CS 603: Minimum Spanning Trees: Prim and Kruskal's algorithms

Ellen Veomett

University of San Francisco

Outline

1. Minimum Spanning Trees
2. Kruskal's Algorithm
3. Prim's Algorithm

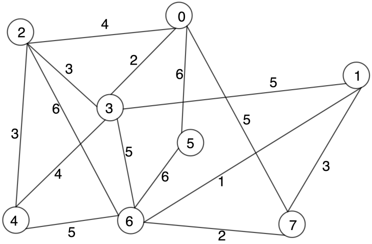
Minimum Spanning Trees

Minimum Spanning Trees

- Suppose we have several locations that we need to connect with something
  - Ethernet Cable
  - Electricity
- There is a cost in connecting each location to each other.

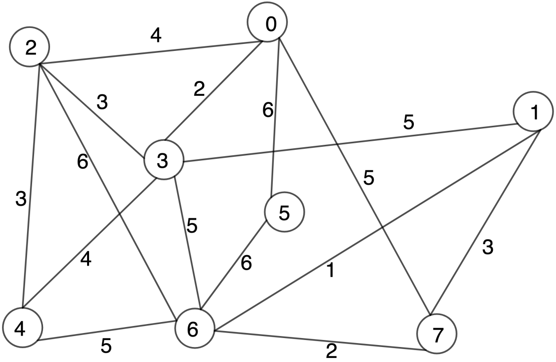
Minimum Spanning Trees

- Suppose we have several locations that we need to connect with something
  - Ethernet Cable
  - Electricity
- There is a cost in connecting each location to each other.
- We want to find the minimum cost!

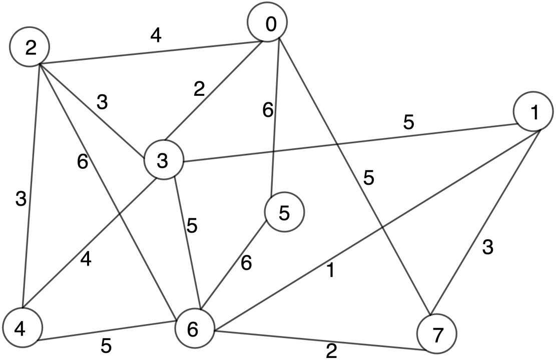


Claim: If the cost is non-negative, the min cost can always be achieved on a tree. (Why?)

Two Greedy Algorithms to find the Min Spanning Tree

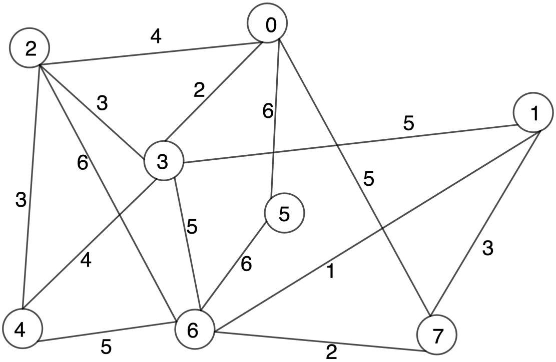


Two Greedy Algorithms to find the Min Spanning Tree



- Kruskal's Algorithm: add cheapest edges that don't create a cycle

Two Greedy Algorithms to find the Min Spanning Tree



- Kruskal's Algorithm: add cheapest edges that don't create a cycle
- Prim's Algorithm: add cheapest edges from current sub-tree to new node

Outline

1. Minimum Spanning Trees
2. Kruskal's Algorithm
3. Prim's Algorithm

Kruskal's Algorithm: Basics

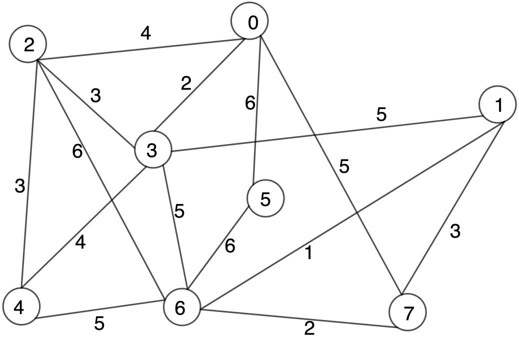
Kruskal's Algorithm: Basics

- Until we have $n-1$ total edges (a spanning tree):

