Algorithms

MIDTERM EXAM: Thursday March 9

Lectures 1 to 10

Office Hours today: 1:30-2:30

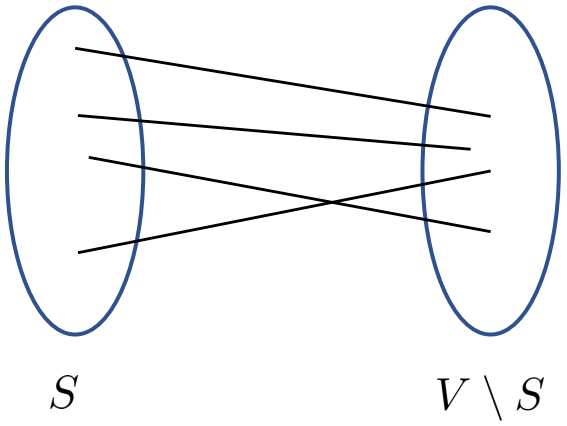
Minimum Spanning Tree (MST)

Connected

Let $G = (V, E)$ be an input graph.

For any subset $S \subset V$, consider the two groups of vertices S and $V \setminus S$.

Such a partition of V is called a ***cut*** and denoted $(S, V \setminus S)$.



The edges going *across* the cut are called ***cut edges***.

Since the graph is connected, there are cut edges.

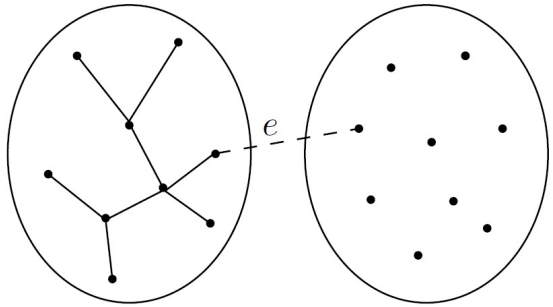
Claim. The lightest cut edge belongs to the MST. ***Cut property.***

Greedy Algorithms for MST

Prim's Algorithm.

Initial tree = any single vertex.

Repeatedly add to the tree the lightest edge connecting a tree vertex to a non-tree vertex until all vertices belong to the tree.

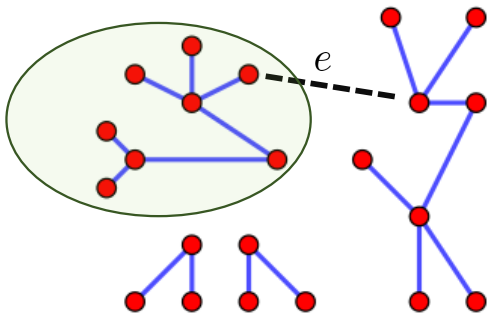


Kruskal's Algorithm.

At any time, we have a *forest* i.e. a collection of trees.

Initially, each vertex is a tree by itself.

We repeatedly add the lightest edge that joins two of the trees until we have a single tree.



The correctness of both algorithms follows from the cut property.

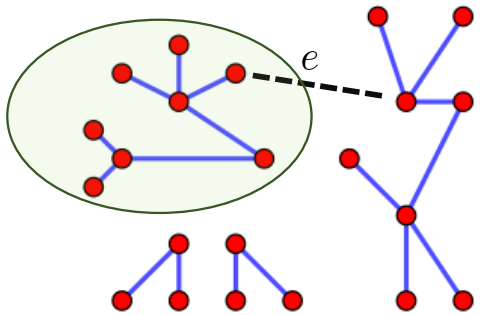
Kruskal's Algorithm

Kruskal's Algorithm.

At any time, we have a *forest* i.e. a collection of trees.

Initially, each vertex is a tree by itself.

We repeatedly add the lightest edge that joins two of the trees until we have a single tree.



Implementation?

Pseudocode.

```
Sort the edges in non-decreasing order of their weights.
Let e_1, ··· , e_m be the sorted order of edges.
for i = 1 to m:
    if the end-points of e_i are in different trees:
        add e_i to the MST
```

How do we implement this?

Kruskal's Algorithm

This ***disjoint set*** data structure allows us to store a collection of sets and supports the following operations.

