# lecture 8

minimal spanning trees

- definition of MST
- cut property
- Prim's algorithm
- Kruskal's algorithm

## Resources

Sedgewick Algorithms 2
week 2

https://class.coursera.org/algs4partII-002/lecture/12

https://class.coursera.org/algs4partII-002/lecture/13

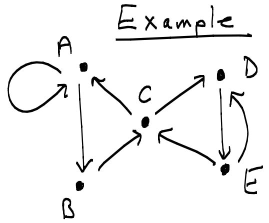
Roughgarden Algorithms 2
week 1

https://class.coursera.org/algo2-2012-001/lecture/25

[Recall lecture 6 - a cycle in a graph is a sequence of vertices $v_1 — v_k$ such that $(v_i, v_{i+1})$ is an edge and $v_1 = v_k$. ]

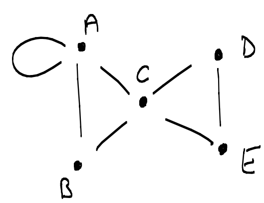
Simple cycle (directed or undirected): a cycle with no repeating vertices (except $v_1, v_k$) and no repeating edges.

Example




| cycles | simple ? |
|---|---|
| AA | yes |
| ABCA | yes |
| ABCAA | no |
| ABCDECA | no |
| DED | yes |

Example (undirected graph)

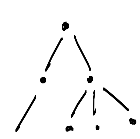



| cycles | simple ? |
|---|---|
| AA | yes |
| ABCA | yes |
| ABA | no |
| ABCAA | no |
| ABCDECA | no |

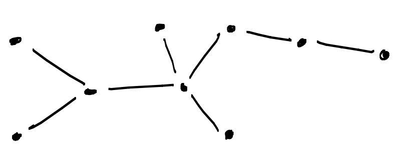
because the edge repeats

A tree is a connected undirected graph that has no (simple) cycles.

rooted tree



non-rooted tree



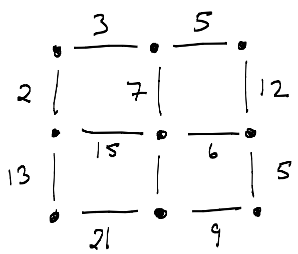
Note there is always exactly one (simple) path between each pair of vertices.

A spanning tree of a connected undirected graph is a subgraph that

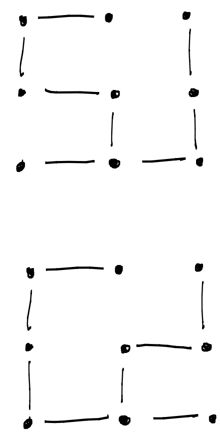
- is a tree
- contains all vertices of the graph.

[Exercise: if the graph has $|V|$ vertices then any spanning tree has $|V|-1$ edges.]

(Weighted) graph




examples of spanning trees



Q: of all spanning trees, which one minimizes the total edge cost?

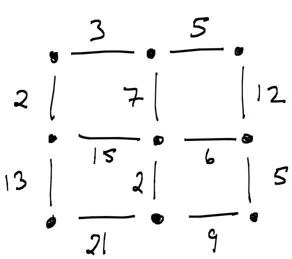
## Classical application

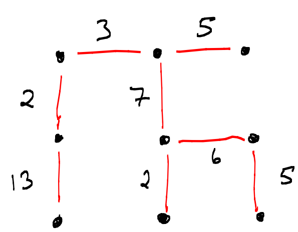
Suppose you have a set of $n$ physical locations (vertices in graph) that you would like to connect with "wires". For each pair of locations, there would be a cost of connecting them with a wire (thus, $\frac{n(n-1)}{2}$ edges). e.g. digging a tunnel in the ground, ....

How can we choose the edges that path connect all the vertices and that minimize the total cost?

## Minimal Spanning Tree (MST)

An MST is a spanning tree who total edge weights are as small as possible.

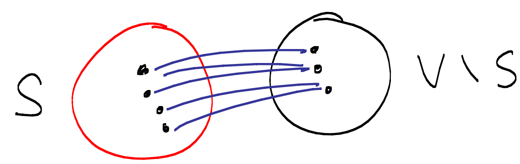



[Recall from Dijkstra's algorithm .... ]

A cut in a graph is a partition of the vertices into $S$ and $V \setminus S$.

