊛

On the other hand, the dual variable can be interpreted as the amount of flow between  $s_i$  and  $t_i$  that is routed along the path  $P$ . This is called the *maximum throughput multi-commodity flow* problem, where we don't care about individual demands, but only the overall flow. The dual tries to assign an amount of flow  $y_P$  to each path  $P$  so that the total flow on each edge is at most the capacity of the edge.

**Note.** The flow conservation constraints are automatically satisfied and the endpoints of the path  $P$  determine which kind of commodity is routed along the path.

Now, we will show that the integrality gap of the primal linear program is  $O(\log k)$  using a randomized rounding algorithm [CKR05].

**Note.** In fact, the first algorithm that achieved an  $O(\log k)$ -approximation for **multi-min-cut** is due to Garg, Vazirani, and Yannakakis [GVY93] (see [Vaz01; WS11]), and it's based on the region growing technique [LR99]. The reason we choose to present the randomized rounding algorithm is due to its future application for metric embedding.

The goal is to find a procedure to cut (i.e., decompose) the graph into components that satisfies the requirement of being a **multi-cut**, i.e.,

- each component has diameter at most 1 when the edge capacity is induced by the primal solution;
- the probability that edge  $e$  is cut is at most  $\alpha x_e$ ,

then we will get an  $\alpha$ -approximation algorithm.

**Intuition.** It essentially reduces to *low-diameter decomposition* in a metric space.

To do this, we consider the following algorithm.

---

**Algorithm 1.13:** Random Partition [CKR05]

---

**Data:** A connected graph  $G = (V, E)$  with edge capacity  $c: E \rightarrow \mathbb{R}_+$ ,  $\{(s_i, t_i) \mid s_i, t_i \in V\}_{i=1}^k$

**Result:** A **multi-min-cut**  $F$

```

1  $\{x_e\}_{e \in E} \leftarrow \text{LP-Solve}(\text{Multi-Min-Cut-LP}(G, c, \{(s_i, t_i)\}_{i=1}^k))$            // Solve the primal
2  $\theta \leftarrow \text{Uniform}([0, 1/2))$ 
3  $\sigma \leftarrow \text{Uniform}(S([k]))$            // Random permutation from permutation group  $S([k])$ 
4 for  $i = 1, \dots, k$  do
5    $V_{\sigma(i)} \leftarrow B_x(s_{\sigma(i)}, \theta) \setminus \bigcup_{j < i} V_{\sigma(j)}$ 
6  $F \leftarrow \bigcup_{i=1}^k \delta(V_i)$ 
7 return  $F$ 
```

---

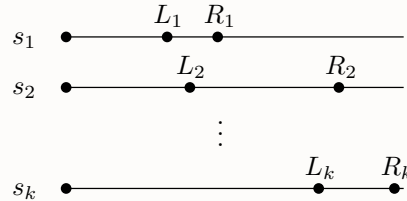
**Lemma 1.4.1.** Algorithm 1.13 outputs a feasible **multi-cut** for the given instance.

**Proof.** Suppose not, then there exists a pair  $(s_i, t_i)$  are still connected in  $G - F$ . This can only happen if  $s_i$  is “grabbed” by some terminals  $s_j$  which is proceeded before  $s_i$ , i.e., there exists some  $V_j \subseteq B_x(s_j, \theta)$  that contains both  $s_i$  and  $t_i$ . However, if  $s_j$  grabs both  $s_i$  and  $t_i$ , it means the distance between  $s_i$  and  $t_i$  is at most  $2\theta < 1$ , a contradiction to the feasibility of  $\{x_e\}_{e \in E}$ . ■

**Lemma 1.4.2.** For any  $e \in E$ ,  $\Pr(e = uv \text{ is cut by Algorithm 1.13}) \leq 2H_k x_e \leq O(\log k) x_e$ .<sup>a</sup>

<sup>a</sup> $H_k$  is the  $k^{\text{th}}$  harmonic number.

**Proof.** Let  $L_i := \min(d_x(s_i, u), d_x(s_i, v))$  and  $R_i := \max(d_x(s_i, u), d_x(s_i, v))$ . Without loss of generality, we can renumber  $s_i$ 's such that  $L_1 \leq L_2 \leq \dots \leq L_k$ .

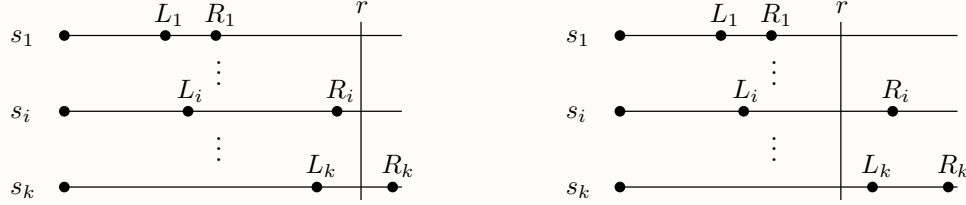


Let  $A_i$  be the event that  $s_i$  cut  $e = uv$  first, i.e.,  $A_i$  is the event that  $|V_i \cap \{u, v\}| = 1$  and  $|V_j \cap \{u, v\}| = 0$  for all  $j$  such that  $\sigma(j) < \sigma(i)$ , where  $|V_i \cap \{u, v\}| = 1$  simply says that  $s_i$  cuts the edge  $e$ . If  $A_i$  happens, then for all  $j$  that come before  $i$  in  $\sigma$ , neither  $u$  nor  $v$  can be in  $V_j$  since:

- if only one of  $u$  and  $v$  is in  $V_j$ , then  $s_j$  cuts  $e$ ;
- if both  $u$  and  $v$  are in  $V_j$ ,  $s_i$  can't cut  $e$  as the cut set only grabs the leftover vertices (line 5).

