

Minimum Spanning Trees

CS/MCS 401: Computer Algorithms I

Lecture 15

Professor: Jonathan Liu

Icebreaker: What's your favorite tree?

Announcements

Make sure to think about who you want to work with for the project! You'll have to let me know by next Friday. You may work with a partner, unless you are a graduate student in which case you must work alone.

I have opened up Midterm Redos for Question 1, if you'd prefer that.

News in Algorithms-Land

Name(s)	Year	Runtime	How?	Good n
[everyone]	<1960	$O(n^2)$	All human ways	$n < 30$ ish
Karatsuba	1960	$O(n^{1.59})$	Divide and Conquer	$n < 1000$ s
Toom and Cook	1963	$O(n^{(\log 2k-1)/(\log k)})$	Split into k pieces instead of 2	
Schönhage–Strassen	1971	$O(n \cdot \log n \cdot \log \log n)$	Fast Fourier Transform (FFT)	$n \sim 1000$ s
Fürer	2007	$O(n \cdot \log n \cdot 2^{\log^* n})$	Recursive FFT	
Harvey and van der Hoeven	2019	$O(n \cdot \log n)$	Multidimensional FFT	$2^{1729^{12}}$

This is a *galactic algorithm* – asymptotically great, practically useless.

News in Algorithms-Land

Integer multiplication below $n \log n$

OpenAI

September 23, 2026

Abstract

We give a deterministic algorithm that multiplies two n -bit integers in $O(n(\lg n)^{1-\kappa})$ worst-case time, with $\kappa = 2^{-182}$, on one fixed finite-alphabet Turing machine with a fixed finite number of one-dimensional tapes. The algorithm is exact for every input length and disproves the $n \log n$ optimality conjecture of Schönhage and Strassen in this model.

This is a *galactic algorithm* – asymptotically great, practically useless.

News in Algorithms-Land



Doug Colkitt ✓

@0xdoug

...

We are publishing an update to OpenAI problem #109 (integer multiplication) with further tightening.

$\kappa = 2^{-34}$ (tightened from $\kappa = 2^{-182}$)

Runtime: $O(n (\log n)^{0.999999999927})$

The exact witness is 8.3×10^{-11} , a roughly 48 million fold improvement over the previous result and a 2^{148} fold improvement over original OAI result.

The improvement came from removing the spacing penalty behind the quadratic bottleneck. This was done by moving compact control bits instead of entire windows.

We give
case time,
of one-dim
optimality

κ) worst-
e number
e $n \log n$

This is a *galactic algorithm* – asymptotically great, practically useless.

Today's Overview

Today, we are learning about two (three?) algorithms for a classic algorithms problem: Minimum Spanning Trees.

Today will combine this unit and the previous one: we'll be using greedy concepts to develop a greedy algorithm for a graph problem, but future problems will use those algorithms as a baseline to do new things.

Minimum Spanning Trees



Problem 10: Pink Roads

You're doing a city-wide promotion where you want to paint some roads pink so that anyone following the roads can get to your shop. You want to figure out the minimum amount of paint you need to do so.

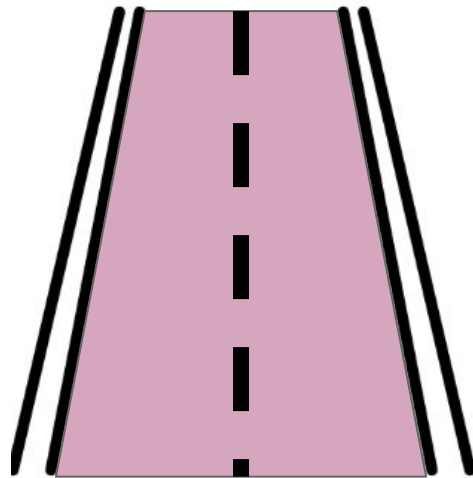
Problem Modeling

What is the input data?

A graph of the city's roads, where each road is weighted by the road's area, and the vertex where your shop is.

What is the output data?

The roads you need to paint to ensure that their area is minimized but every vertex can get to your shop.



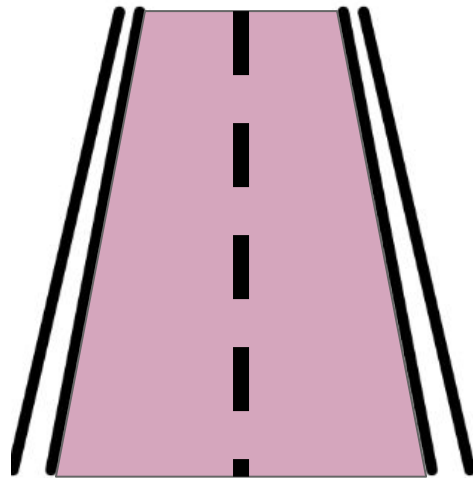
Problem 10: Pink Roads

You're doing a city-wide promotion where you want to paint some roads pink so that anyone following the roads can get to your shop. You want to figure out the minimum amount of paint you need to do so.

What must be true about the subgraph consisting of the roads that you end up painting?

- The roads must reach every point on the graph.
- The roads must be connected.
- The roads must be acyclic.

This is called a *spanning tree*.

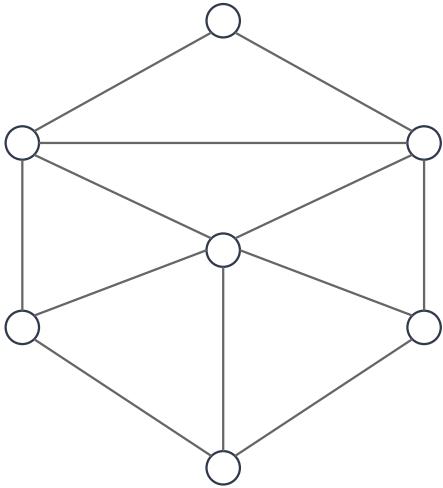


Spanning Trees

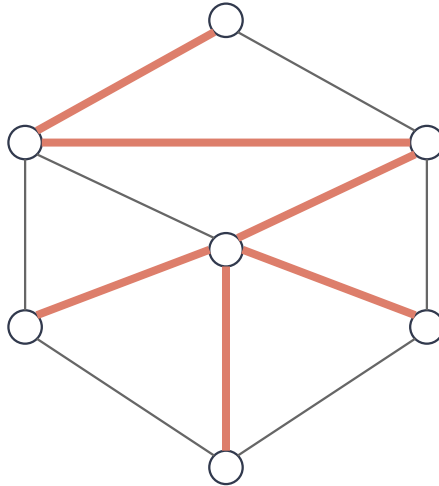
A *tree* is a connected acyclic graph.

A *spanning tree* of a graph is a subgraph that is a tree but is connected to every vertex on the graph.

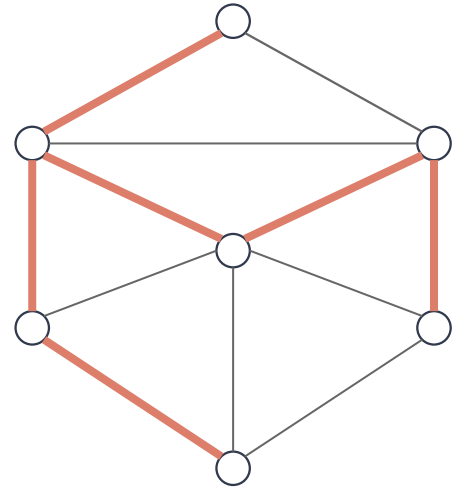
Our Graph



Spanning Tree

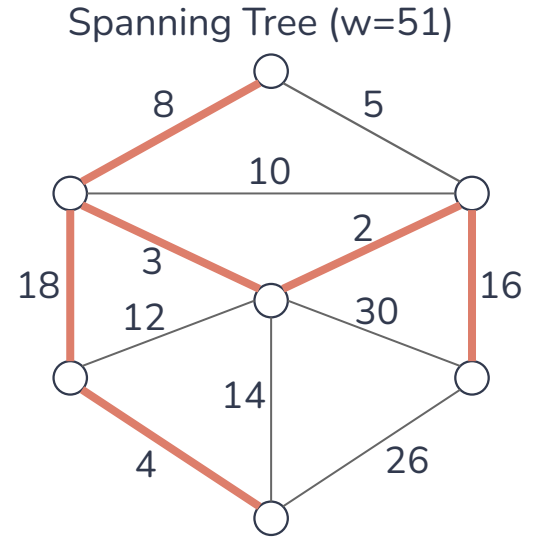
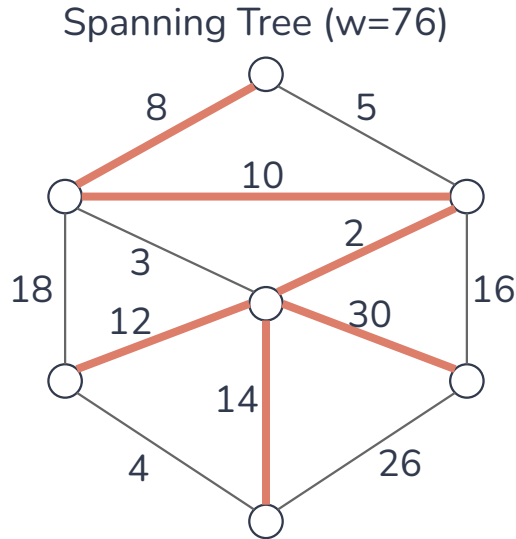
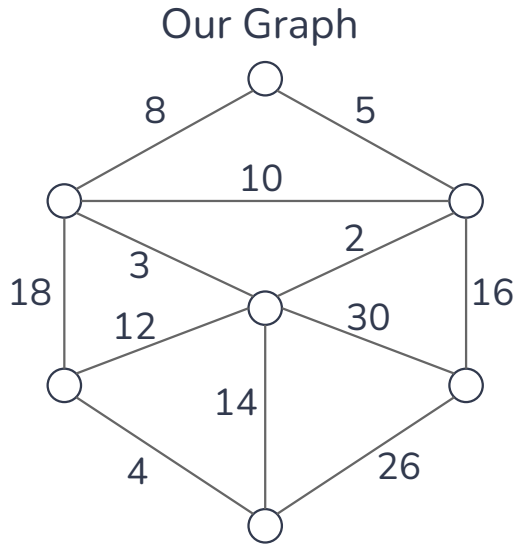


Spanning Tree



Minimum Spanning Trees

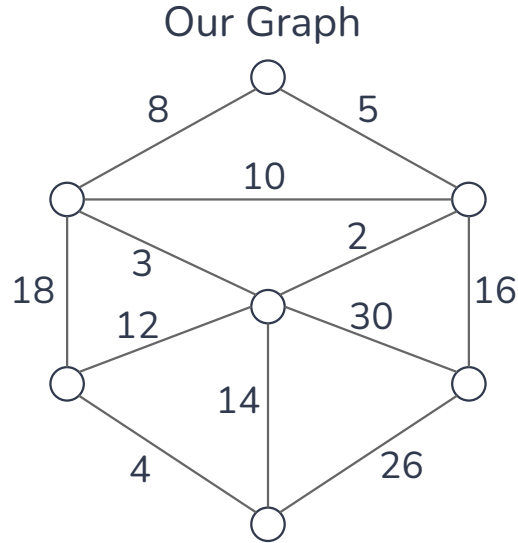
On a weighted graph, each spanning tree has a weight as well.



