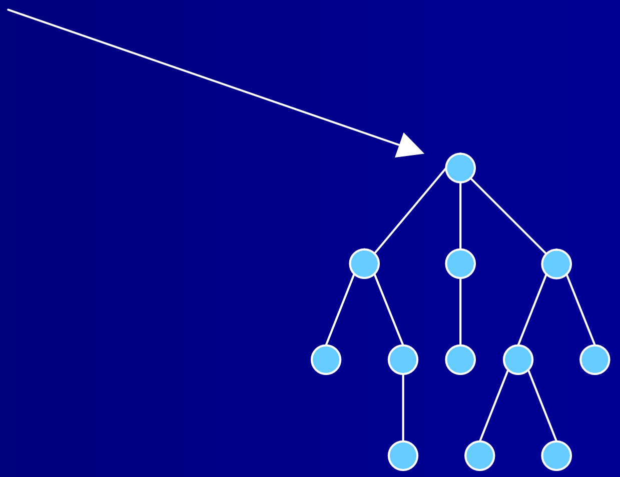
Minimal Spanning Trees

What is a minimal spanning tree (MST) and how to find one

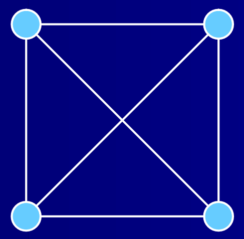
Tree

- A tree contains a root, the top node.
- Each node has:
  - One parent
  - Any number of children