The crossing edges of a cut are the set of vertices $(u,v)$ such that $u \in S$ and $v \in V \setminus S$.

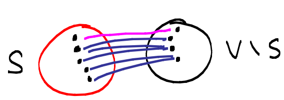


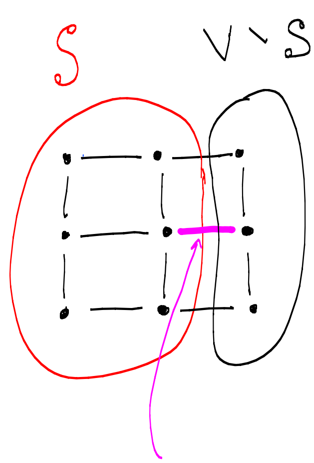

## Claim: "Cut property" of MST

Let $S, V \setminus S$ be a cut of an undirected graph $G = (V, E)$.

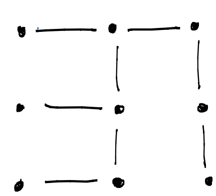
Let $e \in E$ be the crossing edge with strictly smallest weight (if such a unique edge exists).

Then $e$ belongs to every MST.




crossing edge $e$
with smallest weight



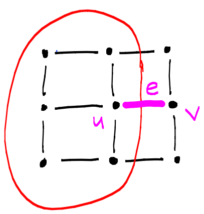
Spanning tree $T$ that does not contain $e$

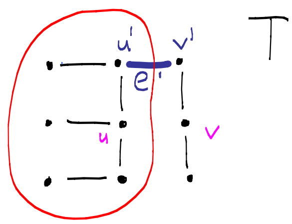
## Proof ("exchange argument"):

Consider a spanning tree $T$ that doesn't contain $e = (u,v)$.

Let $e' = (u',v')$ be a crossing edge in $T$ that is on a path from $u$ to $v$ in $T$.

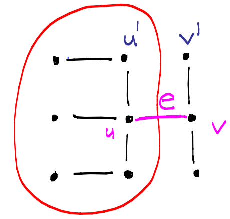
By definition of $e$, $\text{cost}(e) < \text{cost}(e')$.




Next, adding $(u,v)$ to the spanning tree $T$ would create a cycle. Why?

Then removing $(u',v')$ would break that cycle (why?), and create a new spanning tree $T^*$.




$$c(T^*) = c(T) + c(e) - c(e')$$
$$< c(T)$$

Thus, any spanning tree $T$ that doesn't contain $e$ cannot be a MST.

---

**Claim:** If the edge weights (costs) of a graph are all distinct

e.g. $c(e_1) < c(e_2) < \dots < c(e_m)$

then there exists a unique MST.

**Proof:** Suppose we had two MST's, each with $n-1$ edges.

$$T_1 = \{e_{i_1}\ e_{i_2}\ e_{i_3} \cdots e_{i_{n-1}}\}$$

$$T_2 = \{e_{j_1}\ e_{j_2}\ e_{j_3} \cdots e_{j_{n-1}}\}$$

[Exercise: finish the proof]

---

Given a connected undirected graph, how can we compute a MST?

## General Approach:

```
T = empty set
repeat {
    · define a cut
    · choose a minimal crossing edge
    · add that edge to T
} until T is a spanning tree
```

---

# lecture 8

minimal spanning trees

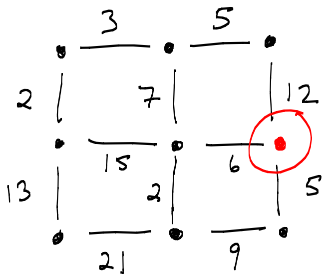
- definition of MST
- cut property
- Prim's algorithm
- Kruskal's algorithm