A minimum spanning tree is the spanning tree with lowest weight.

Try it yourself!

Find a minimum spanning tree of this graph.



A Greedy Algorithm for Minimum Spanning Trees



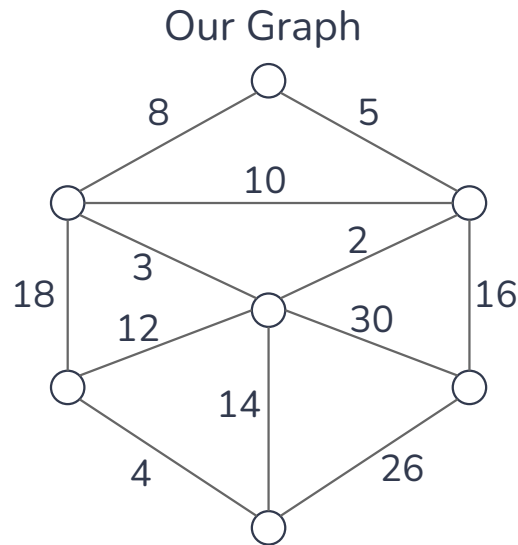
Designing a Greedy Algorithm

Note that a graph with n vertices can have up to n^{n-2} spanning trees!

Pseudocode:

```
while not done:  
    pick the best thing  
return the result
```

What heuristic should we use?



Designing a Greedy Algorithm

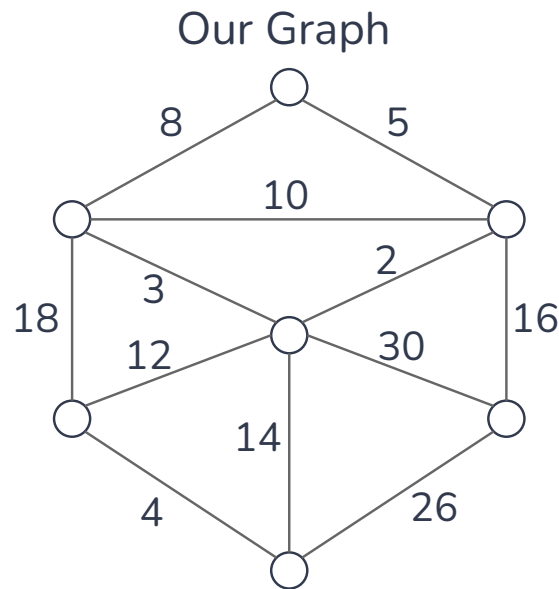
Pseudocode:

$T = \{\}$

while there are unconnected vertices:

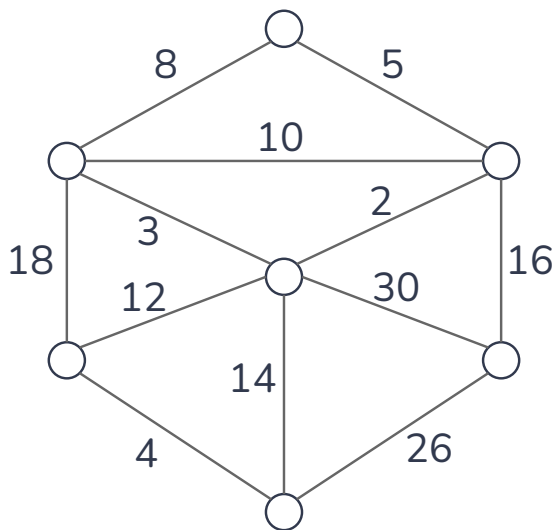
 take the smallest edge unless it
 makes a cycle

Return T



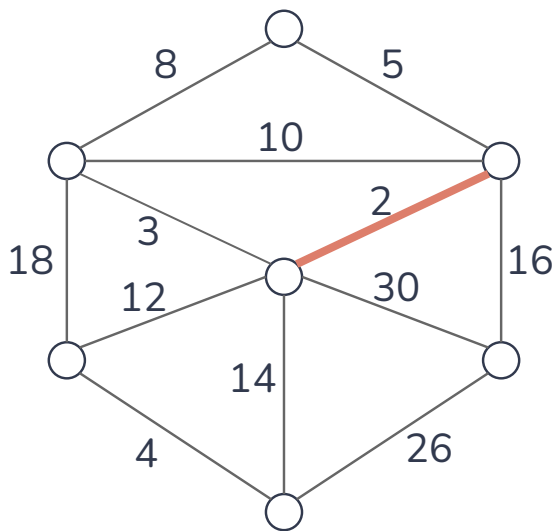
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



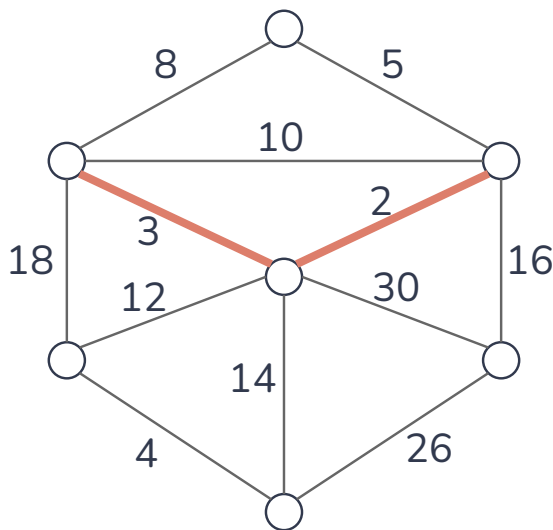
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



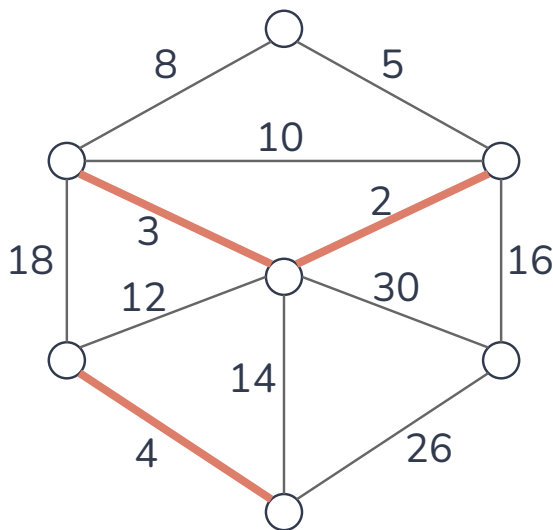
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



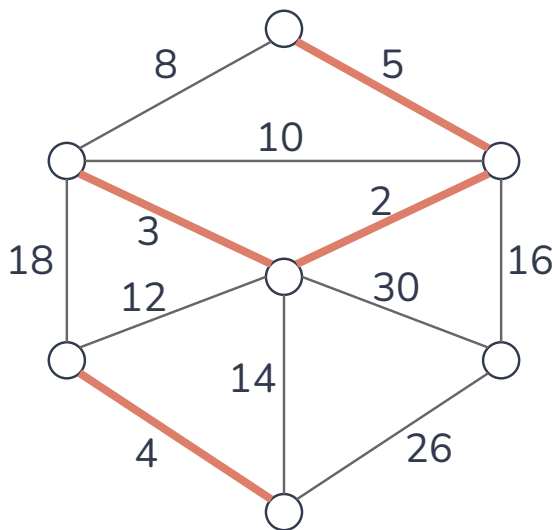
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



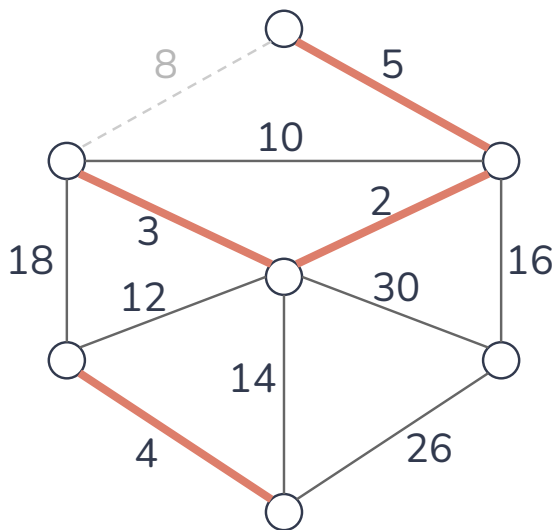
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



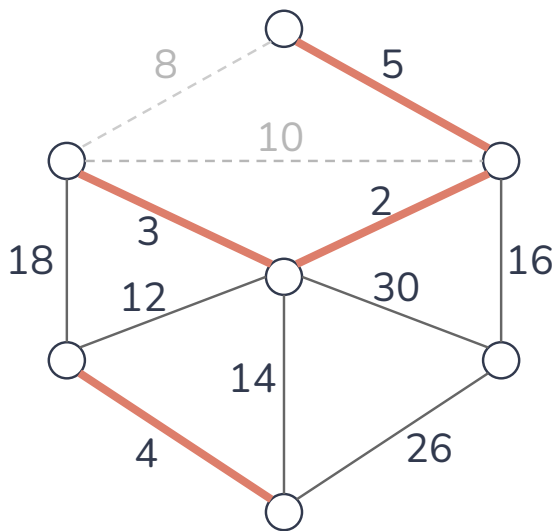
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