Kruskal's Algorithm: Basics

- Until we have $n - 1$ total edges (a spanning tree):
  - Add the cheapest edge *that does not create a cycle*

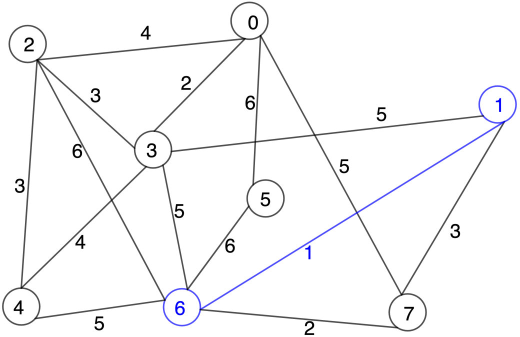
Example



Kruskal's Algorithm: Basics

- Until we have $n - 1$ total edges (a spanning tree):
  - Add the cheapest edge *that does not create a cycle*

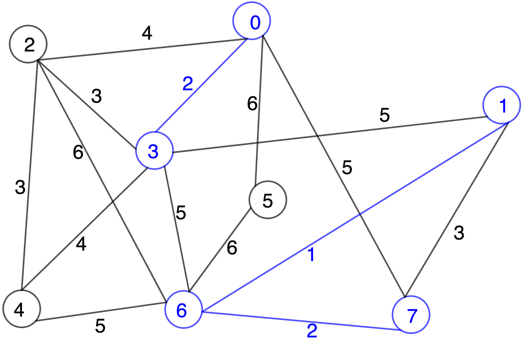
Example



Kruskal's Algorithm: Basics

- Until we have $n - 1$ total edges (a spanning tree):
  - Add the cheapest edge *that does not create a cycle*

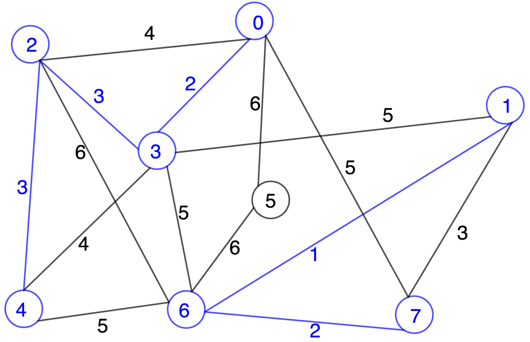
Example



Kruskal's Algorithm: Basics

- Until we have $n - 1$ total edges (a spanning tree):
  - Add the cheapest edge *that does not create a cycle*

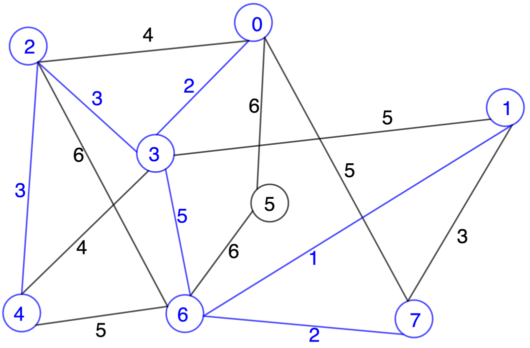
Example



Kruskal's Algorithm: Basics

- Until we have $n - 1$ total edges (a spanning tree):
  - Add the cheapest edge *that does not create a cycle*

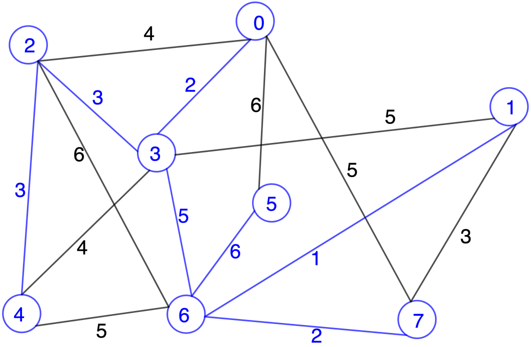
Example



Kruskal's Algorithm: Basics

- Until we have $n - 1$ total edges (a spanning tree):
  - Add the cheapest edge *that does not create a cycle*

