6.1220 LECTURE 8

BRENDON JIANG

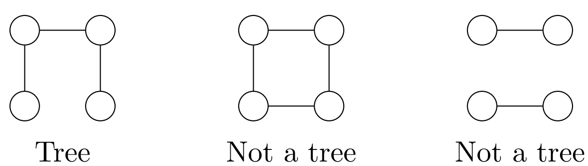
Contents

1. Minimum Spanning Trees

1.1. Spanning Trees

Definition 1.1: Tree

A tree is connected undirected graph with no cycles.



Definition 1.2: Spanning Tree

A **spanning tree** of an undirected graph $G = (V, E)$ is a tree $T = (V', E')$ with vertex set $V = V'$ and edge set $E' \leq E$.

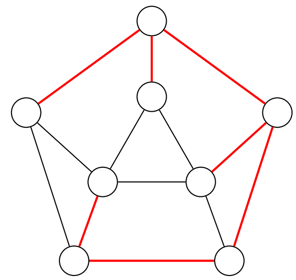
Essentially, the spanning tree is the thinnest tree one can create while still maintaining the connectivity of G. Now we look at properties of a spanning tree:

Proposition 1.3: Spanning Tree Properties

A spanning tree has these properties:

(1) The number of edges is $|V| - 1$.
(2) Not necessarily unique (there are multiple spanning trees in a graph).
(3) Suppose $e = (u, v)$ is not in spanning tree T. Then

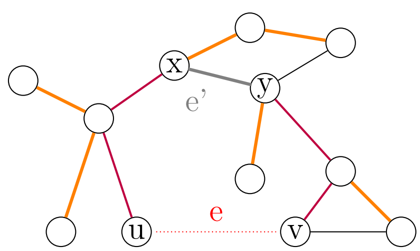
Date: September 2024



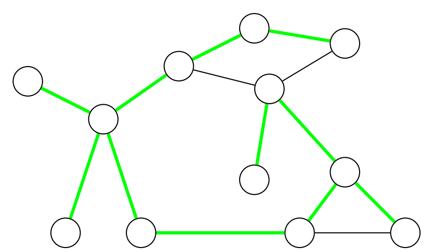
A spanning tree highlighted in red.

- There is a unique path P_e in T connecting u and v.
- $P_e \cup \{e\}$ forms a cycle called the fundamental cycle of e

(4) Suppose $e = (u, v)$ is not in T, and $e' \in P_e$. Then, the graph $T' = T \cup \{e\} \backslash \{e'\}$ is also a spanning tree.



T (orange) and the fundamental cycle of e (purple).



T’, the new spanning tree

Proof of property 4

It is easy to show that T' has $|V| - 1$ edges. We now want to argue vertices in V remain connected by T'.
Take any two vertices u', v' of vertices in T'. If their path in T doesn't use the edge e', they remain connected (all edges remain in T'). If the path between u' and v' uses the edge e', we can use the path $P_e \cup e \backslash \{e'\}$ in place of e' to achieve the same purpose.

1.2. Minimum Spanning Tree Problem

We now present the minimum spanning tree problem.

Definition 1.4: Minimum Spanning Tree (MST) Problem

Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$. We want to find a minimum spanning tree, which is a spanning tree T of G with $w(T) := \sum_{e \in T} w(e)$ minmized.

We can approach this problem using all sorts of greedy algorithms. Here is a template for these greedy algorithms, which maintains an invariant that ensures the set A at all stages is part of some MST:

Algorithm 1: Meta-Greedy-MST template

```
A ← ∅
while A is not a spanning tree do
    Find some "safe" e such that A ∪ {e} always is the subset of some MST
    A ← A ∪ {e}
return A.
```

Example 1.5: Idea for a finding a safe e

Suppose $A = \emptyset$. Then a minimum weight edge e is safe.

