

CS598
Topics in Graph Algorithms

Pingbang Hu

August 27, 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

1	Introduction	2
1.1	Minimum Spanning Tree	2

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 $O(m \ln^3 n C)$, 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 maxflow). The tradition running time is $O(mn \ln m/n)$, and it's later improved to be $O(m\sqrt{n} \ln n C)$. Recently, [Chen et-al] improve to $O(m^{1+o(1)})$, which is almost-linear.^a

cite

^aThis 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 $G = (V, E)$ and edge weight $c: E \rightarrow \mathbb{R}$, 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**,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**.

¹This 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.

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 $G = (V, E)$ and edge weight $c: E \rightarrow \mathbb{R}$

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

```

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

Lemma 1.1.3. *Kruskal's algorithm* takes $O(m \ln n)$.

Proof. Sorting takes $O(m \ln n)$ time. The standard solution for this 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 $O(m \ln n)$. ■

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 $G = (V, E)$ and edge weight $c: E \rightarrow \mathbb{R}$

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

Lemma 1.1.4. Jarnik-Prim's algorithm takes $O(m + n \ln n)$.

Proof. To find the cheapest edge leaving T , one typically uses a priority queue data structure 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 u is added to T the algorithm scans the edges in $\delta(u)$ to update the keys of neighbors of v . Thus, one sees that there are a total of $O(m)$ decrease-key operations, $O(n)$ delete-min operations, and initially we set up an empty queue. Standard priority queues implement decrease-key and delete-min in $O(\ln n)$ time each, so the total time is $O(m \ln n)$. 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 $O(m + n \ln n)$. ■

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

Boruvka's Algorithm

Boruvka's algorithm seems to be the first MST algorithm. It has very nice properties and essentially uses no data structures. The algorithm works in phases. We describe it recursively to simplify the description. 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 that 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. One can easily implement this in $O(m)$ time by a linear scan of the adjacency lists. It then shrinks the connected components induced by F and recurses on the resulting graph $H = (V', E')$. Computing H 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 since we add an edge leaving v to F . Thus the algorithm terminates in $O(\ln n)$ phases for a total of $O(m \ln n)$ time. Note that this algorithm is easy to parallelize unlike the other two algorithms.

Algorithm 1.3: Boruvka's algorithm

Data: A connected graph $G = (V, E)$ and edge weight $c: E \rightarrow \mathbb{R}$

Result: A MST $T = (V, F)$

1.1.2 Faster Algorithms

A natural question is whether there is a linear time MST algorithm, that is an algorithm that runs in $O(m)$. Very early on, Yao, in 1975, obtained an algorithm that ran in $O(\ln \ln n)$ [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 $O(m \ln^* n)$.² This was further improved to $O(m \ln \ln^* n)$ [Gab+86]. Karger, Klein, and Tarjan [KKT95] obtained a linear time randomized algorithm that will be the main topic of this lecture. Finally, 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 , is T an MST of G ?

Clearly, 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. It is based on several non-trivial ideas and data structures and was first developed in the RAM model³ by Dixon, Rauch, and Tarjan [DRT92] based on insights from Komlós [Kom85], with the simplification done by King [Kin97].

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

³The RAM model allows bit-wise operation on $O(\ln n)$ bit words in $O(1)$ time.

Theorem 1.1.1 (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 .

Remark. The original algorithm is quite complicated, and it has been simplified over the years. See lecture notes of [Gupta](#) and [Assadi](#) for accessible explanation, also the MST surveys by Eisner [[Eis97](#)] and Mareš [[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 $O(m \ln \ln n)$ time algorithm by combining [Boruvka's algorithm](#) and [Jarnik-Prim's algorithm](#).

Recall that [Jarnik-Prim's algorithm](#) takes $O(m + n \ln n)$ time via Fibonacci heaps, and this is already very good if m is large. The bottleneck is when $m = o(n \ln n)$. On the other hand, [Boruvka'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.

Intuition. Suppose we run [Boruvka'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 [Boruvka's algorithm](#), and $O(m + n/2^k \ln n/2^k)$ for the [Jarnik-Prim's algorithm](#) on the reduced graph. Thus, if we choose $k = \ln \ln n$, we obtain a total run-time of $O(m \ln \ln n)$.

- Run the boruksr algorithm for h phases.
- Let H be the reduced graph. Run JP algorithm with Fibonacci heap on H .

This gives $O(mh + m' + n' \log n')$, where m' and n' is for the reduced graph, i.e.,

$$O\left(mh + m + \frac{n}{2^h} \ln \frac{n}{2^h}\right) = O(m \ln \ln n)$$

by choosing $h := \log \log n$.

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. 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 following the [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.

Algorithm 1.4: Fredman-Tarjan's algorithm

Data: A connected graph $G = (V, E)$ and edge weight $c: E \rightarrow \mathbb{R}$

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

1 **return** (V, F)

Note. Note that the heap, while growing T_2 , 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.

The algorithm proceeds in this fashion by picking new roots and growing them until all nodes are marked. We see that the algorithm correctly adds a set of **MST** edges F .

Claim. Number of vertices after shrinking the connected components induced by F is $\leq 2m/t$.

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

$$2m = \sum_{v \in V} \deg(v) = \sum_{i=1}^h \sum_{v \in C_i} \deg(v) \geq ht,$$

which implies $h \leq 2m/t$. To see why this is the case, consider the growth of a tree T_j by the algorithm:

- If we stop T_j because heap size exceeds t , then each of the vertex in the heap is a witness to a unique edge incident to T_j , hence the property holds.
- If T_j 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 the algorithm terminates a tree because all vertices are already included in it, but then the phase finishes the algorithm. \circledast

Theorem 1.1.2. The **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 delete-min operations take $O(n \ln t)$. Note that we're utilizing the fact that the initialization of each data structure is easy because it starts as an empty one. Thus, we have obtained a new algorithm that, in a single phase, takes $O(m + n \ln t)$ time and reduces the number of vertices to $2m/t$ based on the **previous claim**.

Note. This can be seen as a parameterized version of **Boruvka's algorithm**.

The question reduces to choosing t .

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

Now we need to bound the overall time, i.e., the number of phases. 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 iteration i , with $m_1 = m$ and $n_1 = n$. From the **claim**, $n_{i+1} \leq 2m_i/t_i$, giving

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

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 $\beta(m, n)$ phases, giving the total time $O(m\beta(m, n))$. \blacksquare

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

Linear-Time Randomized Algorithm

Lemma 1.1.5 (Sampling lemma). Suppose $G = (V, E)$ is a graph. Let $E' \subseteq E$ be obtained by sampling each edge e with probability $1/2$. Let F be a minimum spanning tree in E' . Then the expected number of F -light edge in G is less than $2n$.

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.
- [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.
- [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.
- [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.