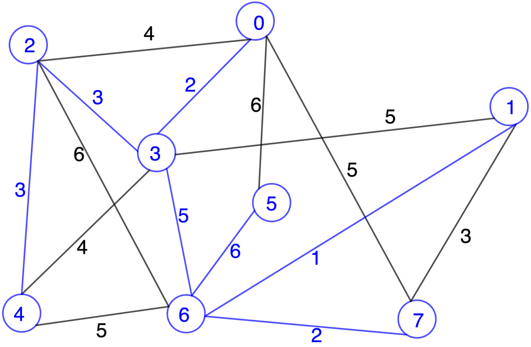
Example



Kruskal's Algorithm: Basics

- Until we have $n-1$ total edges (a spanning tree):
  - Add the cheapest edge *that does not create a cycle*

Example



Thus, the weight of the minimum spanning tree is:

$$1+2+2+3+3+5+6=22$$

Kruskal's Algorithm: Implementation

Kruskal's Algorithm: Implementation

- To make sure we don't create a cycle:

Kruskal's Algorithm: Implementation

- To make sure we don't create a cycle:
  - Use Disjoint Sets

Kruskal's Algorithm: Implementation

- To make sure we don't create a cycle:
  - Use Disjoint Sets
  - Initially each vertex is in its own set

Kruskal's Algorithm: Implementation

- To make sure we don't create a cycle:
  - Use Disjoint Sets
  - Initially each vertex is in its own set
  - When an edge (u, v) is considered, if u and v are in different sets, that edge can be added and we take the union of those sets.

Kruskal's Algorithm: Implementation

- To make sure we don't create a cycle:
  - Use Disjoint Sets
  - Initially each vertex is in its own set
  - When an edge (u, v) is considered, if u and v are in different sets, that edge can be added and we take the union of those sets.
  - If u and v are in the same set, then we can't add that edge (b/c it would create a cycle)

Kruskal's Algorithm: Implementation

- To make sure we don't create a cycle:
  - Use Disjoint Sets
  - Initially each vertex is in its own set
  - When an edge (u, v) is considered, if u and v are in different sets, that edge can be added and we take the union of those sets.
  - If u and v are in the same set, then we can't add that edge (b/c it would create a cycle)
- To find the min edges available:

Kruskal's Algorithm: Implementation

- To make sure we don't create a cycle:
  - Use Disjoint Sets
  - Initially each vertex is in its own set
  - When an edge (u, v) is considered, if u and v are in different sets, that edge can be added and we take the union of those sets.
  - If u and v are in the same set, then we can't add that edge (b/c it would create a cycle)
- To find the min edges available:
  - Use a Min Heap

Kruskal's Algorithm: Computational Complexity

Kruskal's Algorithm: Computational Complexity

- Create Disjoint Sets

Kruskal's Algorithm: Computational Complexity

- Create Disjoint Sets
    - $O(n)$
    - 1 time

Kruskal's Algorithm: Computational Complexity

- Create Disjoint Sets
    - $O(n)$
    - 1 time
- Check if we can add an edge (two finds and maybe one union)

Kruskal's Algorithm: Computational Complexity

- Create Disjoint Sets
  - $O(n)$
  - 1 time
- Check if we can add an edge (two finds and maybe one union)
  - $O^*(1)$
  - $O(m)$ times

Kruskal's Algorithm: Computational Complexity

- Create Disjoint Sets
  - $O(n)$
  - 1 time
- Check if we can add an edge (two finds and maybe one union)
  - $O^*(1)$
  - $O(m)$ times
- Create min heap

Kruskal's Algorithm: Computational Complexity

- Create Disjoint Sets
  - $O(n)$
  - 1 time
- Check if we can add an edge (two finds and maybe one union)
  - $O^*(1)$
  - $O(m)$ times
- Create min heap
  - $O(m)$
  - 1 time