---

## Prim's Algorithm for finding a MST

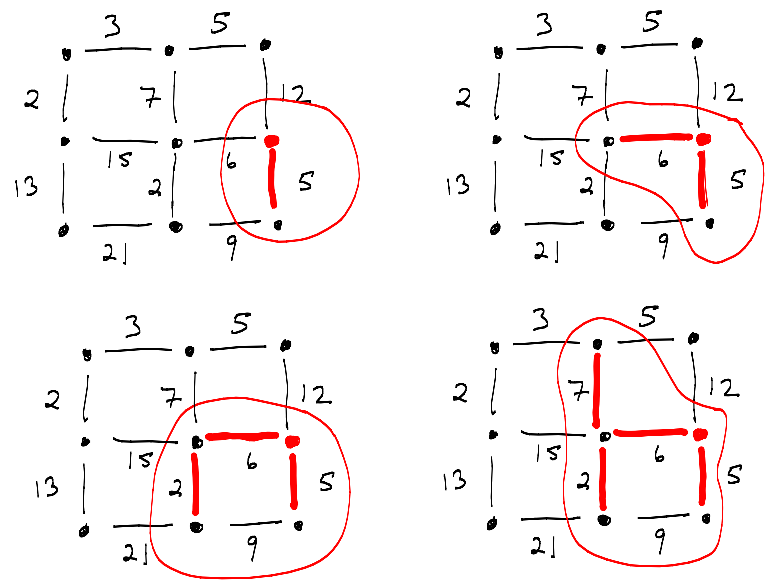
```
choose any vertex s
S = {s}        // set of vertices
T = { }        // set of edges
while |T| < |V| - 1 {
    find minimal cost crossing edge e = (u,v)
        where u ∈ S and v ∈ V \ S
    add v to S
    add (u,v) to T
}
return T
```

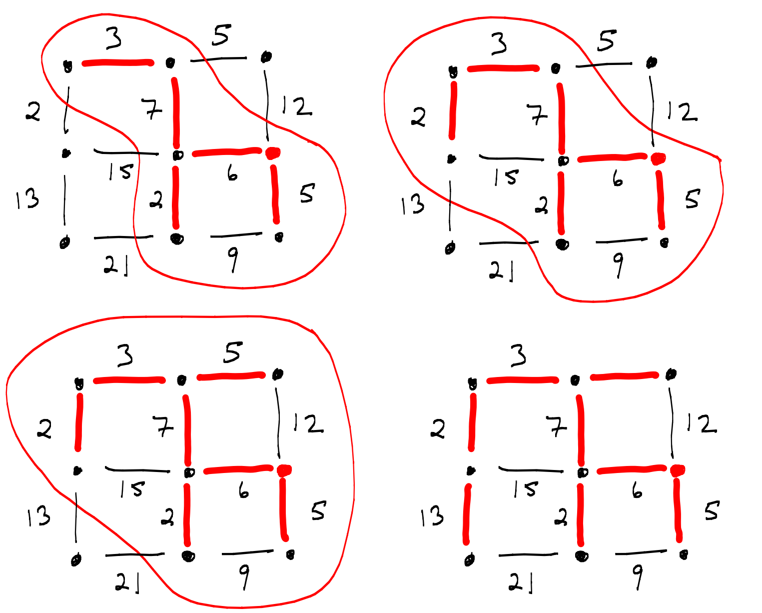
Example



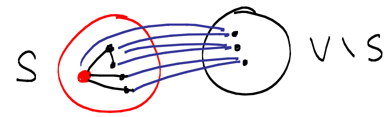

arbitrarily pick this vertex

Now what ?





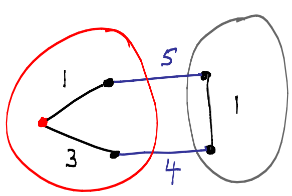
Prim's MST algorithm is almost identical to Dijkstra's shortest path algorithm.




Both grow a tree from a starting vertex s.

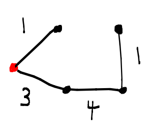
Prim chooses the crossing edge that has minimum cost.

Dijkstra chooses the crossing edge that minimizes shortest path from s.



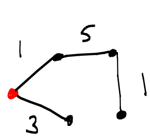

Prim chooses the 4 edge.

$\Rightarrow$




Dijkstra chooses the 5 edge.

$\Rightarrow$




Claim: If the edges weights are distinct, then Prim's algorithm computes the MST.

Proof: At the end of each iteration, we have a cut S and V \ S.

Prim chooses the minimal crossing edge, which we know must belong to any MST. Thus all the edges chosen by Prim belong to the MST (and the MST is unique - see 8 slides ago).

But does Prim find all of these edges? Could Prim terminate before finding all these edges?