Proof

Suppose T is a MST not containing minimum weight edge e. Then, we can take $T' = T \cup \{e\} \backslash \{e'\}$, which is also a spanning tree by property 4 of spanning trees. Since e is has the minimal weight, we could not have increased the total weight, so T' is a minimum spanning tree. In equation form, this is

$$w(T') = w(T) - w(e') + w(e) \leq w(T)$$

since $w(T)$ is minimal, $w(T')$ is also minimal.

Example 1.6: Continuing idea for a finding a safe e

Now we suppose A is a forest (trees and isolated vertices). We claim that any minimum weight edge $e \notin A$ that does not create a cycle is safe.

This is the nature extension of the previous example. However, we can come up with a stronger condition, called the strong safe edge statement (don't know if this is an official name lol):

Theorem 1.7: Strong Safe Edge Statement

Let A be any subset of edges of $G = (V, E)$ and a strict subset of some MST of G.

- Consider the set $S \subseteq V$ such that there are no edge $(u, v) \in A$ satisfies $u \in S, v \in V \backslash S$.
- Suppose $e = (x, y)$ such that $x \in S$, $y \in V \backslash S$ and of minimum weight.

Then, $A \cup \{e\}$ is also in some MST, equivalently e is safe.

Proof of the Strong Safe Edge Statement

Suppose T is a MST containing A. If $e = (a, b) \in T$, then we are done.

Otherwise, consider P_e which is the unique path in T connecting a and b. P does not have e, so it needs to have some other "cut" edge e' that goes across from S to $V \backslash S$. By spanning tree property 4, $T' = T \cup \{e\} \backslash \{e'\}$ is a spanning tree. But again,

$$w(T') = w(T) + w(e) - w(e') \leq w(T)$$

so T' is a MST that contains e.

1.3. MST Algorithms

Now we show two algorithms that follow this greedy algorithm template.

Definition 1.8: Kruskal's algorithm

We always keep a forest A. In each iteration, we add to A the lightest edge $e = (a, b)$ (minimum weight) connecting two trees of this forest.

The correctness of Kruskal's algorithm follow immediately from the strong safe edge statement, as we can take cut S to be the tree in A that contains the vertex a.

To implement Kruskal's algorithm, we can use the union-find data structure. The trees in A are sets in the union-find data structure. The time complexity of the algorithm is $O(m \log m)$, or $O(m \cdot \alpha(n))$ if edges already sorted.

Algorithm 2: Implementation of Kruskal's Algorithm

```
A ← ∅
for v ∈ V do
    Make-Set(v)
Sort E in non-decreasing order of w
for (u, v) ∈ E in sorted order: do
    if Find-set(u) ≠ Find-set(v) then
        Union(u,v)
return A
```

Definition 1.9: Prim's Algorithm

We always keep A as a tree and a bunch of isolated vertex. At each iteration, we add to A the lightest edge adjacent to tree that doesn't create a cycle.

Again, we can use the cut on the tree A represents to prove correctness. To implement Prim's algorithm, we use a priority queue to store the current distances of vertice from the tree we are maintaining.

In this algorithm, there are only m decrease-key operations. Doing run time analysis on this algorithms gives a run time of $O(m + n \log n)$ using fibonacci heap.

Examples showing the execution of Kruskal's and Prim's can be found either on canvas or online.

Algorithm 3: Implementation of Prim's Algorithm

$A \leftarrow \emptyset$
for $v \in V$ **do**
  key(v) $\leftarrow \infty$
  parent(v) $\leftarrow$ NIL.
key(s) $\leftarrow 0$
Q $\leftarrow$ empty priority queue
for $v \in V$ **do**
  Q.insert(v, key(v))
while $Q \neq \emptyset$ **do**
  $u \leftarrow$ Q.extract-min
  $A \leftarrow A \cup \{u, parent(u))\}$
  for $v \in Q$ *with* $w(u, v) < key(v)$ **do**
    Q.decrease-key$(v, w(u, v))$
    parent(v) $\leftarrow u$
return A