Kruskal's Algorithm: Computational Complexity

- Create Disjoint Sets
  - $O(n)$
  - 1 time
- Check if we can add an edge (two finds and maybe one union)
  - $O^*(1)$
  - $O(m)$ times
- Create min heap
  - $O(m)$
  - 1 time

- Remove min

Kruskal's Algorithm: Computational Complexity

- Create Disjoint Sets
  - $O(n)$
  - 1 time
- Check if we can add an edge (two finds and maybe one union)
  - $O^*(1)$
  - $O(m)$ times
- Create min heap
  - $O(m)$
  - 1 time
- Remove min
  - $O(\log(m))$
  - $O(m)$ times

Kruskal's Algorithm: Computational Complexity

- Create Disjoint Sets
  - $O(n)$
  - 1 time
- Check if we can add an edge (two finds and maybe one union)
  - $O^*(1)$
  - $O(m)$ times
- Create min heap
  - $O(m)$
  - 1 time
- Remove min
  - $O(\log(m))$
  - $O(m)$ times

In total: $O(m\log(m)) = O(m\log(n))$ (b/c $n - 1 \leq m < n^2$).

Why does Kruskal's Algorithm Work?

Why does Kruskal's Algorithm Work?

- Maybe you already know the following well-known tree theorem (we'll prove next class):

Theorem

Suppose a graph on n vertices has any two of the following three properties:

1. *Connected*
2. *Acyclic*
3. *Has $n - 1$ edges*

Then that graph must be a tree. (Connected and acyclic).

Why does Kruskal's Algorithm Work?

- Maybe you already know the following well-known tree theorem (we'll prove next class):

Theorem

Suppose a graph on n vertices has any two of the following three properties:

1. *Connected*
2. *Acyclic*
3. *Has $n - 1$ edges*

Then that graph must be a tree. (Connected and acyclic).

- Can prove Kruskal's Algorithm works using the following:

Lemma

Suppose T and T' are both spanning trees of the same graph. Suppose, further, that e is in T but not in T'. Then we can find an e' in T' such that $T' + e - e'$ is a tree.

Outline

1. Minimum Spanning Trees
2. Kruskal's Algorithm
3. Prim's Algorithm

Prim's Algorithm: Basics

Prim's Algorithm: Basics

- Start with single vertex (any)

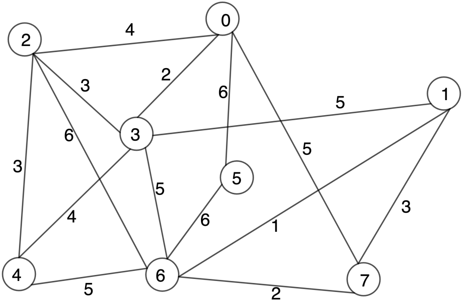
Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.

Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.
- From the current sub-tree, add the cheapest edge to a vertex not in the current sub-tree until you have a tree

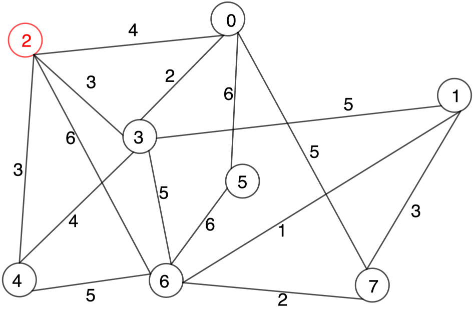
Example



Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.
- From the current sub-tree, add the cheapest edge to a vertex not in the current sub-tree until you have a tree

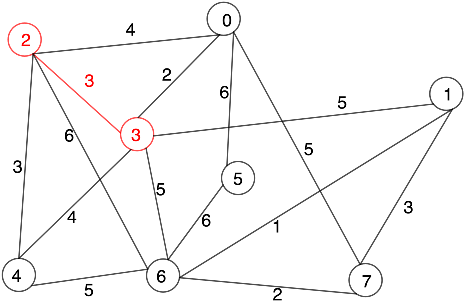
Example



Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.
- From the current sub-tree, add the cheapest edge to a vertex not in the current sub-tree until you have a tree