Let  $A$  be the event that  $e$  is cut, which is the union of the disjoint events  $A_i$ 's, hence  $\Pr(A) = \sum_{i=1}^k \Pr(A_i)$ . Now, for any fixed  $r \in [0, 1/2)$ , we see that

- $r \notin [L_i, R_i]$ : This is easy to understand as if  $r \notin [L_i, R_i]$ ,  $s_i$  is impossible to cut  $e$ ;
- $r \in [L_i, R_i]$ : Consider some  $j < i$  and suppose  $j$  comes before  $i$  in the permutation (i.e.,  $\sigma(j) < \sigma(i)$ ). Since  $j < i$ ,  $L_j \leq L_i \leq r$ . Hence, at least one of  $u$  and  $v$  is inside the ball of radius  $r$  centered at  $s_j$ . Consequently,  $s_i$  can't be the first to cut  $e$ , resulting in the fact that  $s_i$  is the first to cut the edge  $e$  if  $\sigma(i) < \sigma(j)$  for all  $j < i$ .



Since  $\sigma$  is a random permutation,  $i$  appears before  $j$  for all  $j < i$  with probability  $1/i$ . Hence,

$$\begin{cases} \Pr(A_i \mid \theta = r) = 0, & \text{if } \theta \notin [L_i, R_i]; \\ \Pr(A_i \mid \theta = r) \leq 1/i, & \text{if } \theta \in [L_i, R_i]. \end{cases}$$

As  $\theta$  is independent of  $\sigma$ , we have

$$\Pr(A_i) \leq \frac{1}{i} \cdot \mathbb{P}(\theta \in [L_i, R_i]) = \frac{2}{i} (R_i - L_i) \leq \frac{2x_e}{i}$$

from the triangle inequality  $R_i \leq L_i + x_e$ . This finally leads to

$$\Pr(A) = \sum_{i=1}^k \Pr(A_i) \leq \sum_{i=1}^k \frac{2x_e}{i} \leq 2H_k x_e,$$

and we conclude the theorem by noting that  $H_k = O(\log k)$ . ■

**Theorem 1.4.2.** Algorithm 1.13 is an  $O(\log k)$ -approximation (in expectation) algorithm for the multi-min-cut problem. Furthermore, the integrality gap of the multi-min-cut linear program is  $O(\log k)$ .

**Proof.** Let  $F$  be the set of edges outputted by Algorithm 1.13. For each edge  $e$ , let  $\xi_e = \mathbb{1}_{e \in F}$  in an indicator random variable. Hence, we have

$$\mathbb{E}[\xi_e] = \mathbb{P}(\xi_e = 1) \leq 2H_k x_e$$

from Lemma 1.4.2. This leads to

$$\mathbb{E}[c(F)] := \mathbb{E} \left[ \sum_{e \in F} c(e) \right] = \mathbb{E} \left[ \sum_{e \in E} c(e) \xi_e \right] = \sum_{e \in E} c(e) \Pr(\xi_e) \leq 2H_k \sum_{e \in E} c(e) x_e = 2H_k \text{OPT}_{\text{LP}}$$

where  $\text{OPT}_{\text{LP}}$  is the optimal value of the linear program. Since  $\text{OPT}_{\text{LP}} \leq \text{OPT}$  where  $\text{OPT}$  is the optimum value of the multi-min-cut problem, we have

$$\mathbb{E}[c(F)] \leq 2H_k \text{OPT}_{\text{LP}} \leq 2H_k \text{OPT} = O(\log k) \text{OPT}.$$

This also implies that there exists a set of edges  $F$  such that the total capacity of edges in  $F$  is at most  $2H_k \text{OPT}_{\text{LP}}$ , i.e.,  $\text{OPT}_{\text{LP}} \leq \text{OPT} \leq 2H_k \text{OPT}_{\text{LP}}$ , which proves the integrality gap result. ■

**Note.** The expected cost analysis can be used to obtain a randomized algorithm via repetition that outputs an  $O(\log k)$ -approximation with high probability. The algorithm can also be derandomized, but it's not straight forward.

**Remark (Flow-cut gap).** Recall that when  $k = 1$ , we have the [max-flow min-cut theorem](#). The integrality gap of the standard linear program for [multi-min-cut](#) is the same as the relative gap between flow and cut when  $k$  is arbitrary. The upper bound on the integrality gap gives an upper bound on the gap.