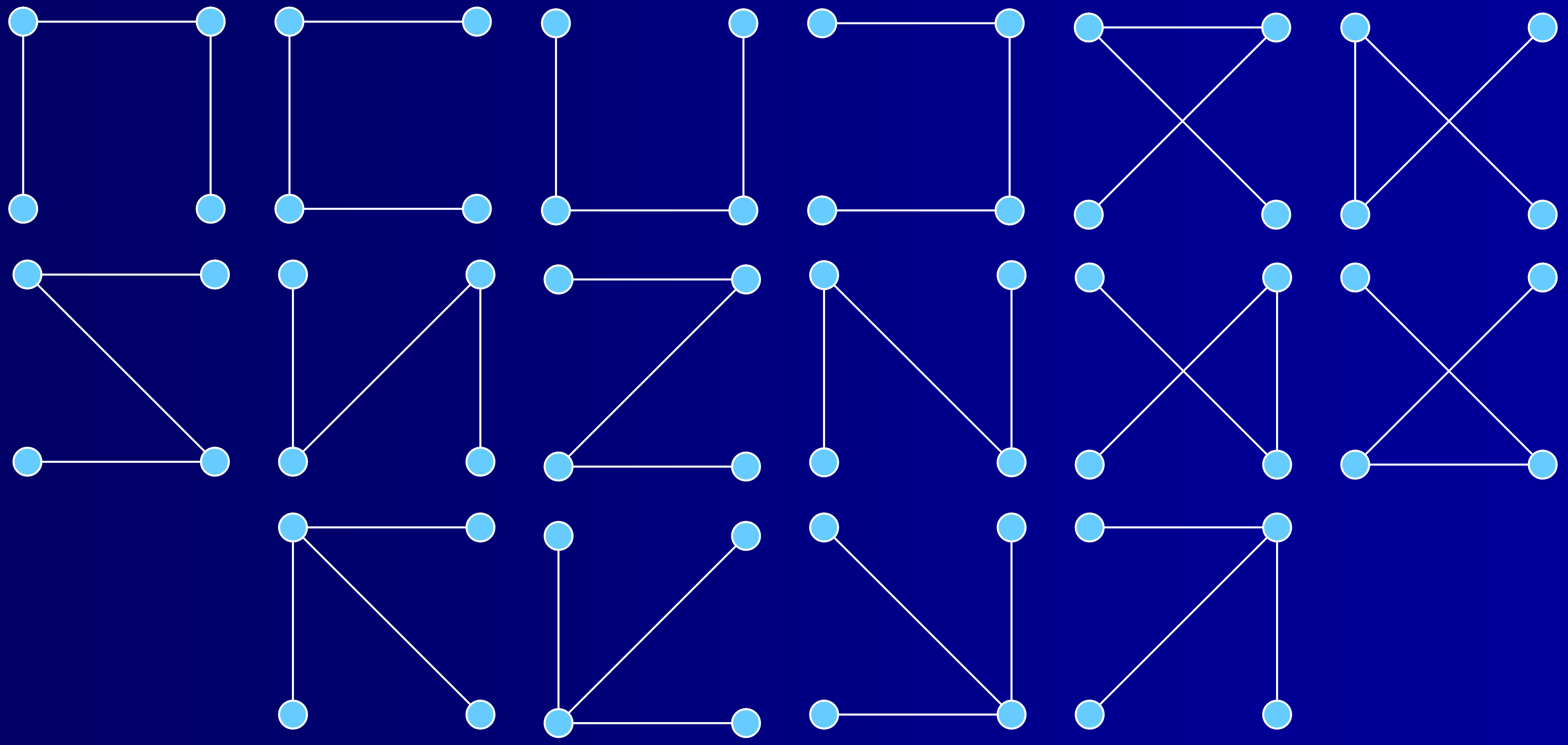
Spanning Tree

- A spanning tree of a graph is a subgraph that contains all the vertices and is a tree (connected).
- A graph may have many spanning trees, for example



has 16 spanning trees.

Spanning Tree Example



Minimal Spanning Tree

- Now suppose the edges were weighted.
- How do we find the spanning tree with the minimum sum of edges.
- This is called the minimal spanning tree.

Sample Problem

- The standard application is to a problem like phone network design.
- You want to lease phone lines to connect several offices with each other.
- The phone company charges different amounts of money to connect different pairs of cities.
- You want a set of lines that connects all your offices with a minimum total cost.
- It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

How To Find a MST

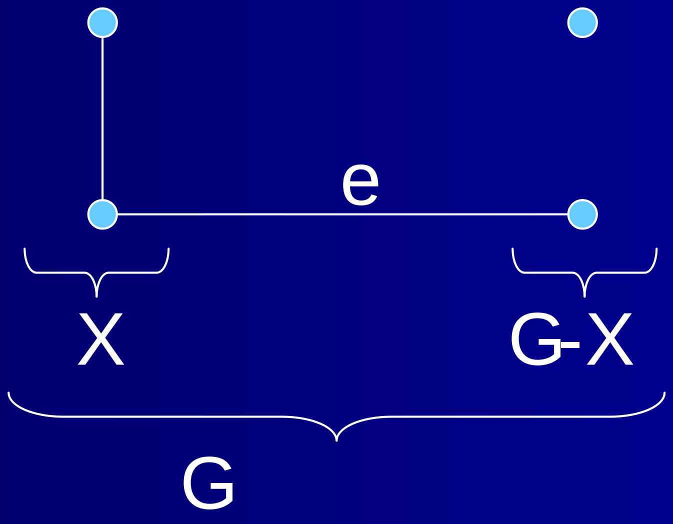
- A slow method is to list all the spanning trees and find the minimum from the list.
- But there are far too many trees (16 in our example for v = 4).
- A better idea is to find some key property of the MST that lets us be sure that some edge is part of it, and use this property to build up the MST one edge at a time.

Assumption

- For simplicity, we assume that there is a unique MST.
- You can get ideas like this to work without this assumption, but it becomes harder to state your theorems or write your algorithms precisely.

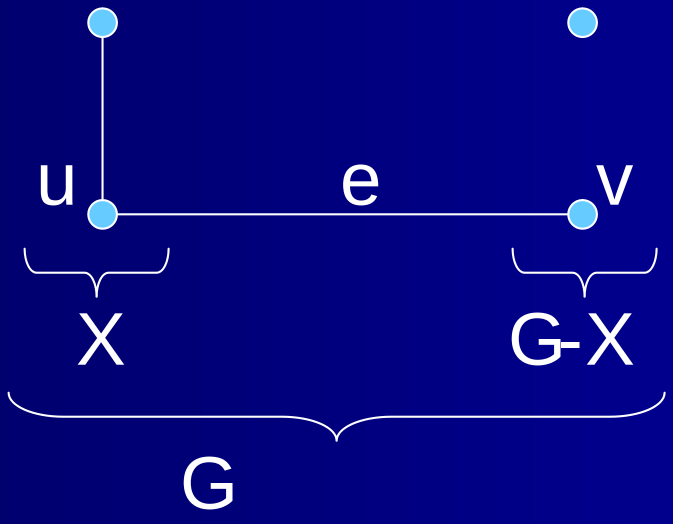
Lemma

- Let X be any subset of the vertices of G, and let edge e be the smallest edge connecting X to G - X (vertexes not in X).
- Then e is part of the MST.