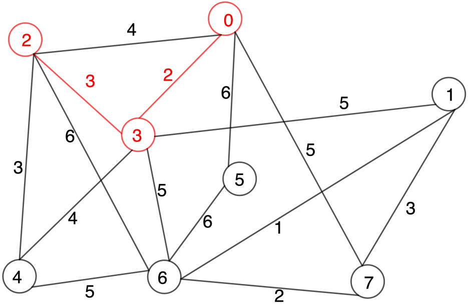
Example



Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.
- From the current sub-tree, add the cheapest edge to a vertex not in the current sub-tree until you have a tree

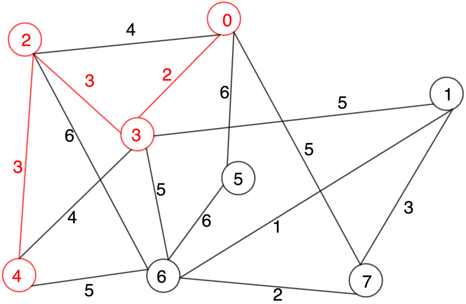
Example



Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.
- From the current sub-tree, add the cheapest edge to a vertex not in the current sub-tree until you have a tree

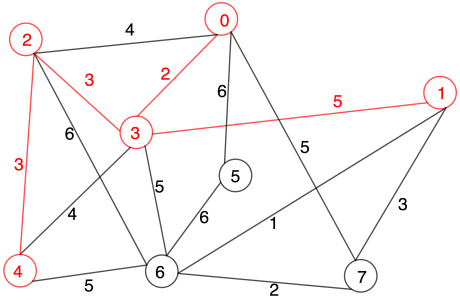
Example



Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.
- From the current sub-tree, add the cheapest edge to a vertex not in the current sub-tree until you have a tree

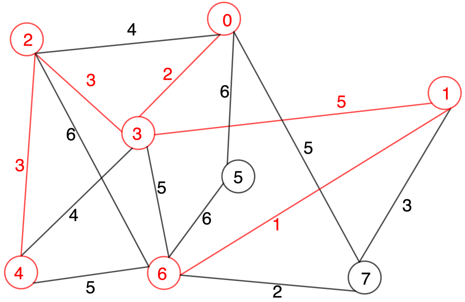
Example



Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.
- From the current sub-tree, add the cheapest edge to a vertex not in the current sub-tree until you have a tree

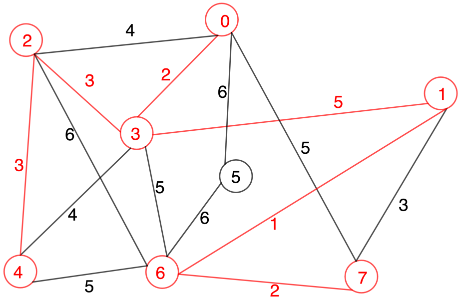
Example



Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.
- From the current sub-tree, add the cheapest edge to a vertex not in the current sub-tree until you have a tree

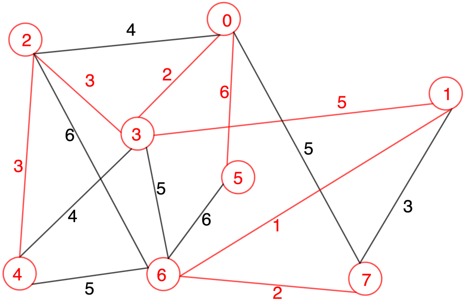
Example

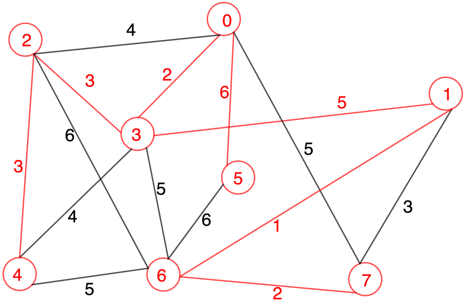


Prim's Algorithm: Basics

- Start with single vertex (any)
- Add cheapest edge incident to that vertex; this is a sub-tree.
- From the current sub-tree, add the cheapest edge to a vertex not in the current sub-tree until you have a tree

Example





Thus, the weight of the minimum spanning tree is:

$$3 + 2 + 3 + 5 + 1 + 2 + 6 = 22$$

- Of course, we got the same minimum weight.
- But note that there can be more than one minimum spanning tree

Prim's Algorithm: Implementation

- Use Array/Hash Map to keep track of which vertices not yet in subtree.

