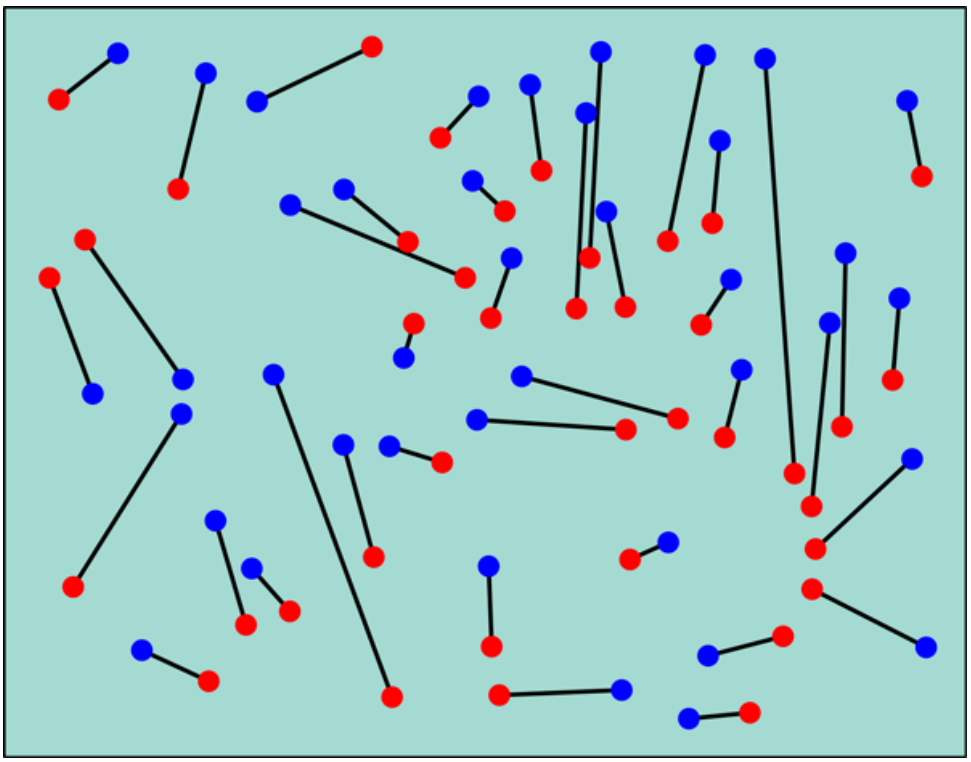
CS598
Topics in Graph Algorithms

Pingbang Hu

August 29, 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

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 $O(m \log^3 nC)$, where C is the largest absolute value of the *integer* length.

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

cite

cite

cite

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

cite

[a] 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 weight $c \colon E \to \mathbb{R}$, 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,[1] 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.

[1] 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^{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 weight $c\colon E \to \mathbb{R}$
Result: A MST $T = (V, F)$

```
1 Sort the edges such that c(e_1) ≤ c(e_2) ≤ ··· ≤ c(e_m)        // O(m log n)
2 F ← ∅                                                          // Initialize the tree
3 for i = 1, ..., m do
4     if e_i + F has no cycle then
5         F ← 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 weight $c\colon E \to \mathbb{R}$
Result: A MST $T = (V, F)$

```
1 r ← uniform(V)                                          // Sample a root
2 V' ← {r}, F ← ∅                                         // Initialize the tree
3 while V' ≠ V do
4     e ← arg min_{e=(u,v)∈δ(V'),u∈V'} c(e)
5     F ← F + e, V' ← 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 weight $c\colon E \to \mathbb{R}$
Result: A MST $T = (V, F)$

```
1 F = ∅                                                   // Initialize the tree
2 S ← {S_v = {v}}                                         // Collection of all sets
3 while |S| > 1 do
4     for S ∈ S do
5         e_S = (u, v) ← arg min_{e∈δ(S)} c(e)
6         S ← S − {S_u, S_v} + S_u ∪ S_v                  // Merge (i.e., shrink)
7         F ← 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 $O(m \log^* n)$.[2] 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 Jarnik-Prim's algorithm.

As previously seen. Jarnik-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^{th} phases, reduces the number of nodes to $n/2^i$; each phase takes $O(m)$ times.

[2] 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 weight $c\colon E \to \mathbb{R}$
Result: A MST $T = (V, F)$

```
V' ← V, F ← ∅                                              // Initialize the tree
while |V| > 1 do
    T ← Grow(G)                                            // First phase
    F ← F ∪ E(T)                                           // Update the tree
    Shrink G w.r.t. T, update V and E                      // Second phase
return (V', F)