[ Exercise: Show Prim indeed finds all edges in the MST. ]

**Prim's algorithm using a priority queue (vertex version):**

initialize empty priority queue **pq**
**T = { }**
for all v in V
    **pq.add(v, infinity )**     **// start by adding all vertices to pq**
    parent[ v ] = null

pick an arbitrary vertex s in V as root of spanning tree
**pq.changePriority( s, 0 )**

while **pq** is not empty
    u = **pq.getMinVertex()**
    **distFromS = pq.removeMin()**
    add u to S
    **if parent[u] != null**
        **add (parent[u], u ) to T**     **// T will become the MST**
    for each edge (u, v)   // v in adjacency list of u
        if v not in S and cost(u,v) < **pq.getPriority( v ) {**
            **pq.changePriority**( v , cost(u, v) )
            parent[ v ] = u }

main difference from Dijkstra's shortest path algorithm

**Prim's algorithm (edge version -- relevant to Assignment 2)**

// Keep a priority queue of edges with at least one endpoint in S.
// Each node of the pq stores an edge and its cost (priority).

S = {s}
**T = { }**
for each edge e = (s, v)
    **pq.add( e, cost(e) )**

while |S| < |V|-1
    **(u,v) = pq.getMinEdge()**
    **pq.removeMin()**     // we don't do anything with edge cost
    if both u and v are in S, do nothing
    else     // one vertex is in S, so let u be in S, v in V \ S
        **add edge (u, v) to T**
        add v to S
        for each edge (v,w)     // w in v's adjacency list
            pq.add( (v, w) , cost(v, w) )

different from the edge version of Dijkstra

lecture 8

minimal spanning trees
( undirected graphs)

- cut property
- Prim's algorithm
- Kruskal's algorithm

Prim expands a vertex set S and an edge set T.

It choose from crossing edges of S and V \ S, and grows a rooted tree T which consists of vertices from S.

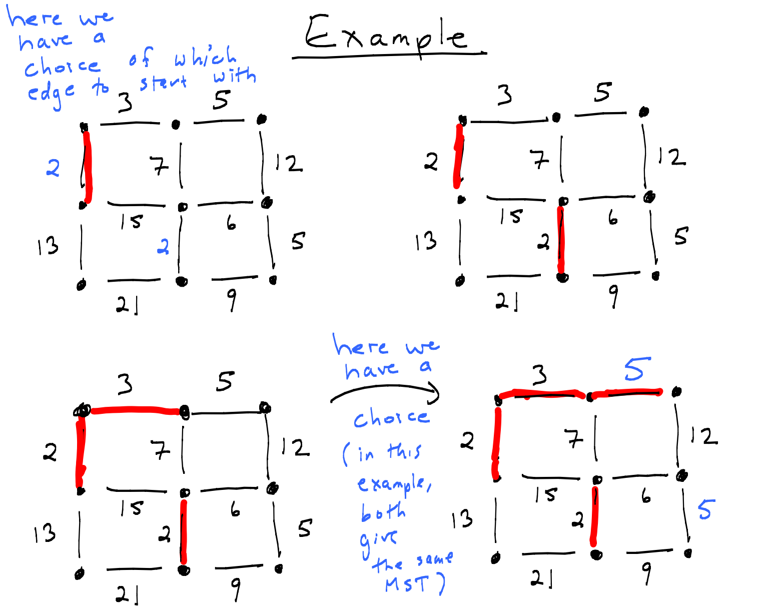
Kruskal grows a set of edges T. It terminates when T is the MST. (but along the way it might be a forest of trees)

**Kruskal's Algorithm (1956)**

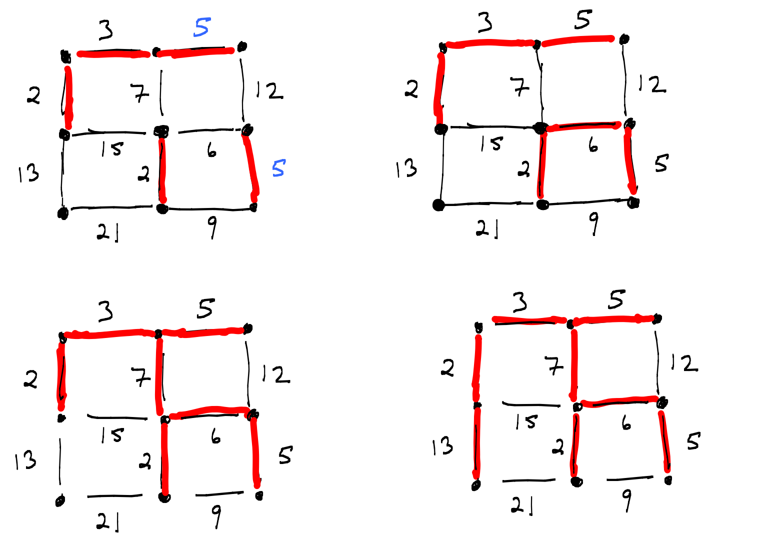
// Recall the MST has |V|-1 edges.