Prim's Algorithm: Implementation

- Use Array/Hash Map to keep track of which vertices not yet in subtree.
- Use Min Heap to keep track of cheapest edges to add
  - Items in min-heap are vertices, ordered by their distance from the sub-tree.
  - Initially we say the distance from any vertex to the sub-tree is ∞

Prim's Algorithm: Implementation

- Use Array/Hash Map to keep track of which vertices not yet in subtree.
- Use Min Heap to keep track of cheapest edges to add
  - Items in min-heap are vertices, ordered by their distance from the sub-tree.
  - Initially we say the distance from any vertex to the sub-tree is ∞
  - In each step, we add a vertex u to the sub-tree, by removing the vertex of min distance from the heap.
    - When we add vertex u to our sub-tree, we consider all edges from u to a vertex v not in our sub-tree
    - If v's current distance to the tree is larger than weight of that edge, update the distance to be the weight of that edge
    - Otherwise, don't update v's distance to the tree.
  - Continue until each vertex is pulled.

Prim's Algorithm: Computational Complexity

Prim's Algorithm: Computational Complexity

- Create Min Heap

Prim's Algorithm: Computational Complexity

- Create Min Heap
  - $O(n)$
  - 1 time

Prim's Algorithm: Computational Complexity

- Create Min Heap
  - $O(n)$
  - 1 time
- Remove Min

Prim's Algorithm: Computational Complexity

- Create Min Heap
  - $O(n)$
  - 1 time
- Remove Min
  - $O(\log(n))$
  - n times

Prim's Algorithm: Computational Complexity

- Create Min Heap
  - $O(n)$
  - 1 time
- Remove Min
  - $O(\log(n))$
  - n times
- Update distance in min heap

Prim's Algorithm: Computational Complexity

- Create Min Heap
  - $O(n)$
  - 1 time
- Remove Min
  - $O(\log(n))$
  - n times
- Update distance in min heap
  - $O(\log(n))$
  - $O(m)$ times

Prim's Algorithm: Computational Complexity

- Create Min Heap
  - $O(n)$
  - 1 time
- Remove Min
  - $O(\log(n))$
  - n times
- Update distance in min heap
  - $O(\log(n))$
  - $O(m)$ times
- Update Array/Hash Map listing which nodes are in subtree

Prim's Algorithm: Computational Complexity

- Create Min Heap
  - $O(n)$
  - 1 time
- Remove Min
  - $O(\log(n))$
  - n times
- Update distance in min heap
  - $O(\log(n))$
  - $O(m)$ times
- Update Array/Hash Map listing which nodes are in subtree
  - $O(1)$
  - $O(n)$ times

Prim's Algorithm: Computational Complexity

- Create Min Heap
  - $O(n)$
  - 1 time
- Remove Min
  - $O(\log(n))$
  - n times
- Update distance in min heap
  - $O(\log(n))$
  - $O(m)$ times

- Update Array/Hash Map listing which nodes are in subtree
  - $O(1)$
  - $O(n)$ times

In total: $O(m\log(n))$.

Why does Prim's Algorithm Work?

We can prove using the same Lemma used to prove Kruskal's algorithm works.

Why does Prim's Algorithm Work?

We can prove using the same Lemma used to prove Kruskal's algorithm works.

Note: There's yet one more greedy algorithm to find the Minimum Spanning tree!

Why does Prim's Algorithm Work?

We can prove using the same Lemma used to prove Kruskal's algorithm works.

Note: There's yet one more greedy algorithm to find the Minimum Spanning tree!

"Inverse Kruskal:". Remove the most expensive edges which do not disconnect the graph, until you have a tree.

All of these greedy algorithms imply a kind of robustness for the MST problem.