$makeset(x)$	*create a new set* $\{x\}$
$union(x, y)$	*combine the set containing* x *and the set containing* y *into one set*
$find(x)$	*return a unique identifier for the set containing* x

How do we use this data structure for Kruskal's algorithm?

We think of each tree in the forest as a set (of vertices) in our collection.

Initially, we create a singleton for each vertex using $makeset$.

We process the edges in non-decreasing the order of their weights.

We add an edge (u, v) to the MST iff $find(u) \neq find(v)$.

If an edge (u, v) is added, we execute $union(u, v)$.

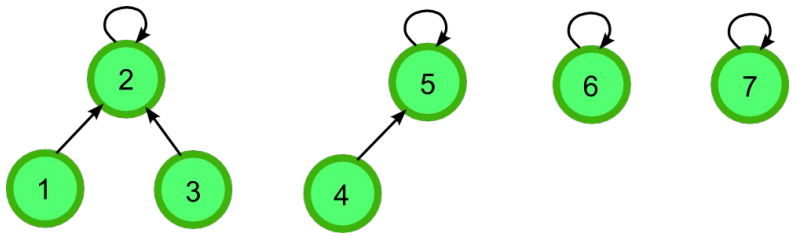
Disjoint Set Data Structure

How do we implement the data structure?

We represent each set in our collection using a tree.

This tree is internal to the data structure and has nothing to do with the MST we're building.

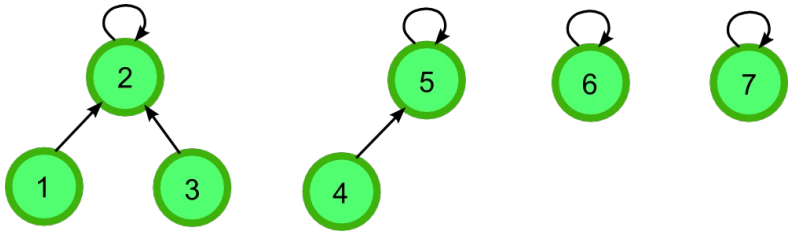
The unique identifier for a set is the root of the tree representing it.



Every node has a pointer to its parent (***not*** to the chidren).

If the node is the root of a tree, its parent pointer points to itself.

Disjoint Set Data Structure



Each node also has a ***rank***.

$makeset(x)$: Creates a single node containing x.

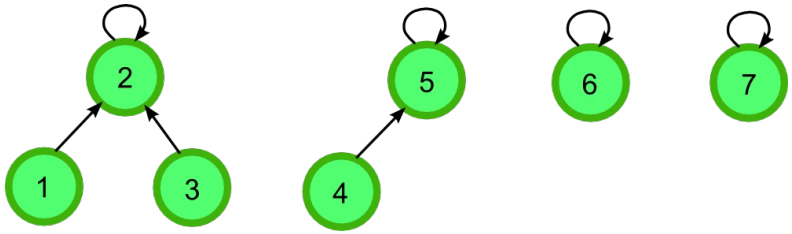
Its parent pointer points to itself.

Its rank is 0.

$find(x)$: Starting at the node containing x, we follow the parent pointer until we reach a root node.

We return a pointer to this node.

Disjoint Set Data Structure



Each node also has a ***rank***.

$union(x, y)$: Let $r_1 = find(x)$ and $r_2 = find(y)$.

If $r_1 \neq r_2$:

If $rank(r_1) \leq rank(r_2)$:

Set $parent(r_1) = r_2$.

Else:

Set $parent(r_2) = r_1$.

If $rank(r_1) == rank(r_2)$: increment $rank(r_2)$

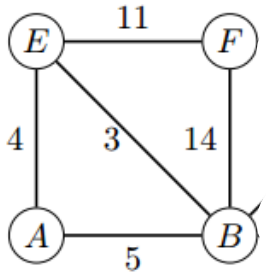
If the ranks are equal, we increment the rank of the parent after merging.

This procedure ensures that the rank of a node is strictly smaller than its parent.

Disjoint Set Data Structure

Each node also has a ***rank***.

$union(x, y)$: Let $r_1 = find(x)$ and $r_2 = find(y)$.