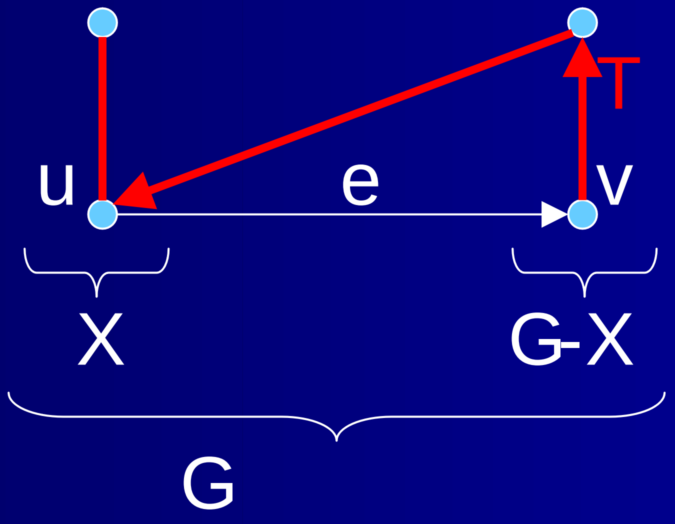
Proof of Lemma

- Suppose you have a tree T not containing e.
- Then we must prove that T is not the MST.
- Let e connect u and v, with u in X and v not in X.



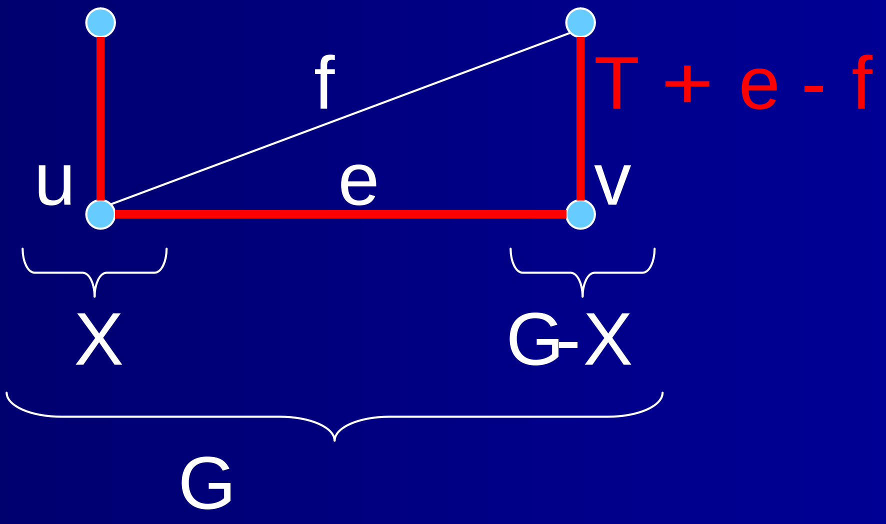
Proof of Lemma

- Then because T is a spanning tree it contains a unique path from u to v, which together with e forms a cycle in G.



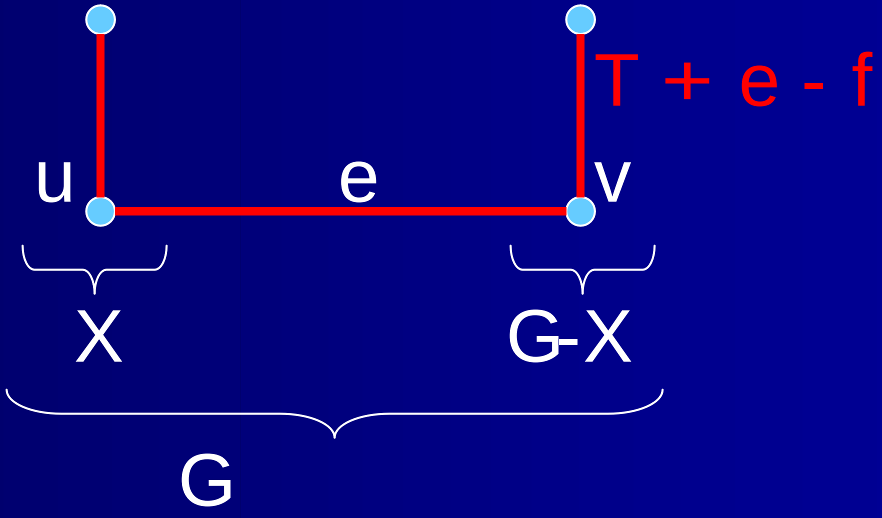
Proof of Lemma

- This path has to include another edge f connecting X to G - X.
- T + e - f is another spanning tree (same number of edges, and remains connected).