Grow(G):
    V' ← ∅, F ← ∅, T ← (V', F)                             // Initialize the forest
    while V' ≠ V do
        r ← uniform(V − V')                                // Pick an unmarked vertex
        T' ← ({r}, ∅)                                      // Initialize a tree
        while |N(T')| < t or V(T') ∩ V' ≠ ∅ do
            Run one more step of Jarnik-Prim(r, T')        // Starting at r, maintaining T'
        V' ← V' ∪ V(T')                                    // Mark
        F ← F ∪ E(T')                                      // Update the forest by merging the tree
    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 $C_1, \ldots, C_h$ 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. ⊛

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 $t_i := 2^{2m/n_i}$,[a] where n_i and m_i are the number of vertices and edges at the beginning of the i^{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 $O(m\beta(m, n))$. ■

[a] 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 $1 - 1/\mathsf{poly}(m)$.

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[a] in $G' = (V, E')$. Then the expected number of F-light edge in G is less than $(n - 1)/p$.

[a] 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 weight $c\colon E \to \mathbb{R}$, probability $p \in (0, 1)$
Result: A minimum spanning forest F and the set of F-light edges A

Sort the edges such that $c(e_1) \le c(e_2) \le \cdots \le c(e_m)$
$A \leftarrow \varnothing$, $F \leftarrow \varnothing$, $E' \leftarrow \varnothing$
for $i = 1, \ldots, m$ **do**
    $r \leftarrow$ `Ber(`p`)` `// Toss a biased coin`
    if $r = 1$ **then**
        $E' \leftarrow E' + e_i$
        if $F + e_i$ *is a forest* **then**
            $F \leftarrow F + e_i$
            $A \leftarrow A + e_i$
    else if e_i *is* F*-light* **then**
        $A \leftarrow A + e_i$
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 weight $c\colon E \to \mathbb{R}$, probability $p \in (0, 1)$
Result: A minimum spanning forest F and the set of F-light edges A

Sort the edges such that $c(e_1) \le c(e_2) \le \cdots \le c(e_m)$
$A \leftarrow \varnothing$, $F \leftarrow \varnothing$
for $i = 1, \ldots, m$ **do**
    if e_i *is* F*-light* **then** `// Sorting implies` $F + e_i$ `is a forest` $\Leftrightarrow$ e_i `is` F`-light`
        $A \leftarrow A + e_i$
        $r \leftarrow$ `Ber(`p`)` `// Toss a biased coin`
        if $r = 1$ **then**
            $F \leftarrow F + e_i$
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, $p\mathbb{E}[|A|] = \mathbb{E}[|F|] \le n - 1$, hence $\mathbb{E}[|A|] \le (n-1)/p$. ■

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 weight $c\colon E \to \mathbb{R}$, 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 weight $c\colon E \to \mathbb{R}$
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 Ber(1/2) to obtain E_1 ⊆ E
5 (V, F_1) ←Karger-Klein-Tarjan((V, E_1))                  // Recursively compute MSF
6 E_2 ←Light-Edge(G, F_1)            // Compute all F_1-light edges with Theorem 1.1.4
7 (V, F_2) ←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:

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

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 $G = (V, E)$[a] with edge weight $c\colon E \to \mathbb{R}$
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') ←Borůvka(G, c, 2)   // Run two iterations with |V'| ≤ |V|/4.
5
6  Sample each edge in G' i.i.d. from Ber(1/2) to obtain E_1 ⊆ E'
7  (V', F_1) ←Karger-Klein-Tarjan((V', E_1))                 // Recursively compute MSF
8  E_2 ←Light-Edge(G_1, F_1)                 // Compute F_2-light edges with Theorem 1.1.4
9  (V', F_2) ←Karger-Klein-Tarjan((V', E_2))                 // Recursively compute MSF
10 return (V, F' ∪ F_2)
```

[a] 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

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

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 $m \ge n - 1$.[a] By a simple inductive proof, we can show that $T(m, n) = O(n + m)$. ■

[a] 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. ■

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. ⊛

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:

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

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 ∞ 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, i.e., $\tau_{\text{frac}}(G) = \min_{P \in \mathcal{P}} c(E_P)/(|P|-1)$ 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 efficient 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 finding 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. It's shown that a $(1-\epsilon)$-approximation fraction tree packing can be found in $O(m \log^3 n/\epsilon^2)$ time using an adaption of the *multiplicative weights update* (MWU) method and data structures for MST maintenance. We may see it later in the course.

Appendix

Bibliography

[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.

[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.

[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.

[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.

[Mar08] Martin Mareš. "The saga of minimum spanning trees". In: *Computer Science Review* 2.3 (2008), pp. 165–221.

[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.

[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.