If $r_1 \neq r_2$:

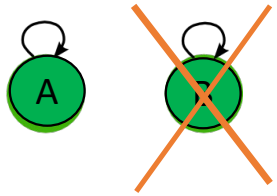
If $rank(r_1) \leq rank(r_2)$:

Set $parent(r_1) = r_2$.

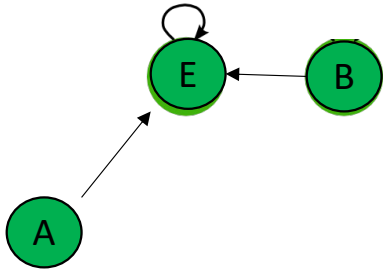
Else:

Set $parent(r_2) = r_1$.

If $rank(r_1) == rank(r_2)$: increment $rank(r_2)$



------ r_i=0



r_E=1

Complexity?

And so on ….

Disjoint Set Data Structure

Claim. If the root of tree has rank r, the tree contains $\geq 2^r$ nodes.

Proof. By induction. Base case: $r = 0$. The tree has 1 node. ✓

Inductive Hypothesis: the statement is true for $r = k$.

We need to show that the statement is true for $r = k + 1$.

When does the rank of a node increase from k to $k + 1$?

When we combine two trees whose roots have rank are k.

By our inductive hypothesis, both trees have $\geq 2^k$ nodes.

Thus, after merging, the tree has 2^{k+1} nodes. ■

If n is the total number of elements in the data structure, what is the maximum rank any node can have? $\lfloor \log_2 n \rfloor$

Disjoint Set Data Structure

$makeset(x)$	*create a new set* $\{x\}$
$union(x, y)$	*combine the set containing* x *and the set containing* y *into one set*
$find(x)$	*return a unique identifier for the set containing* x

How much time does $makeset(\cdot)$ take? $O(1)$

How much time does $find(\cdot)$ take?

$O(\log n)$ since rank increases monotonically on the path from a node to the root of its tree, and the maximum rank is $\leq \log_2 n$.

How much time does $union(\cdot, \cdot)$ take?

$O(\log n)$ since, apart from two find operations, it takes $O(1)$ time.

Let $r_1 = find(x)$ and $r_2 = find(y)$.
If $r_1 \neq r_2$:
  If $rank(r_1) \leq rank(r_2)$:
    Set $parent(r_1) = r_2$.
  Else:
    Set $parent(r_2) = r_1$.
  If $rank(r_1) == rank(r_2)$: increment $rank(r_2)$

What is the running time of Kruskal's algorithm using this data structure?

$O(m \log m)$ time for sorting the edges by weight.

```
Sort the edges in non-decreasing order of their weights.
Let e_1, ..., e_m be the sorted order of edges.
for i = 1 to m:
    if the end-points of e_i are in different trees:
        add e_i to the MST
```

How many of each data structure operation do we need?

O(*n*) O(*2m log n*) O(*n-1*)
n makesets, $2m$ finds, $n-1$ unions : **O(*m log m + n+ 2m log n + n-1*)=O(*n + m log m*)**

Note: $\log m \leq 2 \log n$ since $m \leq n^2$. **So final complexity is: O(*n + m log n*)** (same as Prim)

Exercise

Minimum Spanning Tree (MST)

Fact. If all the edge weights are distinct, then there is a unique MST.

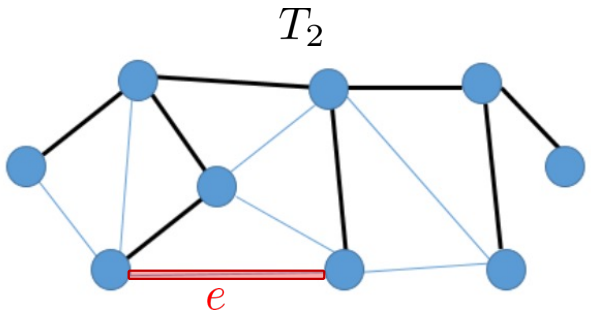
Proof. Suppose that each edge of a graph G has distinct weight but there are two distinct MST's T_1 and T_2.