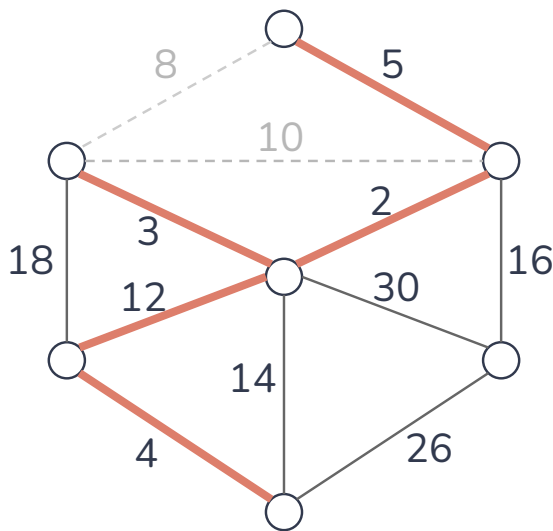
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



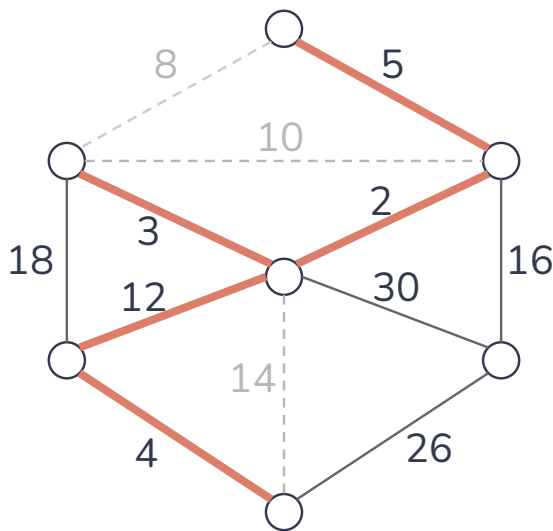
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



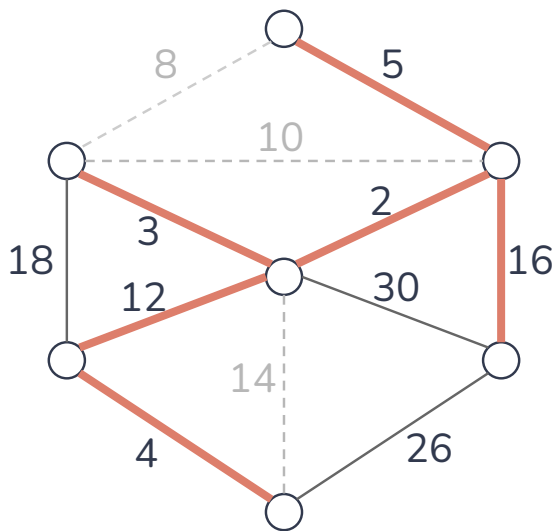
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



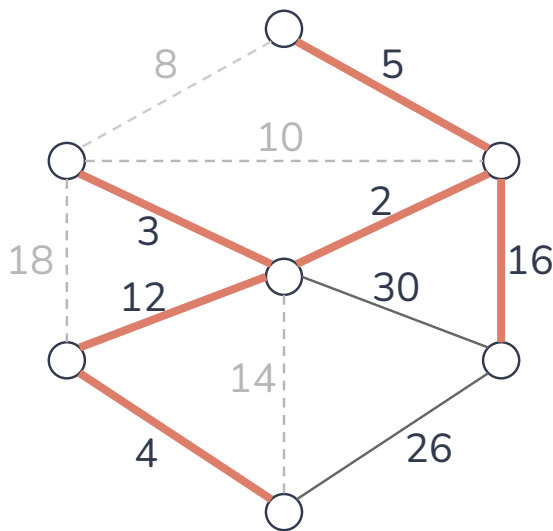
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



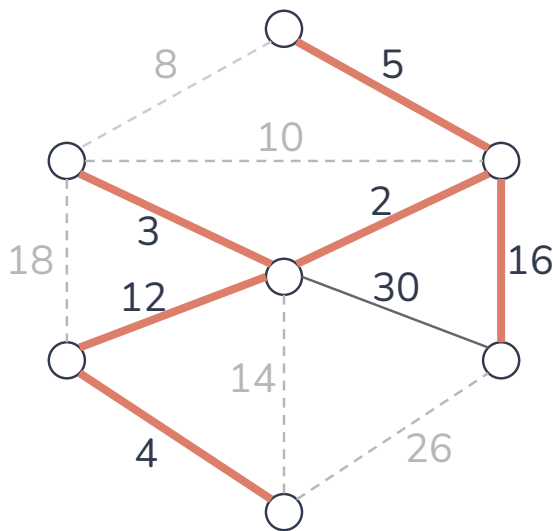
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



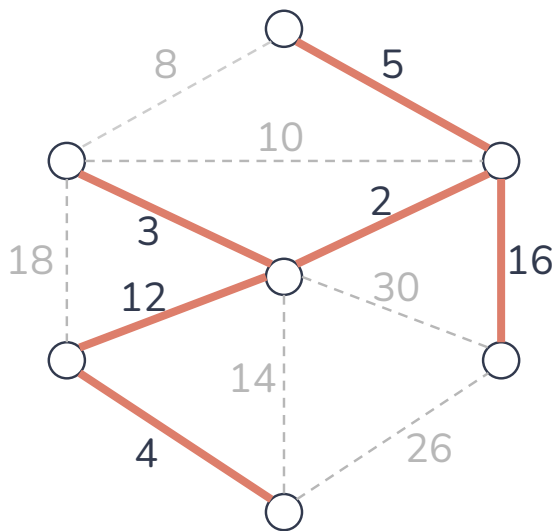
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



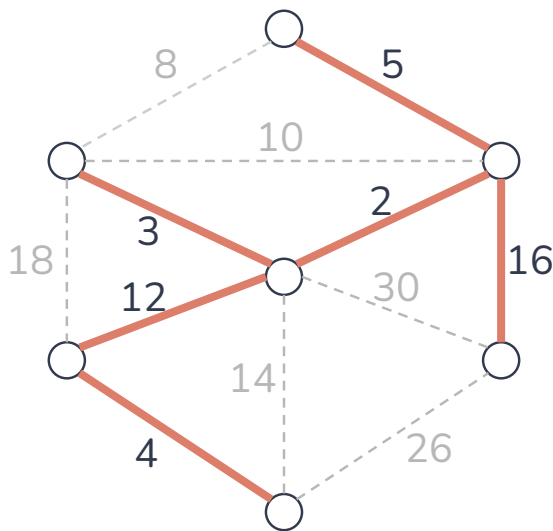
Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



Kruskal's Algorithm (Joseph Kruskal, 1956)

We just wrote Kruskal's Algorithm! Try it on this graph.



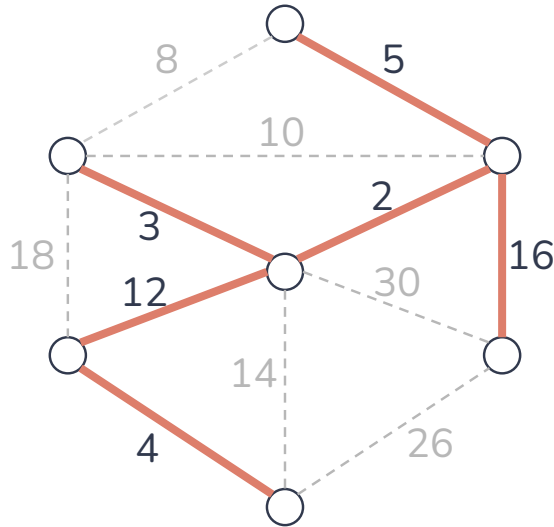
Weight = 42

Proving the Correctness of MST Algorithms



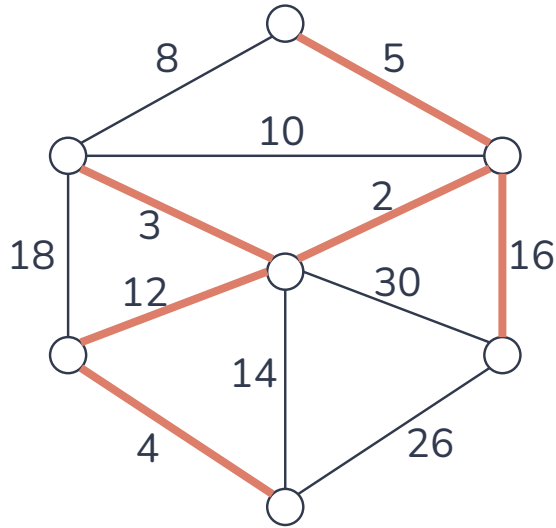
Proving Correctness

How do we prove that our algorithm works?



Proving Correctness

How do we prove that our algorithm works?

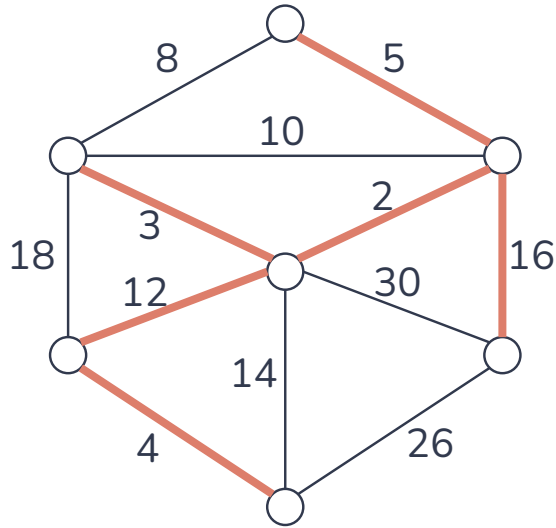


Blueprint:

1. Assume that there is some optimal solution that is different from the greedy solution.
2. Label the first selection where they differ.
3. Argue that, if that selection was replaced with what the greedy algorithm selected, the solution would remain optimal.
4. Conclude that no optimal solution can be better than the greedy solution.