Proof of Lemma

- It has smaller weight than t since e has smaller weight than f.
- So T was not minimum, which is what we wanted to prove.



Kruskal’s Algorithm

- We'll start with Kruskal's algorithm, which is the easiest to understand and probably the best one for solving problems by hand:

```
sort the edges of G in increasing order by
length
keep a subgraph S of G, initially empty
for each edge e in sorted order
  if the endpoints of e are disconnected in S
    add e to S
return S
```

Kruskal’s Algorithm

- Note that, whenever you add an edge (u,v), it's always the smallest connecting the part of S reachable from the rest of G, so by the lemma it must be part of the MST.
- This algorithm is a greedy algorithm, because it chooses at each step the cheapest edge to add to S.
- The greedy idea works in Kruskal's algorithm because of the key property we proved.

Kruskal Analysis

- The line testing whether two endpoints are disconnected looks like it should be slow (linear time per iteration, or O(mn) total).
- The slowest part turns out to be the sorting step.
- Therefore it is important to choose a fast sorting algorithm.
- Using quicksort takes O(m log m) time, which is effectively the total run-time.

Kruskal Demonstration

- Kruskal's Algorithm

Prim’s Algorithm

- Rather than build a subgraph one edge at a time, Prim's algorithm builds a tree one vertex at a time:

```
let T be a single vertex x
while (T has fewer than n vertices)
  find the smallest edge connecting T to G-T
  add it to T
```

Prim’s Algorithm

- Since each edge added is the smallest connecting T to G - T, the lemma we proved shows that we only add edges that should be part of the MST.

Prim’s Algorithm With a Heap

- Again, it looks like the loop has a slow step in it, $O(n^2)$.
- We can speed it up.
- The idea is to use a heap to remember, for each vertex, the smallest edge connecting T with that vertex.

Prim’s Algorithm With a Heap

```
make a heap of values (vertex,edge,weight(edge))
  initially (v,-,infinity) for each vertex
let T be a single vertex x
for each edge f=(u,v)
  add (u,f,weight(f)) to heap

while (T has fewer than n vertices)
  let (v,e,weight(e)) be the edge  with the
                      smallest weight on the heap
  remove (v,e,weight(e)) from the heap
  add v and e to T
  for each edge f=(u,v)
    if u is not already in T
      find value (u,g,weight(g)) in heap
      if weight(f) < weight(g)
        replace (u,g,weight(g)) with
                (u,f,weight(f))
```

Prim Analysis

- We perform n steps in which we remove the smallest element in the heap, and at most m steps in which we reduce the weight of the smallest edge connecting T to G - T.
- For each of those steps, we might replace a value on the heap, reducing it's weight.
- You also have to find the right value on the heap, but that can be done easily enough by keeping a pointer from the vertices to the corresponding values.

Prim Analysis

- You can reduce or delete the weight of an element of the heap in O(log n) time.
- Alternately by using a more complicated data structure known as a Fibonacci heap, you can reduce the weight of an element to constant time.
- Deletion is done n times and reduction is done at most m times.
- The result is a total time bound of O((m + n) log n) or just O(m log n).