T = { }   // empty set
Sort the edges by cost $\{e_1 e_2 \dots e_n\}$
k = 0
while |T| < |V| - 1 {
    k = k + 1
    if $T \cup \{e_k\}$ doesn't create a cycle
        add $e_k$ to T
}

## Example




## Example



**Q:** Why does Kruskal's algorithm give a <u>spanning</u> tree?

**A:** We assume the original graph is connected. Kruskal doesn't create a cycle, so the only problem could occur if Kruskal terminates without T being connected. But that can't happen since there would exist crossing edges for T and $V \setminus T$ and Kruskal would have added one of them.

<u>**Q:**</u> Why does Kruskal's algorithm give a <u>minimal</u> spanning tree? [modified Jan. 31]

**A:** Suppose $(u,v)$ is an edge in the spanning tree found by Kruskal.

Let S be the connected component of u before this edge was added.

Then $v \in V \setminus S$, since otherwise adding $(u,v)$ would have created a cycle and Kruskal wouldn't have added it. We need to show that $(u,v)$ is the crossing edge between S and $V \setminus S$ with minimal cost.

Any edge e that is already in T at the time $(u,v)$ is added cannot be a crossing edge and so it must have both vertices in S or both vertices in $V \setminus S$. Thus, because Kruskal chooses $(u,v)$, any crossing edges other than $(u,v)$ must have cost greater than the cost of $(u,v)$. Thus, $(u,v)$ is the minimum cost crossing edge. By the cut property, it must be long to the MST.

How to implement Kruskal's algorithm?

$O(|E| \log |E|)$

```
T = { }
Sort the edges by cost {e1, e2 .... en}
k = 0
while |T| < |V| - 1 {
    k = k + 1
    if T ∪ {ek} doesn't create a cycle
        add ek to T
}
```

how long?

## Recall union-find (disjoint sets)

$T = \{\}$
sort the edges by cost $\{e_1, e_2, \ldots, e_n\}$
$k = 0$
while $|T| < |V| - 1$ {
  $k = k + 1$
  if $T \cup \{e_k\}$ doesn't create a cycle // $e_k = (u, v)$
    add $e_k$ to $T$
}

union( u, v )

same set( u, v )

sorting

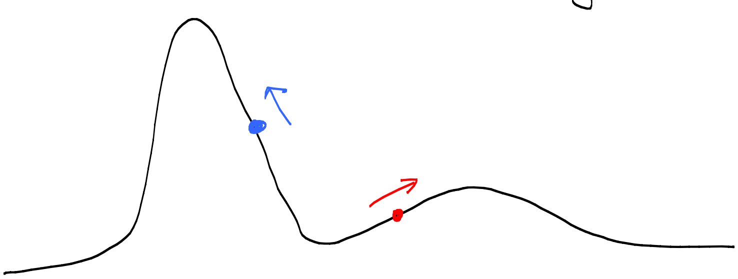
$O(|E| \log^* |V|) < O(|E| \log |E|)$

## ASIDE: the notion of Greedy Algorithms

Dijkstra, Prim, Kruskal are all examples of "greedy" algorithms. There is no mathematical definition of a "greedy algorithm" but the basic idea is that they choose a "path" to a solution that is "locally optimal" in the hope that it will lead to a solution that is "globally optimal."

We will see other examples later in the course.

Suppose you want to find the highest hilltop. A "greedy algorithm" is to walk uphill (called "hill climbing" in AI)



That will work some times but not others.

The next few graph algorithms we will see are not greedy. They allow the intermediate solution worsen in order to eventually reach the solution that is optimal.