Proving Correctness

How do we prove that our algorithm works?

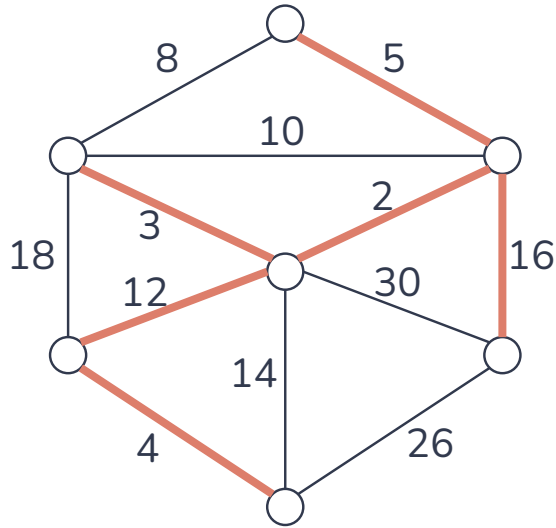


Blueprint:

1. Assume that there is some optimal solution that is different from the greedy solution.
2. Label the first selection where they differ.
3. Argue that, if that selection was replaced with what the greedy algorithm selected, the solution would remain optimal.
4. Conclude that no optimal solution can be better than the greedy solution.

Proving Correctness

How do we prove that our algorithm works?



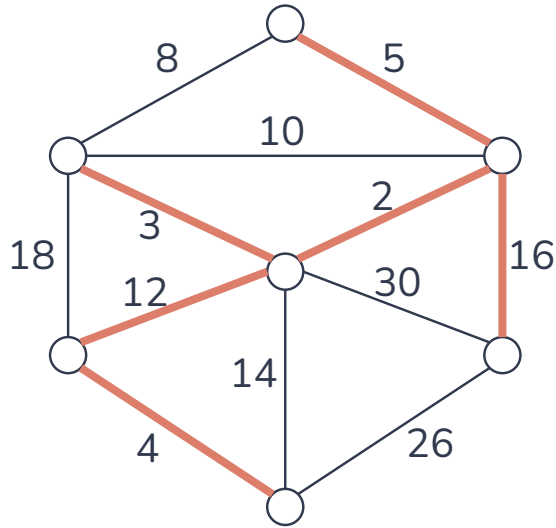
What does “first” mean here?

Blueprint:

1. Assume that there is some optimal solution that is different from the greedy solution.
2. **Label the first selection where they differ.**
3. Argue that, if that selection was replaced with what the greedy algorithm selected, the solution would remain optimal.
4. Conclude that no optimal solution can be better than the greedy solution.

Proving Correctness

How do we prove that our algorithm works?



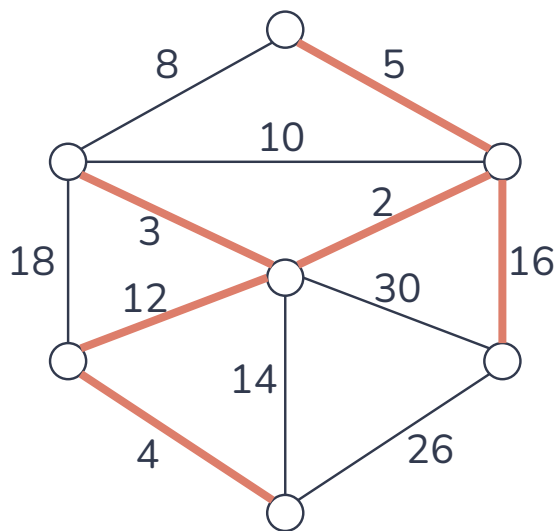
It's unclear, so we'll fix it.

Blueprint:

1. Assume that there is some optimal solution that is different from the greedy solution.
2. **Label the first selection where they differ.**
3. Argue that, if that selection was replaced with what the greedy algorithm selected, the solution would remain optimal.
4. Conclude that no optimal solution can be better than the greedy solution.

Proving Correctness

How do we prove that our algorithm works?



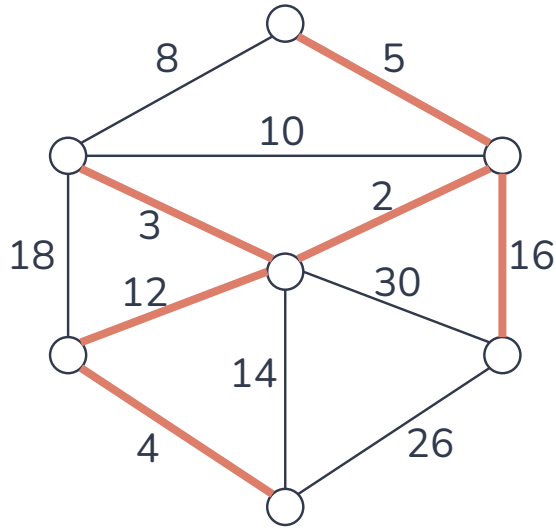
It's unclear, so we'll fix it.

Blueprint:

1. Assume that there is some optimal solution that is different from the greedy solution.
2. **Find a selection where they differ.**
3. Argue that, if that selection was replaced with what the greedy algorithm selected, the solution would remain optimal.
4. Conclude that no optimal solution can be better than the greedy solution.

Proving Correctness

How do we prove that our algorithm works?



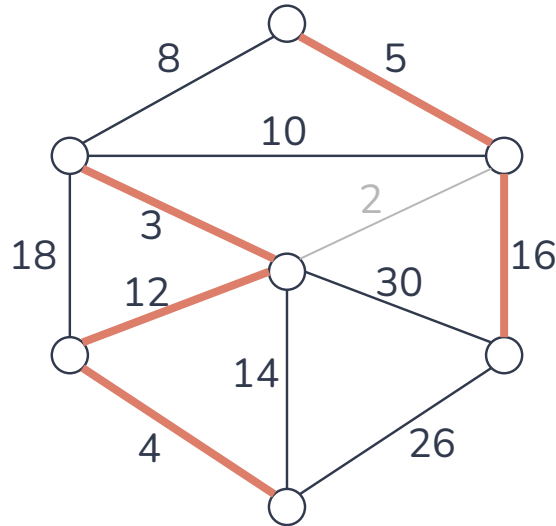
Blueprint:

1. Assume that there is some optimal solution that is different from the greedy solution.
2. Find a selection where they differ.
3. **Argue that, if that selection was replaced with what the greedy algorithm selected, the solution would remain optimal.**
4. Conclude that no optimal solution can be better than the greedy solution.

Proving Correctness

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.

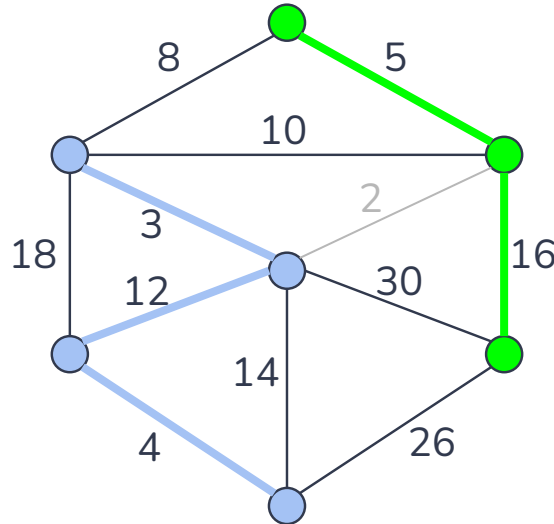
Let's look at what happens if the optimal solution didn't pick the edge of weight 2. Removing that edge disconnects our tree and leaves two connected components.



Proving Correctness

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.

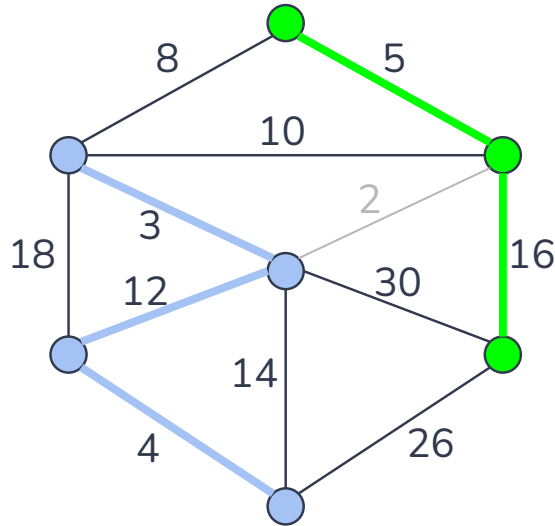
Let's look at what happens if the optimal solution didn't pick the edge of weight 2. Removing that edge disconnects our tree and leaves two connected components.



Proving Correctness

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.

Let's look at what happens if the optimal solution didn't pick the edge of weight 2. Removing that edge disconnects our tree and leaves two connected components.



What can we conclude about the optimal solution?

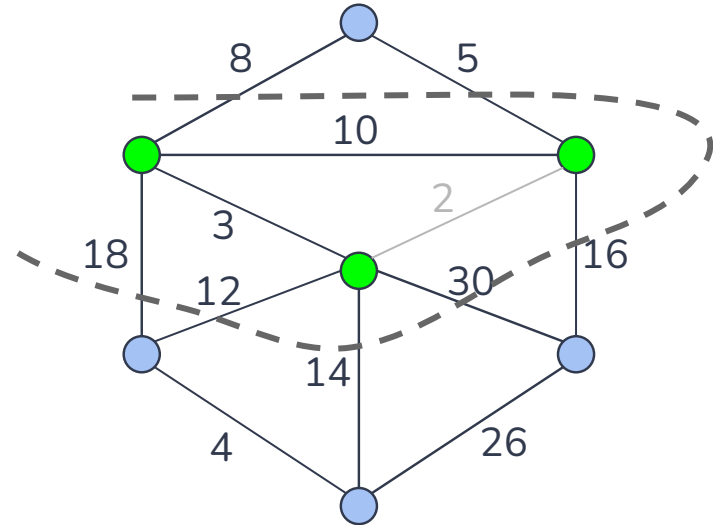
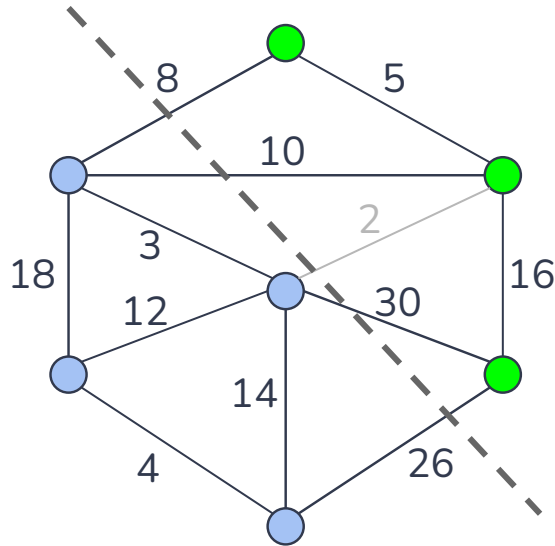
If it didn't take the 2 edge, it must have taken one of the other edges that connect these two sides.