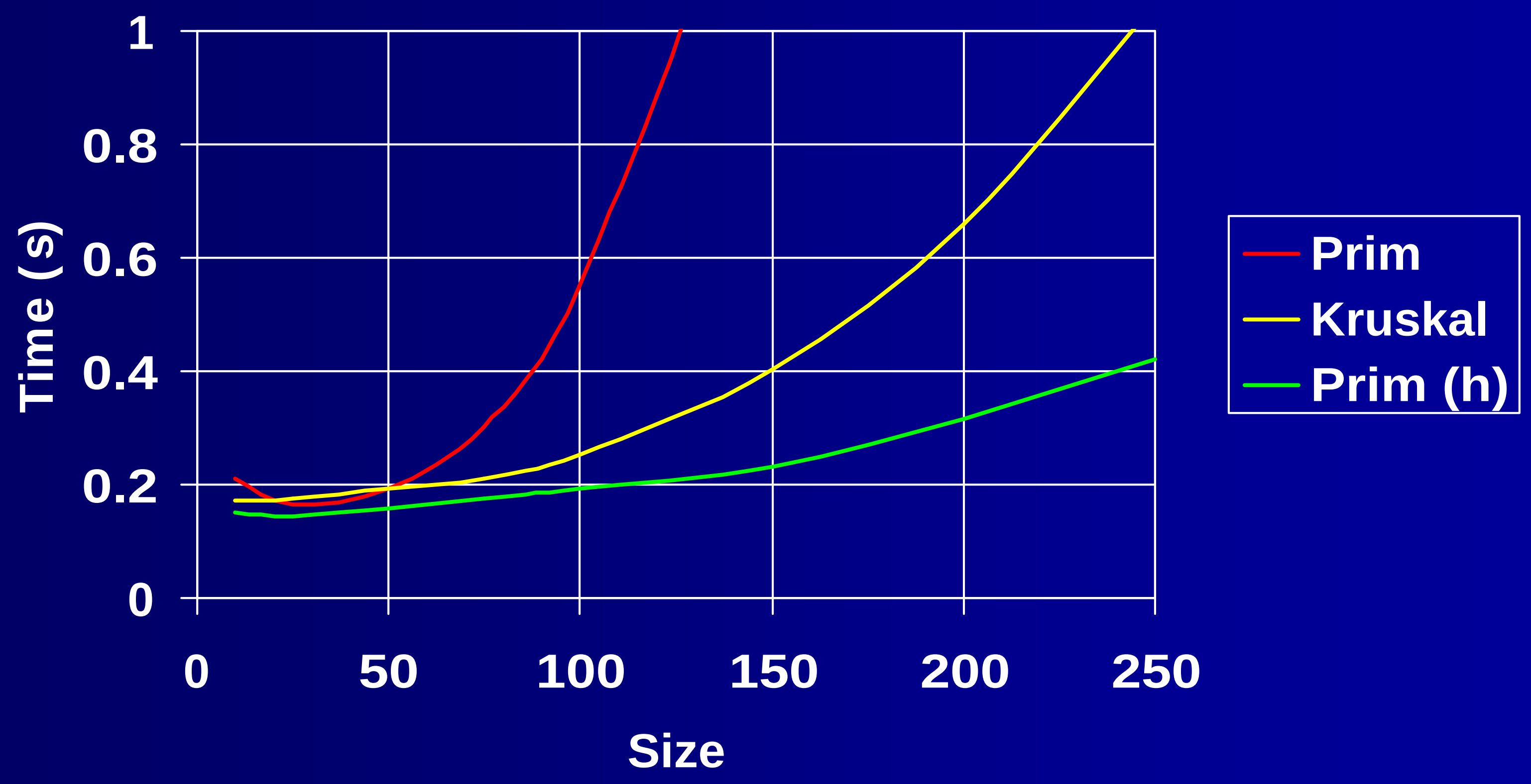
Prim Demonstration

- Prim's Algorithm

Comparison

- They both produce a MST:
  - Prim & Kruskal's Algorithms

Time Comparison



Which One To Choose?

- Prim with a heap is about twice as fast as Kruskal.
- Kruskal is easier to code.
- Try them both and choose the one you prefer.
- Learn how to do both, since sometimes one is much better than the other:
  - Although Kruskal is slower, you may find it much easier to use.

Trail Maintenance (IOI 2003)

- Cows want to maintain certain trails between fields.
- The total length of trails maintained must be minimized.
- They start off with no trails , and after each week they discover a new path.
- Given the trails they discover each week, you need to determine the total distance of the trails after each week.

Trail Maintenance Solution 1

- Recompute the MST after each week, considering all trails ever seen.
- Use Prim’s or Kruskal’s algorithm to compute the MST.
- O(m^2 log m), where m is the total number of paths.
- Will get about 50%.

Trail Maintenance Solution 2

- The same as the previous solution, considering only the best trail between any two fields.
- O(m^2 log n), where n is the number of fields.
- Will get about 60%.

Trail Maintenance Solution 3

- The same as the previous solution, considering only the trails in the previous MST and the new trail.
- O(nm log n).
- Will receive 100%.

Trail Maintenance Solution 4

- Use a true incremental MST.
- Each week, determine the path between the endpoints of the new trail and find the maximum length in that trail.
- If the length of this maximum trail is greater than the new trail, delete that trail and add the new one.
- O (nm).
- Will receive 100%.