Consider the *lightest* edge e that is present in one MST but not the other.

W.L.O.G. assume that e is in T_1 but not T_2.

If we add e to T_2, we will get a cycle C.

Can e be the *heaviest* edge in this cycle?



No, since that would mean that all edges in the cycle were present in T_1.

Then, replacing the heaviest edge in the cycle by e yields a lighter MST!

Minimum Spanning Tree (MST)

There are faster algorithms for Computing the MST.

There is a randomized algorithm that takes $O(m+n)$ time *in expectation*.

https://en.wikipedia.org/wiki/Expected_linear_time_MST_algorithm

There is a deterministic algorithm that takes $O(m\,\alpha(m,n))$ time.

https://www.cs.princeton.edu/courses/archive/fall05/cos528/handouts/A%20Minimum%20Spanning.pdf

α is the inverse Ackermann function

Huffman Encoding

Each quantized value is encoded in binary.

However, in general some quantized values are more frequent than other.

Consider a simple example with four quantized values represented by the symbols A, B, C and D whose frequencies are:

$$A: 75 \text{ million}, \quad B: 2 \text{ million}, \quad C: 22 \text{ million} \quad \text{and} \quad D: 31 \text{ million}.$$

One option would be to encode each value using two bits.

How many bits does this require? *260 million*

Can we do better by encoding the values with codes of different lengths?

Yes! Here is a better encoding. $A: 0$, $B: 100$, $C: 101$ and $D: 11$.

How many bits does this require? *209 million*

Huffman Encoding

We have four quantized values A, B, C and D with frequencies:

$$A : 75 \text{ million}, \ B : 2 \text{ million}, \ C : 22 \text{ million} \ \text{ and } D : 31 \text{ million}.$$

The encoding $A : 0$, $B : 100$, $C : 101$ and $D : 11$ requires 209 million bits.

Can we do even better? How about the following encoding?

$$A : 0, \ B : 01, \ C : 11 \text{ and } D : 1.$$

This encoding is not good because it is not uniquely decipherable.

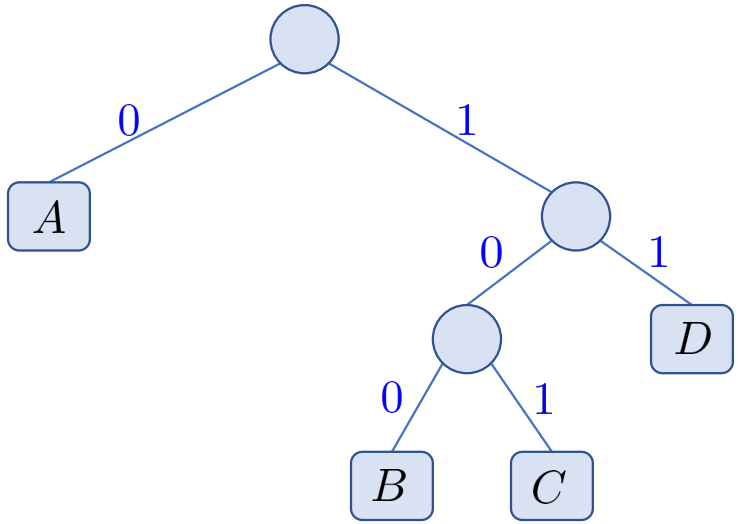
For example 11 can either be C or DD.

We will use a ***prefix-free code*** to avoid this. *No code is a prefix of another.*

Huffman Encoding

The encoding $A : 0$, $B : 100$, $C : 101$ and $D : 11$ is prefix free.

Any encoding that is prefix free can be represented as a binary tree whose leaves are the symbols.



The length of the encoding of any symbol is equal to its depth in the tree.

Let $f(A), f(B), f(C)$ and $f(D)$ denote the frequencies of A, B, C and D.

Then the length of the encoding is: $1 \cdot f(A) + 3 \cdot f(B) + 3 \cdot f(C) + 2 \cdot f(D)$.

Huffman Encoding

In general, there are n symbols whose frequences are given in an array $f[1..n]$.

We want a tree representing a prefix free codes that minimizes

$$\sum_{i=1}^{n} depth(i) \cdot f(i).$$