But, if it took a different one, it'd be better off taking the 2 instead!

The Cut Property

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.

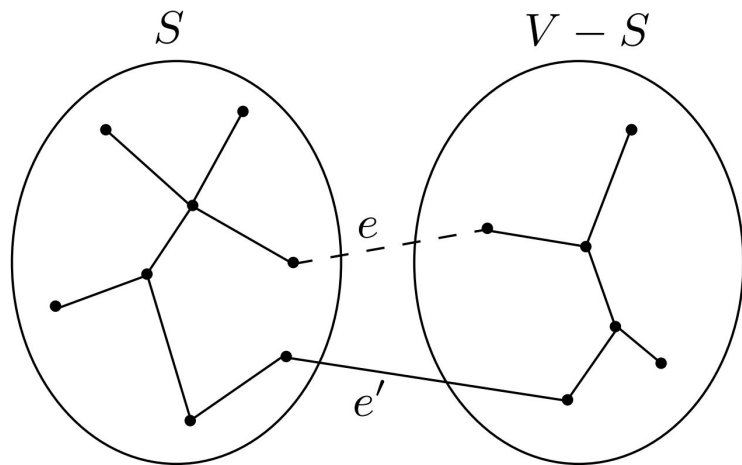
A **cut** is a split of a graph's vertices into two sets, S and $V \setminus S$. You can view this as coloring each vertex of the graph, or as a literal cut through the graph.



The Cut Property

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.

A **cut** is a split of a graph's vertices into two sets, S and $V \setminus S$. You can view this as coloring each vertex of the graph, or as a literal cut through the graph.



The Cut Property

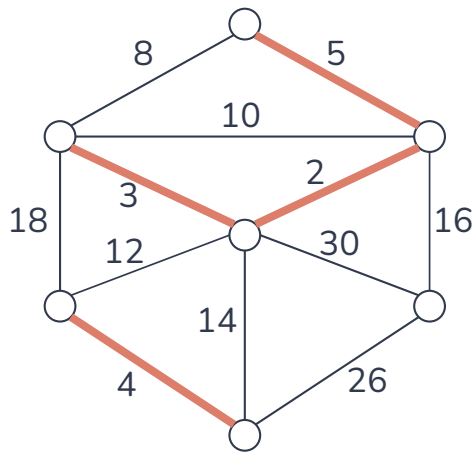
The lowest-weight edge along any cut must be part of a minimum spanning tree.

Proof: Contradiction!

Assume, for contradiction, that there is an MST T that does not contain the lowest-weight edge e along a cut. It is a spanning tree, so it must contain some edge along that cut, e' . But replacing e' with e would keep it a spanning tree and lower the total weight, so T is not minimal.

Back to our proof!

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.



Informally: Say we're considering adding e .

Kruskal's Algorithm:

$T = \text{empty}$

while T is not spanning:

 add to T the

 lowest-weight edge that

 doesn't make a cycle in T

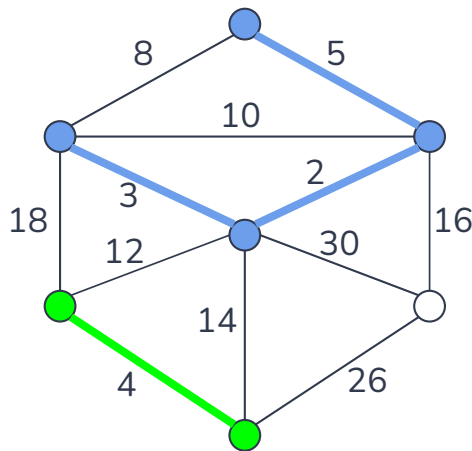
return T

The Cut Property

The lowest-weight edge along any cut must be part of a minimum spanning tree.

Back to our proof!

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.



Informally: Say we're considering adding e . Adding it doesn't create a cycle, so it's connecting two separate components of the tree.

Kruskal's Algorithm:

$T = \text{empty}$

while T is not spanning:

 add to T the

 lowest-weight edge that

 doesn't make a cycle in T

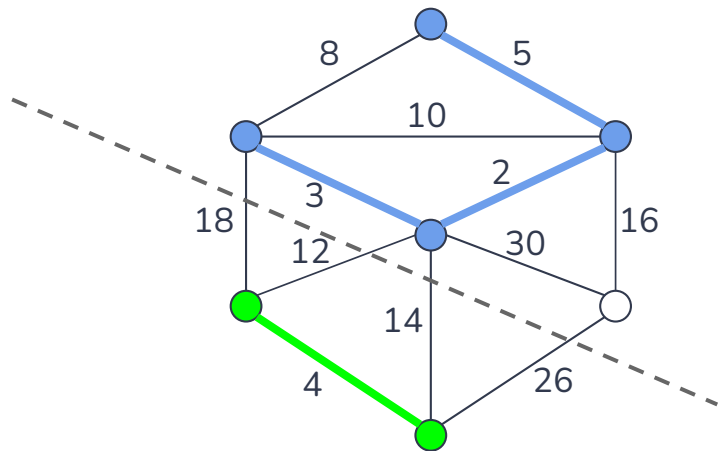
return T

The Cut Property

The lowest-weight edge along any cut must be part of a minimum spanning tree.

Back to our proof!

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.



Informally: Say we're considering adding e . Adding it doesn't create a cycle, so it's connecting two separate components of the tree. Because e is the lowest-weight edge we can add, it's also the lowest-weight edge along any cut between those two components.

Kruskal's Algorithm:

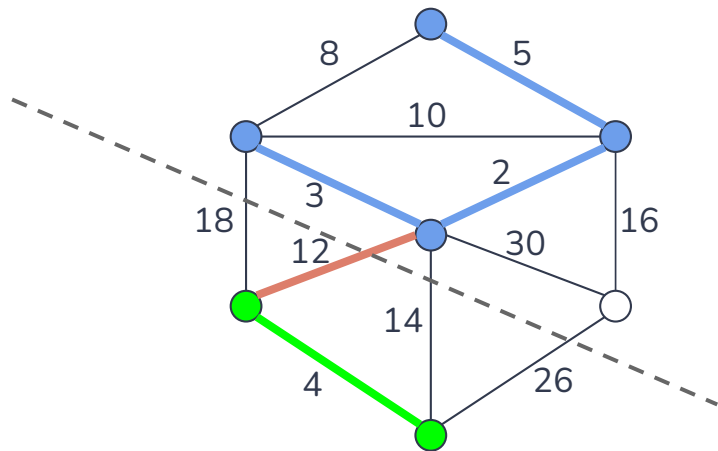
```
T = empty
while T is not spanning:
    add to T the
    lowest-weight edge that
    doesn't make a cycle in T
return T
```

The Cut Property

The lowest-weight edge along any cut must be part of a minimum spanning tree.

Back to our proof!

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.



Informally: Say we're considering adding e . Adding it doesn't create a cycle, so it's connecting two separate components of the tree. Because e is the lowest-weight edge we can add, it's also the lowest-weight edge along any cut between those two components. Thus any optimal solution must take e .

Kruskal's Algorithm:

```
T = empty
while T is not spanning:
    add to T the
    lowest-weight edge that
    doesn't make a cycle in T
return T
```

The Cut Property

The lowest-weight edge along any cut must be part of a minimum spanning tree.

A Formal Proof

Goal: Argue that, if the optimal solution didn't pick what we picked, it could have picked what we picked instead.

Assume Kruskal's algorithm returned some T , and there is some optimal MST T' that is not T . Then there is some edge e in T but not in T' .

Because e was added to T , its addition connected two previously-disjoint components. Thus, for any cut that separates those two components, e is the minimal weight edge along that cut. T' must contain an edge along this cut, but by the Cut Property, that edge must be e . So $T=T'$.

Kruskal's Algorithm:

```
T = empty
while T is not spanning:
    add to T the
    lowest-weight edge that
    doesn't make a cycle in T
return T
```

The Cut Property

The lowest-weight edge along any cut must be part of a minimum spanning tree.

What We've Shown

First we showed that the Cut Property is true: that if you take any cut in the graph, the lowest-weight edge in that cut goes in the MST.

Second, we showed that any edge picked by Kruskal's Algorithm must take the lowest-weight edge along some cut.

We can conclude then that no spanning tree can do better than what Kruskal's Algorithm has done.

Kruskal's Algorithm:

```
T = empty
while T is not spanning:
    add to T the
    lowest-weight edge that
    doesn't make a cycle in T
return T
```

The Cut Property

The lowest-weight edge along any cut must be part of a minimum spanning tree.

Runtime

How long does this take?

- Sorting the edges takes $O(E \log E)$ time.
- How long does it take to determine if an edge adds a cycle?
 - If you do it manually, it might take $O(V+E)$ per added edge!
 - With a good data structure (out of scope for this class), you can get it down to $O(\log V)$ per added edge, so that would take $O(V \log V)$ for the whole loop.
- Because $E \geq V$ (why?), the runtime is **$O(E \log V)$** .

Kruskal's Algorithm:

$T = \text{empty}$

while T is not spanning:

 add to T the

 lowest-weight edge that

 doesn't make a cycle in T

return T

Other *MST* Algorithms



Wait a Second...