# Appendix

# Bibliography

- [AKT21] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. “Subcubic algorithms for Gomory–Hu tree in unweighted graphs”. In: *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. 2021, pp. 1725–1737.
- [Cha00] Bernard Chazelle. “A minimum spanning tree algorithm with inverse-Ackermann type complexity”. In: *Journal of the ACM (JACM)* 47.6 (2000), pp. 1028–1047.
- [CKR05] Gruia Calinescu, Howard Karloff, and Yuval Rabani. “Approximation algorithms for the 0-extension problem”. In: *SIAM Journal on Computing* 34.2 (2005), pp. 358–372.
- [CQ17] Chandra Chekuri and Kent Quanrud. “Near-linear time approximation schemes for some implicit fractional packing problems”. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2017, pp. 801–820.
- [CQ21] Chandra Chekuri and Kent Quanrud. “Isolating Cuts, (Bi-)Submodularity, and Faster Algorithms for Global Connectivity Problems”. In: *CoRR* abs/2103.12908 (2021). arXiv: [2103.12908](https://arxiv.org/abs/2103.12908). URL: <https://arxiv.org/abs/2103.12908>.
- [CQX20] Chandra Chekuri, Kent Quanrud, and Chao Xu. “LP relaxation and tree packing for minimum k-cut”. In: *SIAM Journal on Discrete Mathematics* 34.2 (2020), pp. 1334–1353.
- [DKL76] Efim A Dinitz, Alexander V Karzanov, and Michael V Lomonosov. “On the structure of the system of minimum edge cuts of a graph”. In: *Issledovaniya po Diskretnoi Optimizatsii* (1976), pp. 290–306.
- [DRT92] Brandon Dixon, Monika Rauch, and Robert E Tarjan. “Verification and sensitivity analysis of minimum spanning trees in linear time”. In: *SIAM Journal on Computing* 21.6 (1992), pp. 1184–1192.
- [Eis97] Jason Eisner. “State-of-the-art algorithms for minimum spanning trees”. In: *Unpublished survey* (1997).
- [FT87] Michael L Fredman and Robert Endre Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms”. In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.
- [Gab+86] Harold N Gabow et al. “Efficient algorithms for finding minimum spanning trees in undirected and directed graphs”. In: *Combinatorica* 6.2 (1986), pp. 109–122.
- [GVY93] Naveen Garg, Vijay V Vazirani, and Mihalis Yannakakis. “Approximate max-flow min-(multi) cut theorems and their applications”. In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 1993, pp. 698–707.
- [HO94] JX Hao and James B Orlin. “A faster algorithm for finding the minimum cut in a directed graph”. In: *Journal of Algorithms* 17.3 (1994), pp. 424–446.
- [Kar00] David R Karger. “Minimum cuts in near-linear time”. In: *Journal of the ACM (JACM)* 47.1 (2000), pp. 46–76.
- [Kar95] David Ron Karger. *Random sampling in graph optimization problems*. stanford university, 1995.
- [Kar98] David R Karger. “Random sampling and greedy sparsification for matroid optimization problems”. In: *Mathematical Programming* 82.1 (1998), pp. 41–81.
- [Kin97] Valerie King. “A simpler minimum spanning tree verification algorithm”. In: *Algorithmica* 18 (1997), pp. 263–270.
- [KKT95] David R Karger, Philip N Klein, and Robert E Tarjan. “A randomized linear-time algorithm to find minimum spanning trees”. In: *Journal of the ACM (JACM)* 42.2 (1995), pp. 321–328.

- [Kom85] János Komlós. “Linear verification for spanning trees”. In: *Combinatorica* 5.1 (1985), pp. 57–65.
- [KS96] David R Karger and Clifford Stein. “A new approach to the minimum cut problem”. In: *Journal of the ACM (JACM)* 43.4 (1996), pp. 601–640.
- [LP20] Jason Li and Debmalya Panigrahi. “Deterministic min-cut in poly-logarithmic max-flows”. In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2020, pp. 85–92.
- [LR99] Tom Leighton and Satish Rao. “Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms”. In: *Journal of the ACM (JACM)* 46.6 (1999), pp. 787–832.
- [Mar08] Martin Mareš. “The saga of minimum spanning trees”. In: *Computer Science Review* 2.3 (2008), pp. 165–221.
- [NI92] Hiroshi Nagamochi and Toshihide Ibaraki. “Computing edge-connectivity in multigraphs and capacitated graphs”. In: *SIAM Journal on Discrete Mathematics* 5.1 (1992), pp. 54–66.
- [PR02] Seth Pettie and Vijaya Ramachandran. “An optimal minimum spanning tree algorithm”. In: *Journal of the ACM (JACM)* 49.1 (2002), pp. 16–34.
- [Sch+03] Alexander Schrijver et al. *Combinatorial optimization: polyhedra and efficiency*. Vol. 24. 2. Springer, 2003.
- [Vaz01] Vijay V Vazirani. *Approximation Algorithms*. 2001.
- [WS11] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.
- [Yao75] Andrew Chi-Chih Yao. “An  $O(|E| \log \log |V|)$  algorithm for finding minimum spanning trees”. In: *Information Processing Letters* 4 (1975), pp. 21–23.