If the cut property is true, that implies a much broader set of algorithms:

Generic MST Algorithm:

```
T = empty
while T is not spanning:
    find a cut of the graph
    find the minimum-weight edge
    along that cut
    add that edge to T
return T
```

Take a few minutes to try it!

Kruskal's Algorithm:

```
T = empty
while T is not spanning:
    add to T the
    lowest-weight edge that
    doesn't make a cycle in T
return T
```

The Cut Property

The lowest-weight edge along any cut must be part of a minimum spanning tree.

Another MST Algorithm

Discovered by Vojtěch Jarník in 1929 and published in 1930.



Another MST Algorithm

Discovered by Vojtěch Jarník in 1929 and published in 1930.

Discovered by Joseph Kruskal in 1956 (with the previous algorithm).



Another MST Algorithm

Discovered by Vojtěch Jarník in 1929 and published in 1930.

Discovered by Joseph Kruskal in 1956 (with the previous algorithm).

Discovered by Robert Prim in 1957 (died in 2021, at age 100!).



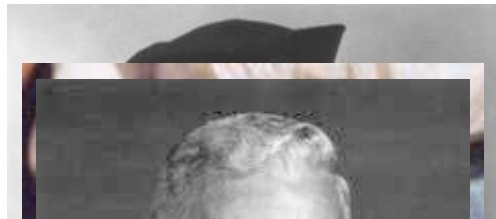
Another MST Algorithm

Discovered by Vojtěch Jarník in 1929 and published in 1930.

Discovered by Joseph Kruskal in 1956 (with the previous algorithm).

Discovered by Robert Prim in 1957 (died in 2021, at age 100!).

Discovered by Harry Loberman and Arnold Weinberger in 1957.



Formal Procedures for Connecting Terminals with a Minimum Total Wire Length*

H. LOBERMAN AND A. WEINBERGER

National Bureau of Standards, Washington, D. C.



Another MST Algorithm

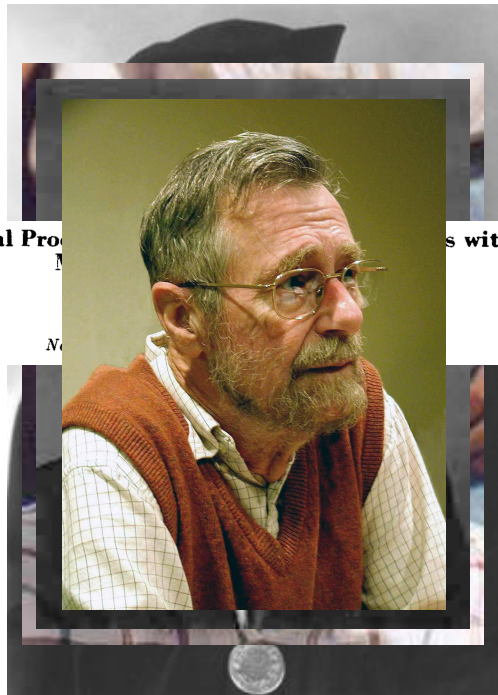
Discovered by Vojtěch Jarník in 1929 and published in 1930.

Discovered by Joseph Kruskal in 1956 (with the previous algorithm).

Discovered by Robert Prim in 1957 (died in 2021, at age 100!).

Discovered by Harry Loberman and Arnold Weinberger in 1957.

Discovered by Edsger Dijkstra in 1958.



Prim's Algorithm

Discovered by Vojtěch Jarník in 1929 and published in 1930.

Discovered by Joseph Kruskal in 1956 (with the previous algorithm).

Discovered by Robert Prim in 1957 (died in 2021, at age 100!).

Discovered by Harry Loberman and Arnold Weinberger in 1957.

Discovered by Edsger Dijkstra in 1958.

Sometimes called Prim-Dijkstra algorithm or the DJP algorithm.



Prim's Algorithm

Prim's Algorithm:

T = empty

S = some vertex v \\ set of visited vertices

while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T

Generic MST Algorithm:

T = empty

while T is not spanning:

 find a cut of the graph

 find the minimum-weight edge

 along that cut

 add that edge to T

return T

Prim's Algorithm

Prim's Algorithm:

T = empty

S = some vertex v $\setminus$ set of visited vertices

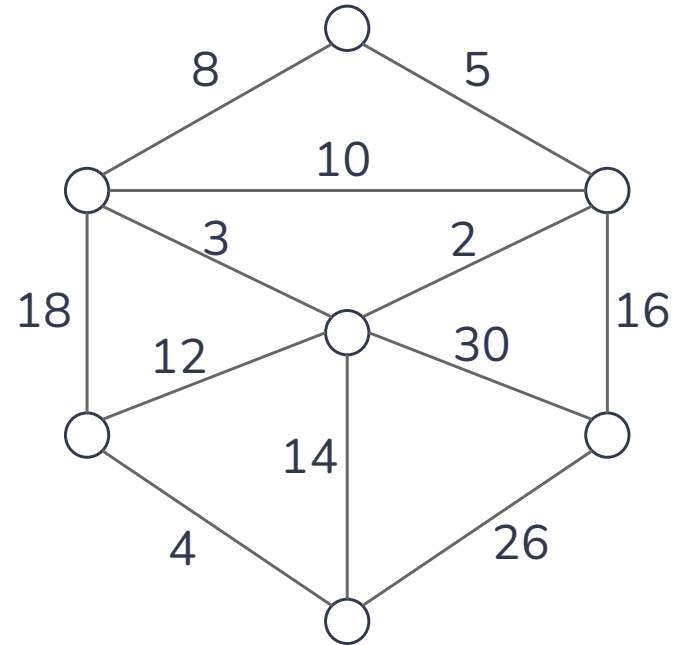
while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

$T = \text{empty}$

$S = \text{some vertex } v$ $\setminus \setminus$ set of visited vertices

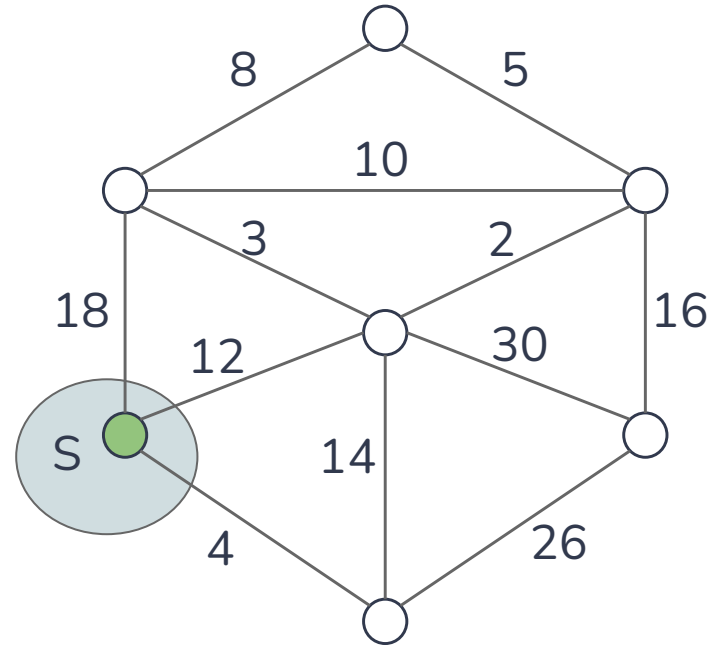
while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

$T = \text{empty}$

$S = \text{some vertex } v$ $\setminus \setminus$ set of visited vertices

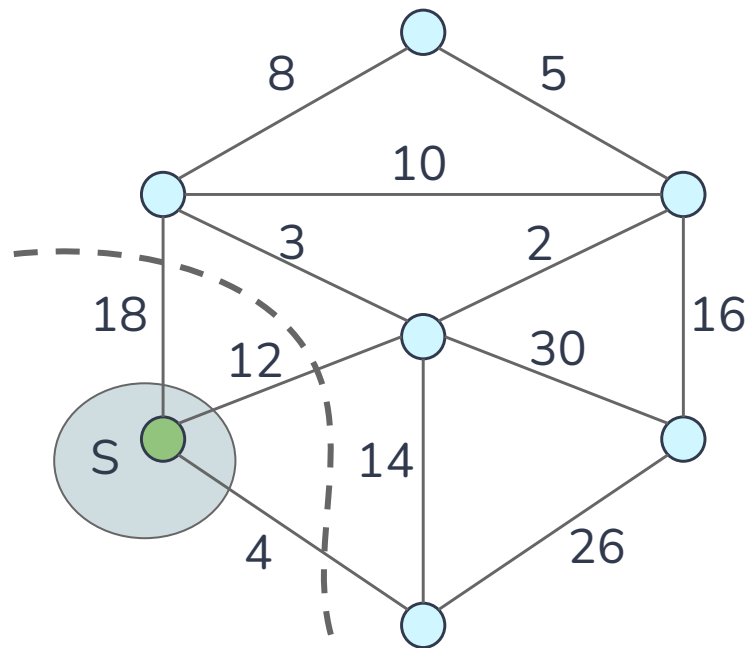
while T is not spanning:

select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

$T = \text{empty}$

$S = \text{some vertex } v$ $\setminus \setminus$ set of visited vertices

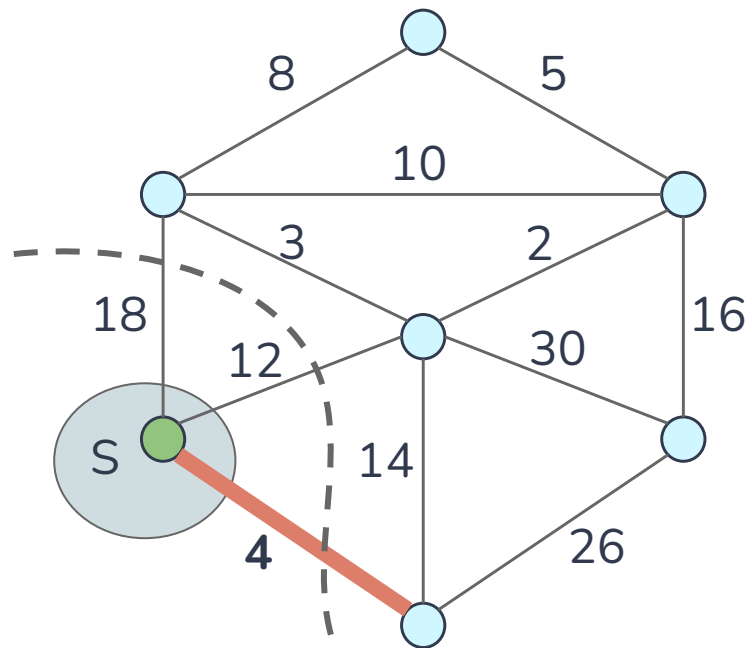
while T is not spanning:

 select, as your cut, S and $V \setminus S$

pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

T = empty

S = some vertex v $\setminus$ set of visited vertices

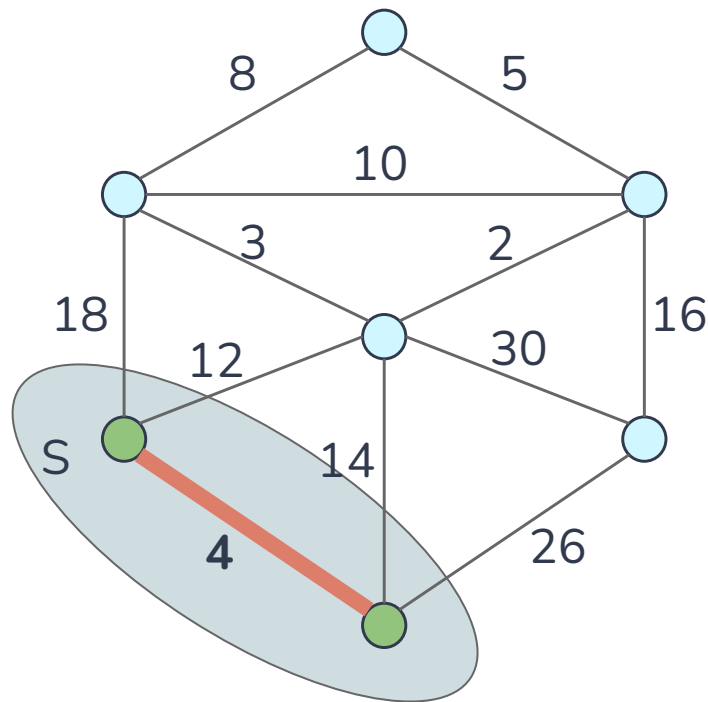
while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

$T = \text{empty}$

$S = \text{some vertex } v$ $\setminus \setminus$ set of visited vertices

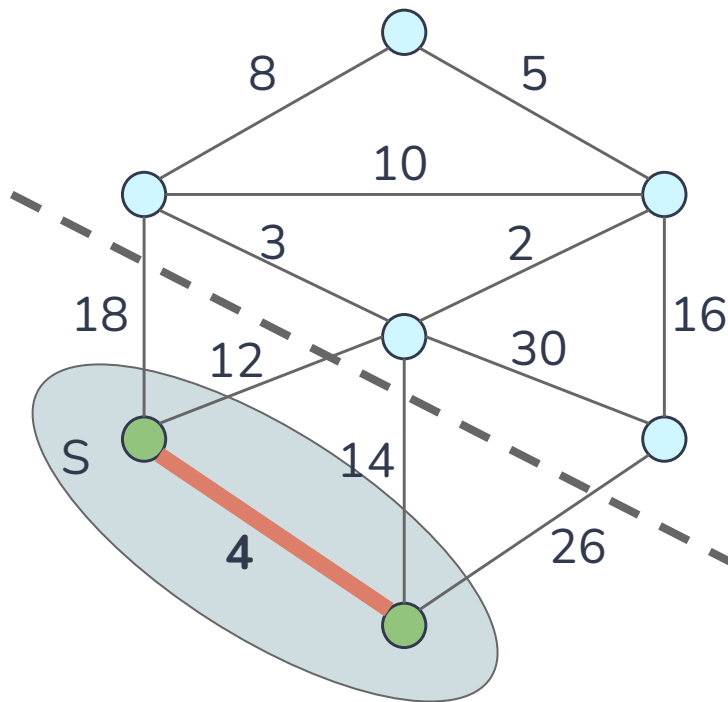
while T is not spanning:

select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

$T = \text{empty}$

$S = \text{some vertex } v$ $\setminus \setminus$ set of visited vertices

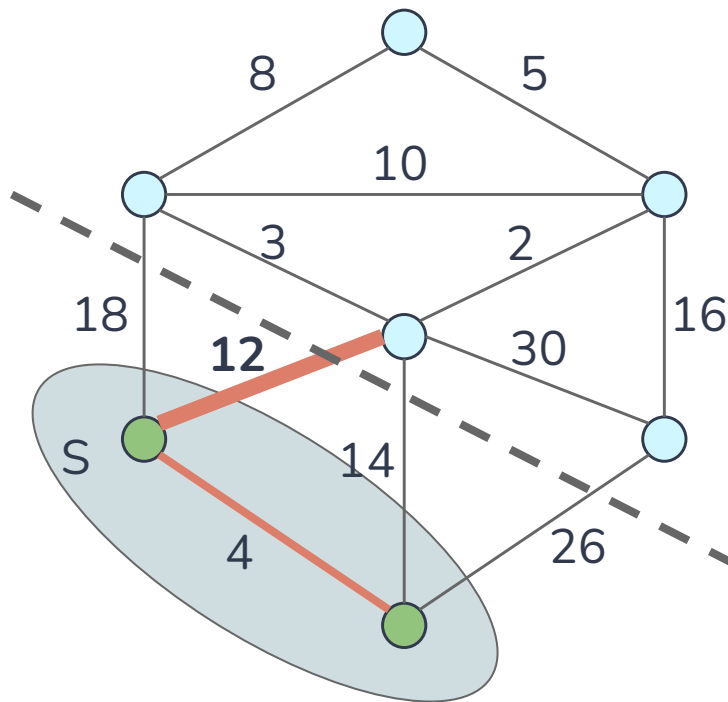
while T is not spanning:

 select, as your cut, S and $V \setminus S$

pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

T = empty

S = some vertex v \\ set of visited vertices

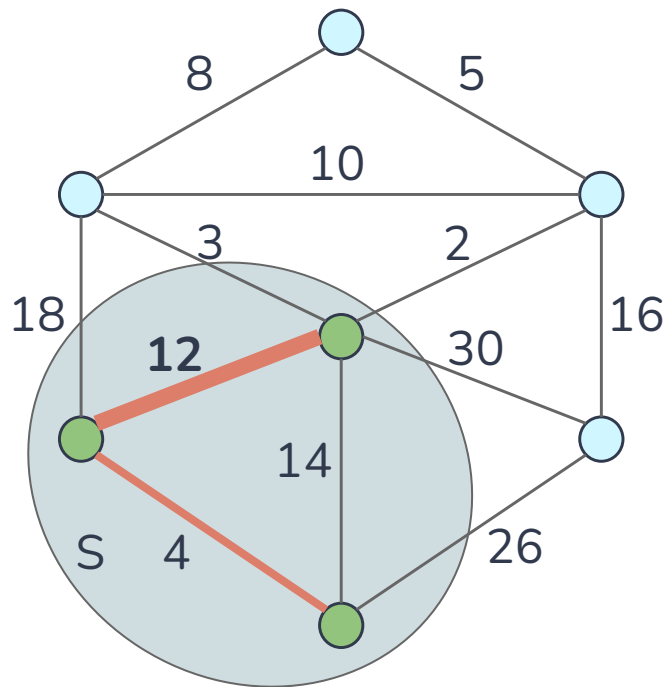
while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

$T = \text{empty}$

$S = \text{some vertex } v$ $\setminus \setminus$ set of visited vertices

while T is not spanning:

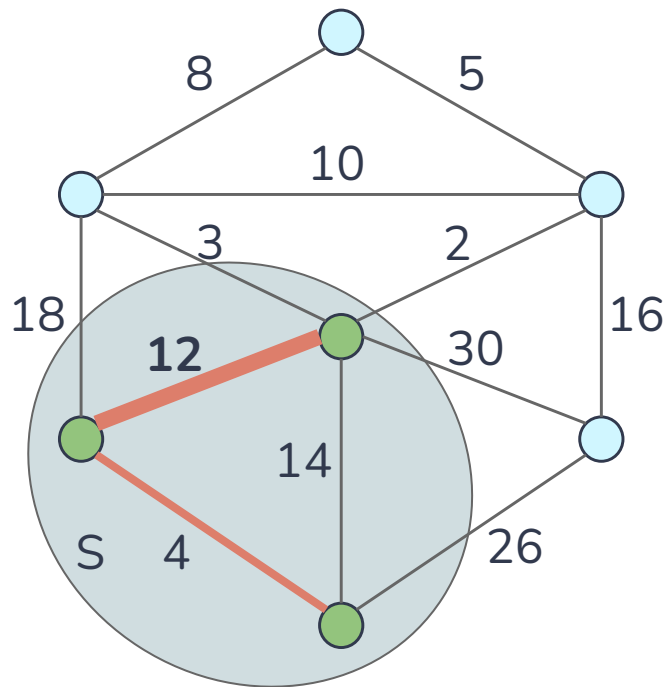
 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T

What would we add next?



Prim's Algorithm

Prim's Algorithm:

$T = \text{empty}$

$S = \text{some vertex } v$ $\setminus \setminus$ set of visited vertices

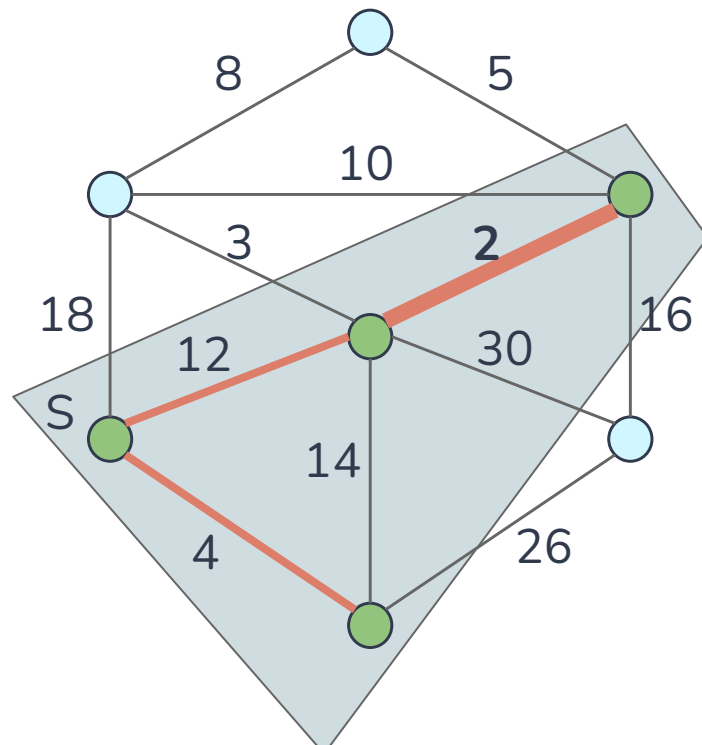
while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

$T = \text{empty}$

$S = \text{some vertex } v$ $\setminus \setminus$ set of visited vertices

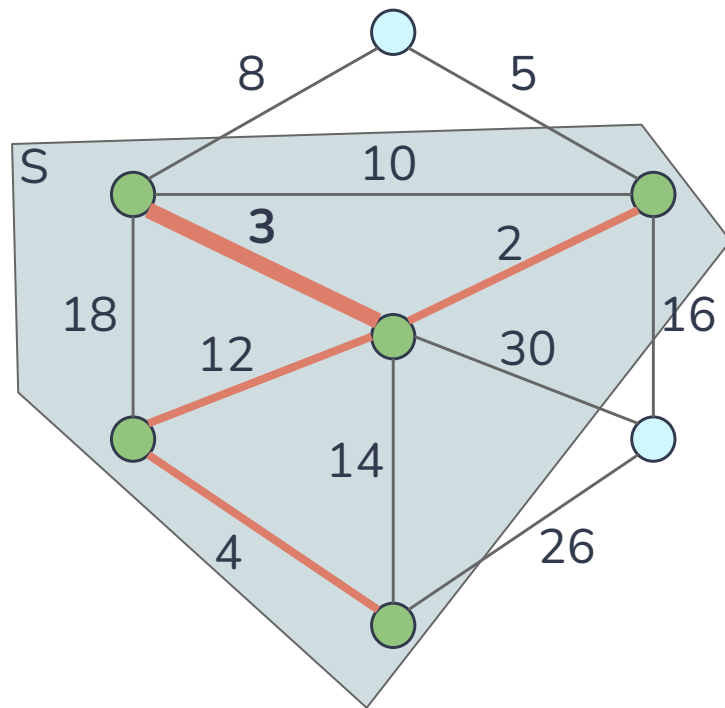
while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

$T = \text{empty}$

$S = \text{some vertex } v$ $\setminus \setminus$ set of visited vertices

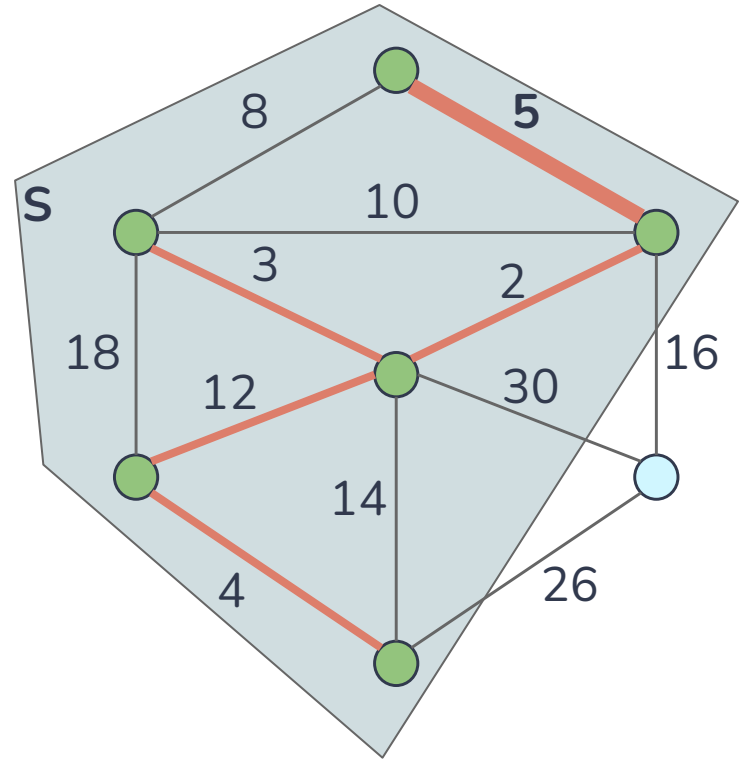
while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

T = empty

S = some vertex v \\ set of visited vertices

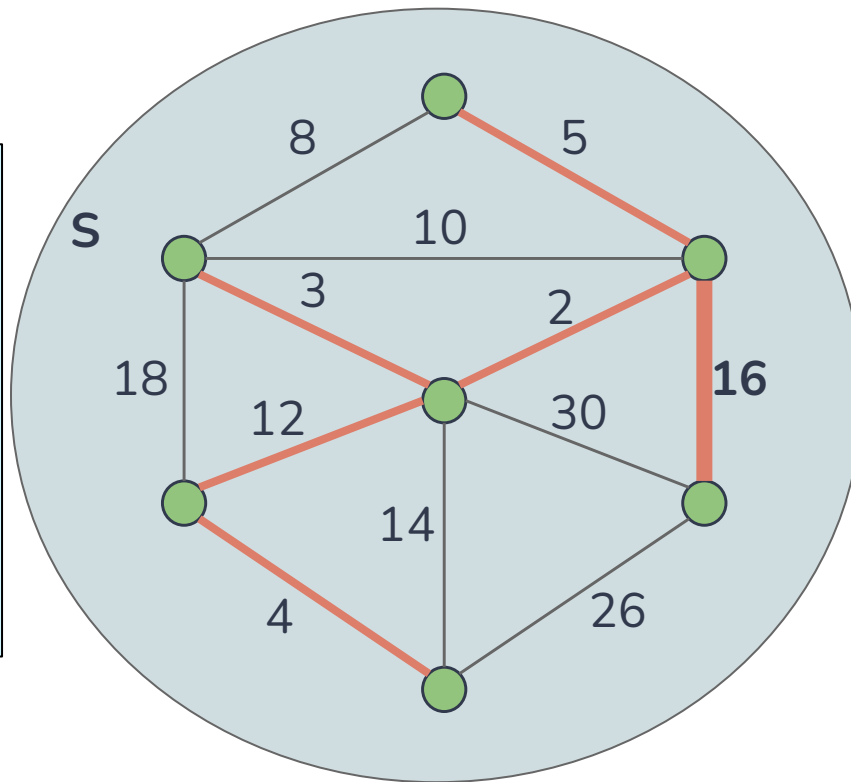
while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm

Prim's Algorithm:

T = empty

S = some vertex v $\setminus$ set of visited vertices

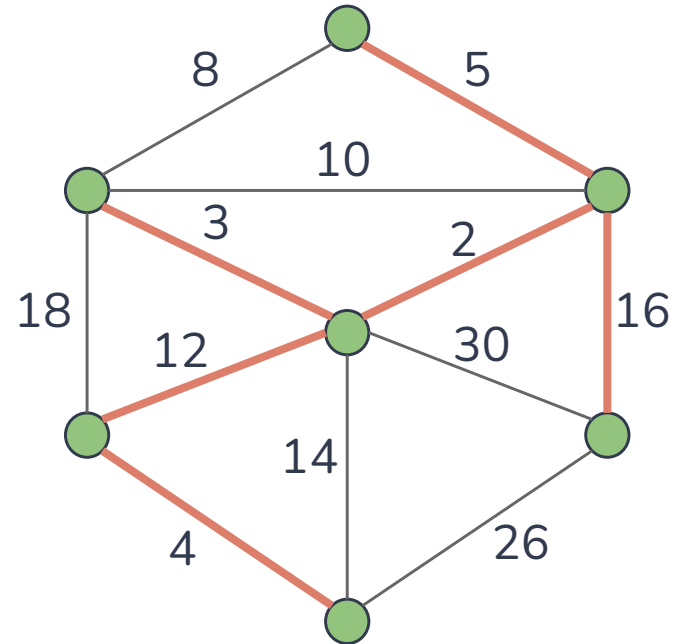
while T is not spanning:

 select, as your cut, S and $V \setminus S$

 pick the minimum-weight edge along that cut

 add that edge to T and the other vertex to S

return T



Prim's Algorithm vs. Dijkstra's Algorithm

Prim's Algorithm($G=(V,E)$, starting vertex s):

$S = \{s\}$ $\quad \quad \quad \backslash \backslash$ set of visited vertices
 $T = \text{empty}$ $\quad \quad \quad \backslash \backslash$ spanning edges
while $S \neq V$:
 select, as your cut, S and $V \setminus S$
 pick the minimum-weight edge along that cut
 add that edge to T and the other vertex to S
return T

As a result, they also have the same runtime: $O(E \log V)$ with the right data structures.

Notice the structural similarities between these algorithms! They follow the same principle: slowly grow your “visited region” until you’ve seen every vertex on the graph.

Dijkstra's($G=(V,E)$, starting vertex s):

$S \leftarrow \{s\}$ $\quad \quad \quad \backslash \backslash$ set of visited vertices
 $d[s] \leftarrow 0$ $\quad \quad \quad \backslash \backslash$ distances from s to every vertex
while $S \neq V$:
 get the unvisited vertex v minimizing $d[u] + l_{(u,v)}$
 add v to S
 $d[v] \leftarrow d[u] + l_{(u,v)}$

Lecture Recap

Today we saw three different algorithms for Minimum Spanning Trees.

They all follow the same underlying principle: the minimum-weight edge along any cut must go in the spanning tree.

Their implementations differ though. For this class, you may use whichever you prefer.

As a final note, in practice Borůvka's Algorithm (1926, before everyone listed today) tends to be faster, but not asymptotically, and it's the hardest to implement. Kruskal published his algorithm because he thought Borůvka's was "unnecessarily elaborate".